./

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 |
| 2 | 12/04/2021 | Suvradeep Dutta / 99003788 |  |  |  |

Table of Contents

[Table of Contents 2](#_Toc68937016)

[1. Embedded Linux- 5](#_Toc68937017)

[1.1 What is Embedded Linux? 5](#_Toc68937018)

[1.2 What is Embedded System? 5](#_Toc68937019)

[Error! Bookmark not defined.](#_Toc68937020)

[1.3 What is Linux? 6](#_Toc68937021)

[1.4 The Linux operating system comprises several different pieces: **Error! Bookmark not defined.**](#_Toc68937022)

[1.4.1 Kernel 7](#_Toc68937023)

[Error! Bookmark not defined.](#_Toc68937024)

[1.4 Functions of Kernel **Error! Bookmark not defined.**](#_Toc68937025)

[1.4.1 Device management **Error! Bookmark not defined.**](#_Toc68937026)

[1.4.2 Memory management **Error! Bookmark not defined.**](#_Toc68937027)

[1.4.3 Process management **Error! Bookmark not defined.**](#_Toc68937028)

[1.5 System Libraries **Error! Bookmark not defined.**](#_Toc68937029)

[1.6 System Tools **Error! Bookmark not defined.**](#_Toc68937030)

[1.7 Development Tools **Error! Bookmark not defined.**](#_Toc68937031)

[1.8 End User Tools **Error! Bookmark not defined.**](#_Toc68937032)

[1.9 Basic Linux Commands 9](#_Toc68937033)

[ ls – 9](#_Toc68937034)

[ cd /var/log – 9](#_Toc68937035)

[ grep – 10](#_Toc68937036)

[ su / sudo command – 10](#_Toc68937037)

[ pwd – Print Working Directory 10](#_Toc68937038)

[ passwd – 10](#_Toc68937039)

[ mv – Move a file 10](#_Toc68937040)

[ cp – Copy a file 11](#_Toc68937041)

[ rm – 11](#_Toc68937042)

[ mkdir – to make a directory. 11](#_Toc68937043)

[2. Qemu Based Emulation 11](#_Toc68937044)

[2.1 Introduction about QEMU 11](#_Toc68937045)

[2.2 QEMU has multiple operating modes :- 12](#_Toc68937046)

[2.2.1 User-mode emulation 12](#_Toc68937047)

[2.2.2 System emulation 12](#_Toc68937048)

[2.3 Features of QEMU: 12](#_Toc68937049)

[3. Installation of QEMU: 13](#_Toc68937050)

[3.1 Install Qemu, a full system emulator for ARM target architecture 13](#_Toc68937051)

[3.2 Rootfs 13](#_Toc68937052)

[3.3 Toolchain 13](#_Toc68937053)

[3.4 Binutil: 13](#_Toc68937054)

[3.5 C library 13](#_Toc68937055)

[3.6 Debugger 14](#_Toc68937056)

[3.7 Installation of Toolchain: 14](#_Toc68937057)

[4. Emulation and Simulation: 14](#_Toc68937058)

[4.1 First Boot (Emulation): 14](#_Toc68937059)

[5. Building Custom Kernel (QEMU): 15](#_Toc68937060)

[5.1 Download Kernel Source: 15](#_Toc68937061)

[5.2 Obtain Configuration File: 15](#_Toc68937062)

[5.3 Customization: 15](#_Toc68937063)

[5.4 How to build the Kernel: 16](#_Toc68937064)

[5.5 How to test the Built outcome: 16](#_Toc68937065)

[6. Cross Compiling Code: 17](#_Toc68937066)

[6.1 Simple Hello Module: 17](#_Toc68937067)

[6.2 Multi file Programming: 17](#_Toc68937068)

[6.3 Static Library: 18](#_Toc68937069)

[6.4 Dynamic Linking: 18](#_Toc68937070)

[7. Working with U-Boot: 19](#_Toc68937071)

[7.1 Cross Building: 19](#_Toc68937072)

[7.2 Simple Boot – Rootfs in SD Card: 19](#_Toc68937073)

[7.3 Prepare Partitioned in SD Card: 20](#_Toc68937074)

[7.4 Rootfs in Partitioned SD Card: 20](#_Toc68937075)

[7.5 Setup TFTP on Host: 21](#_Toc68937076)

[7. 6 Booting kernel using Networking: 21](#_Toc68937077)

[7.7 Setup TFTP on host: 22](#_Toc68937078)

[8. Device Tree: 23](#_Toc68937079)

[8.1 Introduction of Device Tree 23](#_Toc68937080)

[8.2 High Level View about Device Tree 23](#_Toc68937081)

[8.2.1 Platform Identification: 23](#_Toc68937082)

[8.2.2 Runtime configuration 24](#_Toc68937083)

[8.2.3 Device population 24](#_Toc68937084)

[8.3 Booting With Device Tree 24](#_Toc68937085)

[8.4 Compatibility mode for DT booting 24](#_Toc68937086)

[9. WHAT DO YOU MEAN BY KERNEL? 24](#_Toc68937087)

[9.1 There are three types of kernels: 25](#_Toc68937088)

[9.1.1 A monolithic kernel 25](#_Toc68937089)

[9.1.2 A micro kernel 25](#_Toc68937090)

[9.1.3 Hybrid Kernel 25](#_Toc68937091)

[10. WHAT DO YOU MEAN BY MODULES? 29](#_Toc68937092)

[11 Activity QEMU installation 29](#_Toc68937093)

[12 Activity TOOLCHAIN Installation 29](#_Toc68937094)

[13 .Building Kernel Modules: 29](#_Toc68937095)

[13.1 Simple Hello Module: 29](#_Toc68937096)

[13.2 Simple hello Module with init and exit function 30](#_Toc68937097)

[13.3 Hello module with parameters 32](#_Toc68937098)

[13.4 Module Dependency simple 32](#_Toc68937099)

[13.5 Module Dependency sample 33](#_Toc68937100)

[13.6 ADDING KCONFIG ENTRIES 33](#_Toc68937101)

[13.7 Version 2 for K Config entries: 33](#_Toc68937102)

[14 WHAT DO YOU MEAN BY SYSTEM CALLS? 34](#_Toc68937103)

[14.1 Services Provided by System Calls : 35](#_Toc68937104)

[14.2 Adding a system call: 35](#_Toc68937105)

[15 Pseudo Char Driver: 36](#_Toc68937106)

[15.1 Statistically registration device driver 37](#_Toc68937107)

[int register\_chrdev\_region(dev\_t first, unsigned int count, char \*name) 37](#_Toc68937108)

[15.2 Dynamically registration of Character Device Driver 37](#_Toc68937109)

[15.3 Un-registration of character device driver 37](#_Toc68937110)

[15.4 Device file Creation: 38](#_Toc68937111)

[15.4.1 Manually, Automatically 38](#_Toc68937112)

[15.5 Create the class: 39](#_Toc68937113)

[15.5.1 Create Device: 39](#_Toc68937114)

[15.5.2 Device Destroy: 39](#_Toc68937115)

[16. Kernel Data Structures: 39](#_Toc68937116)

[16.1 Kfifo API: 39](#_Toc68937117)

[17 IPC in Kernel: 41](#_Toc68937118)

[17.1 Semaphors: 41](#_Toc68937119)

[17.2 Semaphore API 42](#_Toc68937120)

[17.3 Mutex: 42](#_Toc68937121)

[17.4 Spin Locks: 43](#_Toc68937122)

[17.5 Reader-writer spinlocks: 44](#_Toc68937123)

[17.6 Wait Queue API: 45](#_Toc68937124)

[17.7 Generate Race Conditions in Pseudo Driver: 46](#_Toc68937125)

[18 IOCTL usage: 47](#_Toc68937126)

[19. References: 48](#_Toc68937127)

# Embedded Linux

## **What is Embedded Linux?**

Embedded Linux is an operating system based on the Linux kernel that is designed to be installed and used within embedded devices and appliances. It is a compact version of Linux that offers features and services in line with the operating and application requirement of the embedded system.

## **1.2 What is Embedded System?**

"A computer system with a dedicated function within a larger mechanical or electrical system, often with real-time computing constraints." I find it simple enough to say that an embedded system is a computer that most people don't think of as a computer. Its primary role is to serve as an appliance of some sort, and it is not considered a general-purpose computing platform.

# 

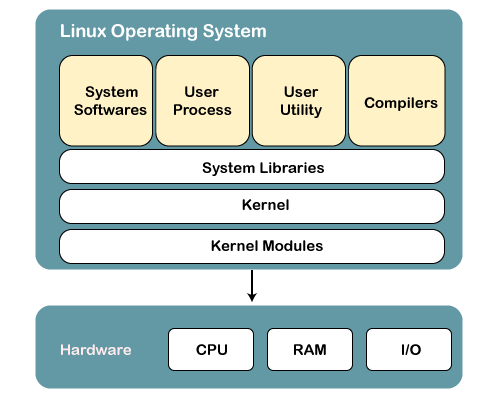
Figure. Architecture Diagram

### 

### 1.3 Evolution of Linux OS

The [Linux OS](https://www.javatpoint.com/linux-tutorial) was developed by Linus Torvalds in 1991, which sprouted as an idea to improve the UNIX OS. He suggested improvements but was rejected by UNIX designers. Therefore, he thought of launching an OS, designed in a way that could be modified by its users.

