./

Learning Report – EMBEDDED LINUX AND KERNEL PROGRAMMING



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ver. Rel. No.** | **Release Date** | **Prepared. By** | **Reviewed By** | **To be approved By** | **Remarks/Revision Details** |
|  |  | Name/PS No | Name/PS No | Module Owner Name | Comments |
| 1 | 07/04/21 | Neha Tabassum |  |  |  |

**INDEX**

Table of Contents

[Table of Contents 3](#_Toc68945062)

[Table of Figures 4](#_Toc68945063)

[Table of Tables 4](#_Toc68945064)

[INTRODUCTION TO QEMU 5](#_Toc68945065)

[QEMU SETUP 8](#_Toc68945066)

[BOOT EMULATION 9](#_Toc68945067)

[KERNEL BUILDING 11](#_Toc68945068)

[CROSS COMPILING 15](#_Toc68945069)

[U-BOOT 17](#_Toc68945070)

[SD CARD CREATION 17](#_Toc68945071)

[BOOTING KERNEL USING NETWORKING 20](#_Toc68945072)

[TFTP ON HOST 20](#_Toc68945073)

[KERNEL PROGRAMMING 24](#_Toc68945074)

[COMPONENTS OF THE APPLICATION: 24](#_Toc68945075)

[KERNEL FUNCTONS 27](#_Toc68945076)

[SYSTEM CALLS 29](#_Toc68945077)

[PSEUDO FILESYSTEMS 29](#_Toc68945078)

[LINUX DRIVER MODEL 30](#_Toc68945079)

[CHAR DEVICE DRIVERS 30](#_Toc68945080)

[CONCURRENCY & IPC IN KERNEL 31](#_Toc68945081)

[IOCTL OPERATIONS 32](#_Toc68945082)

## **Table of Figures**

Figure 1: QEMU Logo 6

Figure 2: LINARO 9

Figure 3: Custom Setup 12

Figure 4: Device Driver Setup 12

Figure 5: Kernel Building SS 14

Figure 6: Qemu Window 15

Figure 7: Static and Dynamic files 17

Figure 8: Partitioned SD Card 19

Figure 9: TFTP Commands 21

Figure 10: Network established 22

Figure 11: Bootloading 23

Figure 12: Components of OS 24

Figure 13: Monolithic Kernel 26

Figure 14: Micro Kernel 26

Figure 15: Linux Kernel Architechture 28

Figure 16: Kernel Device Driver 30

Figure 17: (a) Message Passing (b) Shared Memory 32

## **Table of Tables**

Table 1: Uname Commands 10

**EMBEDDED LINUX**

**QEMU BASED EMULATION**

# INTRODUCTION TO QEMU

QEMU is a generic and open source machine emulator and virtualizer. QEMU is a hosted virtual machine monitor, it emulates the machine's processor through dynamic binary translation and provides a set of different hardware and device models for the machine, enabling it to run a variety of guest operating systems. It also can be used with Kernel-based Virtual Machine (KVM) to run virtual machines at near-native speed (by taking advantage of hardware extensions such as Intel VT-x). QEMU can also do emulation for user-level processes, allowing applications compiled for one architecture to run on another.

**Licensing :**

QEMU was written by [Fabrice Bellard](https://en.wikipedia.org/wiki/Fabrice_Bellard) and is [free software](https://en.wikipedia.org/wiki/Free_software), mainly licensed under the [GNU General Public License](https://en.wikipedia.org/wiki/GNU_General_Public_License) (GPL for short). Various parts are released under the [BSD license](https://en.wikipedia.org/wiki/BSD_license), [GNU Lesser General Public License](https://en.wikipedia.org/wiki/GNU_Lesser_General_Public_License) (LGPL) or other GPL-compatible licenses. This is the reason that we include following commands in some codes:

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Your name");

MODULE\_DESCRIPTION("A Simple Module");

QEMU is a FAST! processor emulator using dynamic translation to achieve good emulation speed.

QEMU has multiple operating modes:

* **Xen Hosting:** QEMU is involved only in the emulation of hardware; the execution of the guest is done within Xen and is totally hidden from QEMU.
* **KVM Hosting:** Here QEMU deals with the setting up and migration of [KVM](https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine) images. It is still involved in the emulation of hardware, but the execution of the guest is done by KVM as requested by QEMU.
* **System emulation:** In this mode QEMU emulates a full computer system, including peripherals. It can be used to provide virtual hosting of several virtual computers on a single computer. QEMU can boot many guest operating systems, including Linux, Solaris, Microsoft Windows, DOS, and BSD, it supports emulating several instruction sets, including x86, MIPS, 32-bit ARMv7, ARMv8, PowerPC, SPARC, ETRAX CRIS and MicroBlaze.
* **User-mode emulation:** In this mode QEMU runs single Linux or Darwin/macOS programs that were compiled for a different instruction set. System calls are thunked for endianness and for 32/64 bit mismatches. Fast cross-compilation and cross-debugging are the main targets for user-mode emulation.



Figure 1: QEMU Logo

**Qemu has the following Features:**

QEMU can save and restore the state of the virtual machine with all programs running. Guest operating systems do not need patching in order to run inside QEMU.

QEMU full system emulation has the following features:

* QEMU uses a full software MMU for maximum portability.
* QEMU can optionally use an in-kernel accelerator, like kvm. The accelerators execute most of the guest code natively, while continuing to emulate the rest of the machine.
* Various hardware devices can be emulated and in some cases, host devices (e.g. serial and parallel ports, USB, drives) can be used transparently by the guest Operating System. Host device passthrough can be used for talking to external physical peripherals (e.g. a webcam, modem or tape drive).
* Symmetric multiprocessing (SMP) support. Currently, an in-kernel accelerator is required to use more than one host CPU for emulation.

QEMU user mode emulation has the following features:

* Generic Linux system call converter, including most ioctls.
* clone() emulation using native CPU clone() to use Linux scheduler for threads.
* Accurate signal handling by remapping host signals to target signals.

QEMU can run without a host kernel driver and yet gives acceptable performance. It uses dynamic translation to native code for reasonable speed, with support for self-modifying code and precise exceptions.

It is portable to several operating systems (GNU/Linux, \*BSD, Mac OS X, Windows) and architectures.

It performs accurate software emulation of the FPU.

QEMU supports the emulation of various architectures, including:

1. IA-32 (x86) PCs
2. x86-64 PCs
3. MIPS64 Release 6[7] and earlier variants
4. Sun's SPARC sun4m
5. Sun's SPARC sun4u
6. ARM development boards (Integrator/CP and Versatile/PB)
7. SH4 SHIX board
8. PowerPC (PReP and Power Macintosh)
9. ETRAX CRIS
10. MicroBlaze
11. RISC-V

The virtual machine can interface with many types of physical host hardware, including the user's hard disks, CD-ROM drives, network cards, audio interfaces, and USB devices. USB devices can be completely emulated, or the host's USB devices can be used, although this requires administrator privileges and does not work with all devices.

QEMU can emulate network cards (of different models) which share the host system's connectivity by doing network address translation, effectively allowing the guest to use the same network as the host. The virtual network cards can also connect to network cards of other instances of QEMU or to local TAP interfaces. Network connectivity can also be achieved by bridging a TUN/TAP interface used by QEMU with a non-virtual Ethernet interface on the host OS using the host OS's bridging features.

QEMU integrates several services to allow the host and guest systems to communicate; for example, an integrated SMB server and network-port redirection (to allow incoming connections to the virtual machine). It can also boot Linux kernels without a bootloader.

QEMU does not depend on the presence of graphical output methods on the host system. Instead, it can allow one to access the screen of the guest OS via an integrated VNC server. It can also use an emulated serial line, without any screen, with applicable operating systems.

While setting up qemu, one should have a clean workspace as well. Following are the do’s and don’ts while working with the workspace or setting up the workspace:

* Choose a clean directory for all the work.
* Don’t use Desktop, Downloads, Documents, Music, Videos, Pictures etc, which are meant for other purpose.
* You may choose directory like workspace/eworkspace/kworkspace/ebuildws/ews or with any sensible name under home directory
* Avoid spaces or special symbols in path names Under this workspace keep different sub directories for downloaded packages, extracted source to build, configuration files, examples etc.

# QEMU SETUP

Setting up Qemu with step by step installation steps is very important for proper functioning. Following are the steps for installation:

1.) Enter the following commands for successful installation of the Qemu.

**” sudo apt install qemu-system-arm”**

2.) After installation, enter any of the commands to check whether it is properly installed or not.

**“qemu-system-arm -v”**

**“qemu-system-arm -M ? “**

**“qemu-system-aarch64 -v”**

3.) Download core-image-minimal-qemuarm.ext4 from the below link for rootfs image:

“ <http://downloads.yoctoproject.org/releases/yocto/yocto-2.5/machines/qemu/qemuarm/> “

4.) Now rename core-image-minimal-qemuarm.ext4 as rootfs.img

5.) Use the following commands to Align the size of rootfs:

**“ e2fsck -f rootfs.img “**

**“ resize2fs rootfs.img 16M “**

This step is to make sure there is enough space for the files to be accomodated.

Issues faced: rootfs image was not installed properly or got corrupted while installation.

Debugging: The rootfs image was installed again and loaded propery.

