SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – I - EMBEDDED OS AND DEVICE DRIVERS – SECA5204

1. OVERVIEW OF EMBEDDED SYSTEMS

(Embedded System Architecture fundamentals. Hardware and Software abstraction models)

1.1 Overview of Embedded Application Architecture

Embedded systems, an emerging area of computer technology, combine multiple technologies, such as computers, semiconductors, microelectronics, and the Internet, and as a result, are finding ever-increasing application in our modern world. With the rapid development of computer and communications technologies and the growing use of the Internet, embedded systems have brought immediate success and widespread application in the post-PC era, especially as the core components of the Internet of Things. They penetrate into every corner of modern life from the mundane, such as an automated home thermostat, to industrial production, such as in robotic automationin manufacturing. Embedded systems can be found in military and national defense, healthcare, science, education, and commercial services, and from mobile phones, MP3 players, and PDAs to cars, planes, and missiles.

This chapter provides the concepts, structure, and other basic information about embedded systems and lays a theoretical foundation for embedded application development, of which application development for Android OS is becoming the top interest of developers.

1.2 Introduction to Embedded Systems

Since the advent of the first computer, the ENIAC, in 1946, the computer manufacturing process has gone from vacuum tubes, transistors, integrated circuits, and large-scale integration (LSI), to very-large-scale integration (VLSI), resulting in computers that are more compact, powerful, and energy efficient but less expensive (per unit of computing power).

After the advent of microprocessors in the 1970s, the computer-using world witnessed revolutionary change. Microprocessors are the basis of microcomputers, and personal computers (PCs) made them more affordable and practical, allowing many private users to own them. At this stage, computers met a variety of needs: they were sufficiently versatile to satisfy various demands such as computing, entertainment, information sharing, and office automation. As the adoption of microcomputers was occurring, more people wanted to embed them into specific systems to intelligently control the environment. For example, microcomputers were used in machine tools in factories. They were used to control signals and monitor the operating state through the configuration of peripheral sensors. When microcomputers were embedded into such environments, they were prototypes of embedded systems.

As the technology advanced, more industries demanded special computer systems. As a result, the development direction and goals of specialized computer systems for specific environments and general-purpose computer systems grew apart. The technical requirement of general-purpose computer systems is fast, massive, and diversified computing, whereas the goal of technical development is faster computing speed and larger storage capacity. However, the

technical requirement of embedded computer systems is targeted more toward the intelligent control of targets, whereas the goal of technical development is embedded performance, control, and reliability closely related to the target system.

Embedded computing systems evolved in a completely different way. By emphasizing the characteristics of a particular processor, they turned traditional electronic systems into modern intelligent electronic systems. Figure 1-1 shows an embedded computer processor, the Intel Atom N2600 processor, which is 2.2 × 2.2 cm, alongside a penny.

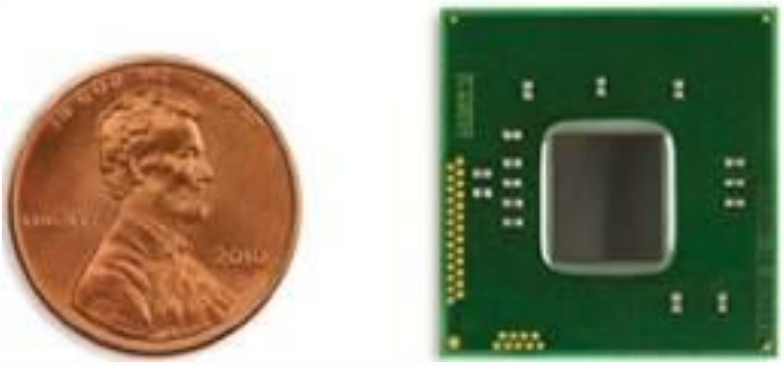


Figure 1-1. Comparison of an embedded computer chip to a US penny. This chip is an Intel Atom processor

The emergence of embedded computer systems alongside general-purpose computer systems is a milestone of modern computer technologies. The comparison of general-purpose computers and embedded systems is shown in Table 1-1.

Table 1-1. Comparison of General-Purpose Computers and Embedded Systems

Item	General-purpose computer systems	Embedded systems
Hardware	High-performance hardware, large storage media	Diversified hardware, single-processor solution
Software	Large and sophisticated OS	Streamlined, reliable, real-time systems
Development	High-speed, specialized development team	Broad development sectors

Today, embedded systems are an integral part of people's lives due to their mobility. As mentioned earlier, they are used everywhere in modern life. Smartphones are a great example of embedded systems.

1.2.1 Mobile Phones

Mobile equipment, especially smartphones, is the fastest growing embedded sector in recent years. Many new terms such as *extensive embedded development* and *mobile development* have been derived from mobile software development. Mobile phones not only are pervasive but also have powerful functions, affordable prices, and diversified applications. In addition to basic telephone functions, they include, but are not limited to, integrated PDAs, digital cameras, game consoles, music players, and wearables.

1.2.2 Consumer Electronics and Information Appliances

Consumer electronics and information appliances are additional big application sectors for embedded systems. Devices that fall into this category include personal mobile devices and home/entertainment/audiovisual devices. Personal mobile devices usually include smart handsets such as PDAs, as well as wireless Internet access equipment like mobile Internet devices (MIDs). In theory, smartphones are also in this class; but due to their large number, they are listed as a single sector.

Home/entertainment/audiovisual devices mainly include network television like interactive television; digital imaging equipment such as digital cameras, digital photo frames, and video players; digital audio and video devices such as MP3 players and other portable audio players; and electronic entertainment devices such as handheld game consoles, PS2 consoles, and so on. Tablet PCs (tablets), one of the newer types of embedded devices, have become favorites of consumers since Apple released the iPad in 2010.

1.3 General architecture of an Embedded System

Figure 1-2 shows a configuration diagram of a typical embedded system consisting of two main parts: embedded hardware and embedded software. The embedded hardware primarily includes the processor, memory, bus, peripheral devices, I/O ports, and various controllers. The embedded software usually contains the embedded operating system and various applications. Input and output are characteristics of any open system, and the embedded system is no exception. In the embedded system, the hardware and software often collaborate to deal with various input signals from the outside and output the processing results through some form.

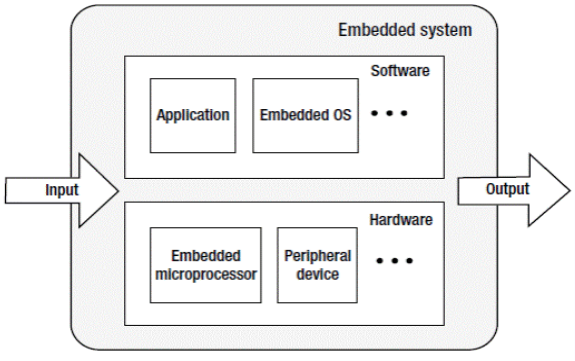


Figure 1-2. Basic architecture of an embedded system

The input signal may be an ergonomic device (such as a keyboard, mouse, or touch screen) or the output of a sensor circuit in another embedded system. The output may be in the form of sound, light, electricity, or another analog signal, or a record or file for a database.

1.3.1 Hardware Architecture of Embedded systems

The basic computer system components—microprocessor, memory, and input and output modules are interconnected by a system bus in order for all the parts to communicate and execute a program (see Figure 1-3).

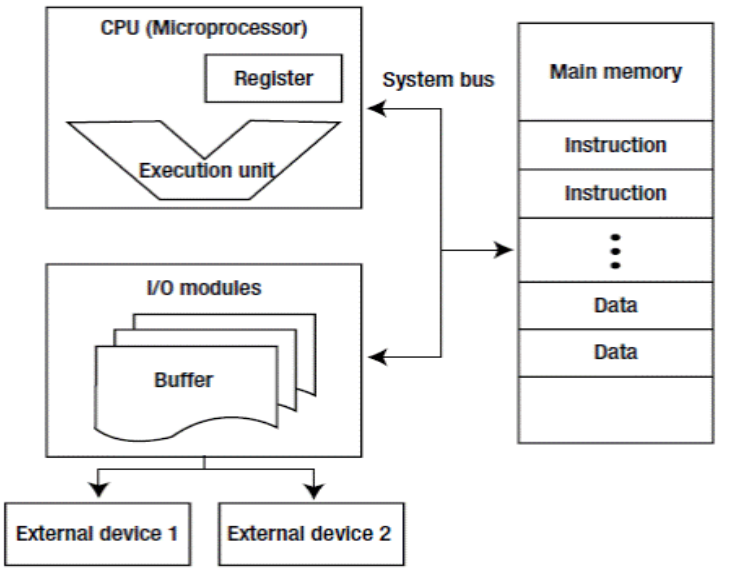


Figure 1-3. Hardware architecture of Embedded System

In embedded systems, the microprocessor's role and function are usually the same as those of the CPU in a general-purpose computer: control computer operation, execute instructions, and process data. In many cases, the microprocessor in an embedded system is also called the CPU. Memory is used to store instructions and data. I/O modules are responsible for the data exchange between the processor, memory, and external devices.

External devices include secondary storage devices (such as flash and hard disk), communications equipment, and terminal equipment. The system bus provides data and controls signal communication and transmission for the processor, memory, and I/O modules.

There are basically two types of architecture that apply to embedded systems: Von Neumann architecture and Harvard architecture. In a Von-Neumann architecture, the same memory and bus are used to store both data and instructions that run the program. Since you cannot access program memory and data memory simultaneously, the Von Neumann architecture is susceptible to bottlenecks and system performance is affected.

1.3.2 Von Neumann Architecture

Von Neumann architecture (also known as Princeton architecture) was first proposed by John von Neumann. The most important feature of this architecture is that the software and data use the same memory: that is, "The program is data, and the data is the program" (as shown in Figure 1-4).

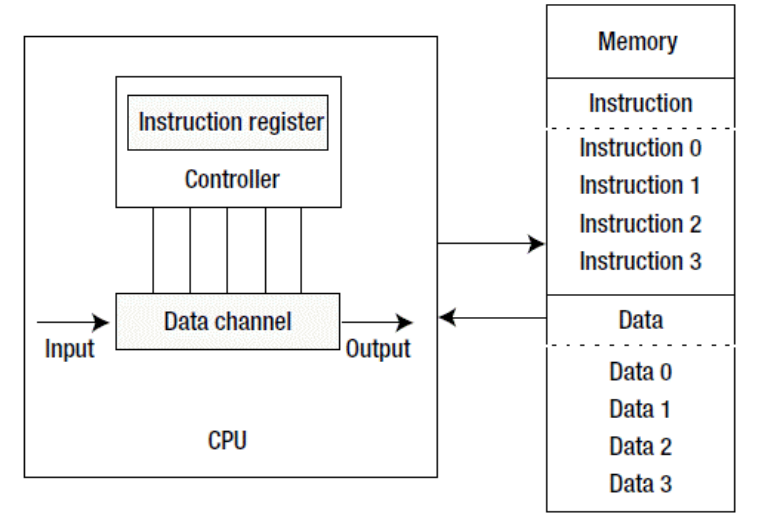


Figure 1-4. Von Neumann architecture

In the Von Neumann architecture, an instruction and data share the same bus. In this architecture, the transmission of information becomes the bottleneck of computer performance and affects the speed of data processing; so, it is often called the *Von Neumann bottleneck*. In reality, cache and branch-prediction technology can effectively solve this issue.