Linux is undoubtedly the fastest-growing operating system today. It is used in areas such as embedded devices all the way to mainframes. One of the interesting and most important facts about Linux is that it is open-sourced. Linux OS is known for Multi-user, Multi-tasking and Multi programming capabilities. It means that multiple users can access the various system resources at the same time. It also permits the system to run multiple applications at a time as well.



**Figure: Linux Embedded**

## **1.4 What is Linux?**

Just like Windows, iOS, and Mac OS, Linux is an operating system. In fact, one of the most popular platforms on the planet, Android, is powered by the Linux operating system. An operating system is software that manages all of the hardware resources associated with your desktop or laptop. To put it simply, the operating system manages the communication between your software and your hardware. Without the operating system (OS), the software would not function.

### 1.5 Kernel

Linux kernel is the main component of a Linux operating system and is the core interface between a computer’s hardware and its processes. It establishes communication between devices and software. Moreover, it manages system resources.

# 

Figure: Kernel

**Functions of Kernel**

### Process management

Process management is focused on the execution of processes. In the kernel, these are called threads and represent an individual virtualization of the processor (thread code, data, stack, and CPU registers). In user space, the term process is typically used, though the Linux implementation does not separate the two concepts (processes and threads). The kernel provides an application program interface (API) through the SCI to create a new process (fork, exec, or Portable Operating System Interface [POSIX] functions), stop a process (kill, exit), and communicate and synchronize between them (signal, or POSIX mechanisms).

### Memory management

Another important resource that’s managed by the kernel is memory. For efficiency, given the way that the hardware manages virtual memory, memory is managed in what are called pages (4KB in size for most architectures). Linux includes the means to manage the available memory, as well as the hardware mechanisms for physical and virtual mappings. But memory management is much more than managing 4KB buffers. Linux provides abstractions over 4KB buffers, such as the slab allocator. This memory management scheme uses 4KB buffers as its base, but then allocates structures from within, keeping track of which pages are full, partially used, and empty. This allows the scheme to dynamically grow and shrink based on the needs of the greater system.

### Device drivers

The vast majority of the source code in the Linux kernel exists in device drivers that make a particular hardware device usable. The Linux source tree provides a drivers subdirectory that is further divided by the various devices that are supported, such as Bluetooth, I2C, serial, and so on. You can find the device driver sources in ./linux/drivers.

### System call interface

The SCI is a thin layer that provides the means to perform function calls from user space into the kernel. As discussed previously, this interface can be architecture dependent, even within the same processor family. The SCI is actually an interesting function-call multiplexing and demultiplexing service. You can find the SCI implementation in ./linux/kernel, as well as architecture-dependent portions in ./linux/arch.

### 1.6 System Libraries

* System libraries are special programs that help in accessing the kernel's features.
* A kernel has to be triggered to perform a task, and this triggering is done by the applications. But applications must know how to place a system call because each kernel has a different set of system calls. Programmers have developed a standard library of procedures to communicate with the kernel. Each operating system supports these standards, and then these are transferred to system calls for that operating system.
* The most well-known system library for Linux is Glibc (GNU C library).

### 1.7 System Tools

* Linux OS has a set of utility tools, which are usually simple commands. It is a software which GNU project has written and publish under their open source license so that software is freely available to everyone.
* With the help of commands, you can access your files, edit and manipulate data in your directories or files, change the location of files, or anything.

## **1.8 Development Tools**

With the above three components, your OS is running and working. But to update your system, you have additional tools and libraries. These additional tools and libraries are written by the programmers and are called toolchain. A toolchain is a vital development tool used by the developers to produce a working application.

## **1.9 End User Tools**

These end tools make a system unique for a user. End tools are not required for the operating system but are necessary for a user. Some examples of end tools are graphic design tools, office suites, browsers, multimedia players, etc.

* 1. **Basic Linux Commands**

### **ls –**

In Linux, the ls command is used to list out files and directories. Some versions may support color-coding. The names in blue represent the names of directories.

$ ls -l filename

### **cd /var/log –**

Change the current directory. The forward slash is to be used in Linux. The example is a Linux directory that comes with all versions of Linux.

When use ls -l will be able to see more details of the contents in the directory

It will list down the

$ cd /var/log

### **grep –**

The grep command searches through many files at a time to find a piece of text you are looking for.

Grep PATTERN[FILE]

grep failed transaction.log

The above command will find all of the words in the files that matched the word ‘failed’.

$ grep ‘failed’ transaction.log

### **su / sudo command –**

SU command changes the shell to be used as a super user and until you use the exit command you can continue to be the super user

Sudo- If you just need to run something as a super user, you can use the sudo command. This will allow you to run the command in elevated rights and once the command is executed you will be back to your normal rights and permissions.

$ sudo shutdown 2

$ sudo shutdown –r 2

### **pwd – Print Working Directory**

It displays the current working directory path and is useful when directory changes are often

$ pwd

### **passwd –**

his command is used to change the user account password. You could change your password or the password of other users. Note that the normal system users may only change their own password, while root may modify the password for any account.

$ passwd admin

### **mv – Move a file**

To move a file or rename a file use the mv command

$ mv first.txt second.txt

### **cp – Copy a file**

To copy a file in the same directory

$ cp second.txt third.txt

### **rm –**

This command is used to remove files in a directory or the directory itself. A directory cannot be removed if it is not empty.

$ rm file1

$ rm -r myproject

### **mkdir – to make a directory.**

To create a directory in the name ‘myproject’ type

$ mkdir myproject

* Chmod-

To change mode of a file system object. Files can have r – read, w- write and x-execute permissions.

$ chmod 744 script.sh

# Qemu Based Emulation

## **2.1 Introduction about QEMU**

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 quest 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.

Qemu is quick it's a hypervisor that allows you to run virtual machines with complete operating systems that operate like any other program on your desktop. This can be useful for general purpose computing and black box testing. The software is open-source and cross-platform. It targets a range of computer architectures beyond standard IBM PCs such as ARM and PowerPC. On Linux, it also has user-mode emulation where standard executables of one architecture can seamlessly run on another.

Qemu is a virtualization platform application that works as an emulator that emulates an existing system to run more than one operating system together at the same time. It has its own configuration and it is easy to install and use. It is used to emulate many subsystems like hardware and software. It emulates the processors as well like multiprocessing systems.

## **2.2 QEMU has multiple operating modes**

### 2.2.1 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. This allow the process execution from one CPU to another CPU. It performs dynamic translation of instructions for host CPU.

### 2.2.2 System emulation

In this mode QEMU emulates a full computer system, including [peripherals](https://en.wikipedia.org/wiki/Peripheral). It can be used to provide virtual hosting of several virtual computers on a single computer. QEMU can boot many guest [operating systems](https://en.wikipedia.org/wiki/Operating_system), including [Linux](https://en.wikipedia.org/wiki/Linux), [Solaris](https://en.wikipedia.org/wiki/Solaris_(operating_system)), [Microsoft Windows](https://en.wikipedia.org/wiki/Microsoft_Windows), [DOS](https://en.wikipedia.org/wiki/DOS), and [BSD](https://en.wikipedia.org/wiki/BSD);[[6]](https://en.wikipedia.org/wiki/QEMU#cite_note-yfwuu-6) it supports emulating several instruction sets,including [x86](https://en.wikipedia.org/wiki/X86), [MIPS](https://en.wikipedia.org/wiki/MIPS_architecture),32 bit [ARMv7](https://en.wikipedia.org/wiki/ARMv7), [ARMv8](https://en.wikipedia.org/wiki/ARMv8), [PowerPC](https://en.wikipedia.org/wiki/PowerPC), [SPARC](https://en.wikipedia.org/wiki/SPARC), [ETRAXCRIS](https://en.wikipedia.org/wiki/ETRAX_CRIS) and [MicroBlaze](https://en.wikipedia.org/wiki/MicroBlaze).

This allow the emulation of the complete system that includes hardware as well as peripheral related components.

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.

Xen Hosting

* QEMU is involved only in the emulation of hardware; the execution of the guest is done within [Xen](https://en.wikipedia.org/wiki/Xen) and is totally hidden from QEMU.

## 

## **2.3 Features of QEMU**

* 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 supports the emulation of various architectures, including:
  + RISC -V
  + MicroBlaze
  + SH4 SHIX board
  + ARM development boards (Integrator/CP and Versatile/PB)
  + IA-32 (x86) PCs
  + x86-64 PCs
  + Sun's SPARC sun4m
  + Sun's SPARC sun4u
* Virtual disk images can be stored in a special format (qcow, qcow2) that only takes up as much disk space as the guest OS actually uses.
* 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.