**ToolChain**

The Linaro toolchain:

**Linaro** is an engineering organization that works on [free and open-source software](https://en.wikipedia.org/wiki/Free_and_open-source_software) such as the [Linux kernel](https://en.wikipedia.org/wiki/Linux_kernel), the [GNU Compiler Collection](https://en.wikipedia.org/wiki/GNU_Compiler_Collection) (GCC), power management, graphics and multimedia interfaces for the [Arm](https://en.wikipedia.org/wiki/Arm_architecture) family of [instruction sets](https://en.wikipedia.org/wiki/Instruction_set) and implementations thereof as well as for the [Heterogeneous System Architecture](https://en.wikipedia.org/wiki/Heterogeneous_System_Architecture) (HSA). The company provides a collaborative engineering forum for companies to share engineering resource and funding to solve common problems on Arm software.



Figure 2: LINARO

The Linaro Toolchain works on all aspects of system-level tools - the core development toolchain (compiler, assembler, linker, debugger), core system libraries (dynamic linker, c-library), emulation, profiling and analysis (oprofile, performance events) and instrumentation (ftrace). The team also provides Linaro toolchain binary releases and Linaro Toolchain package releases. Linaro Toolchain works directly with upstream communities: GCC, Binutils, GDB, glibc, LLVM, QEMU.

**Install linaro toolchain from ubuntu package manager**

“ sudo apt install gcc-arm-linux-gnueabi “ --------> for soft float

“ sudo apt install gcc-arm-linux-gnueabihf “----- > for hard float

Rootfs already comes pre-loaded with floating point computation so hard float is not required.

# BOOT EMULATION

Mainly two files are required for boot emulation the kernel image and the supporting file like vexpress-v2p-ca9.dtb. One should make sure that both the files are in the same folder.

Emulate using Qemu – sdcard approach:

The command to emulate qemu using sdcard approach is given below,

**“qemu-system-arm -M vexpress-a9 -m 1024 -serial stdio \**

**-kernel zImage -dtb vexpress-v2p-ca9.dtb \**

**-sd rootfs.img -append "console=ttyAMA0 root=/dev/mmcblk0 rw" “**

Emulate using Qemu – initrd approach:

The command for emulation using initrd is given below,

**“ qemu-system-arm -M vexpress-a9 -m 1024 -serial stdio \**

**-kernel zImage -dtb vexpress-v2p-ca9.dtb \**

**-initrd rootfs.img -append "console=ttyAMA0 root=/dev/ram0 rw" “**

Here we are using the memory size of 1GB so we put as 1024.

Generally we used Qemu sdcard approach for all the simulations.

Following are the mandatory steps which should be checked on target window:-

1. **Uname -r** :-returns system information in the structure pointed to by *buf*. The *utsname* struct is defined in *<sys/utsname.h>.* Eg.: uname –r, uname –v, uname –a.
2. ***/proc/cpuinfo***bz**:-** It is a short, read-only, [plain text](http://www.linfo.org/plain_text.html) file that contains [information](http://www.linfo.org/information.html) about the CPUs (central processing units) on a computer.

It can easily be read with a command such as [*cat*](http://www.linfo.org/cat.html), i.e.,

cat /proc/cpuinfo

1. **free –m:-** The *free* [command](http://www.linfo.org/command.html) provides [information](http://www.linfo.org/information.html) about unused and used [*memory*](http://www.linfo.org/memory.html) and [*swap space*](http://www.linfo.org/swap_space.html) on any [computer](http://www.linfo.org/computer.html) running [Linux](http://www.linfo.org/linuxdef.html) or another [Unix-like](http://www.linfo.org/unix-like.html) [operating system](http://www.linfo.org/operating_systems_list.html).
2. **Df –kh :-** **To display information of device name, total blocks, total disk space, used disk space, available disk space and mount points on a file system.**
3. **Mount:- mount command** is used to **mount** the filesystem found on a device to big tree structure(**Linux** filesystem) rooted at '/'.
4. **Dmesg:- dmesg command** also called as “driver message” or “display message” is used to examine the kernel ring buffer and print the message buffer of kernel.

Table 1: Uname Commands

|  |
| --- |
| **Uname commands:-** |
|  |

# KERNEL BUILDING

**Building Custom Kernel :**

1.) Download the latest kernel source for best performance from linux kernel website.

2) For better compatibility with Qemu download 4.14.x version. From the bellw link:

wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.14.202.tar.xz tar –xvf linux-4.14.202.tar.xz

3.) Obtaining configuration file:

Locate default config available in KSRC/arch/arm/configs, Or collect any well tested configuration file as base config, here we use vexpress\_defconfig for Versatil Express target being used for Qemu emulation.

4.) Enter the commands:

**make ARCH=arm mrproper**

**make ARCH=arm vexpress\_defconfig**

5.) Now copy custom config file as .config under KSRC

**Note:-** Please note that mrproper will remove built files, including the configuration. So run this only for any new build.

6) Now for further customization, following commands should be run :-

“make ARCH=arm menuconfig”

Issues:- If the above command doesn’t work then install the following library:

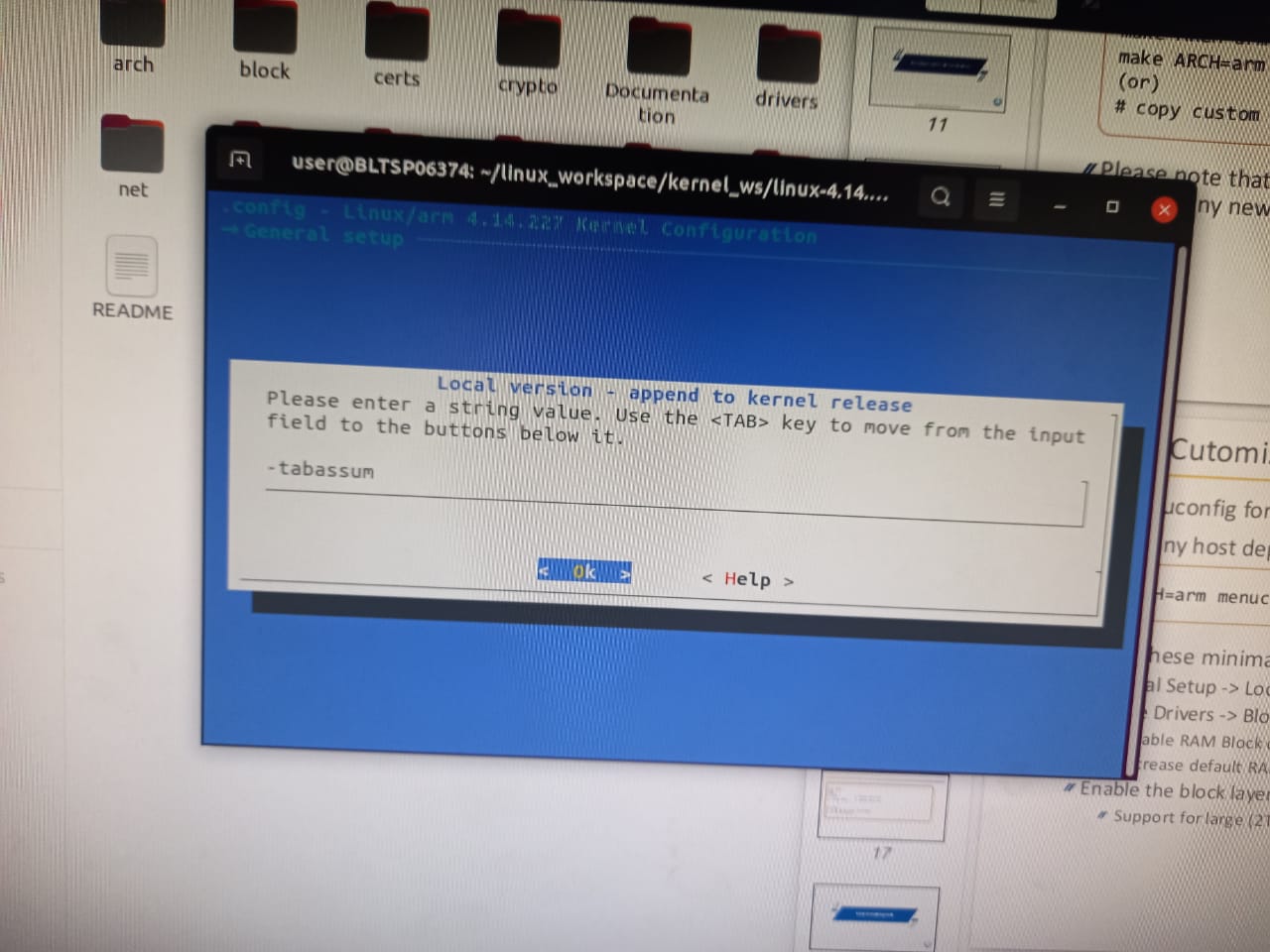
**sudo apt install libncurses5-dev,**

**sudo apt install flex,**

**sudo apt install bison**

Now following are the minimal changes needed:

General Setup -> Local Version -> "-custom”

  
Figure 3: Custom Setup

1. Device Drivers -> Block Devices -> Enable RAM Block device support
2. Increase default RAM disk size to suitable limit, say 65536
3. Enable the block layer
4. Support for large (2TB+)