1.3.3 Harvard Architecture

The Harvard architecture was first named after the Harvard Mark I computer. Compared with the Von Neumann architecture, a Harvard architecture processor has two outstanding features. First, instructions and data are stored in two separate memory modules; instructions and data do not coexist in the same module. Second, two independent buses are used as dedicated communication paths between the CPU and memory; there is no connection between the two buses. The Harvard architecture is shown in Figure 1-5.

To efficiently perform memory reads/writes, the processor is not directly connected to the main memory, but to the cache. Commonly, the only difference between the Harvard architecture and the Von Neumann architecture is single or dual L1 cache. In the Harvard architecture, the L1 cache is often divided into an instruction cache (I cache) and a data cache (D cache), but the Von-Neumann architecture has a single cache.

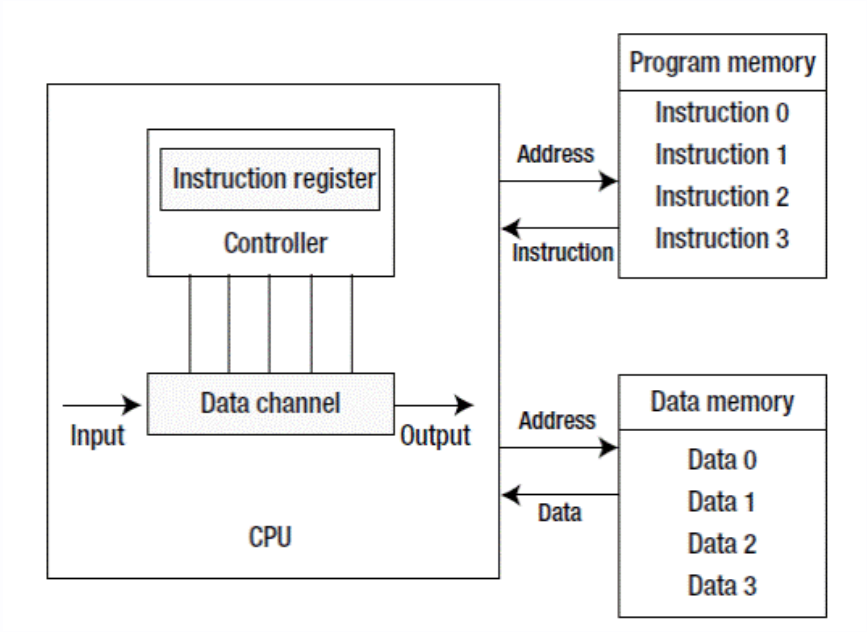


Figure 1-5. Harvard architecture

Because the Harvard architecture has separate program memory and data memory, it can provide greater data-memory bandwidth, making it the ideal choice for digital signal processing. Most systems designed for digital signal processing (DSP) adopt the Harvard architecture. The Von Neumann architecture features simple hardware design and flexible program and data storage and is usually the one chosen for general-purpose and most embedded systems.

1.3.4 Microprocessor Architecture for Embedded Systems

The microprocessor is the core in embedded systems. By installing a microprocessor into a special circuit board and adding the necessary peripheral circuits and expansion circuits, a practical embedded system can be created. The microprocessor architecture determines the instructions, supporting peripheral circuits, and expansion circuits. There are wide ranges of microprocessors: 8-bit, 16-bit, 32-bit and 64-bit, with clock performance from MHz to GHz, and ranging from a few pins to thousands of pins.

In general, there are two types of embedded microprocessor architecture: *reduced instruction set computer (RISC)* and *complex instruction set computer (CISC)*. The RISC processor uses a small, limited, simple instruction set. Each instruction uses a standard word length and has a short execution time, which facilitates the optimization of the instruction pipeline. To compensate for the command functions, the CPU is often equipped with a large number of general-purpose registers. The CISC processor features a powerful instruction set and different instruction lengths, which facilitates the pipelined execution of instructions.

Currently, microprocessors used in most embedded systems have five architectures: RISC, CISC, MIPS, PowerPC, and SuperH. The details follow.

1.3.4.1 CISC Architecture

The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction. Computers based on the CISC architecture are designed to decrease the memory cost (figure 1.6).

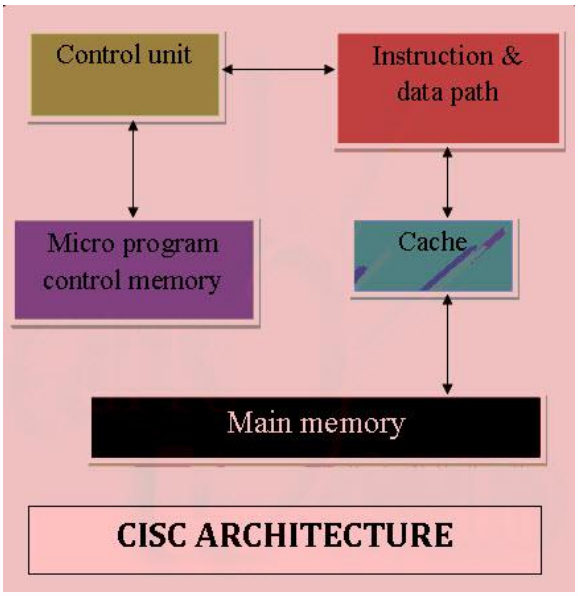


Figure 1.6 CISC Architecture

Because, the large programs need more storage, thus increasing the memory cost and large memory becomes more expensive. To solve these problems, the number of instructions per program can be reduced by embedding the number of operations in a single instruction, thereby making the instructions more complex.

Characteristics of CISC processor

- MUL loads two values from the memory into separate registers in CISC.
- CISC uses minimum possible instructions by implementing hardware and executes operations.
- Instruction-decoding logic will be Complex.
- One instruction is required to support multiple addressing modes.
- Less chip space is enough for general purpose registers for the instructions that are operated directly on memory.
- Various CISC designs are set up two special registers for the stack pointer, handling interrupts, etc.
- MUL is referred to as a "complex instruction" and requires the programmer for storing functions.

Note: Instruction Set Architecture is a medium to permit communication between the programmer and the hardware. Data execution part, copying of data, deleting or editing is

the user commands used in the microprocessor and with this microprocessor the Instruction set architecture is operated.

Examples of CISC PROCESSORS

- **IBM 370/168** – It was introduced in the year 1970. CISC design is a 32 bit processor and four 64-bit floating point registers.
- **VAX 11/780** – CISC design is a 32-bit processor and it supports many numbers of addressing modes and machine instructions which is from Digital Equipment Corporation.
- **Intel 80486** – It was launched in the year 1989 and it is a CISC processor, which has instructions varying lengths from 1 to 11 and it will have 235 instructions.

1.3.4.2 RISC Architecture

RISC (Reduced Instruction Set Computer) processors take simple instructions and are executed within a clock cycle. The first RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s. The IBM 801, Stanford MIPS, and Berkeley RISC 1 and 2 were all designed with a similar philosophy which has become known as RISC. Certain design features have been characteristic of most RISC processors:

- *one cycle execution time*: RISC processors have a CPI (clock per instruction) of one cycle. This is due to the optimization of each instruction on the CPU and a technique called pipelining.
- *pipelining*: A techique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions;
- *large number of registers*: the RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory

RISC is used in portable devices due to its power efficiency. For Example, Apple iPod and Nintendo DS. RISC is a type of microprocessor architecture that uses highly-optimized set of instructions. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program Pipelining is one of the unique feature of RISC. It is performed by overlapping the execution of several instructions in a pipeline fashion. It has a high performance advantage over CISC.

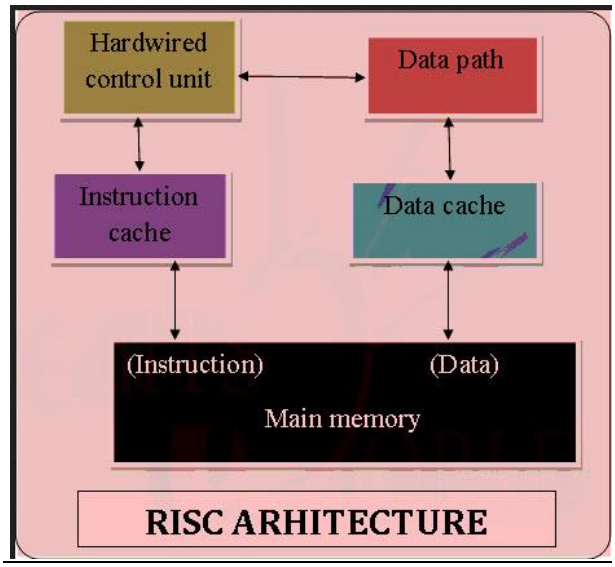


Figure 1.7 RISC Architecture

RISC ARCHITECTURE CHARACTERISTICS

- Simple Instructions are used in RISC architecture.
- RISC helps and supports few simple data types and synthesize complex data types.
- RISC utilizes simple addressing modes and fixed length instructions for pipelining.
- RISC permits any register to use in any context.
- One Cycle Execution Time
- The amount of work that a computer can perform is reduced by separating "LOAD" and "STORE" instructions.
- RISC contains Large Number of Registers in order to prevent various number of interactions with memory.
- In RISC, Pipelining is easy as the execution of all instructions will be done in a uniform interval of time i.e. one click.
- In RISC, more RAM is required to store assembly level instructions.
- Reduced instructions need a less number of transistors in RISC.
- RISC uses Harvard memory model means it is Harvard Architecture.
- A compiler is used to perform the conversion operation means to convert a high-level language statement into the code of its form.

A comparison of RISC and CISC is given in Table 1-2.

Table 1-2. Comparison of RISC and CISC

	RISC	CISC
Instruction system	Simple and efficient instructions. Realizes uncommon functions through combined instructions.	Rich instruction system. Performs specific functions through special instructions; handles special tasks efficiently.
Memory operation	Restricts the memory operation and simplifies the controlling function.	Has multiple memory operation instructions and performs direct operation.
Program	Requires a large amount of memory space for the assembler and features complex programs for special functions.	Has a relatively simple assembler and features easy and efficient programming of scientific computing and complex operations.
Interruption	Responds to an interrupt only at the proper place in instruction execution.	Responds to an interruption only at the end of execution.
CPU	Features fewer unit circuits, small size, and low power consumption.	Has feature-rich circuit units, powerful functions, a large area, and high power consumption.
Design cycle	Features a simple structure, a compact layout, a short design cycle, and easy application of new technologies.	Features a complex structure and long design cycle.
Usage	Features a simple structure, regular instructions, simple control, and easy learning and application.	Features a complex structure, powerful functions, and easy realization of special functions.
Application scope	Determines the instruction system per specific areas, which is more suitable for special machines.	Becomes more suitable for general-purpose machines.