# 3. Installation of QEMU

## **3.1 Setup Qemu**

## Install Qemu, a full system emulator for ARM target architecture

* sudo apt install qemu-system-arm
* qemu-system-arm –v
* qemu-system-arm –M ?
* qemu-system-aarch64 -v

## **Rootfs**

* Download core-image-minimal-qemuarm.ext4 from http://downloads.yoctoproject.org/releases/yocto/yocto-2.5/machines/qemu/qemuarm/
* Rename core-image-minimal-qemuarm.ext4 as rootfs.img
* Align the size of rootfs

e2fsck -f rootfs.img resize2fs

rootfs.img 16M

## **3.3 Toolchain**

**Install linaro toolchain from ubuntu package manager**

* sudo apt install gcc-arm-linux-gnueabi # soft float
* sudo apt install gcc-arm-linux-gnueabihf # hard float

## **3.4 Binutil**

The GNU Binutils is the first component of a toolchain. The GNU Binutils contains two very important tools:

* + - as, the assembler, that turns assembly code (generated by GCC) to binary.
    - ld, the linker, that links several object code into a library, or an executable.

## **3.5 C library**

The C library implements the traditional POSIX API that can be used to develop userspace applications. It interfaces with the kernel through system calls and provides higher-level services.

## **3.6 Debugger**

The debugger is also usually part of the toolchain, as a cross-debugger is needed to debug applications running on your target machine. In the embedded Linux world, the typical debugger is GDB.

## **3.7 Installation of Toolchain:**

* We can install linaro toolchain form Ubuntu packages manager using following steps:
* 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.

# 4. Emulation and Simulation:

* An emulator is hardware or software that enables one computer system (called the host) to behave like another computer system(called the guest).
* A simulator is a software that helps your computer run certain programs built for a different Operating System.
* Emulation advantages are include better graphic quality, save space, emulation in video games, add post-processing effects, etc.
* Simulation advantages include increase safety and efficiency, avoid danger and loss of life, slowed down to study behavior more closely, etc.

## **4.1 First Boot (Emulation):**

* First collect prebuild zImage, vexpress-v2p-ca9.dtb file.
* We have to ensure that the rootfs.img is also in the same location.
* Emulate using Qemu – sdcard approach:
* 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:
* 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"4.2

**4.2 First Steps on Target:**

1. Uname -r
2. uname -v
3. uname -a
4. cat /proc /cpuinfo
5. free -m
6. df -kh
7. mount
8. dmesg

# 5. Building Custom Kernel (QEMU):

A kernel is an important program of every device out there. Android is a famous operating system that features a lot of custom kernel out there for almost every phone nowadays. Custom Kernels not only offer security updates, but also various improvements over the Stock Kernel.

## **5.1 Download Kernel Source:**

* We have to download any recent LTS version of kernel source.
* It can be downloaded from the steps also:
  + git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
  + cd linux
  + git checkout tags/v4.14 –b v4.14

## **5.2 Obtain Configuration File:**

* Locate default config available in KSRC/arch/arm/configs, we’ll refer vexpress\_defconfig for Versatil Express target being used for Qemu emulation.
* Collect any well tested configuration file as base config
* make ARCH=arm mrproper
* make ARCH=arm vexpress\_defonfig
* Note that mrproper will remove built files, including the configuration.
* So run this only for any new build.

## **5.3 Customization:**

* Run the menuconfig for further customization.
* make ARCH=arm menuconfig

## **5.4 How to build the Kernel:**

* Run menuconfig for further customization.
* Build kernel image
* make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi- zImage
* Build Device Tree Binaries
* make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi- dtbs firmware
* Build dynamic modules
* make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi- modules
* make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi- modules\_install \ INSTALL\_MOD\_PATH=<tempdir> # or mount point of target rootfs

## 

## **5.5 How to test the Built outcome:**

* Collect built outcome to a temporary location
* cp $KSRC/arch/arm/boot/zImage
* cp $KSRC/arch/arm/boot/dts/\*.dtb
* Ensure rootfs.img is also in same location.
* Emulate using Qemu
* 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"
  + # In target
* uname –r
* uname –v
* ls /boot
* ls /lib/modules
  + # In host
* ls –lh $KSRC/arch/arm/boot/zImage
* ls –lh $KSRC/vmlinux

# 6. Cross Compiling Code

Cross-compiling means compiling a software binary that is targeted at, or intended to run on, a CPU architecture that is different to the one on which it is compiled. This is commonly done because compiling is a CPU-intensive task and the RPI has a modest CPU. Compiling the u-boot.bin binary on your local computer would be a great deal faster than doing it directly on the RPI, and with cross-compilation, you can create native RPI binaries from the comfort of the regular development environment on your host PC.

* 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.

## **6.1 Simple Hello Module:**

* Write a simple hello world code and save it.
* Generate its output file with the command –
* arm-linux-gnueabi-gcc hello.c –o h1.out
* arm-linux-gnueabi-gcc hello.c –o h2.out –o static
* file h1.out h2.out
* ls –lh h1.out h2.out
* ldd h1.out
* ldd h2.out
* Copy the output file to the target rootfs using the command mount, copy and umount-
* sudo mount -o loop,rw,sync rootfs.img /mnt/rootfs
* sudo cp h1.out h2.out /mnt/rootfs/home/root
* sudo umount /mnt/rootfs

## **6.2 Multi file Programming:**

* Create one .c test file.
* In that file simple mathematical functions will be there and one main code will be there.
* Create one Makefile for the same test file.
* It will create all the necessary file which will be further used.
* After the files are created copy all the output files to target rootfs and test.
* We can create output file by the following commands-
  + arm-linux-gnueabi-gcc test.c –c
  + arm-linux-gnueabi-gcc sum.c -c
  + arm-linux-gnueabi-gcc sqr.c –c
  + arm-linux-gnueabi-gcc test.o sum.o sqr.o -o all.out

## **6.3 Static Library:**

When we click the .exe (executable) file of the program and it starts running, all the necessary contents of the binary file have been loaded into the process’s virtual address space. However, most programs also need to run functions from the system libraries, and these library functions also need to be loaded.

In the simplest case, the necessary library functions are embedded directly in the program’s executable binary file. Such a program is statically linked to its libraries, and statically linked executable codes can commence running as soon as they are loaded.

* A static library or statically-linked library is a set of routines, external functions and variables
* In static library follow the same above steps and prepare the source code and generate the output files.
* Create one Makefile for this.
* We can do the steps using the following commands:
* arm-linux-gnueabi-ar sum.o sqr.o –o libsample.a
* arm-linux-gnueabi-gcc –L. test.o –lsample –o s1.out
* arm-linux-gnueabi-gcc –L. test.o –lsample –o s2.out -static

## **6.4 Dynamic Linking:**

Every dynamically linked program contains a small, statically linked function that is called when the program starts. This static function only maps the link library into memory and runs the code that the function contains. The link library determines what are all the dynamic libraries which the program requires along with the names of the variables and functions needed from those libraries by reading the information contained in sections of the library.

* In the dynamic linking we will follow same steps as static library.
* Copy the libsample.so, d1.out to target rootfs and execute using the following commands.

# On Host

* + - arm-linux-gnueabi-gcc –shared libsample.so sum.o sqr.o
    - arm-linux-gnueabi-gcc –L. test.o –lsample –o d1.out

#On Target

* + - LD\_LIBRARY\_PATH=. ./d1.out

# 7. Working with U-Boot:

* Das U-Boot (Universal Boot Loader and shortened to U-Boot)
* U-Boot is both a first-stage and second-stage bootloader.
* It is loaded by the system's ROM or BIOS from a supported boot device, such as an SD card, SATA drive, NOR flash (e.g. using SPI or I2C), or NAND flash.
* If there are size constraints, U-Boot may be split into stages: the platform would load a small SPL (Secondary Program Loader), which is a stripped-down version of U-Boot, and the SPL would do initial hardware configuration and load the larger, fully featured version of U-Boot.

* U-Boot boots an operating system by reading the kernel and any other required data (e.g. device tree or ramdisk image) into memory, and then executing the kernel with the appropriate arguments.

## **7.1 Cross Building:**

make ARCH=arm vexpress\_ca9x4\_defconfig

make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi-

# Locate generated u-boot and copy to a tempdir

## **7.2 Simple Boot – Rootfs in SD Card:**

* It will create an image of 64Mb.
* qemu-img create simplesd.img 64M
* sudo mkfs.vfat simplesd.img
* sudo mount -o loop,rw,sync simplesd.img /mnt/sdcard
* After that copy the zImage, vexpress-v2p-ca9.dtb, rootfs.img to /mnt/sdcard umount /mnt/sdcard
* #copy simplesd.img to tempdir, where generated u-boot is copied.
* qemu-system-arm -M vexpress-a9 -m 1024 -serial stdio -kernel u-boot -sd sdcard.img
* # Stop autoboot by hitting any key, Run the following commands in U-Boot shell
* mmcinfo fatls mmc 0:0
* fatload mmc 0:0 0x60200000 zImage
* fatload mmc 0:0 0x60100000 vexpress-v2p-ca9.dtb
* fatload mmc 0:0 0x62000000 rootfs.img
* setenv bootargs 'console=ttyAMA0 root=/dev/ram0 rw rootfstype=ext4
* initrd=0x62000000, 16777216’
* bootz 0x60200000 - 0x60100000
* # 16777216 is size of loaded rootfs image
* # for this method ramdisk support should be enabled at kernel level (menuconfig --> Device Drivers

Block Devices --> RAM Block Device Support)