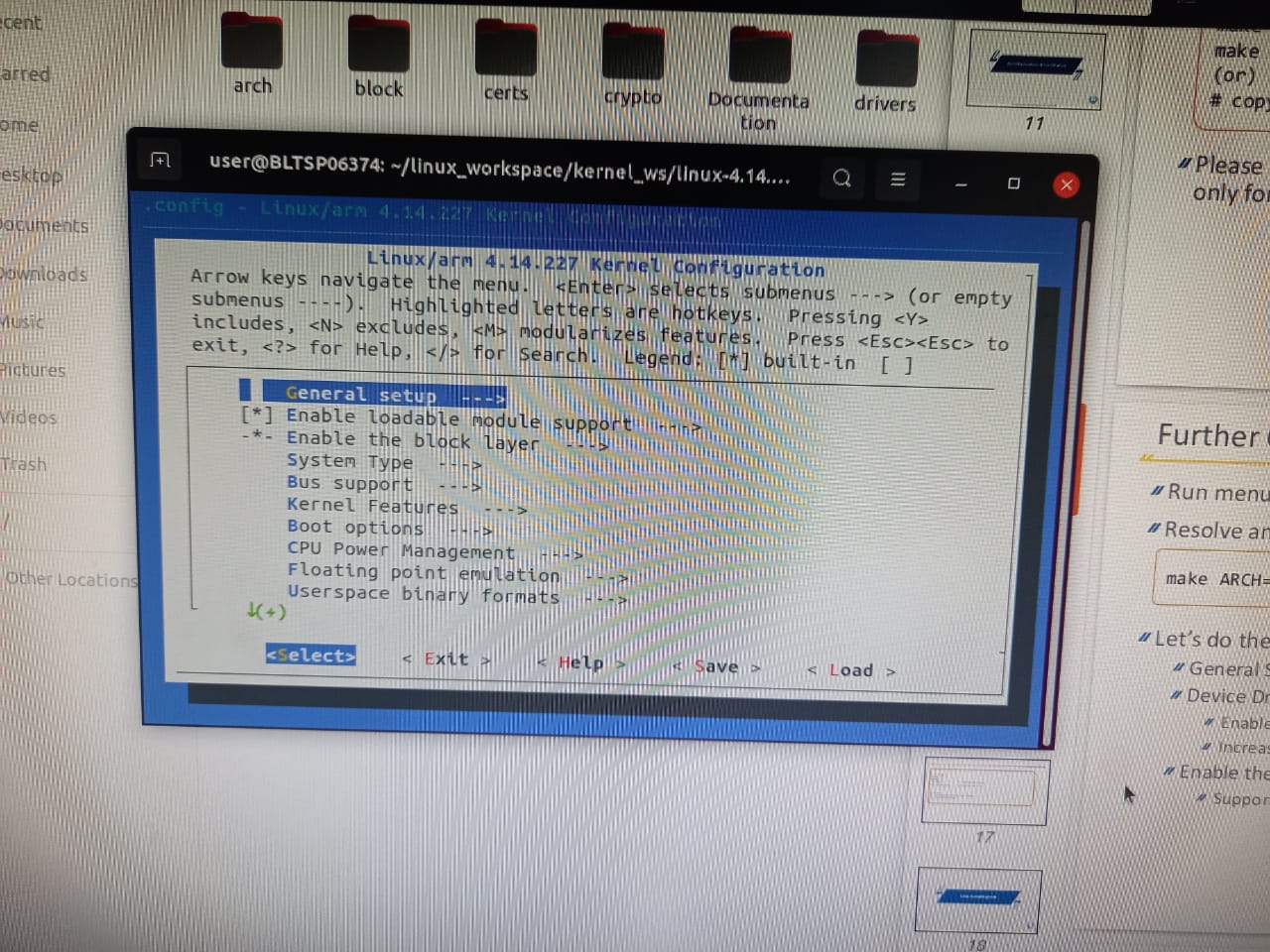


Figure 4: Device Driver Setup

**To Build the kernel:**

1.) Build kernel image by entering the following command:

**make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi- zImage -j <n>**

Here ‘n’ is the number of cores the central processing unit has.

N has been set to 6 or 4 in or system while building the kernel.

2.) Now build the Device Tree binaries by the following command:

**make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi- dtbs firmware**

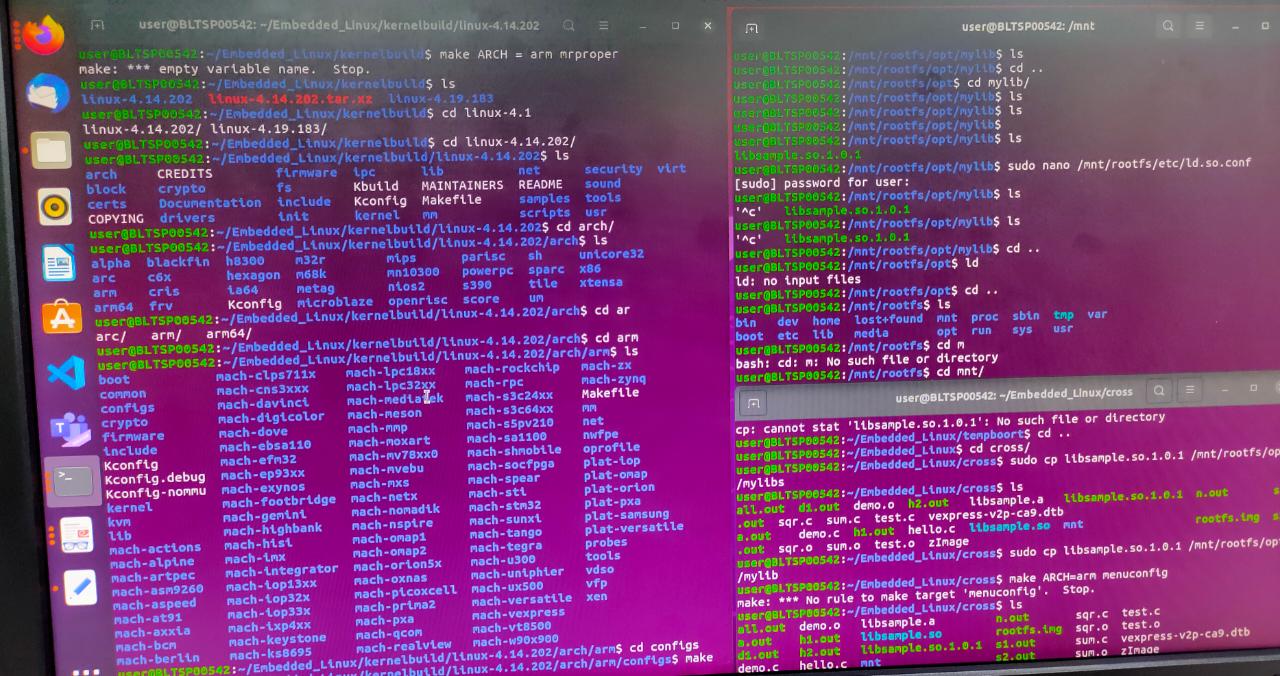
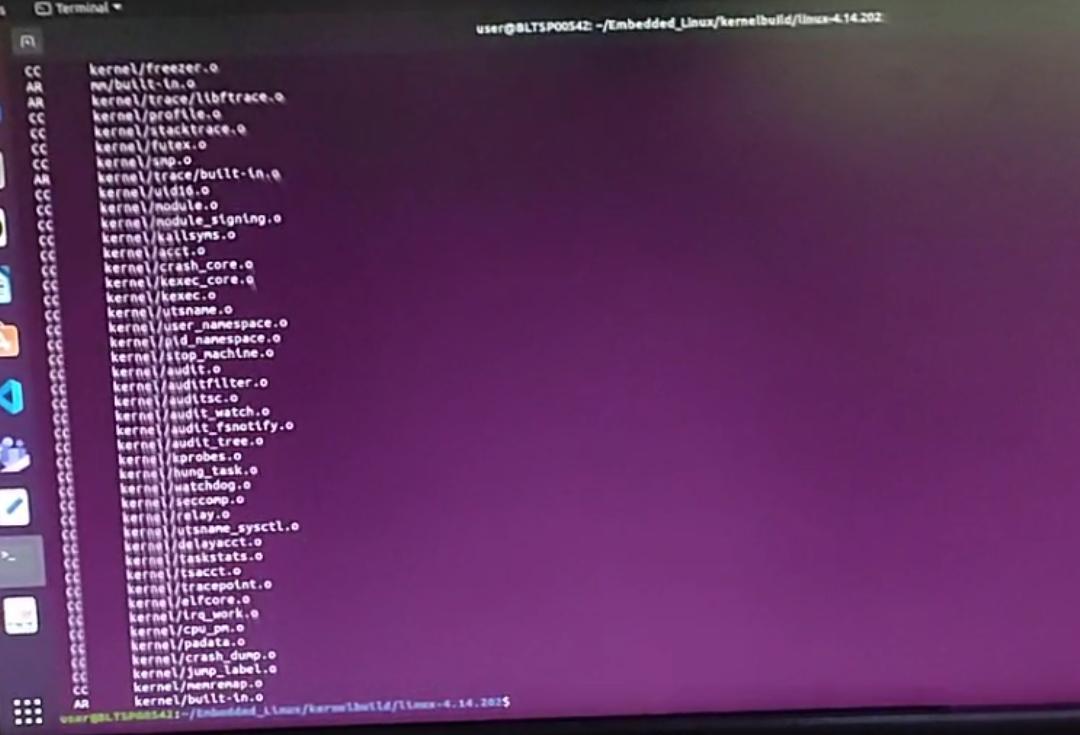


Figure 5: Kernel Building SS

**Testing the Build:**

Here we test the build by running in an emulator Qemu.