RISC and CISC have distinct characteristics and advantages, but the boundaries between RISC and CISC begin to blur in the microprocessor sector. Many traditional CISCs absorb RISC advantages and use a RISC-like design. Intel x86 processors are typical of them. They are considered CISC architecture. These processors translate x86 instructions into RISC-like instructions through a decoder and comply with the RISC design and operation to obtain the benefits of RISC architecture and improve internal operation efficiency. A processor's internal instruction execution is called *micro operation*, which is denoted as *micro-OP* and abbreviated *mu-op* (or written *m-op* or *mop*). In contrast, the x86 instruction is called *macro operation* or *macro-op*. The entire mechanism is shown in Figure 1-6.

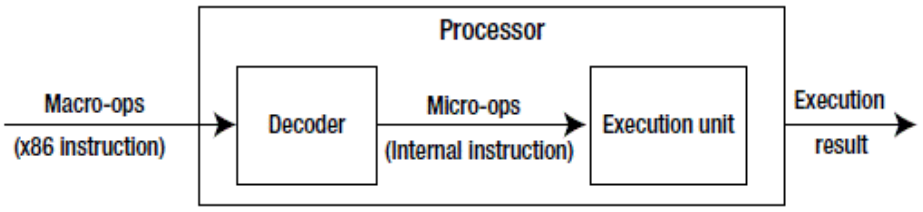


Figure 1-7. Micro and macro operations of an Intel processor

Normally, a macro operation can be decoded into one or more micro operations to execute, but sometimes a decoder can combine several macro operations to generate a micro operation to execute. This process is known as x86 *instruction fusion* (macro-ops fusion). For example, the processor can combine the x86 CMP (Compare) instruction and the x86 JMP (Jump) instruction to produce a single micro operation—the compare and jump instruction. This combination has obvious benefits: there are fewer instructions, which indirectly enhances the performance of the processor execution. And the fusion enables the processor to maximize the parallelism between the instructions and consequently improve the implementation efficiency of the processor.

1.3.4.3 MIPS Architecture

Microprocessor without Interlocked Piped Stages (MIPS) is also a RISC processor. Its mechanism is to make full use of the software to avoid data issues in the pipeline. It was first developed by a research team led by Professor John Hennessy of Stanford University in the early 1980s and later was commercialized by MIPS Technologies. Like ARM, MIPS Technologies provides MIPS microprocessor cores to semiconductor companies through intelligence property (IP) cores and allows them to further develop embedded microprocessors in the RISC architecture. The core technology is a multiple-issue capability: split the idle processing units in the processor to virtualize as another core and improve the utilization of processing units. The major c*omponents of the MIPS architecture are*:

- Program counter (PC)
- Instruction register (IR)
- Registers (general purpose)
- Arithmetic and logic unit (ALU)
- Memory

Figure 1.8 shows the basic elements of MIPS architecture based microprocessor.

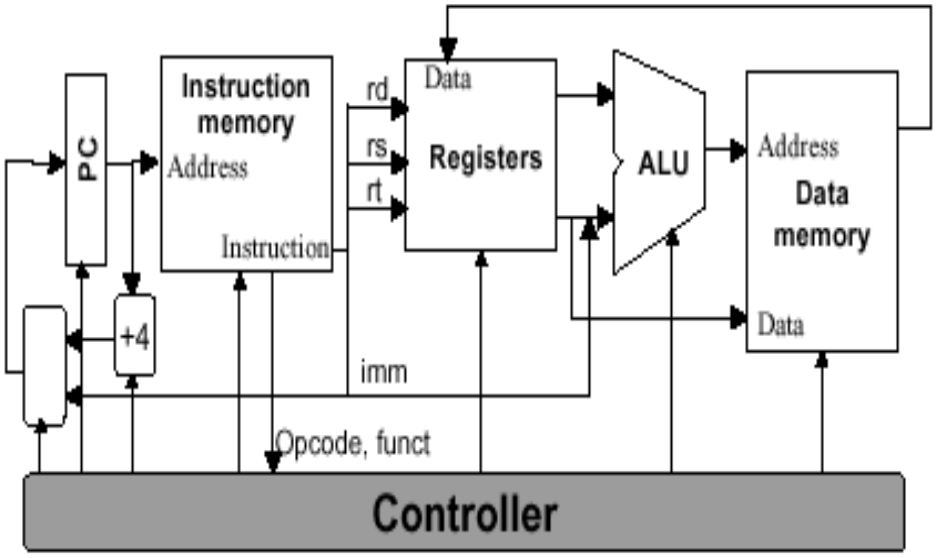


Figure 1.8. Elements of MIPS architecture based microprocessor

1.3.4.4 PowerPC Architecture

PowerPC is a CPU in the RISC architecture. It derives from the POWER architecture, and its basic design comes from the IBM PowerPC 601 microprocessor Performance Optimized with Enhanced RISC (POWER). In the 1990s, IBM, Apple, and Motorola successfully developed the PowerPC chip and created a PowerPC-based multiprocessor computer. The PowerPC architecture features scalability, convenience, flexibility, and openness: it defines an instruction set architecture (ISA), allows anyone to design and manufacture PowerPC-compatible processors, and freely uses the source code of software modules developed for PowerPC. PowerPC has a broad range of applications from mobile phones to game consoles, with wide application in the communications and networking sectors such as switches, routers, and so on. The Apple Mac series used PowerPC processors for a decade until Apple switched to the x86 architecture.

1.3.4.5 SuperH Architecture

SuperH (SH) is a highly cost-effective, compact, embedded RISC processor. The SH architecture was first developed by Hitachi and was owned by Hitachi and ST Microelectronics. Now it has been taken over by Renesas. SuperH includes the SH-1, SH2, SH-DSP, SH-3, SH-3-DSP, SH-4, SH-5, and SH-X series and is widely used in printers, faxes, multimedia terminals, TV game consoles, set-top boxes, CD-ROM, household appliances, and other embedded systems.

Some real world processor architectures families and its manufacturer details are listed in Table 1.3.

Table 1.3 Real world embedded processors and their architectures.

Architecture	Processor	Manufacturer
AMD	Au1xxx	Advanced Micro Devices, …
ARM	ARM7, ARM9, …	ARM, …
C16X	C167CS, C165H, C164CI, …	Infineon, …
ColdFire	5282, 5272, 5307, 5407, …	Motorola/Freescale, …
I960	I960	Vmetro, …
M32/R	32170, 32180, 32182, 32192, …	Renesas/Mitsubishi, …
M Core	MMC2113, MMC2114, …	Motorola/Freescale
MIPS32	R3K, R4K, 5K, 16, …	MTI4kx, IDT, MIPS Technologies, …
NEC	Vr55xx, Vr54xx, Vr41xx	NEC Corporation, …
PowerPC	82xx, 74xx,8xx,7xx,6xx,5xx,4xx	IBM, Motorola/Freescale, …
68k	680x0 (68K, 68030, 68040, 68060, …), 683xx	Motorola/Freescale, …
SuperH (SH)	SH3 (7702,7707, 7708,7709), SH4 (7750)	Hitachi, …
SHARC	SHARC	Analog Devices, Transtech DSP, Radstone, …
strongARM	strongARM	Intel, …
SPARC	UltraSPARC II	Sun Microsystems, …
TMS320C6xxx	TMS320C6xxx	Texas Instruments, …
x86	X86 [386,486,Pentium (II, III, IV)…]	Intel, Transmeta, National Semiconductor, Atlas, …
TriCore	TriCore1, TriCore2, …	Infineon, …

1.4 Structure of an Embedded System

A microprocessor is the center of the system, with storage devices, input and output peripherals, a power supply, human-computer interaction devices, and other necessary supporting facilities. In an actual embedded system, the hardware is generally tailormade for the application. To save cost, the peripherals may be quite compact, and only the basic peripheral circuits are retained for the processor and applications. The typical hardware structure of an embedded system is shown in Figure 1-7.

With the development of integrated circuit design and manufacturing technology, integrated circuit design has gone from transistor integration, to logic-gate integration, to the current IP integration or system on chip (SoC). The SoC design technology integrates popular circuit modules on a single chip. SoC usually contains a large number of peripheral function modules such as microprocessor/microcontroller, memory, USB controller, universal asynchronous

receiver/transmitter (UART) controller, A/D and D/A conversion, I2C, and Serial Peripheral Interface (SPI). Figure 1-8 is an example structure of SoC-based hardware for embedded systems.

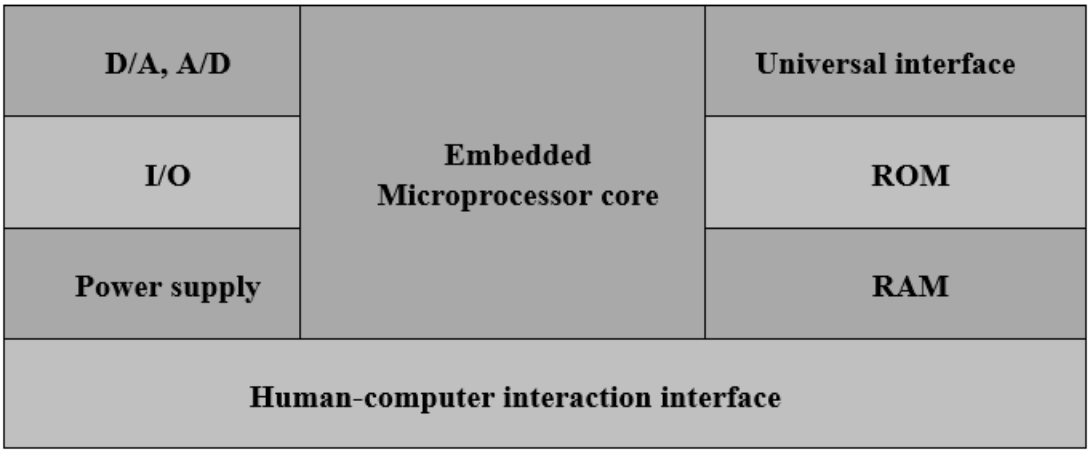


Figure 1-8. Typical hardware structure of an embedded system

1.4.1 Systems on Chip (SoC) for embedded applications

Syste on Chip architecture refers to integrated circuits (ICs) that has the complete embedded system on a single chip . It usually includes:

- Programmable processor(s)
- Memory
- Accelerating function units
- Input/output interfaces
- Software
- Re-usable intellectual property blocks (HW + SW)

A system on a programmable chip (SoPC) advocates that an electronic system be integrated onto a silicon chip with programmable logic technology. Therefore, SoPC is a special type of SoC, in that the main logic function of the entire system is achieved by a single chip. Because it is a programmable system, its functions can be changed via software. It can be said that the SoPC combines the benefits of the SoC, programmable logic device (PLD), and field-programmable gate array (FPGA).

One of the development directions of embedded system hardware is centered on SoC/SoPC, where a hardware application system through the minimum external components and connectors is built to meet the functional requirements of applications.