## **7.3 Prepare Partitioned in SD Card:**

* In this, we will follow the below codes in order to partition the SD card successfully.
* dd if=/dev/zero of=sdcard.img bs=1M count=128
  + # create two primary partitions in sdcard.img using cfdisk
  + # Keep first partition size as small as possible, say 16M
* sudo fdisk -l sdcard.img # 1048576 is 2048x512, 2048 is start of first
  + partition # 17825792 is 34816x512, 34816 is start of second partition
  + sudo losetup -o 1048576 /dev/loop20 sdcard.img
  + sudo losetup -o 17825792 /dev/loop21 sdcard.img
  + sudo mkfs.vfat /dev/loop20 sudo mkfs.ext4 /dev/loop21
  + sudo mount -o loop,rw,sync /dev/loop20 /mnt/boot
  + sudo mount -o loop,rw,sync /dev/loop21 /mnt/rootfs
  + #copy zImage, vexpress-v2p-ca9.dtb to /mnt/boot
  + # extract core-image-minimal-qemuarm.tar.bz2 to /mnt/rootfs
* tar -jxvf core-image-minimal-qemuarm.tar.bz2 -C /mnt/rootfs
* sudo umount /mnt/boot
* sudo umount /mnt/rootfs
* sudo losetup -d /dev/loop20
* sudo losetup -d /dev/loop21

## **7.4 Rootfs in Partitioned SD Card:**

* qemu-system-arm -M vexpress-a9 -m 1024 -serial stdio -kernel u-boot -sd sdcard.img
* #Stop autoboot by hitting any key, Run the following commands in 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

## **7.5 Setup TFTP on Host:**

* TFTP stands for Trivial File Transfer Protocol.
* TFTP is used to transfer a file either from client to server or from server to client without the need of FTP feature.
* Software of TFTP is smaller than FTP.
* TFTP works on 69 Port number and its service is provided by UDP.
* TFTP does not need authentication for communication.
* TFTP is mainly used for transmission of configurations to and from network devices.
* We can install tftp by following the commands:
* sudo apt install tftpd
  + - * # create /etc/xinetd.d/tftp
      * # with specified content
      * # replace server\_args as per your machine
      * /etc/init.d/xinetd restart

service tftp

{

protocol = udp

port = 69

socket\_type = dgram

wait = yes

user = nobody

server = /usr /sbin /in.tftpd

server\_args = /\* \*/

disable = no

}

**For Check tftp Commands** ->

* sudo modprobe tun
* sudo ifconfig tap0 192.168.0.1

## **7. 6 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.

## **7.7 Setup TFTP on host:**

* First we need to install the tftpd, in order to install execute the following command,
* sudo apt install tftpd
* Now create the tftp file with the required code in the “/etc/xinetd.d/tftp” location
* Restart to update the changes,
* /etc/init.d/xinetd restart
* TUN/TAP provides packet reception and transmission for user space programs
  + To run the tun command,
* sudo modprobe tun
* Now in order to setup the TFTP on the target machine, we need to follow series of commands
  + 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

* To set the ipaddress, we take an environment variable ipaddr, the command is
  + setenv ipaddr 192.168.0.2
* To set the server ip,
  + - setenv serverip 192.168.0.1
* To check the status of the network connectivity between host and target, we can use the ping command.
* ping 192.168.0.1
* We load the zImage into board via Network by using the tftp protocol.
  + - tftp 0x60200000 zImage
* In similar fashion, we load the vexpress-v2p-ca9.dtb.
  + - tftp 0x60100000 vexpress-v2p-ca9.dtb
* Now to boot into the target, use the following command,
  + - setenv bootargs ‘console=ttyAMA0 root=/dev/mmcblk0p2 rootfstype=ext4 ‘
* The range of bootloader in the target board is from 0x60200000 to 0x60100000. Command is
  + - bootz 0x60200000 – 0x60100000

# 8. Device Tree:

## **8.1 Introduction of Device Tree**

* The “Open Firmware Device Tree”, or simply Device Tree (DT), is a data structure and language for describing hardware. More specifically, it is a description of hardware that is readable by an operating system so that the operating system doesn’t need to hard code details of the machine.
* Structurally, the DT is a tree, or acyclic graph with named nodes, and nodes may have an arbitrary number of named properties encapsulating arbitrary data. A mechanism also exists to create arbitrary links from one node to another outside of the natural tree structure.
* Conceptually, a common set of usage conventions, called ‘bindings’, is defined for how data should appear in the tree to describe typical hardware characteristics including data busses, interrupt lines, GPIO connections, and peripheral devices.

## **8.2 High Level View about Device Tree**

* The most important thing to understand is that the DT is simply a data structure that describes the hardware. There is nothing magical about it, and it doesn’t magically make all hardware configuration problems go away. What it does do is provide a language for decoupling the hardware configuration from the board and device driver support in the Linux kernel (or any other operating system for that matter). Using it allows board and device support to become data driven; to make setup decisions based on data passed into the kernel instead of on per-machine hard coded selections.
* The DT is simply a data structure that describes the hardware.
* Linux uses DT data for three major purposes:

1. Platform identification
2. Runtime configuration
3. Sevice population

### 8.2.1 Platform Identification:

First and foremost, the kernel will use data in the DT to identify the specific machine. In a perfect world, the specific platform shouldn’t matter to the kernel because all platform details would be described perfectly by the device tree in a consistent and reliable manner. Hardware is not perfect though, and so the kernel must identify the machine during early boot so that it can run machine-specific fixups.

The kernel will use data in the DT to identify the specific machine.

Hardware is not perfect though, and so the kernel must identify the machine during early boot so that it has the opportunity to run machine-specific fixups.

### 8.2.2 Runtime configuration

DT will be the sole method of communicating data from firmware to the kernel, so also gets used to pass in runtime and configuration data like the kernel parameters string and the location of an initrd image.

chosen {

bootargs = "console=ttyS0,115200 loglevel=8";

initrd-start = <0xc8000000>;

initrd-end = <0xc8200000>;

};

### 8.2.3 Device population

After the board has been identified, and after the early configuration data has been parsed, then kernel initialization can proceed in the normal way. At some point in this process, unflatten\_device\_tree() is called to convert the data into a more efficient runtime representation.

## 8.3 Booting With Device Tree

* The kernel no longer contains the description of the hardware, it is located in a separate binary: the device tree blob
* The bootloader loads two binaries: the kernel image and the DTB
  + Kernel image remains uImage or zImage
  + DTB located in arch/arm/boot/dts, one per board
* U-Boot command:

bootm <kernel img addr> - <dtb addr>

## **8.3 Compatibility mode for DT booting**

Some bootloaders have no specific support for the Device Tree, or the version used on a particular device is too old to have this support. To ease the transition, a compatibility mechanism was added:

CONFIG\_ARM\_APPENDED\_DTB.

I It tells the kernel to look for a DTB right after the kernel image. I There is no built-in Makefile rule to produce such kernel, so one must manually do:

# 9. WHAT DO YOU MEAN BY KERNEL?

* A kernel is the central part of an operating system. It manages the operations of the computer and the hardware, most notably memory and CPU time
* It decides which process should be allocated to processor to execute and which process should be kept in main memory to execute.

## **9.1 There are three types of kernels:**

### 9.1.1 A monolithic kernel

* It is one of types of kernel where all operating system services operate in kernel space. It has dependencies between systems components. It has huge lines of code which is complex
* Advantage

It has good performance.

* Disadvantage

It has dependencies between system component and lines of code in millions.

### 9.1.2 A micro kernel

* It is kernel types which has minimalist approach. It has virtual memory and thread scheduling. It is more stable with less services in kernel space. It puts rest in user space.
* Advantage

It is more stable.

* Disadvantage

There are lots of system calls and context switches.

### 9.1.3 Hybrid Kernel

* It is the combination of both monolithic kernel and microkernel. It has speed and design of monolithic kernel and modularity and stability of microkernel.
* Advantage

It combines both monolithic kernel and microkernel.