**“ cp $KSRC/arch/arm/boot/zImage “**

**“ cp $KSRC/arch/arm/boot/dts/\*.dtb “**

Make sure that rootfs.img is also in the same location

Now to emulate, check the following command,

**qemu-system-arm -M vexpress-a9 -m 1024 -serial stdio \**

**-kernel zImage -dtb vexpress-v2p-ca9.dtb \**

**-sd rootfs.img -append "console=ttyAMA0 root=/dev/mmcblk0 rw"**

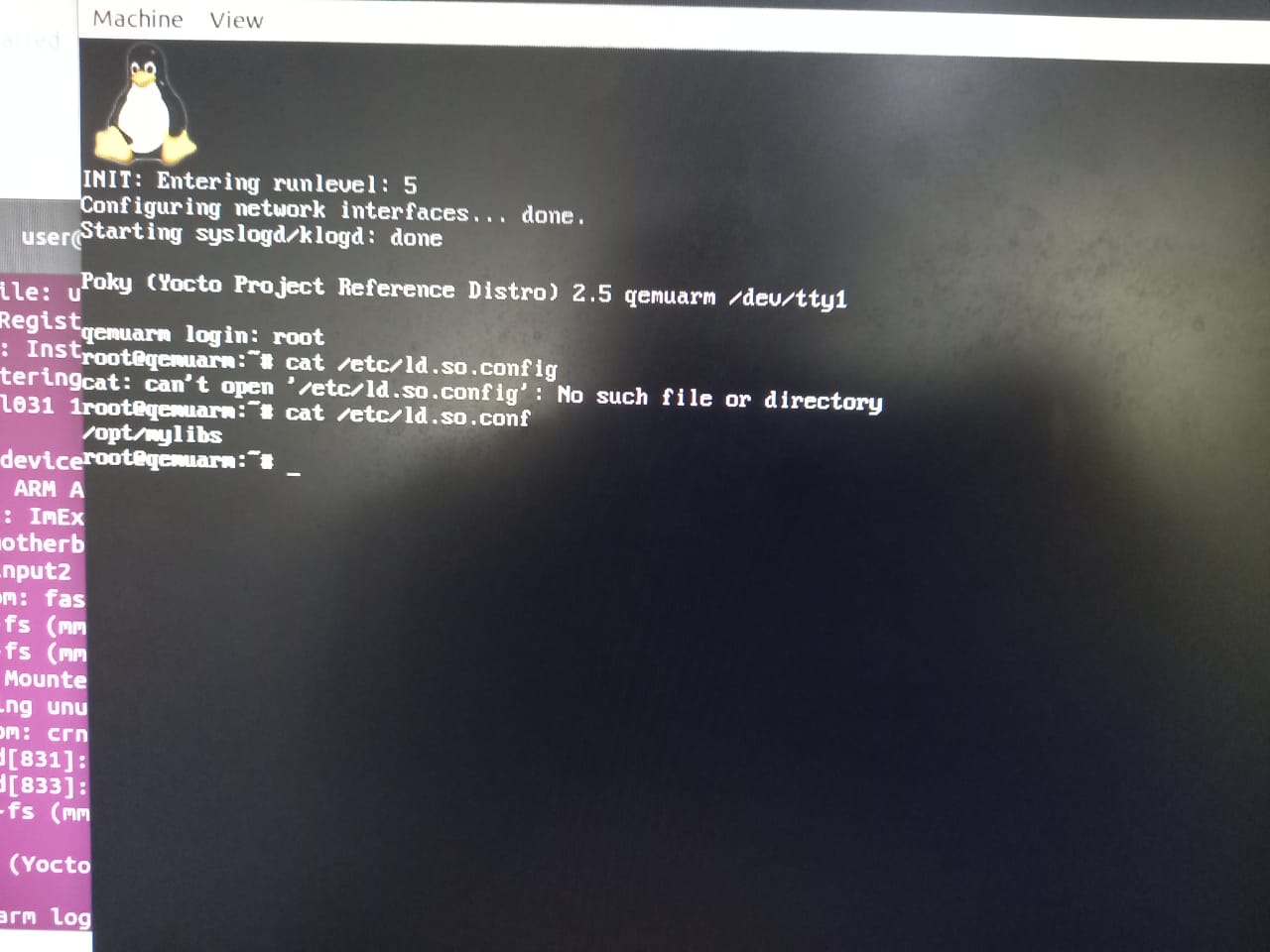


Figure 6: Qemu Window

# CROSS COMPILING

The cross compiling is a very essential aspect in embedded linux development. It is very helpful to create files which are emulated to run in machines other than the host.

Cross compiling is the technique in which coding or development is done in one architecture and it is compiled to work in another other than the host architecture. Every board cannot be with us all the time and it is also not feasible too. Hence we require special softwares which can simulate the conditions or architecture of the target device. These softwares are called as emulators.

In our design, we use Qemu emulator.

Static Library:

A static library or statically-linked library is a set of routines, external functions and variables which are resolved in a caller at compile-time and copied into a target application by a compiler, linker, or binder, producing an object file and a stand-alone executable. This executable and the process of compiling it are both known as a static build of the program. Historically, libraries could only be static.

In order to create static libraries, input the following command:

**“ arm-linux-gnueabi-ar rc libsample.a sum.o sqr.o -o “**

Here libsample.a is the static library that will be created,

sum.o, sqr.o are the output files.

Now input these commands,

**arm-linux-gnueabi-gcc -L. test.o –lsample -o s1.out**

**arm-linux-gnueabi-gcc -L. test.o –lsample -o s2.out –static**

Dynamic Library:

A Dynamic Library is a collection of object files that can be linked to any program at run-time by inserting the location of the function in memory to the executable. This provides a way for code to be used even though it could be loaded anywhere in memory.

Once a dynamic library is loaded into memory, it’s code can be used by any program that needs it. This is much different than Static Libraries and keeps the size of executables low.

Because Dynamic Libraries provide a way to use code that’s anywhere in memory, if the code is updated, you don’t need to recompile. Just because the code was updated, doesn’t necessarily mean it’s location has been changed, so the library still knows where to direct the program. This means that updating code is much simpler and requires less time.

To create Dynamic library, input the following commands,

**arm-linux-gnueabi-gcc -shared libsample.so sum.o sqr.o**

**arm-linux-gnueabi-gcc -L. test.o –lsample -o d1.out**

Here libsample.so is the dynamic library,

d1.out is the dynamically outputted file

In order to load the dynamic library in to the emulator environment, we use the following command,

**“ LD\_LIBRARY\_PATH=. ./d1.out”**

Versioned so files:

This method is developed to maintain the latest version of the dynamic libraries. When we regularly or constantly update the library files, it becomes difficult to always monitor the previously existing dynamic files in the directory. In order to sort this out, the versioned so files concept is practiced.

Whenever a new dynamic library is created, it is maintained in the form of versions rather than directly storing it in the native format. This allows us to update the library files more efficiently.

To create the versioned files, follow the following commands,

**“arm-linux-gnueabi-gcc -shared -Wl,-soname,libsample.so \**

**-o libsample.so.1.0.1 sum.o sqr.o”**

**“ln -s libsample.so.1.0.1 libsample.so”**

**“arm-linux-gnueabi-gcc –L. test.o –lsample –o d1.out”**

This step creates us the dynamic library file as d1.out

Now in order to make sure that the kernel knows where the dynamic library is located,

we need to give the output path where the dynamic library is created.

So here the path is “/opt/mylibs”.

This path we will create in “/mnt/rootfs/etc/ld.so.conf” i.e, in the configuration file.