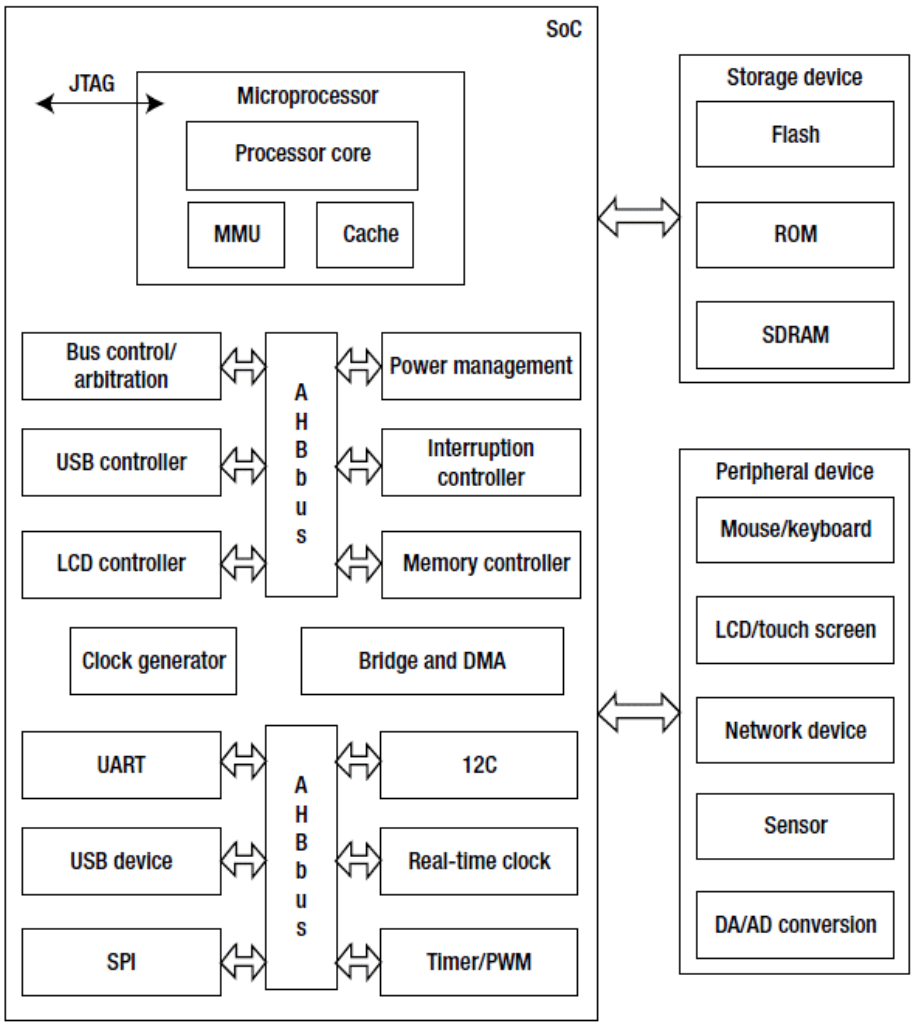


Figure 1-9. Example of an SoC-based hardware system structure

1.5 Software Architecture for Embedded Systems

Like embedded hardware, embedded software architecture is highly flexible. Simple embedded software (such as electronic toys, calculators, and so on) may be only a few thousand lines of code and perform simple input and output functions. On the other hand, complex embedded systems (such as smartphones, robots, and so on) need more complex software architecture, similar to desktop computers and servers. Simple embedded software is suitable for low-performance chip hardware, has very limited functionality, and requires tedious secondary development. Complex embedded systems provide more powerful functions, need more convenient interfaces for users, and require the support of more powerful hardware. With the improvement of hardware integration and processing capabilities, the hardware bottleneck has gradually loosened and even broken, so embedded system software now tends to be fully functional and diversified.

Typical, complete embedded system software has the architecture shown in Figure 1-10.

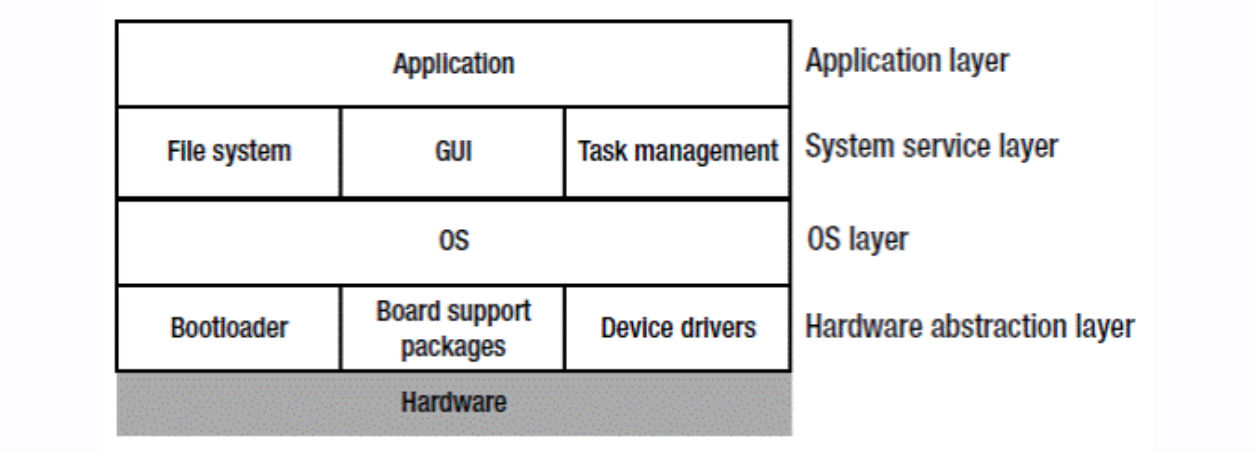


Figure 1-10 Software architecture of an embedded system

An embedded software system is composed of four layers, from bottom to top:

1. Hardware abstraction layer
2. Operating system layer
3. System service layer
4. Application layer

1.5.1 Hardware Abstraction Layer

The hardware abstraction layer (HAL), as a part of the OS, is a software abstraction layer between the embedded system hardware and OS. In general, the HAL includes the bootloader, board support package (BSP), device drivers, and other components. Similar to the BIOS in PCs, the bootloader is a program that runs before the OS kernel executes. It completes the initialization of the hardware, establishes the image of memory space, and consequently enables the hardware and software environment to reach an appropriate state for the final scheduling of the system kernel. From the perspective of end users, the bootloader is used to load the OS. The BSP achieves the abstraction of the hardware operation, empowering the OS to be independent from the hardware and enabling the OS to run on different hardware architectures.

A unique BSP must be created for each OS. For example, Wind River VxWorks BSP and Microsoft Windows CE BSP have similar functions for an embedded hardware development board, but they feature completely different architectures and interfaces. The concept of a BSP is rarely mentioned when various desktop Windows or Linux operating systems are discussed, because all PCs adopt the unified Intel architecture; the OS may be easily migrated to diversified Intel architecture-based devices without any changes. The BSP is a unique software module in embedded systems. In addition, device drivers enable the OS to shield the differences between hardware components and peripherals and provide a unified software interface for operating hardware.

Most embedded hardware requires some type of software initialization and management. The software that directly interfaces with and controls this hardware is called a device driver. All embedded systems that require software have, at the very least, device driver software in their system software layer. Device drivers are the software libraries that initialize the hardware, and manage access to the hardware by higher layers of software. Device drivers are the liaison between the hardware and the operating system, middleware, and application layers. The types of hardware components needing the support of device drivers vary from board to board.

Device drivers are typically considered either architecture-specific or generic. A device driver that is architecture-specific manages the hardware that is integrated into the master processor (the architecture). Examples of architecture-specific drivers that initialize and enable components within a master processor include on-chip memory, integrated memory managers (MMUs), and floating point hardware. A device driver that is generic manages hardware that is located on the board and not integrated onto the master processor. In a generic driver, there are typically architecture-specific portions of source code, because the master processor is the central control unit and to gain access to anything on the board usually means going through the master processor. However, the generic driver also manages board hardware that is not specific to that particular processor, which means that a generic driver can be configured to run on a variety of architectures that contain the related board hardware for which the driver is written. Generic drivers include code that initializes and manages access to the remaining major components of the board, including board buses (I2C, PCI, PCMCIA, etc.), off-chip memory (controllers, level-2+ cache, Flash, etc.), and off-chip I/O (Ethernet, RS-232, display, mouse, etc.).

A *boot loader*, also called a boot manager, is a small program that places the operating system (OS) of a computer into memory. When a computer is powered-up or restarted, the basic input/output system (BIOS) performs some initial tests, and then transfers control to the master boot record (MBR) where the boot loader resides. Most new computers are shipped with boot loaders for some version of Microsoft Windows or the Mac OS. If a computer is to be used with Linux, a special boot loader must be installed.

For Linux, the two most common boot loaders are known as LILO (LInux LOader) and LOADLIN (LOAD LINux). An alternative boot loader, called GRUB (GRand Unified Bootloader), is used with Red Hat Linux. LILO is the most popular boot loader among computer users that employ Linux as the main, or only, operating system. The primary advantage of LILO is the fact that it allows for fast boot-up. LOADLIN is preferred by some users whose computers have multiple operating systems, and who spend relatively little time in Linux. LOADLIN is sometimes used as a backup boot loader for Linux in case LILO fails. GRUB is preferred by many users of Red Hat Linux, because it is the default boot loader for that distribution.

In embedded systems, the boot-loader is a short program used to burn the firmware to the microcontroller without any programmer device either like FLASH or volatile like RAM and jumps to the desired program from there it takes care of execution. The process of burning the provided data to the program memory is controlled by the boot-loader. A boot-loader is a program

which will be the first thing to run and can load other applications into specific places in memory and is provided to the serial interface. Embedded microcontrollers offer various hardware-reset-configuration schemes.

1.5.2 Operating System Layer

An OS is a software system for uniformly managing hardware resources. It abstracts many hardware functions and provides them to applications in the form of services. Scheduling, files synchronization, and networking are the most common services provided by the OS. Operating systems are widely used in most desktop and embedded systems. In embedded systems, the OS has its own unique characteristics: stability, customization, modularity, and real-time processing.

The common embedded OS contains embedded Linux, Windows CE, VxWorks, MeeGo, Tizen, Android, Ubuntu, and some operating systems used in specific fields. Embedded Linux is a general Linux kernel tailored, customized, and modified for mobile and embedded products. Windows CE is a customizable embedded OS that Microsoft launched for a variety of embedded systems and products. VxWorks, an embedded realtime operating system (RTOS) from Wind River, supports PowerPC, 68K, CPU32, SPARC, I960, x86, ARM, and MIPS. With outstanding real-time and reliable features, it is widely used in communications, military, aerospace, aviation, and other areas that require highly sophisticated, real-time technologies. In particular, VxWorks is used in the Mars probes by NASA.

1.5.3 System Service Layer

The system service layer is the service interface that the OS provides to the application. Using this interface, applications can access various services provided by the OS. To some extent, it plays the role of a link between the OS and applications. This layer generally includes the file system, graphical user interface (GUI), task manager, and so on. A GUI library provides the application with various GUI programming interfaces, which enables the application to interact with users through application windows, menus, dialog boxes, and other graphic forms instead of a command line.

1.5.4 Application Layer

The application, located at the top level of the software hierarchy, implements the system functionality and business logic. From a functional perspective, all levels of modules in the application aim to perform system functions. From a system perspective, each application is a separate OS process. Typically, applications run in the less-privileged processor mode and use the API system schedule provided by the OS to interact with the OS.