* Disadvantage

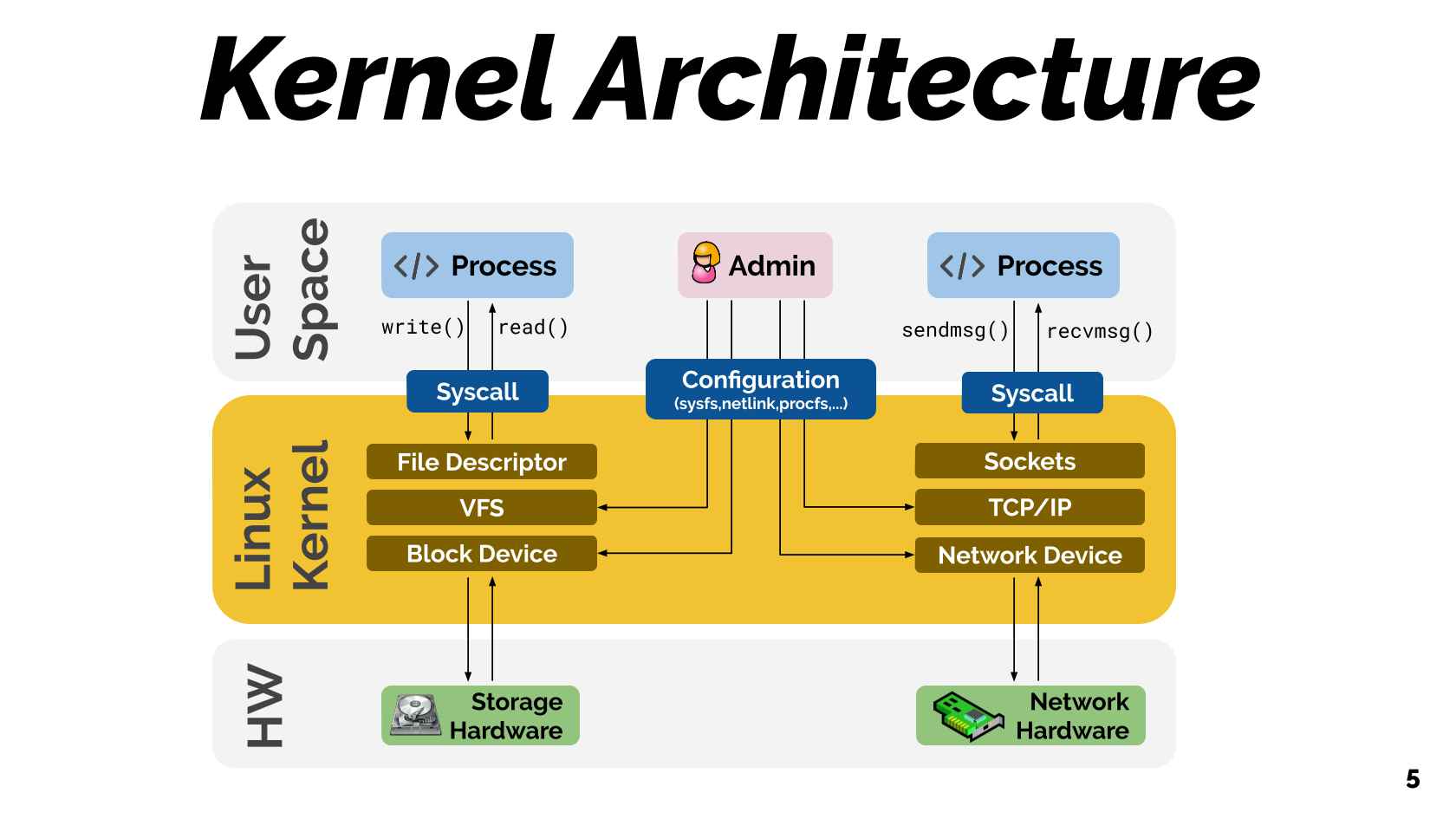
It is still similar to monolithic kernel.

## **9.2 KERNEL VERSIONS**

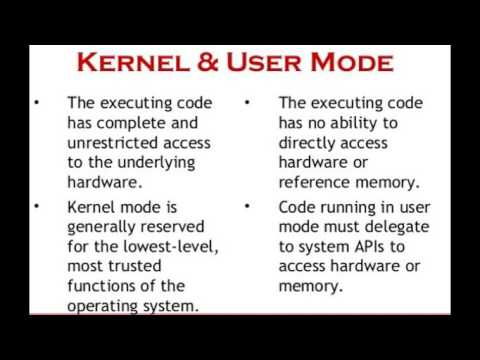
The [Linux kernel](https://en.wikipedia.org/wiki/Linux_kernel) is a [free and open-source](https://en.wikipedia.org/wiki/Free_and_open-source), [monolithic](https://en.wikipedia.org/wiki/Monolithic_kernel), [Unix-like](https://en.wikipedia.org/wiki/Unix-like) [operating system](https://en.wikipedia.org/wiki/Operating_system) [kernel](https://en.wikipedia.org/wiki/Kernel_(operating_system)). It was conceived and created in 1991 by [Linus Torvalds](https://en.wikipedia.org/wiki/Linus_Torvalds). Linux kernels have different support level depending on version, (e.g. version 4.4, released in January 2016, was declared to have [Long-Term Support (LTS)](https://en.wikipedia.org/wiki/Long-term_support)). It has six years of support that way, but it was also defined to have Super Long Term Support (SLTS), i.e. Civil Infrastructure Platform will provide support (for 32-bit ARM and [x86-64](https://en.wikipedia.org/wiki/X86-64) only) until at least 2026, possibly until 2036. It is by now the oldest supported version.

* a.b.c-d format
  + - a.b represents major version
    - c represents release version
    - d represents local version
* Older convention of kernel versions (major, minor, release versions)
* Patch set, optional fourth digit
* Key changes between 2.4.x – 2.6.x
* LTS versions – Long Term Support
* 5.11.x is latest stable version (As on March 2021)

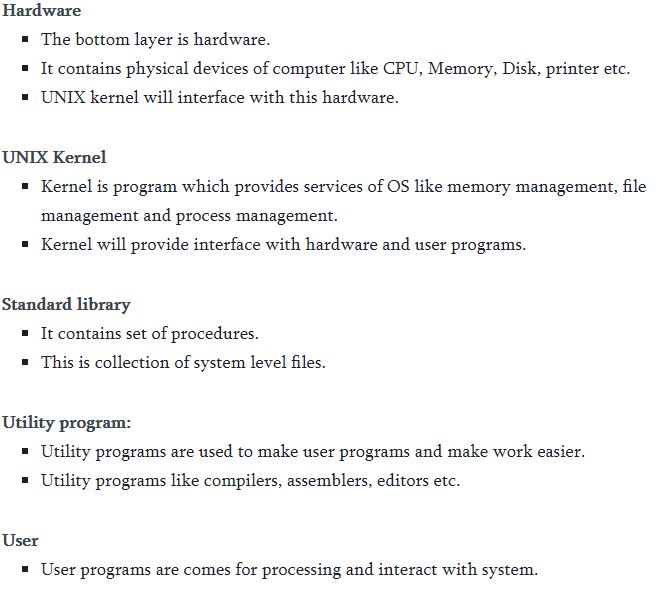
## **ARCHITECTURE OF LINUX KERNEL**



* Mandate component of Operating System
* Resides in memory all the time, rest all depending on kernel
* Provides basic services including memory management, IO management & other management services
* Provides services to application and libraries in the form of SYSTEM CALLS



**Components of OS**



## **OVERVIEW OF KERNEL SOURCE**

* **Makefile** - This file is the top-level Makefile for the whole source tree. It defines a lot of useful variables and rules, such as the default gcc compilation flags.
* **documentation** - This directory contains a lot of useful (but often out of date) information about configuring the kernel, running with a ramdisk, and similar things. The help entries corresponding to different configuration options are not found here, though - they're found in Kconfig files in each source directory.
* **arch** - All the architecture specific code is in this directory and in the include/asm-<arch> directories. Each architecture has its own directory underneath this directory. For example, the code for a PowerPC based computer would be found under arch/ppc. You will find low-level memory management, interrupt handling, early initialization, assembly routines, and much more in these directories.
* **crypto** - This is a cryptographic API for use by the kernel itself.
* **drivers** - As a general rule, code to run peripheral devices is found in subdirectories of this directory. This includes video drivers, network card drivers, low-level SCSI drivers, and other similar things.
* **fs -** Both the generic filesystem code (known as the VFS, or Virtual File System) and the code for each different filesystem are found in this directory. Your root filesystem is probably an ext2 filesystem; the code to read the ext2 format is found in fs/ext2. Not all of the filesystems compile or run, and the more obscure filesystems are always a good candidate for someone looking for a kernel project.
* **include** - Most of the header files included at the beginning of a .c file are found in this directory. Architecture specific include files are in asm-<arch>. Part of the kernel build process creates the symbolic link from asm to asm-<arch>, so that #include <asm/file.h> will get the proper file for that architecture without having to hard code it into the .c file. The other directories contain non-architecture specific header files. If a structure, constant, or variable is used in more than one .c file, it should be probably be in one of these header files.
* **init** - This directory contains the files main.c, version.c, and code for creating "early userspace". version.c defines the Linux version string. main.c can be thought of as the kernel "glue". We'll talk more about main.c in the next section. Early userspace provides functionality that needs to be available while a Linux kernel is coming up, but that doesn't need to be run inside the kernel itself.
* **ipc** - "IPC" stands for "Inter-Process Communication". It contains the code for shared memory, semaphores, and other forms of IPC.
* **kernel** - Generic kernel level code that doesn't fit anywhere else goes in here. The upper level system call code is here, along with the printk() code, the scheduler, signal handling code, and much more. The files have informative names, so you can type ls kernel/ and guess fairly accurately at what each file does.
* **lib** - Routines of generic usefulness to all kernel code are put in here. Common string operations, debugging routines, and command line parsing code are all in here.
* **mm** - High level memory management code is in this directory. Virtual memory (VM) is implemented through these routines, in conjunction with the low-level architecture specific routines usually found in arch/<arch>/mm/. Early boot memory management (needed before the memory subsystem is fully set up) is done here, as well as memory mapping of files, management of page caches, memory allocation, and swap out of pages in RAM (along with many other things).
* **net** - The high-level networking code is here. The low-level network drivers pass received packets up to and get packets to send from this level, which may pass the data to a user-level application, discard the data, or use it in-kernel, depending on the packet. The net/core directory contains code useful to most of the different network protocols, as do some of the files in the net/ directory itself. Specific network protocols are implemented in subdirectories of net/.
* **scripts** - This directory contains scripts that are useful in building the kernel, but does not include any code that is incorporated into the kernel itself. The various configuration tools keep their files in here, for example.
* **security** - Code for different Linux security models can be found here, such as NSA Security-Enhanced Linux and socket and network security hooks.
* Sound - Drivers for sound cards and other sound related code is placed here.
* **usr** - This directory contains code that builds a cpio-format archive containing a root filesystem image, which will be used for early userspace.