Following are the .o files and .so files which we created while compiling static and dynamic codes:

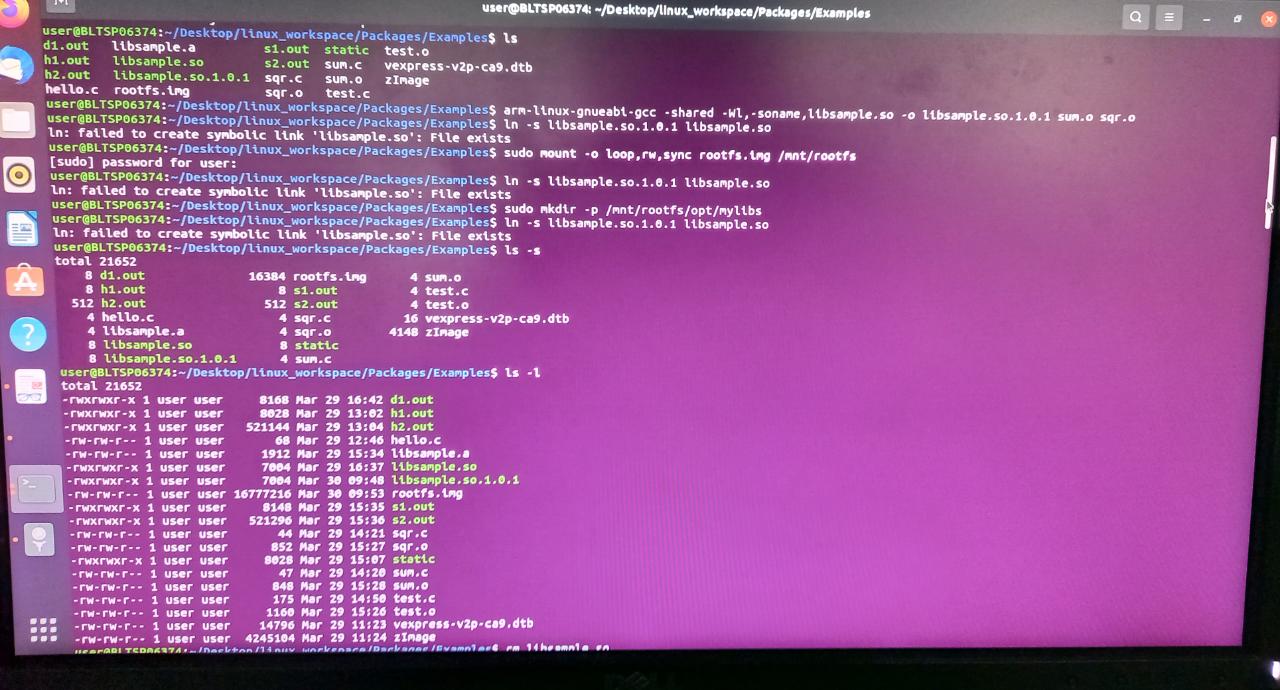


Figure 7: Static and Dynamic files

# U-BOOT

This is a custom bootloading interface which is used to load the bootloader as soon as the system starts.

It is available to download from “<http://ftp.denx.de/pub/u-boot/>”

# SD CARD CREATION

**Creating SD card:**

We will experiment the embedded linux by creating a virtual SD card, partitioning it into two partitions.

The command to create a SD card image is

**“qemu-img create simplesd.img 64M”**

we are using vfat formatting, hence

**“ sudo mkfs.vfat simplesd.img”**

**“ sudo mount -o loop,rw,sync simplesd.img /mnt/sdcard”**

Now copy the kernel image zImage file, vexpress-v2p-ca9.dtb file and rootfs.img into the sd card directory.

copy simplesd.img to tempdir, where generated u-boot is copied

To unmount the SD card,

**“umount /mnt/sdcard”**

To create Partitions in the SD card:

We create a new SD card image sdcard with the following command,

**“****qemu-img create sdcard.img 128M”**

It is of size 128Mb.

Now check the partitions of it using the command,

**“****sudo fdisk -l sdcard.img”**

The command to partition is cfdisk. The following command

**“sudo cfdisk sdcard.img”**

lets you create new partitions in the sdcard.img. Here we create two partitions only.

In the first partition, we have zImage and vexpress-v2p-ca9.dtb files.

In the second partition we have core-image-minimal-qemuarm.tar

For partitions,

**“****sudo losetup -o 1048576 /dev/loop31 sdcard.img ”**

**“ sudo losetup -o 17825792 /dev/loop32 sdcard.img ”**

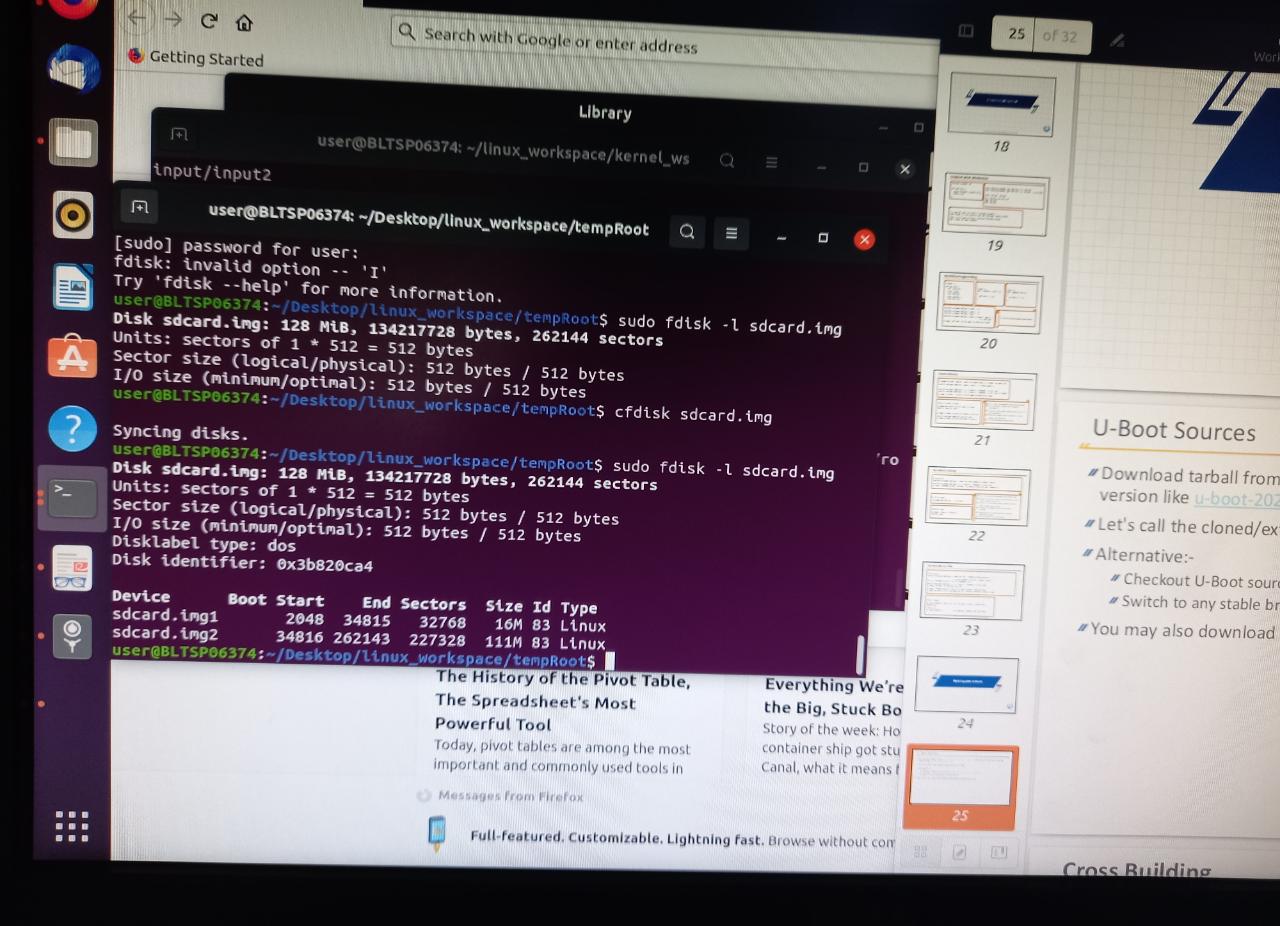


Figure 8: Partitioned SD Card

We need to download a file “core-image-minimal-qemuarm.tar.bz2” and extract it into mnt/rootfs directory.

**“qemu-system-arm -M vexpress-a9 -m 1024 -serial stdio -kernel u-boot -sd sdcard.img”**

The above command revokes the emulator with newly created sdcard image.

Now run the commands in the emulator i.e the U-Boot shell.

**“ mmcinfo”**

**“ fatls mmc 0:1”**

**“ fatload mmc 0:1 0x60200000 zImage”**

**“ fatload mmc 0:1 0x60100000 vexpress-v2p-ca9.dtb”**

**“ setenv bootargs ‘console=ttyAMA0 root=/dev/mmcblk0p2 rw rootfstype=ext4’ ”**

**“ bootz 0x60200000 – 0x60100000”**

# BOOTING KERNEL USING NETWORKING

We can also remotely boot the kernel via networking. For this, we will use TFTP protocol.

Trivial File Transfer Protocol (TFTP) is a simple protocol used for transferring files. TFTP uses the User Datagram Protocol (UDP) to transport data from one end to another. TFTP is mostly used to read and write files/mail to or from a remote server.

It allows a client to get a file from or put a file onto a remote host. One of its primary uses is in the early stages of nodes booting from a Local Area Network. TFTP has been used for this application because it is very simple to implement.

Due to its simple design, TFTP can be easily implemented by code with a small memory footprint. It is therefore the protocol of choice for the initial stages of any network booting strategy like BOOTP, PXE, BSDP, etc., when targeting from highly resourced computers to very low resourced Single-board computers (SBC) and System on a Chip (SoC).

It is also used to transfer firmware images and configuration files to network appliances like routers, firewalls, IP phones, etc. Today, TFTP is virtually unused for Internet transfers.

# TFTP ON HOST

**Setup TFTP on host:**

1. First we need to install the tftpd, in order to install execute the following command,

**“sudo apt install tftpd”**

2.) Now create the tftp file with the required code in the “/etc/xinetd.d/tftp” location

1. Restart to update the changes,

**“/etc/init.d/xinetd restart”**

TUN/TAP provides packet reception and transmission for user space programs. It can be seen as a simple Point-to-Point or Ethernet device, which, instead of receiving packets from physical media, receives them from user space program and instead of sending packets via physical media writes them to the user space program.