1.6 Software Development process for embedded system

Because machine code is the only language the hardware can directly execute, all other languages need some type of mechanism to generate the corresponding machine code. This mechanism

usually includes one or some combination of *preprocessing*, *translation*, and *interpretation*. Depending on the language, these mechanisms exist on the programmer's ***host*** system (typically a non-embedded development system, such as a PC or Sparc station), or the ***target*** system (the embedded system being developed). See Figure 1.10.

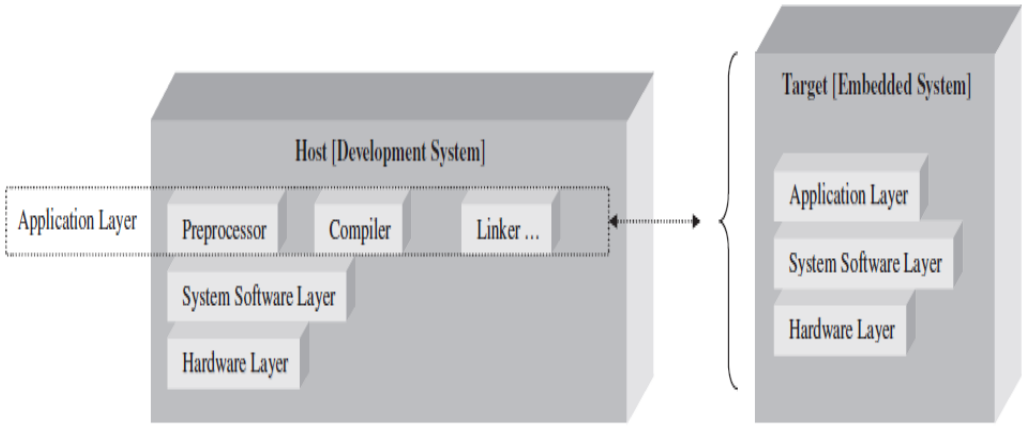


Figure 1.11: Host and target system diagram

Preprocessing is an optional step that occurs before either the translation or interpretation of source code, and whose functionality is commonly implemented by a ***preprocessor***. The preprocessor's role is to organize and restructure the source code to make translation or interpretation of this code easier.

As an example, in languages like C and C++, it is a preprocessor that allows the use of named code fragments, such as *macros*, that simplify code development by allowing the use of the macro's name in the code to replace fragments of code. The preprocessor then replaces the macro name with the contents of the macro during preprocessing. The preprocessor can exist as a separate entity, or can be integrated within the translation or interpretation unit.

1.6.1 Compiler

Many languages convert source code, either directly or after having been preprocessed through use of a ***compiler***, a program that generates a particular target language such as machine code and Java byte code from the source language as depicted in Figure 1.11. A compiler typically "translates" all of the source code to some target code at one time. As is usually the case in embedded systems, compilers are located on the programmer's host machine and generate target code for hardware platforms that differ from the platform the compiler is actually running on. These compilers are commonly referred to as ***cross-compilers***. In the case of assembly language, the compiler is simply a specialized cross-compiler referred to as an ***assembler***, and it always

generates machine code. The language name plus the term “compiler, ”such as“ Java compiler and C compiler, commonly refer to other high-level language compilers.

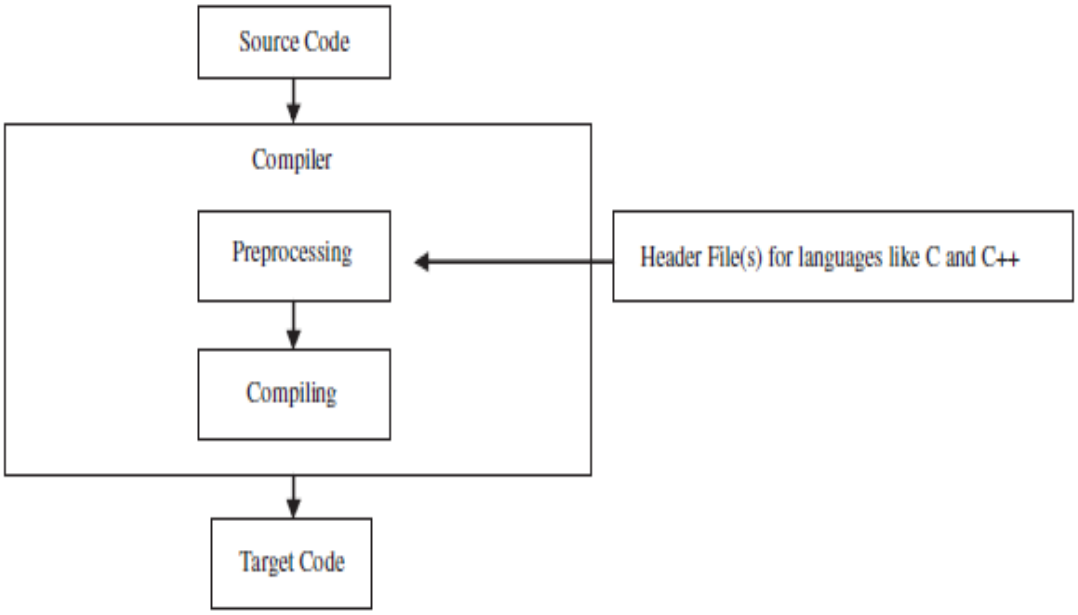


Figure 1.12 General functions of an Embedded software

High-level language compilers vary widely in terms of what is generated. Some generate machine code, while others generate other high-level code, which then requires what is produced to be run through at least one more compiler or interpreter, as discussed later in this section. Other compilers generate assembly code, which then must be run through an assembler. After all the compilation on the programmer’s host machine is completed, the remaining target code file is commonly referred to as an ***object file***, and can contain anything from machine code to Java byte code (discussed later in this section), depending on the programming language used. As shown in Figure 1.13, after linking this object file to any system libraries required, the object file, commonly referred to as an ***executable***, is then ready to be transferred to the target embedded system’s memory.

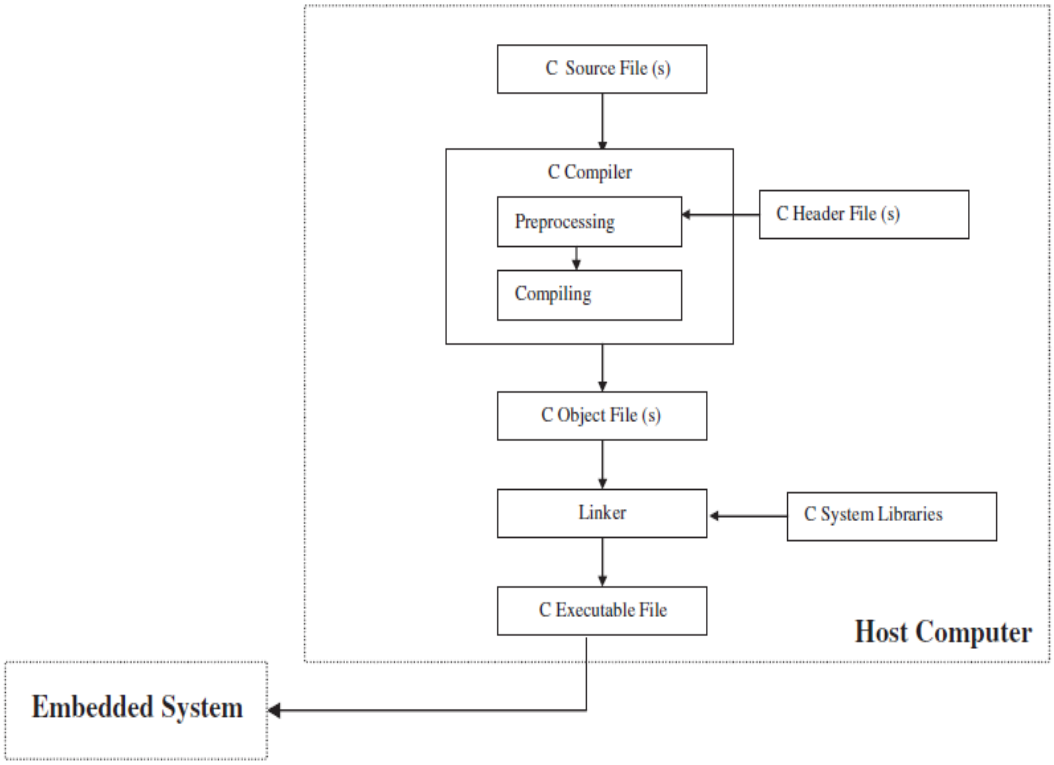


Figure 1.13: C Example compilation/linking steps and object file results

References

[1] Charles Crowley," Operating Systems: A Design-Oriented Approach", MGH, 1st Edition, 2001.

[2] Christopher Hallinan, "Embedded Linux Primer: A practical Real-World approach", Prentice Hall, 2nd Edition, 2011.

[3] Daniel P. Bovet and Marco Cesati, "Understanding the Linux Kernel, O'Reilly, 3rd Edition, 2005.

[4] John Madieu, "Linux Device Drivers Development: Develop customized drivers for embedded Linux", Packt Publishing, 1st Edition, 2017.

[5] Jonathan Corbet, Alessandro Rubini, Greg Kroah, "Linux Device Drivers", O'Reilly, 3rd Edition, 2005.

Exercise Questions

1. What do you understand by Board Support package? Explain it in detail.
2. What is special about device management to device driver in the EOS vs Firmware-stack?
3. What is Hardware Abstraction Layer (HAL)? Why it interacts with the process not the controller?
4. What are the major components of operating system? Write your views about peripheral management, device management and memory management.
5. What are two modes of operation in OS? Why in privileged mode more services are accessible as compare to normal mode?
6. What do you understand by (Board support package)BSP and (MCU support package)MSP ? What does BSP contain?
7. What do you understand by the term device-driver? Explain with suitable examples.
8. What is Firmware? Specify key differences between firmware and typical operating system.
9. Can we customize OS to hardware? If yes/no give a reason in support to your answer.
10. Why on board customization supports to 32 bit and 64 bit architectures, not on 8 bit and 16 bit MCUs. Explain.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – II - EMBEDDED OS AND DEVICE DRIVERS – SECA5204

2. OPERATING SYSTEMS OVERVIEW

An operating system is software that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. An amazing aspect of operating systems is how they vary in accomplishing these tasks in a wide variety of computing environments. Operating systems are everywhere, from cars and home appliances that include "Internet of Things" devices, to smart phones, personal computers, enterprise computers, and cloud computing environments. In order to explore the role of an operating system in a modern computing environment, it is important first to understand the organization and architecture of computer hardware. This includes the CPU, memory, and I/O devices, as well as storage. A fundamental responsibility of an operating system is to allocate these resources to programs

2.1 Role of an operating system

A computer system can be divided roughly into four components: the hardware, the operating system, the application programs, and a user (figure 2.1). The hardware— the central processing unit (CPU), the memory, and the input/output (I/O) devices—provides the basic computing resources for the system. The application programs—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve users' computing problems. **The operating system controls the hardware and coordinates its use among the various application programs for the various users.**

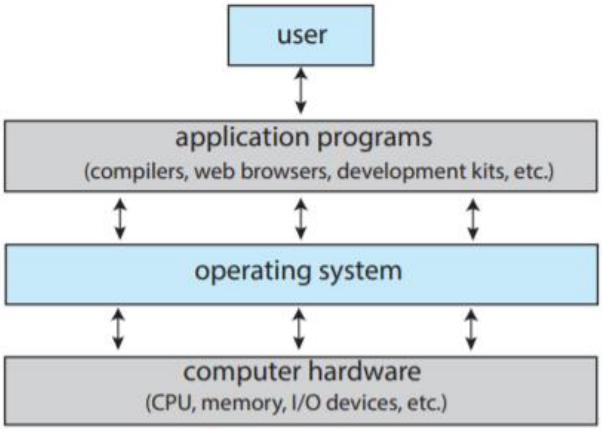


Figure 2.1 Abstract view of the components of a computer system.