# 11. WHAT DO YOU MEAN BY MODULES?

* **Modules are** pieces of code that **can** be loaded and unloaded into the **kernel** upon demand.
* They extend the functionality of the kernel without the need to reboot the system.
* The kernel consists of a set of kernel modules that interact with each other, each performing a specific function. Some kernel modules perform software functions exclusively, while others (such as device drivers) control the operation of system hardware components.

# 12. Activity QEMU installation

* QEMU is a generic and open source machine emulator and virtualizer.
* QEMU is used to emulate devices and certain privileged instructions and requires either the KQEMU or KVM kernel modules and the host operating system

**Installing QEMU on ARM based architecture**

* + sudo apt install qemu-system-arm

**Running QEMU by ZImage and vexpress dtb file**

* 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"

# Activity TOOLCHAIN Installation

* Installing soft load on ARM Architecture
* sudo apt install gcc-arm-linux-gnueabi

**Download Kernel Source**

Downloading from linux tar.xz from the source and extract it in a new folder and then Obtain the zImage and vexpress dtb file.

Linux Commands:

* make ARCH=arm mrproper
* make ARCH=arm vexpress\_defconfig

# **14. Building Kernel Modules**:

## **14.1 Simple Hello Module:**

* Step 1 : Building the hello.c file and writing the contents
* Step 2: make file and writing the contents ( obj-m += hello.o )
* Cross compile using make
  + make –C ${KSRC} M=${PWD} modules ARCH=arm, CROSS\_COMPILE=arm-linux-gnueabi-
* testing on target
  + sudo mount –o loop,rw,sync rootfs.img /mnt/rootfs
  + sudo cp hello.ko /mnt/rootfs/home/root
  + sudo umount /mnt/rootfs

## **14.2 Simple hello Module with init and exit function**

* Building the hello.c file and writing the contents
* make file and writing the contents
  + obj-m += hello.o
  + KSRC = (where you have linux tar.xz location)
  + all: make –C ${KSRC} M=${PWD} modules
  + clean: make –C ${KSRC} M=${PWD} clean
* Cross compile using make command
* Testing on the target

### 3 Testing on Target (Qemu)

# copy the ko file to target rootfs

# if cross compiled for Qemu, skip this if

# compiled natively for ubuntu

sudo mount –o loop,rw,sync rootfs.img /mnt/rootfs

sudo cp hello.ko /mnt/rootfs/home/root

sudo umount /mnt/rootfs

**# Testing module on target**

dmesg -c

insmod hello.ko # sudo

lsmod

cat /proc/modules

dmesg

rmmod hello # sudo

dmesg

### Testing On Host (Ubuntu)

# copy the ko file to target rootfs

# if cross compiled for Qemu, skip this if

# compiled natively for ubuntu

sudo mount –o loop,rw,sync rootfs.img /mnt/rootfs

sudo cp hello.ko /mnt/rootfs/home/root

sudo umount /mnt/rootfs

**# Testing module on target**

dmesg -c

insmod hello.ko # sudo

lsmod

cat /proc/modules

dmesg

rmmod hello # sudo

dmesg

### In-Tree Module : Dynamic

#Create a sub dir in KSRC

mkdir drivers/char/dtest

# place hello.c in dtest

# create a Makefile in dtest

obj-m += hello.o

# add following entry to drivers/char/Makefile

# be cautious about this step, as editing

# existing file

obj-m += dtest/

# Re-build the kernel & redeploy

# Reboot with updated kernel image

# updated rootfs in case of Qemu

find /lib/modules –name hello.ko

dmesg –c

modprobe hello

make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi- zImage modules

sudo mount –o loop,rw.sync rootfs.img /mnt/rootfs

make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi- modules\_install INSTALL\_MOD\_PATH=/mnt/rootfs

sudo umount /mnt/rootfs

### 14.6 Static Module (In-Tree) :

#Create a sub dir in KSRC

mkdir drivers/char/stest

# place sdemo.c in dtest

# create a Makefile in dtest

obj-y += sdemo.o

# add following entry to drivers/char/Makefile

obj-y += stest/

# be cautious , as editing existing file

# Re-build the kernel & redeploy

# reboot with new kernel

# No need to rebuild modules (or) modules\_install

#check /proc/kallsyms

cat /proc/kallsyms | grep sdemo\_init

cat /proc/kallsyms | grep svar

cat /proc/kallsyms | grep sayHello

dmesg | grep sdemo

# check sdemo\_init under generated

# System.map also

arm-linux-gnueabi-nm vmlinux | grep sdemo\_init

arm-linux-gnueabi-objdump -t vmlinux | grep sdemo\_init

# normal nm, objdump in case of native

## **14.7 Hello module with parameters**

* Building the hello.c file and writing the contents
* **The contents added to be are:**
* int ndevices=1
* module\_param(ndevices,int,S\_IRUGO);
* make file and writing the contents

* Now in host i.e QEMU pass the arguments like insmod ndevices = 5 or by default it will be 1

## **14.8 Module Dependency simple**

* Building the hello.c file and writing the contents
  + - The contents added to be are :
    - The functions and variable are present in the hello.c file
    - EXPORT\_SYMBOL\_GPL(xvar);
    - EXPORT\_SYMBOL\_GPL(sayHello);
    - make file and writing the contents
* **obj-m += simple.o**
* **all:  
   make -C /home/user/eworkspace/kernel\_ws/ksrc M=${PWD} modules ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi-  
  clean:**
  + - * **modules ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi-**
      * Now open the emulation using tempboot location
      * run command to print the contents insmod (.ko) file
      * dmesg will display the contents of the file

## **14.9 Module Dependency sample**

* Building the hello.c file and writing the contents
  + - The contents are added to be are apart from simple
    - **extern int xvar;**
    - **extern void sayHello(void);**
    - Then after importing the module from simple we can use the functions defined in the simple module by printing in the sample module
    - We need to first run the simple module and then sample module so that we can use the functions present in the simple module

## **14.10 ADDING KCONFIG ENTRIES**

Version 1 for K config entries:

Name a file hello.c in folder mtest

* + - * + config HELLO

tristate "Hello module"

default n

help

A Hello module

* + - * + Now making Makefile for the program:

obj-$(CONFIG\_SIMPLE) += hello.o

* + - * + Now update the make file present in the outside folder that is char folder

obj-y += mtest/

* + - * + Add the statement to the outside K config

source "drivers/char/mtest/Kconfig"

## **14.11 Version 2 for K Config entries:**

* + - * + Name a file hello.c in folder mtest add into Kconfig blank file
        + menu "My Custom Modules“

config SIMPLE

tristate "Simple module"

default n

help A

Hello module

endmenu

# 15 WHAT DO YOU MEAN BY SYSTEM CALLS?

* A **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.
* System call **provides** the services of the operating system to the user programs via Application Program Interface(API).
* Interface to OS Services, Communication between Kernel mode and User mode.
* System calls initiated by user space, executed by kernel space
* System calls are also referred as software interrupts
* Identified by Unique Number
* Written in C or Assemble within Kernel space
* System call offers the services of the operating system to the user programs via API (Application Programming Interface)
* Follows “standard protocol” for parameter flow and return values
* No common memory between user space and kernel space, hence system calls use REGISTERS for communications
* If arguments are more than available registers, then arguments are packed in structures or blocks and address is passed in register
* ABI (Application Binary Interface) - Identify system call number from header file , Store system call number in specific register (accumulator) , Store parameters in other registers , Initiate TRAP instruction . On execution, system call always returns 0 or positive for SUCCESS & negative for FAIL

In the most literal sense, a system call (also called a "syscall") is an instruction, similar to the "add" instruction or the "jump" instruction. At a higher level, a system call is the way a user level program asks the operating system to do something for it. If we're writing a program, and we need to read from a file, we use a system call to ask the operating system to read the file for us.

Here's how a system call works. First, the user program sets up the arguments for the system call. One of the arguments is the system call number (more on that later). Note that all this is done automatically by library functions unless you are writing in assembly. After the arguments are all set up, the program executes the "system call" instruction. This instruction causes an exception: an event that causes the processor to jump to a new address and start executing the code there. The instructions at the new address save our user program's state, figure out what system call you want, call the function in the kernel that implements that system call, restores your user program state, and returns control back to the user program. A system call is one way that the functions defined in a device driver end up being called. That was the whirlwind tour of how a system call works. Next, we'll go into minute detail for those who are curious about exactly how the kernel does all this. Don't worry if you don't quite understand all of the details - just remember that this is one way that a function in the kernel can end up being called, and that no magic is involved. We can trace the control flow all the way through the kernel - with difficulty sometimes, but we can do it.