4.) To run the tun command,

**“sudo modprobe tun”**

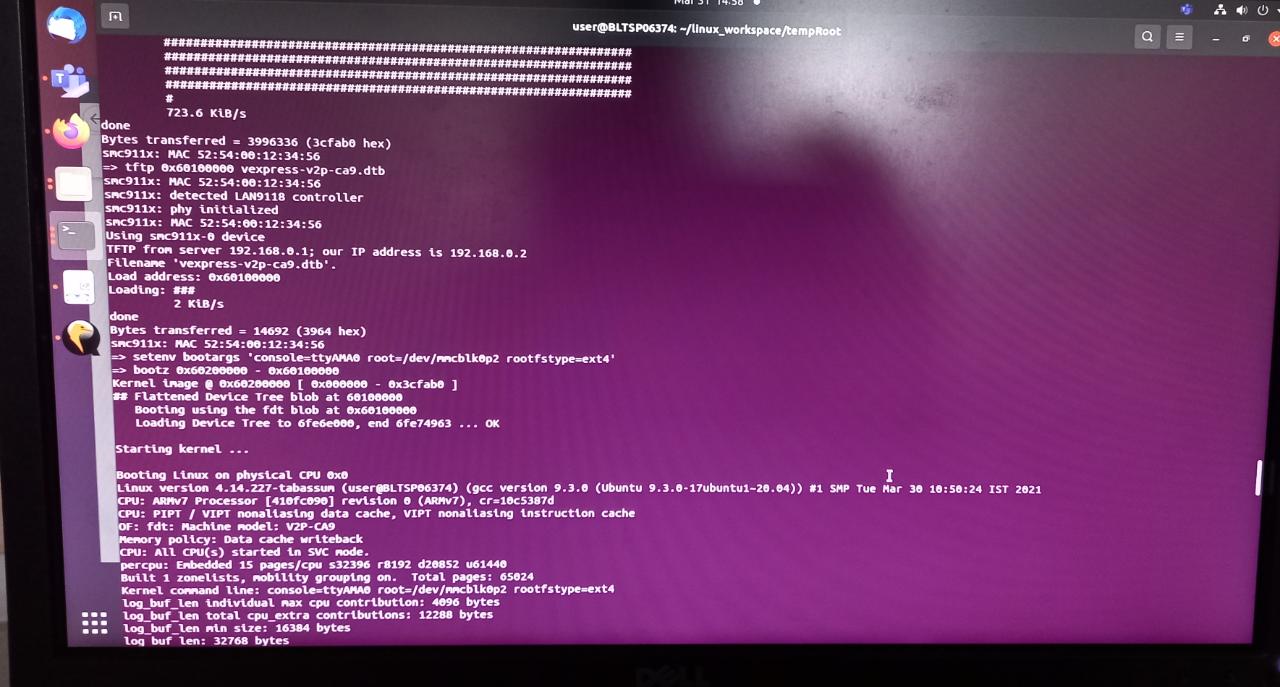
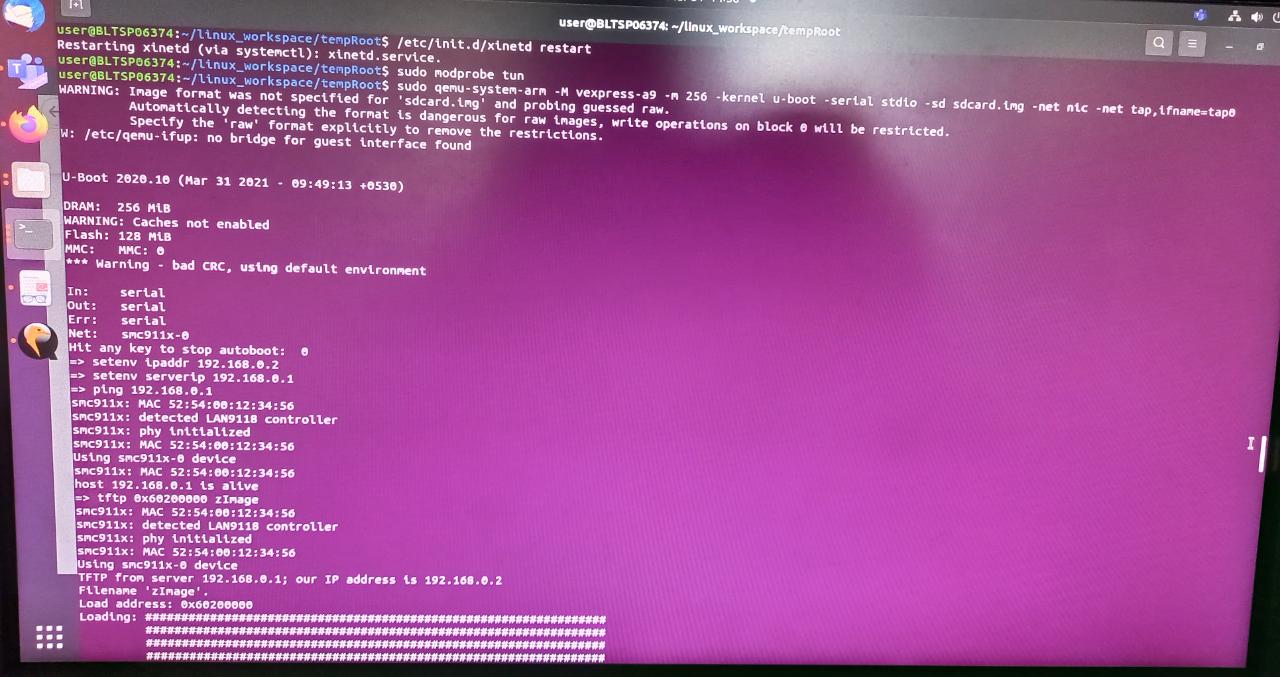


Figure 9: TFTP Commands

Now in order to setup the TFTP on the target machine, we need to follow series of commands.

5.) To run the interface in qemu by establishing a network tap0,

**“ sudo qemu-system-arm -M vexpress-a9 -m 256 -kernel u-boot -serial stdio \**

**-sd sdcard.img -net nic -net tap,ifname=tap0 “**

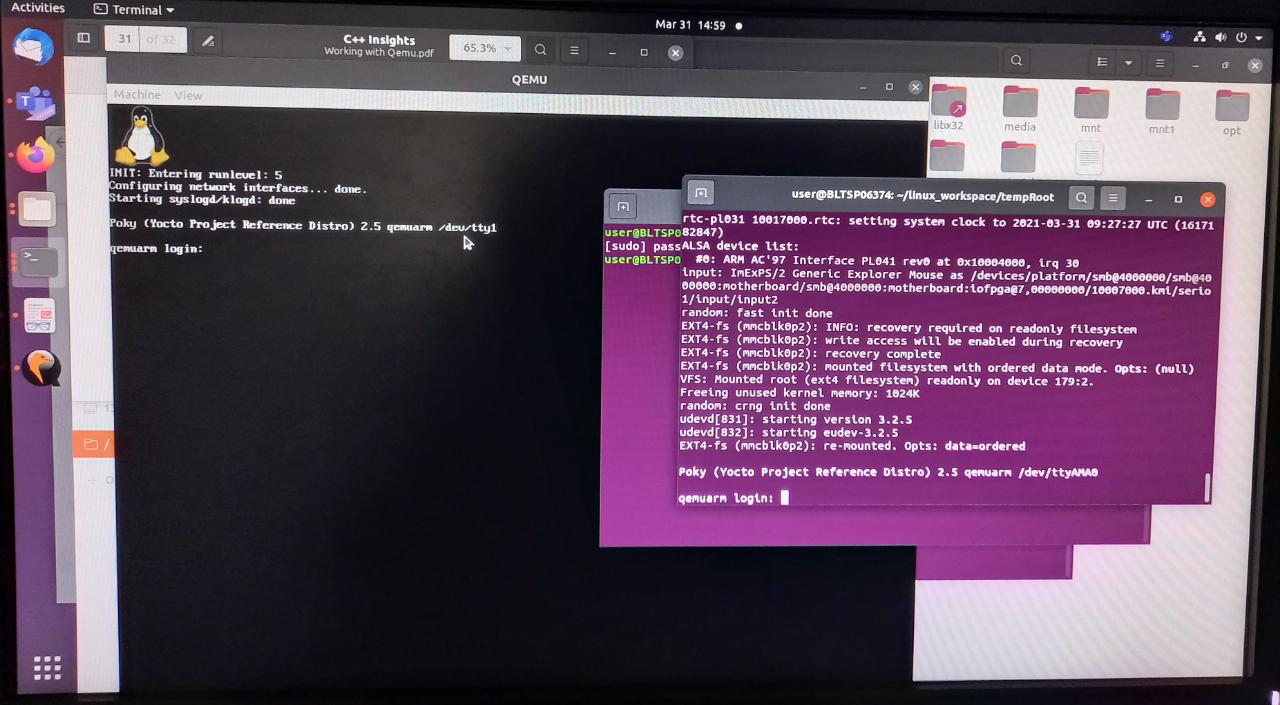


Figure 10: Network established

6.) To set the ipaddress, we take an environment variable ipaddr, the command is

**“ setenv ipaddr 192.168.0.2 ”**

7.) To set the server ip,

**“setenv serverip 192.168.0.1”**

8.) To check the status of the network connectivity between host and target, we can use the ping command.