For a computer to start running— for instance, when it is powered up or rebooted—it needs to have an initial program to run. As noted earlier, this initial program, or bootstrap program, tends to be simple. Typically, it is stored within the computer hardware in firmware. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to start executing that system. To accomplish this goal, the bootstrap program must locate the operating-system kernel and load it into memory. Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel by system programs that are loaded

into memory at boot time to become system daemons, which run the entire time the kernel is running. On Linux, the first system program is "systemd," and it starts many other daemons. Once this phase is complete, the system is fully booted, and the system waits for some event to occur. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt.

Another form of interrupt is a trap (or an exception), which is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed by executing a special operation called a system call.

2.1.1 Multiprogramming and Multitasking

One of the most important aspects of operating systems is the ability to run multiple programs, as a single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Furthermore, users typically want to run more than one program at a time as well. Multiprogramming increases CPU utilization, as well as keeping users satisfied, by organizing programs so that the CPU always has one to execute. In a multi-programmed system, a program in execution is termed a process. The idea is as follows: The operating system keeps several processes in memory simultaneously (figure 2.2). The operating system picks and begins to execute one of these processes. Eventually, the process may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle.

In a multi-programmed system, the operating system simply switches to, and executes, another process. When that process needs to wait, the CPU switches to another process, and so on. Eventually, the first process finishes waiting and gets the CPU back. As long as at least one process needs to execute, the CPU is never idle. Multitasking is a logical extension of multiprogramming. In multitasking systems, the CPU executes multiple processes by switching among them, but the switches occur frequently, providing the user with a fast response time.

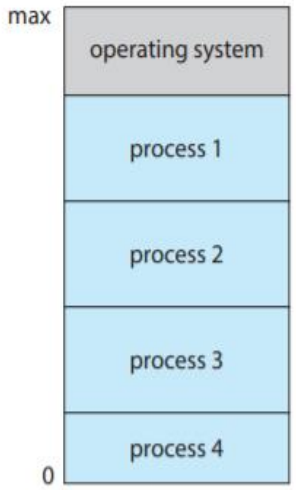


Figure 2.2 Memory layout for a multiprogramming system.

Consider that when a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or touch screen. Since interactive I/O typically runs at "people speeds," it may take a long time to complete. Input, for example, may be bounded by the user's typing speed; seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to another process.

Having several processes in memory at the same time requires some form of memory management. In addition, if several processes are ready to run at the same time, the system must choose which process will run next. Making this decision is CPU scheduling. Finally, running multiple processes concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management. Multiprogramming and multitasking systems must also provide a file system and file system resides on a secondary storage; hence, storage management must be provided. In addition, a system must protect resources from inappropriate use. To ensure orderly execution, the system must also provide mechanisms for process synchronization and communication.

2.2 Functions of an operating System

An operating system performs various functions or services that are illustrated in figure 2.3 and each function is briefly described in the following sections.

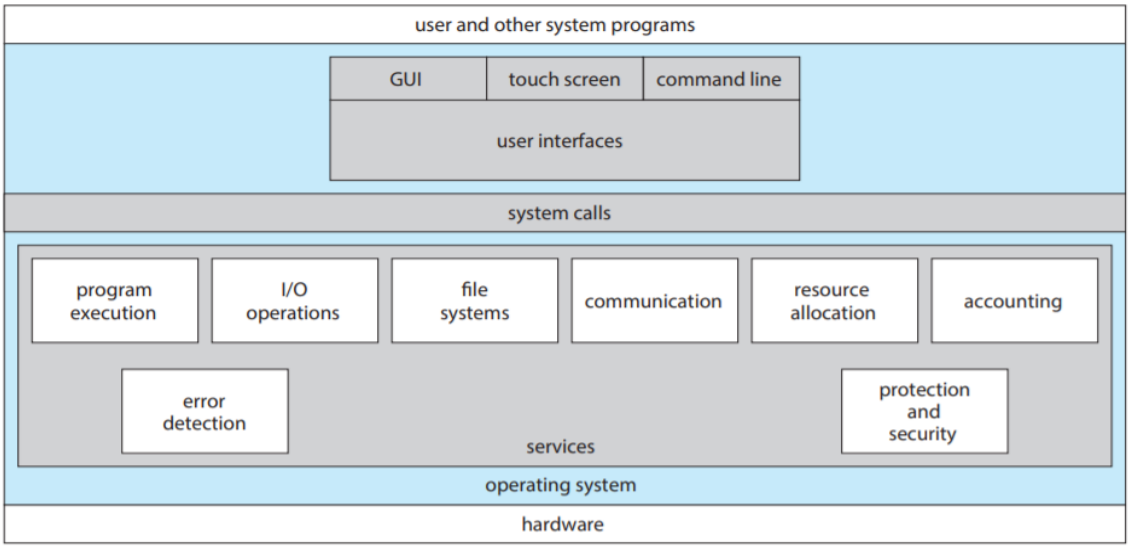


Figure 2.3 A view of operating system services.

2.2.1 User Interface

Almost all operating systems have a user interface (UI). This interface can take several forms. Most commonly, a graphical user interface (GUI) is used. Here, the interface is a window system with a mouse that serves as a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Mobile systems such as phones and tablets provide a touch-

screen interface, enabling users to slide their fingers across the screen or press buttons on the screen to select choices. Another option is a command-line interface (CLI), which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). Some systems provide two or all three of these variations.

2.2.2 Process Management

In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called **process scheduling**. An Operating System does the following activities for processor management −

- Keeps tracks of processor and status of process. The program responsible for this task is known as **traffic controller**.
- Allocates the processor (CPU) to a process.
- De-allocates processor when a process is no longer required.

2.2.3 Memory Management

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.

Main memory provides a fast storage that can be accessed directly by the CPU. For a program to be executed, it must in the main memory. An Operating System does the following activities for memory management −

- Keeps tracks of primary memory, i.e., what part of it are in use by whom, what part are not in use.
- In multiprogramming, the OS decides which process will get memory when and how much.
- Allocates the memory when a process requests it to do so.
- De-allocates the memory when a process no longer needs it or has been terminated.

2.2.4 I/O Device Management

An Operating System manages device communication via their respective drivers. It does the following activities for device management −

- Keeps tracks of all devices. Program responsible for this task is known as the **I/O controller**.
- Decides which process gets the device when and for how much time.
- Allocates the device in the efficient way.
- De-allocates devices.

2.2.5 File Management

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

An Operating System does the following activities for file management −

- Keeps track of information, location, uses, status etc. The collective facilities are often known as **file system**.
- Decides who gets the resources.
- Allocates the resources.
- De-allocates the resources.

The other activities that an Operating System performs are:

- **Security** − By means of password and similar other techniques, it prevents unauthorized access to programs and data.
- **Control over system performance** − Recording delays between request for a service and response from the system.
- **Job accounting** − Keeping track of time and resources used by various jobs and users.
- **Error detecting aids** − Production of dumps, traces, error messages, and other debugging and error detecting aids.
- **Coordination between other softwares and users** − Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

2.3 Kernel and User Mode

Since the operating system and its users share the hardware and software resources of the computer system, a properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs —or the operating system itself— to execute incorrectly. In order to ensure the proper execution of the system, we must be able to distinguish between the execution of operating-system code and user-defined code. The approach taken by most computer systems is to provide hardware support that allows differentiation among various modes of execution. At the very least, we need two separate modes of operation: user mode and kernel mode (also called supervisor mode, system mode, or privileged mode). A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request. This is shown in Figure 1.13.

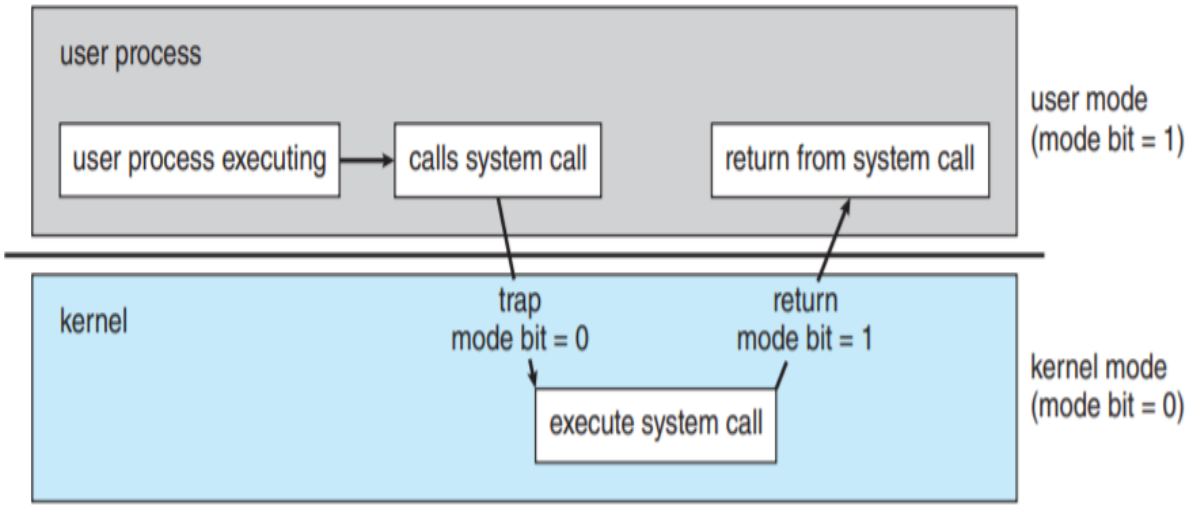


Figure 2.4 Transition from user to kernel mode.

As we shall see, this architectural enhancement is useful for many other aspects of system operation as well. At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program. The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as privileged instructions. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not

execute the instruction but rather treats it as illegal and traps it to the operating system. The instruction to switch to kernel mode is an example of a privileged instruction. Some other examples include I/O control, timer management, and interrupt management. Many additional privileged instructions are discussed throughout the text.

Initial control resides in the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call. Most contemporary operating systems—such as Microsoft Windows, Unix, and Linux— take advantage of this dual-mode feature and provide greater protection for the operating system.

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms,

it is the method used by a process to request action by the operating system. A system call usually takes the form of a trap to a specific location in the interrupt vector. This trap can be executed by a generic trap instruction, although some systems have a specific syscall instruction to invoke a system call. When a system call is executed, it is typically treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode. The system-call service routine is a part of the operating system. The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers). The kernel verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call. We describe system calls more fully in Section 2.3.