## 

## **15.1 Services Provided by System Calls** :

## Process creation and management

## Main memory management

## File Access, Directory and File system management

## Device handling(I/O)

## Protection

## Networking

## **15.2 Adding a system call:**

* We need to add the syscall.h with linkage
  + - asmlinkage long sys\_mytestcall(void);
  + Adding syscall number so that kernel can identify by the number:
    - 398 common mytestcall sys\_mytestcall
  + In kernel folder add mysys.c file :
* kernel/mysys.c
  + Update the kernel/Makefile:
    - * obj-y +=mysys.o
* Write this code in kernel/mysys.c file:

SYSCALL\_DEFINE0(testcall)

{

printk("This is my test call\n");

return 0;

}

Invoking System Call from Userspace:

Method 1: Generic wrapper class

Create a .c file and write this code in that file:

#include<stdio.h>

#include<

#define \_\_NR\_testcall 398

int main()

{

int ret;

ret=syscall(\_\_NR\_testcall);

if(ret<0)

perror(“Testcall”);

return 0;

}

Run the system calls by :

./filename.out

# Pseudo Char Driver:

**Step1 : Register Char Driver**

Registering the new device to the system means assigning a [major number](https://www.embhack.com/introduction-to-major-and-minor-number/) to it, during the initialization routine. The major number is provided by the kernel for any character or block device.

Two types of ways of restering a character device driver

1. Statistically registration of character device driver
2. Dynamically registration of character device driver

## **16.1 Statistically registration device driver**

When we know the [major number](https://www.embhack.com/introduction-to-major-and-minor-number/) in advance we can register the device using this method.

Two functions in the kernel for statistical registration of device driver:

* **register\_chrdev()**

int register\_chrdev(unsigned int major, const char \*name, struct file\_operations \*fops);

* **register\_chrdev\_region()**

## int register\_chrdev\_region(dev\_t first, unsigned int count, char \*name)

## **16.2 Dynamically registration of Character Device Driver**

In this method, Kernel gives the highest available major number to the device.

1. alloc\_chrdev\_region

The prototype of alloc\_chrdev\_region, is declared in <linux/fs.h>:

int alloc\_chrdev\_region(dev\_t \*dev, unsigned int firstminor, unsigned int count, char \*name);

## **16.3 Un-registration of character device driver**

To deallocate an allocated major number use the ***unregister\_chrdev()*** function. The prototype is given below and the parameters of the function are self-explanatory:

void unregister\_chrdev\_region(dev\_t first, unsigned int count);

**Step-2 : Register File Operations**

The various operations a driver can perform on the devices it manages.

open device is identified internally by a file structure, and the kernel uses the file\_operations structure to access the driver’s functions.

The structure, defined in <linux/fs.h>, is an array of function pointers. Each file is associated with its own set of functions (by including a field called f\_op that points to a file\_operations structure).

The operations are mostly in charge of implementing the system calls and are thus named *open*, *read*, and so on.

We can consider the file to be an “object” and the functions operating on it to be its “methods,” using object-oriented programming terminology to denote actions declared by an object to act on itself.

ssize\_t (\*write) (struct file \*, const char \*, size\_t, loff\_t \*);

Testing the Device Driver:

First we register the file by using :

insmod pseudo.ko

upload the module by:

mknod /dev/psample c xxx 0

See output by :

cat /dev/psample

write input by target:

echo "abc" > /dev/psample

Check output by:

dmesg

Remove file by:

rmmod filename

See result by:

rm /dev/psample

## **Device file Creation:**

## The device file allows transparent communication between user-space applications and hardware.

## All device files are stored in /dev directory.

## Use ls command to browse the directory.

## ls -l /dev/

## We can create a dive file in two ways.

### 16.4.1 Manually, Automatically

Manually Creating Device File:

We can create the device file manually by using mknod.

mknod -m

## **16.5 Create the class:**

It will create a structure under/sys/class/.

struct class \* class\_create (struct module \*owner, const char \*name);

### 

### 16.5.1 Create Device:

This function can be used by char device classes. A struct device will be created in sysfs, registered to the specified class.

struct device \*device\_create (struct \*class, struct device \*parent, dev\_t dev, const char \*fmt, ...)

### 16.5.2 Device Destroy:

void device\_destroy (struct class \* class, dev\_t devt);

Step-4: Buffer as pseudo device:

The Z-buffer device is a "pseudo device" in that drawing commands update buffers in memory rather than sending commands to a physical device or file.

To use the Z-buffer as the current graphics device, issue the IDL command:

pbuffer = kmalloc(MAX\_SIZE, GFP\_KERNEL);

Implement read, write operations:

A memory unit stores binary information in groups of bits called words.

Data input lines provide the information to be stored into the memory, Data output lines carry the information out from the memory.

The control lines Read and write specifies the direction of transfer of data.

# 17. Kernel Data Structures:

## **17.1 Kfifo API:**

The kernel FIFO implementation, kfifo, is not that widely used and Stefani Seibold would like to see that change

A kfifo is declared using the DECLARE\_KFIFO() macro which can be used inside of a struct or union declaration.

FIFOs declared with with DECLARE\_KFIFO() must be initialized using INIT\_KFIFO().

DECLARE\_KFIFO(name, size)

INIT\_KFIFO(name)

DEFINE\_KFIFO(name, size)

unsigned int kfifo\_in\_rec(struct kfifo \*fifo,

void \*from, unsigned int n, unsigned int recsize)

List implementation in Kernel:

Linked list is contained inside the node, structure of node.

there were multiple implementations of linked lists in the kernel. A single, powerful linked list implementation was needed to remove duplicate code.

The linked-list code is declared in <linux/list.h> and the data structure is simple:

struct list\_head {

struct list\_head \*next

struct list\_head \*prev;

};

A list\_head by itself is worthless; it is normally embedded inside your own structure:

struct my\_struct {

struct list\_head list;

unsigned long dog;

void \*cat;

}

## **18.CONCURRENCY**

## **KERNEL THREADS**

A **kernel thread** is the schedulable entity, which means that the system scheduler handles **kernel threads**. These **threads**, known by the system scheduler, are strongly implementation-dependent. To facilitate the writing of portable programs, libraries provide user **threads**.

A *kernel thread* is a kernel entity, like processes and interrupt handlers; it is the entity handled by the system scheduler. A kernel thread runs within a process, but can be referenced by any other thread in the system. The programmer has no direct control over these threads, unless you are writing kernel extensions or device drivers. For more information about kernel programming, see *Kernel Extensions and Device Support Programming Concepts*.

A *user thread* is an entity used by programmers to handle multiple flows of controls within a program. The API for handling user threads is provided by the *threads library*. A user thread only exists within a process; a user thread in process *A* cannot reference a user thread in process *B*. The library uses a proprietary interface to handle kernel threads for executing user threads. The user threads API, unlike the kernel threads interface, is part of a POSIX-standards compliant portable-programming model. Thus, a multithreaded program developed on an AIX® system can easily be ported to other systems.

On other systems, user threads are simply called *threads*, and *lightweight process* refers to kernel threads.

# 18.2 IPC in Kernel:

IPC mechanisms as implemented in the Linux 2.4 kernel. It is organized into four sections.

## **18.2.1 Semaphors:**

Semaphores are IPCs, which means Inter-Process Communication Systems used to allow different processes to communicate with each other. It is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multiprogramming operating system.

The functions described in this section implement the user level semaphore mechanisms. Note that this implementation relies on the use of kernel splinlocks and kernel semaphores. To avoid confusion, the term "kernel semaphore" will be used in reference to kernel semaphores. All other uses of the word "sempahore" will be in reference to the user level semaphores.

a semaphore is based on a variable.

binary semaphore;

normal semaphore.

Sequencing, signaling mechanism, used for process/thread synchronization

• Manage and protect access to shared resources

• Kernel level data structure Types of usage

• Binary Semaphore

• Value of semaphore ranges between 0 & 1

• Mutual Exclusion / Access to a single resource

• Counting Semaphore

• Value of semaphore can be 0 (zero) & any positive value

• Accessing/sharing multiple similar resources Two (2) varieties of semaphores

• Traditional System V semaphores

• POSIX semaphores.

Two (2) types of POSIX semaphores

• Named

• Unnamed

**Name is given to semaphore and can be access by parent & child or different processes**

• Uses internal shared memory for resources access

POSIX API’s

#include <semaphore.h>

#include <errno.h>

• sem\_t \*ps; (declare a semaphore variable)

• ps = sem\_open(“/s1", O-CREAT, 0666, 1) (internal shared memory)

• sem\_wait(ps) (lock the semaphore)

• sem\_post(ps) (unlock the semaphore)

• sem\_close(ps) (close semaphore from process)

• sem\_unlink(ps) (remove named semaphore)

All calls return 0 on success, -1 on error and ‘errno’ variable is set to error number.

**No name is given to the Semaphore.**

• Memory is allocated in the program address space

POSIX Unnamed Semaphore API’s

#include <semaphore.h>

#include <errno.h>

• sem\_init(sem\_t \*sem, int pshared, unsigned int value) (Initialize unnamed semaphore)

• sem\_wait(sem\_t \*sem) (Lock the semaphore )

• Check sem\_trywait & sem\_timedwait

• sem\_post(sem\_t \*sem) (Unlock the semaphore)

• sem\_destroy(sem\_t \*sem) (Destroy the semaphore )

All calls return 0 on success, -1 on error and ‘errno’ variable is set to error number.