**“ ping 192.168.0.1 ”**

9.) We load the zImage into board via Network by using the tftp protocol.

**“ tftp 0x60200000 zImage ”**

10.) In similar fashion, we load the vexpress-v2p-ca9.dtb.

**“ tftp 0x60100000 vexpress-v2p-ca9.dtb “**

11.) Now to boot into the target, use the following command,

**“ setenv bootargs ‘console=ttyAMA0 root=/dev/mmcblk0p2 rootfstype=ext4 ‘ “**

12.) The range of bootloader in the target board is from 0x60200000 to 0x60100000. Command is

**“ bootz 0x60200000 – 0x60100000 “**

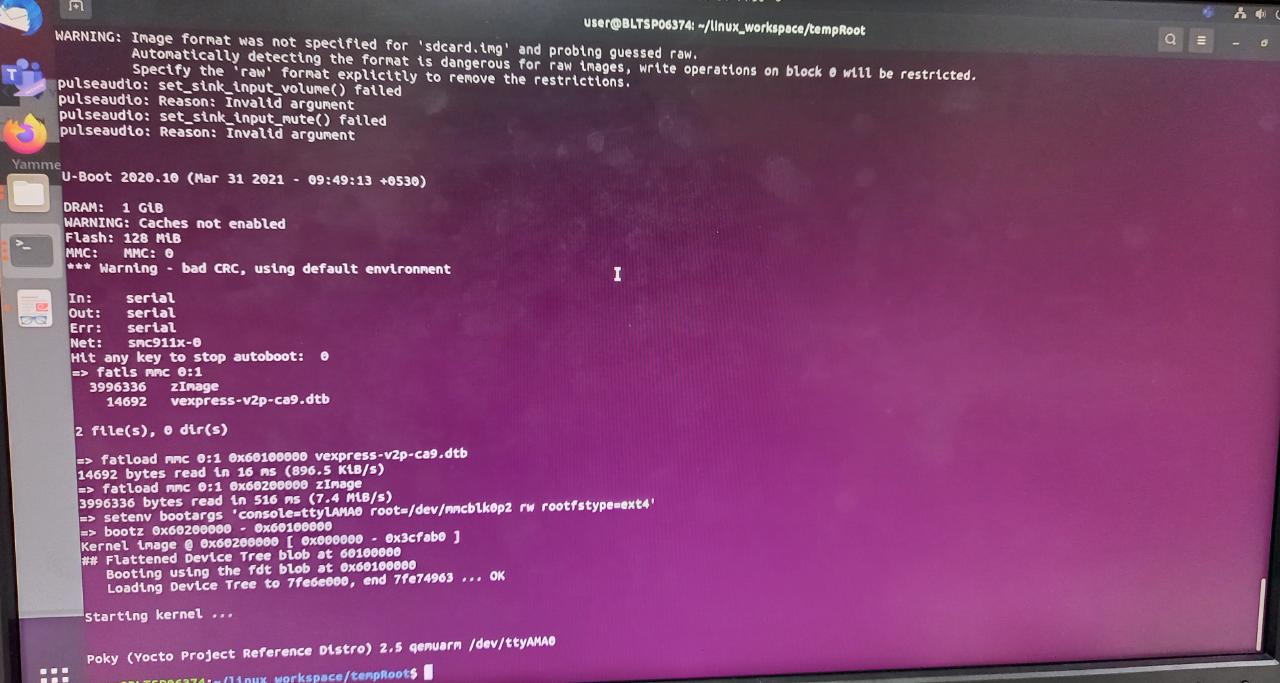


Figure 11: Bootloading

**Issues: sequence of addressing was wrong**

**Debugging: Hex address should be in order.**

# KERNEL PROGRAMMING

Kernel and user are the two most commonly used words in running applications. Their definition is straightforward: The kernel is part of the application with high privileges while the user (space) usually refers to applications with low privileges. Initiated by Linus Torvalds as a hobby project in 1991. It was inspired by Minix OS designed by Andrew S Tanenbaum and driven by large and dynamic community

An application can be defined as a large and complex program divided into smaller pieces using an interface. This program is used to share various OS items including file, input / output, process memory, etc. All split pieces should be well defined parts of the system with i / ps, o / ps & carefully separated function. We know that all programs do not have the same structure but most current operating systems share the system components described below.

The OS is the most important program on the computer because every computer works with an operating system (OS) to run all the programs and applications. The main functions of computer OS are to identify the / p from the keyboard, to send the o / p to the screen, to track files, to store drives, peripheral control devices, such as printers, etc.

# COMPONENTS OF THE APPLICATION:

Application components play a major role in making computer system components work together. The functional components are discussed below:

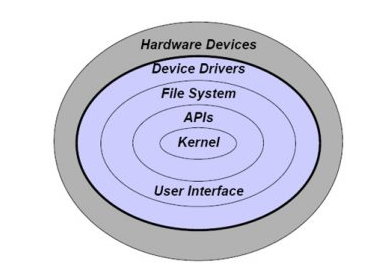


Figure 12: Components of OS

**Kernel:**

The kernel on the OS provides a basic level of control over all computer chemicals. In the app, the kernel is an important pre-loaded object and stays inside the main memory.

**User interface:**

The GUI or user interface (UI) is part of the OS that allows operators to access data. A text-based user interface that displays text and its commands typed in the command line with the help of the keyboard.

**Security:**

If a computer is crowded to allow a faster process for a variety of processes, more processes should be protected from other tasks. This system security is highly dependent on a variety of effective technologies. Current operating systems provide access to many services, which are available to run software on the system, and on external devices such as networks using the kernel.

**Networking:**

Networking can be defined as when a processor communicates with another through a communication line. Network-design design should consider route, connection methods, security, vision problems and security.

**Performing multiple tasks:**

Describes the performance of multiple independent computer programs on the same computer system. Performing multiple tasks on the OS allows operators to perform one or more computer tasks at a time. Since most computers can do one or two tasks simultaneously, this can usually be done with the help of time sharing, where each program uses computer time to perform.

**Memory Management:**

OS performance is nothing but memory management that controls the main memory and moves processes back and forth between disk and large memory during boot.

**Interrupts:**

In the app, interrupts is important because it provides a reliable OS process for communication and response in the environment around them. Disruption is nothing more than a single type of signal between an app and a computer program other than from a computer program that requires the OS to leave and determine exactly what to do next.

**Process Processing:**

The OS provides a interface between the hardware and the application system so that the system can connect to the hardware device simply by following the procedures and principles set out in the OS.

A Kernel operating system is a system that makes hardware and software work together. With the help of device drivers, the Linux kernel acts as a translator that allows communication between software and user interface and hardware. A monolithic kernel is a kernel that controls all hardware and driver functions as part of it, unlike microkernels that run a few basic functions while leaving all the rest as external functions, due to a lack of popular systems using microkernels we will ignore low communication between our OS (app) and our virtual device.

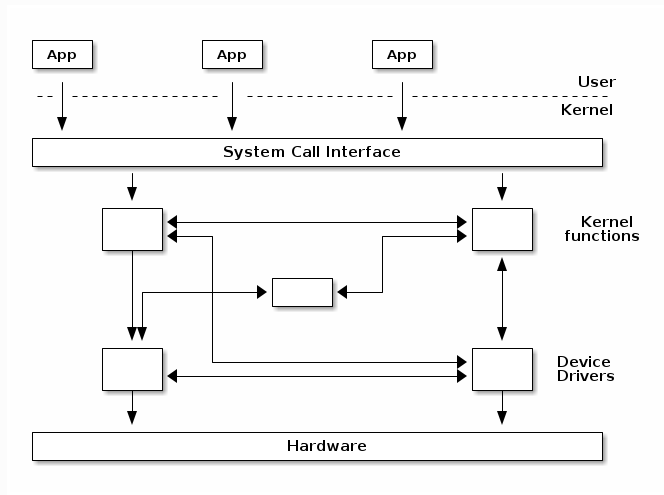


Figure 13: Monolithic Kernel

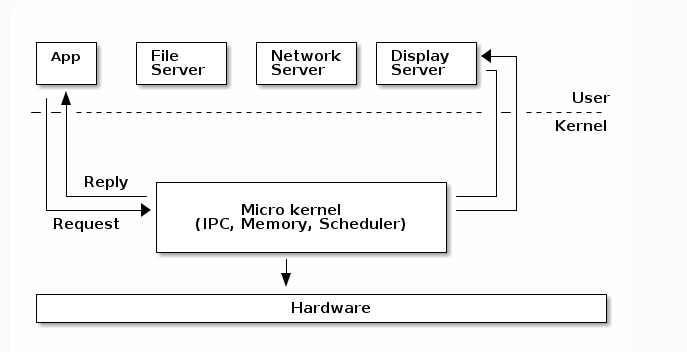


Figure 14: Micro Kernel

Becoming a "Hardware Software Translator" The main function of the kernel is to distribute Hardware resources to software processes, especially memory and processor. The kernel can also restrict memory allocation to devices that fail to prevent crashes.