Once hardware protection is in place, it detects errors that violate modes. These errors are normally handled by the operating system. If a user program fails in some way—such as by making an attempt either to execute an illegal instruction or to access memory that is not in the user's address space— then the hardware traps to the operating system. The trap transfers control through the interrupt vector to the operating system, just as an interrupt does. When a program error occurs, the operating system must terminate the program abnormally. This situation is handled by the same code as a user-requested abnormal termination. An appropriate error message is given, and the memory of the program may be dumped. The memory dump is usually written to a file so that the user or programmer can examine it and perhaps correct it and restart the program.

2.3.1 System Calls

System calls provide an interface to the services made available by an operating system. These calls are generally available as functions written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions.

Before we discuss how an operating system makes system calls available, let's first use an example to illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file. The first input that the program will need is the names of the two files: the input file and the output file. These names can be specified in many ways, depending on the operating-system design. One approach is to pass the names of the two files as part of the command— for example, the UNIX cp command:

```
cp in.txt out.txt
```

This command copies the input file in.txt to the output file out.txt. A second approach is for the program to ask the user for the names. In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files. On mouse-based and icon-based systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified. This sequence requires many I/O system calls.

Once the two file names have been obtained, the program must open the input file and create and open the output file. Each of these operations requires another system call. Possible error conditions for each system call must be handled. For example, when the program tries to open the input file, it may find that there is no file of that name or that the file is protected against access. In these cases, the program should output an error message (another sequence of system calls) and then terminate abnormally (another system call). If the input file exists, then we must create a new output file. We may find that there is already an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (yet another system call). Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the terminal) whether to replace the existing file or to abort the program.

When both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions. On input, the program may find that the end of the file has been reached or that there was a hardware failure in the read (such as a parity error). The write operation may encounter various errors, depending on the output device (for example, no more available disk space). Finally, after the entire file is copied, the program may close both files (two system calls), write a message to the console or window (more system calls), and finally terminate normally (the final system call). This system-call sequence is shown in Figure 2.5.

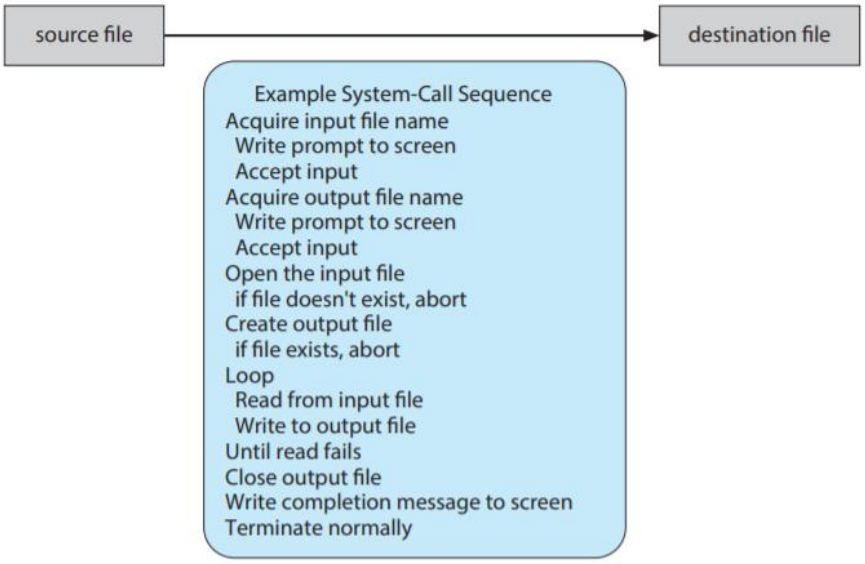


Figure 2.5 Example of how system calls are used.

2.3.2 Application Programming Interface

As you can see, even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second. Most programmers never see this level of detail, however. Typically, application developers design programs according to an application programming interface (API). The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. Three of the most common APIs available to application programmers are the Windows API for Windows systems, the POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux, and macOS), and the Java API for programs that run on the Java virtual machine. A programmer accesses an API via a library of code provided by the operating system. In the case of UNIX and Linux for programs written in the C language, the library is called libc. Note that—unless specified — the system-call names used throughout this text are generic examples. Each operating system has its own name for each system call. Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer. For example, the Windows function CreateProcess() (which, unsurprisingly, is used to create a new process) actually invokes the NTCreateProcess() system call in the Windows kernel.

As an example of a standard API, consider the read() function that is available in UNIX and Linux systems. The API for this function is obtained from the man page by invoking the command man read on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, read() returns −1.

Why would an application programmer prefer programming according to an API rather than invoking actual system calls? There are several reasons for doing so. One benefit concerns

program portability. An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API (although, in reality, architectural differences often make this more difficult than it may appear). Furthermore, actual system calls can often be more detailed and difficult to work with than the API available to an application programmer. Nevertheless, there often exists a strong correlation between a function in the API and its associated system call within the kernel. In fact, many of the POSIX and Windows APIs are similar to the native system calls provided by the UNIX, Linux, and Windows operating systems.

Another important factor in handling system calls is the run-time environment (RTE)— the full suite of software needed to execute applications written in a given programming language, including its compilers or interpreters as well as other software, such as libraries and loaders. The RTE provides a system-call interface that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system. Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers. The system call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call.

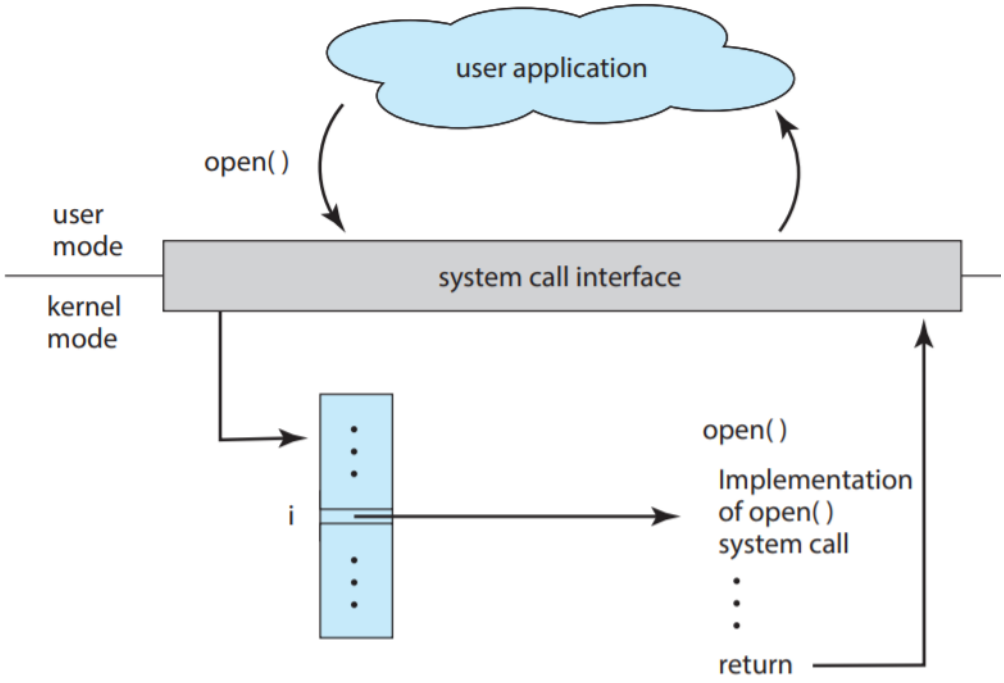


Figure 2.6 The handling of a user application invoking the open() system call

Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the RTE. The relationship among an API, the system-call interface, and the operating system is shown in Figure 2.6, which illustrates how the operating system handles a user application invoking the open() system call.

2.3.3 Types of System Calls

System calls can be grouped roughly into six major categories: process control, fil management, device management, information maintenance, communications, and protection. The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communications	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Figure 2.7 Examples of Windows and Linux system calls

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the printf() statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the write() system call. The C library takes the value returned by write() and passes it back to the user program as illustrated below:

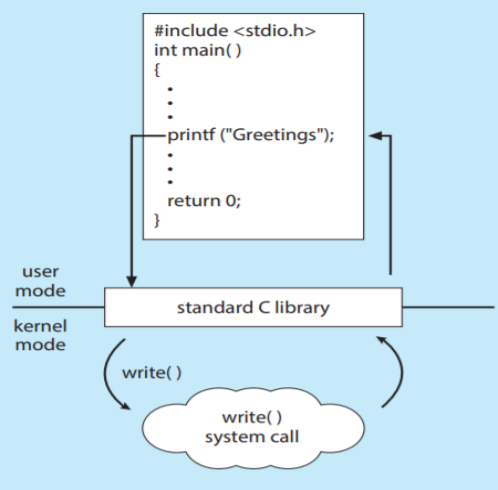


Figure 2.8 The standard C library - system call example

2.4 Operating system interface

User and Operating-System Interface We mentioned earlier that there are several ways for users to interface with the operating system. Here, we discuss three fundamental approaches. One provides a command-line interface, or command interpreter, that allows users to directly enter commands to be performed by the operating system. The other two allow users to interface with the operating system via a graphical user interface, or GUI.

Command Interpreters

Most operating systems, including Linux, UNIX, and Windows, treat the command interpreter as a special program that is running when a process is initiated or when a user first logs on (on interactive systems). On systems with multiple command interpreters to choose from, the interpreters are known as shells. For example, on UNIX and Linux systems, a user may choose among several different shells, including the C shell, Bourne-Again shell, Korn shell, and others. Third-party shells and free user-written shells are also available. Most shells provide similar functionality, and a user's choice of which shell to use is generally based on personal preference. Figure 2.2 shows the Bourne-Again (or bash) shell command interpreter being used on macOS

The main function of the command interpreter is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. The various shells available on UNIX systems operate in this way.

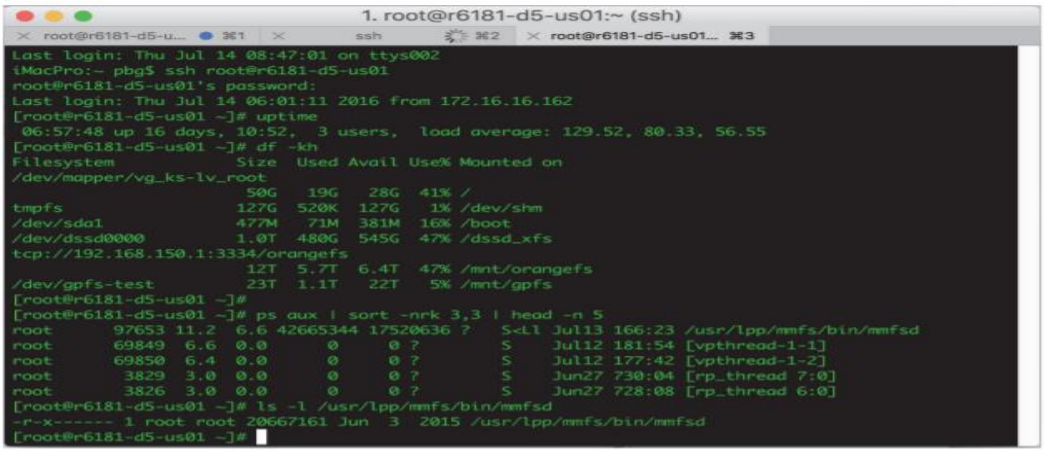


Figure 2.2 The bash shell command interpreter in macOS

Graphical User Interface

A second strategy for interfacing with the operating system is through a userfriendly graphical user interface, or GUI. Here, rather than entering commands directly via a command-line interface, users employ a mouse-based windowand-menu system characterized by a desktop metaphor. The user moves the mouse to position its pointer on images, or icons, on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory—known as a folder—or pull down a menu that contains commands.