## **18.2.2 Semaphore API**

semaphore API is located in the include/linux/semaphore.h header file. The semaphore mechanism is represented by the following structure.

struct semaphore

{

raw\_spinlock\_t lock;

unsigned int count;

struct list\_head wait\_list;

};

in the Linux kernel. The semaphore structure consists of three fields:

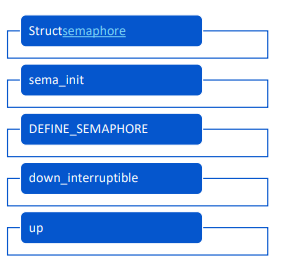
lock - spinlock for a semaphore data protection;

count - amount available resources;

wait\_list - list of processes which are waiting to acquire a lock.

#define DEFINE\_SEMAPHORE(name) \

struct semaphore name = \_\_SEMAPHORE\_INITIALIZER(name, 1)



## **18.2.3 Mutex:**

Mutex is a mutual exclusion object that synchronizes access to a resource. It is created with a unique name at the start of a program. The Mutex is a locking mechanism that makes sure only one thread can acquire the Mutex at a time and enter the critical section.

wait (mutex);

Critical Section

signal (mutex);

Mutual Exclusion

• Only locked Process(es)/Threads can unlock the resources

• Any other Process/Threads trying to unlock is referred as “unauthorized operation”

• Unlocking twice or unlocking before locking is not allowed

• Strictly lock & unlock in the same thread only

• Mutex will have "ownership" as compared to semaphore

• #include <pthread.h>

• pthread\_mutex\_t m1=PTHREAD\_MUTEX\_INITIALIZER (declare & initialize)

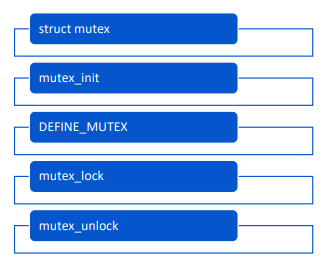
• pthread\_mutex\_init(&m1)

• pthread\_mutex\_lock(&m1) (lock)

• pthread\_mutex\_unlock(&m1) (unlock)

• pthread\_mutex\_destroy (&m1) (destroy)

Always check return value for Success or Failure.



## **18.2.4 Spin Locks:**

A spin lock is a way to protect a shared resource from being modified by two or more processes simultaneously. The first process that tries to modify the resource "acquires" the lock and continues on its way, doing what it needed to with the resource. Any other processes that subsequently try to acquire the lock get stopped; they are said to "spin in place" waiting on the lock to be released by the first process, thus the name spin lock. The spinlock is a low-level synchronization mechanism which in simple words, represents a variable which can be in two states:

* acquired;
* released.

Each process which wants to acquire a spinlock, must write a value which represents spinlock acquired state to this variable and write spinlock released state to the variable. If a process tries to execute code which is protected by a spinlock, it will be locked while a process which holds this lock will release it. In this case all related operations must be [atomic](https://en.wikipedia.org/wiki/Linearizability) to prevent [race conditions](https://en.wikipedia.org/wiki/Race_condition) state. The spinlock is represented by the spinlock\_t type in the Linux kernel.

When two or more processes require dedicated access to a shared resource, they might need to enforce the condition that they are the sole process to operate in a given section of code. The basic form of locking in the Linux kernel is the spinlock.

Spinlocks take their name from the fact that they continuously loop, or spin, waiting to acquire a lock. Because spinlocks operate in this manner, it is imperative not to have any section of code inside a spinlock attempt to acquire a lock twice. This results in deadlock.

Before operating on a spinlock, the spinlock\_t structure must be initialized. This is done by calling spinlock\_init().

The most basic primitive for locking is spinlock.

static DEFINE\_SPINLOCK(xxx\_lock);

unsigned long flags;

spin\_lock\_irqsave(&xxx\_lock, flags);

... critical section here ..

spin\_unlock\_irqrestore(&xxx\_lock, flags);

Documentation/memory-barriers.txt

(5) LOCK operations.

(6) UNLOCK operations.

## **18.2.5 Reader-writer spinlocks:**

If your data accesses have a very natural pattern where you usually tend

to mostly read from the shared variables, the reader-writer locks

(rw\_lock) versions of the spinlocks are sometimes useful.

rwlock\_t xxx\_lock = \_\_RW\_LOCK\_UNLOCKED(xxx\_lock);

unsigned long flags;

read\_lock\_irqsave(&xxx\_lock, flags);

read\_unlock\_irqrestore(&xxx\_lock, flags);

write\_lock\_irqsave(&xxx\_lock, flags);

write\_unlock\_irqrestore(&xxx\_lock, flags);



**18.2.6 Wait Queue API:**

A wait queue is used to wait for someone to wake you up when a certain condition is true. They must be used carefully to ensure there is no race condition. You declare a wait\_queue\_head\_t, and then processes which want to wait for that condition declare a wait\_queue\_t referring to themselves, and place that in the queue. A "**wait queue**" in the **Linux kernel** is a data structure to manage threads that are **waiting** for some condition to become true; they are the normal means by which threads block (or "sleep") in **kernel** space. Over the years, the **wait queue** mechanism has evolved into a fairly elaborate and complicated **kernel** subsystem.

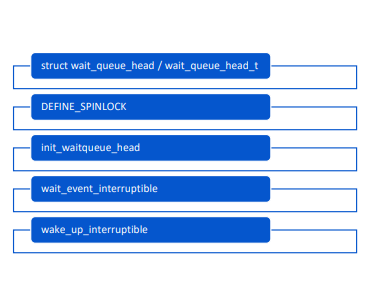
**Declaring**

You declare a wait\_queue\_head\_t using the DECLARE\_WAIT\_QUEUE\_HEAD() macro, or using the init\_waitqueue\_head() routine in your initialization code.

When you write a Linux Driver or Module or Kernel Program. Some processes should wait or sleep for some event. There are several ways of handling sleeping and waking up in Linux, each suited to different needs. Wait queue also one of the methods to handle that case.

Whenever a process must wait for an event (such as the arrival of data or the termination of a process), it should go to sleep. Sleeping causes the process to suspend execution, freeing the processor for other uses. After some time, the process will be woken up and will continue with its job when the event which we are waiting for has arrived.

Wait queue is a mechanism provided in the kernel to implement the wait. As the name itself suggests, wait queue is the list of processes waiting for an event. In other words, A wait queue is used to wait for someone to wake you up when a certain condition is true. They must be used carefully to ensure there is no race condition.



## **18.2.7 Generate Race Conditions in Pseudo Driver:**

A race condition is a concurrency problem that may occur inside a critical section. A critical section is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section.

Two Types of Race Conditions

Race conditions can occur when two or more threads read and write the same variable according to one of these two patterns:

Read-modify-write

Check-then-act

## **DRIVER MODEL IN KERNEL**

* The Linux Kernel Driver Model is a unification of all the disparate driver models that were previously used in the kernel. It is intended to augment the bus-specific drivers for bridges and devices by consolidating a set of data and operations into globally accessible data structures.
* Traditional driver models implemented some sort of tree-like structure (sometimes just a list) for the devices they control. There wasn’t any uniformity across the different bus types.
* The current driver model provides a common, uniform data model for describing a bus and the devices that can appear under the bus. The unified bus model includes a set of common attributes which all busses carry, and a set of common callbacks, such as device discovery during bus probing, bus shutdown, bus power management, etc.
* The common device and bridge interface reflects the goals of the modern computer: namely the ability to do seamless device “plug and play”, power management, and hot plug. In particular, the model dictated by Intel and Microsoft (namely ACPI) ensures that almost every device on almost any bus on an x86-compatible system can work within this paradigm. Of course, not every bus is able to support all such operations, although most buses support most of those operations.

# 20. IOCTL usage:

IOCTL is referred to as Input and Output Control, which is used to talking to device drivers. This system call, available in most driver categories.  The major use of this is in case of handling some specific operations of a device for which the kernel does not have a system call by default.

Some real-time applications of ioctl are Ejecting the media from a “cd” drive, to change the Baud Rate of Serial port, Adjust the Volume, Reading or Writing device registers, etc. We already have the write and read function in our device driver. But it is not enough for all cases.

here are some steps involved to use IOCTL.

* Create IOCTL command in driver
* Write IOCTL function in the driver
* Create IOCTL command in a Userspace application
* Use the IOCTL system call in a Userspace

The **ioctl**() system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with **ioctl**() requests. The argument *fd* must be an open file descriptor.

ioctl() is the most common way for applications to interface with device drivers. It is flexible and easily extended by adding new commands and can be passed through character devices, block devices as well as sockets and other special file descriptors. However, it is also very easy to get ioctl command definitions wrong, and hard to fix them later without breaking existing applications, so this documentation tries to help developers get it right.

**#include <sys/ioctl.h>**

The second argument is a device-dependent request code. Thethird argument is an untyped pointer to memory. It' traditionally char \*argp (from the days before void \* was valid C), and will be so named for this discussion.

# 21. References:

* www.geekfforgeeks.com
* Wikipedia
* Javatpoint
* W3school.com
* www.kernel.org/doc/html/latest/devicetree/usage-model.html