The Linux kernel is a key component of the Linux operating system (OS) and is a visual interface between computer hardware and its processes. Connects between 2, manages resources as efficiently as possible.

The kernel is so named because — like a seed inside a hard shell — it exists inside the OS and controls all the major hardware functions, whether it be a phone, laptop, server, or any other type of computer.

# KERNEL FUNCTONS

**The kernel has 4 functions:-**

**System calls and security:** Receive service requests from processes

**Device drivers:** Act as mediator / interpreter between hardware and processes

**Process Management:** Find out which processes can use the central processing unit (CPU), when, and for how long

**Memory Management:** Keep a record of what is used in memory to store what, and where

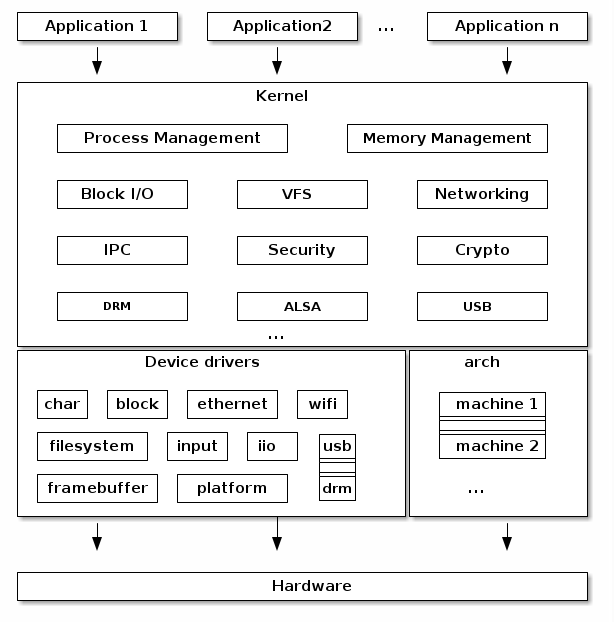


Figure 15: Linux Kernel Architechture

**Features of Linux Kernel:-**

1. Scalable for wide range of architectures and configurations – tiny embedded devices to powerful super computers
2. Standardized and interoperable programming interface (system calls, std compliance)
3. Secure, Stable and Reliable
4. Rich set of generic drivers
5. Rich set of networking drivers & protocol stacks
6. Upstream (mainline) vs Downstream kernel

As Kernel space and user space cannot interact easily so system call and pseudo file systems which includes – device files, procfs, sysfs, debugfs etc. are used to access kernel space.

# SYSTEM CALLS

System calling is a system configuration program that asks for applications from the kernel, and strace is a powerful tool that allows you to follow a small layer between user processes and Linux kernel.

To understand how an operating system works, you first need to understand how a system costs to operate. One of the key functions of the app is to provide extensions to user programs.

# PSEUDO FILESYSTEMS

Pseudo- 'means a lie, a pretense. "Fake file system" therefore means a file system that does not have the files itself - instead, it has visual inputs created by the file itself right there.

For example, / proc on most OS procfs strongly generates clues throughout the process. Similarly, / sys in Linux generates files and references to represent hardware properties. There are many fuse-based system-systemsystems for FUSE.

/ dev can be a real file system (subdirectory for /), or a fake file system (e.g. devfs), or an intermediate point such as Linux devtmpfs (which is a file system full of memory but still creates device nodes out of place. ).

**Note:** Various programs associated with system calls, pseudo filesystems and many more are uploaded on github repositories

# LINUX DRIVER MODEL

The Linux Kernel Driver model is a combination of all the different types of drivers that were previously used in the kernel. It aims to add bus-specific drivers to bridges and devices by integrating data collection and operating in data that are globally accessible.

Driver: between the bus infrastructure and the Linux framework,

In Linux, the driver always integrates with:

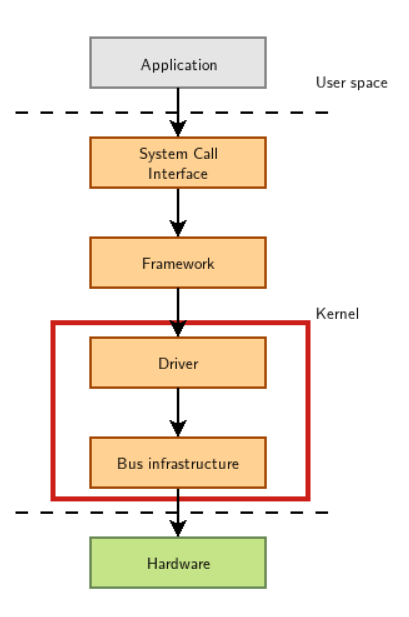
1. A framework that allows the driver to display hardware features in a standard way.
2. Bus infrastructure, device model component, acquisition / communication and hardware.

Figure 16: Kernel Device Driver

# CHAR DEVICE DRIVERS

**Registering Char Device Drivers :**

In UNIX, hardware devices are accessed by the user through special device files. These files are compiled into the / dev identifier, and system calls are open, read, write, close, lseek, mmap etc. In the UNIX world there are two categories of device files and thus device drivers: character and block. This distinction is made by the speed, volume and method of organizing data that will be transferred from the device to the system and vice versa.

Read and write:- This operation is used for reaching and writing access to the device driver as a result of an application space call system for reading or writing system. Write and read function transfers data between device and user location: readable function reads data from device and transfers it to user space, while recording user space data and recording it to device.

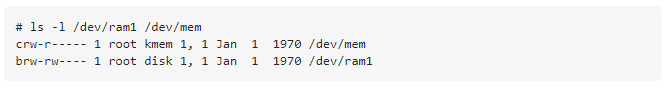
Major and Minor ID in Kernel:-

Each device file has a large ID number and a small ID number. The main ID identifies the standard device category, and is used by the kernel to identify the appropriate driver for this type of device. A small ID separately identifies a specific device within a standard category. The device's large and small file IDs are displayed by the following command:

**ls -l**

Each device driver registers its association with a specific device ID, and this organization provides a connection between the device's unique file and device. Device file name is invalid when the kernel is looking for a device driver.

You cannot have two device locations with the same type and larger ones: smaller ones and expect them to reach two different devices that are logical or portable. When a kernel is found, select one driver based on the type and maximum number (and may be based on the device's node name), and at the meeting the small number selects according to a specific situation or function.



# CONCURRENCY & IPC IN KERNEL

Multi-Threading Model:

The user thread is supported over the kernel and is managed without the support of the kernel but the kernel threads are supported and demolished directly by the application. Almost the entire OS

include kernel threads. Finally there should be a relationship between the user's threads and the kernel

threads. There are three modes for it.

1. Many-to-Many model: It multiplexes user-level thread with a small or equal kernel number

threads. The kernel string number may be specified in a specific application or object

machine.

2. Many-to-one model: It map multiple user-level threads with a single kernel thread. Thread management is done by the library at the user space is therefore efficient; but the whole process will block if the cable makes a blocking call.

3. One-to-one model: It map a string for each user with a kernel thread. It offers more concurrency than

many-to-one model by allowing another thread to run when one thread is making a blocking call.

**Inter process Communication:-**

Process always require communication with each other. In example of the Shell pipe, effect of the first process should be transferred to the second process regularly. So there is a need of communication between processes, preferably in a well-planned manner without using interrupts.

IPC enables one system to control another system, as well as several applications to share the same data without interference.

Inter-process communication (IPC) is a collection of data exchange techniques between multiple threads in one or more processes. Process can be working on one or more computers connected to the network. IPC strategies are divided into categories for synchronization, message transfer, remote control system (RPC) calls and shared memory.

Co-operation process:- The process is independent if it cannot affect or affected by another process. A process is cooperating if it can affect another or be affected by another process. Any process

you share information with another process called collaborative process. There are many reasons to provide

the nature of the cooperative process.

The cooperation process requires IPC. There are two basic methods of IPC.

a. Message Passing

b. Shared memory

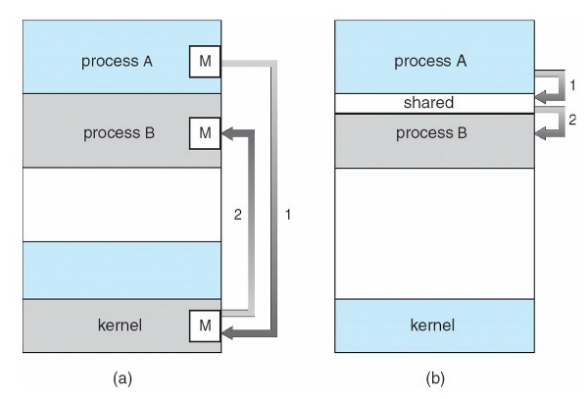


Figure 17: (a) Message Passing (b) Shared Memory

# IOCTL OPERATIONS

Ioctl, which means "input output control" is a type of system call to a specific device. There are only a few system calls on Linux (300-400), which are not enough to display all the functions of unique devices. The driver can then specify ioctl which allows the user app to send orders. However, ioctls are not very flexible and tend to get a little crowded (lots of "magic numbers" that just work ... or not), and can also be unprotected, as the buffer get transferred to the kernel - misbehavior can easily break things. Alternatively, the sysfs interface, is used to set the file under / sys / and read / write to get information from the driver.

**Note:** Codes/programs related to threads, wait queue, IOCTL etc are on Git repositories