Traditionally, UNIX systems have been dominated by command-line interfaces. Various GUI interfaces are available, however, with significant development in GUI designs from various open-source projects, such as K Desktop Environment (or KDE) and the GNOME desktop by the GNU project. Both the KDE and GNOME desktops run on Linux and various UNIX systems and are available under open-source licenses, which means their source code is readily available for reading and for modification under specific license terms.

2.5 Linkers and Loaders

Usually, a program resides on disk as a binary executable file— for example, a.out or prog.exe. To run on a CPU, the program must be brought into memory and placed in the context of a process. In this section, we describe the steps in this procedure, from compiling a program to placing it in memory, where it becomes eligible to run on an available CPU core. The steps are highlighted in Figure 2.11. Source files are compiled into object files that are designed to be loaded into any physical memory location, a format known as an relocatable object fil . Next, the linker combines these relocatable object files into a single binary executable file. During the linking phase, other object files or libraries may be included as well, such as the standard C or math library (specified with the flag -lm). A loader is used to load the binary executable file into memory, where it is eligible to run on a CPU core. An activity associated with linking and loading is relocation, which assigns final addresses to the program parts and adjusts code and data in the program to match those addresses so that, for example, the code can call library functions and access its variables as it executes. In Figure 2.11, we see that to run the loader, all that is necessary is to enter the name of the executable file on the command line. When a program name is entered on the

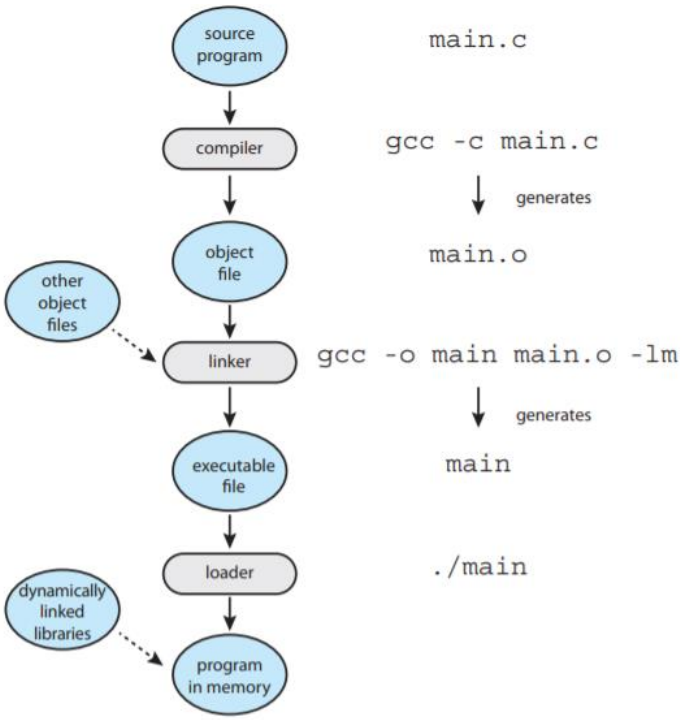


Figure 2.9 The role of the linker and loader

command line on UNIX systems— for example, ./main— the shell first creates a new process to run the program using the fork() system call. The shell then invokes the loader with the exec() system call, passing exec() the name of the executable file. The loader then loads the specified program into memory using the address space of the newly created process. (When a GUI interface is used, double-clicking on the icon associated with the executable file invokes the loader using a similar mechanism.) The process described thus far assumes that all libraries are linked into the executable file and loaded into memory. In reality, most systems allow a program to dynamically link libraries as the program is loaded. Windows, for instance, supports dynamically linked libraries (DLLs). The benefit of this approach is that it avoids linking and loading libraries that may end up not being used into an executable file.

Instead, the library is conditionally linked and is loaded if it is required during program run time. For example, in Figure 2.11, the math library is not linked into the executable file main. Rather, the linker inserts relocation information that allows it to be dynamically linked and loaded as the program is loaded. We shall see in Chapter 9 that it is possible for multiple processes to share dynamically linked libraries, resulting in a significant savings in memory use. Object files and executable files typically have standard formats that include the compiled machine code and a symbol table containing metadata about functions and variables that are referenced in the program. For UNIX and Linux systems, this standard format is known as ELF (for Executable and Linkable Format). There are separate ELF formats for relocatable and executable files. One piece of information in the ELF file for executable files is the program's entry point, which contains the address of the first instruction to be executed when the program runs. Windows systems use the Portable Executable (PE) format, and macOS uses the Mach-O format.

2.6 Types of operating systems

All of this history and development has left us with a wide variety of operating systems, not all of which are widely known. In this section we will briefly touch upon seven of them. We will come back to some of these different kinds of systems later in the book.

Mainframe Operating Systems

At the high end are the operating systems for the mainframes, those room-sized computers still found in major corporate data centers. These computers distinguish themselves from personal computers in terms of their I/O capacity. A mainframe with 1000 disks and thousands of gigabytes of data is not unusual: a personal computer with these specifications would be odd indeed. Mainframes are also making something of a comeback as high-end Web servers, servers for large-scale electronic commerce sites, and servers for business-to-business transactions. The operating systems for mainframes are heavily oriented toward processing many jobs at once, most of which need prodigious amounts of I/O. They typically offer three kinds of services: batch, transaction processing, and timesharing. A batch system is one that processes routine jobs without any interactive user present. Claims processing in an insurance company or sales reporting for a chain of stores is typically done in batch mode. Transaction processing systems handle large numbers of small requests, for example, check processing at a bank or airline reservations. Each unit of work is small, but the system must handle hundreds or thousands per second. Timesharing systems allow

multiple remote users to run jobs on the computer at once, such as querying a big database. These functions are closely related: mainframe operating systems often perform all of them. An example mainframe operating system is OS/390, a descendant of OS/360.

Server Operating Systems

One level down are the server operating systems. They run on servers, which are either very large personal computers, workstations, or even mainframes. They serve multiple users at once over a network and allow the users to share hardware and software resources. Servers can provide print service, file service, or Web service. Internet providers run many server machines to support their customers and Web sites use servers to store the Web pages and handle the incoming requests. Typical server operating systems are UNIX and Windows 2000. Linux is also gaining ground for servers.

Multiprocessor Operating Systems

An increasingly common way to get major-league computing power is to connect multiple CPUs into a single system. Depending on precisely how they are connected and what is shared, these systems are called parallel computers, multicomputers, or multiprocessors. They need special operating systems, but often these are variations on the server operating systems, with special features for communication and connectivity.

Personal Computer Operating Systems

The next category is the personal computer operating system. Their job is to provide a good interface to a single user. They are widely used for word processing, spreadsheets, and Internet access. Common examples are Windows 98, Windows 2000, the Macintosh operating system, and Linux. Personal computer operating systems are so widely known that probably little introduction is needed. In fact, many people are not even aware that other kinds exist.

Real-Time Operating Systems

Another type of operating system is the real-time system. These systems are characterized by having time as a key parameter. For example, in industrial process control systems, real-time computers have to collect data about the production process and use it to control machines in the factory. Often there are hard deadlines that must be met. For example, if a car is moving down an assembly line, certain actions must take place at certain instants of time, if a welding robot welds too early or too late, the car will be ruined. If the action absolutely must occur at a certain moment (or within a certain range), we have a hard real-time system. Another kind of real-time system is a soft real-time system, in which missing an occasional deadline is acceptable. Digital audio or multimedia systems fall in this category. VxWorks and QNX are well-known real-time operating systems.

Embedded Operating System

Continuing on down to smaller and smaller systems, we come to palmtop computers and embedded systems. A palmtop computer or PDA (Personal Digital Assistant) is a small computer that fits in a shirt pocket and performs a small number of functions such as an electronic address

book and memo pad. Embedded systems run on the computers that control devices that are not generally thought of as computers, such as TV sets, microwave ovens, and mobile telephones. These often have some characteristics of realtime systems but also have size, memory, and power restrictions that make them special. Examples of such operating systems are PalmOS and Windows CE (Consumer Electronics).

Smart Card Operating Systems

The smallest operating systems run on smart cards, which are credit card-sized devices containing a CPU chip. They have very severe processing power and memory constraints. Some of them can handle only a single function, such as electronic payments, but others can handle multiple functions on the same smart card. Often these are proprietary systems. Some smart cards are Java oriented. What this means is that the ROM on the smart card holds an interpreter for the Java Virtual Machine (JVM). Java applets (small programs) are downloaded to the card and are interpreted by the JVM interpreter. Some of these cards can handle multiple Java applets at the same time, leading to multiprogramming and the need to schedule them. Resource management and protection also become an issue when two or more applets are present at the same time. These issues must be handled by the (usually extremely primitive) operating system present on the card.

2.7 Operating-System Structure

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components, or modules, rather than have one single system. Each of these modules should be a well-defined portion of the system, with carefully defined interfaces and functions. In this section, we discuss how these components are interconnected and melded into a kernel.

Monolithic Structure

The simplest structure for organizing an operating system is no structure at all. That is, place all of the functionality of the kernel into a single, static binary file that runs in a single address space. This approach—known as a monolithic structure—is a common technique for designing operating systems. An example of such limited structuring is the original UNIX operating system, which consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. We can view the traditional UNIX operating system as being layered to some extent, as shown in Figure 2.10.

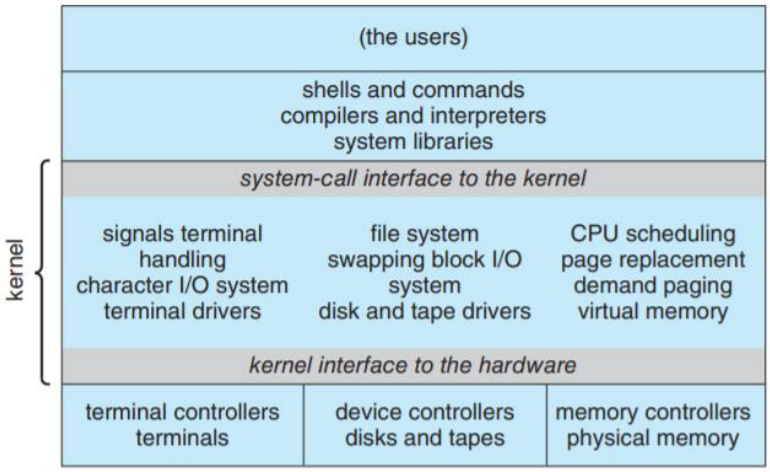


Figure 2.10 Traditional UNIX system structure.

Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operatingsystem functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into one single address space. The Linux operating system is based on UNIX and is structured similarly, as shown in Figure 2.10. Applications typically use the glibc standard C library when communicating with the system call interface to the kernel.

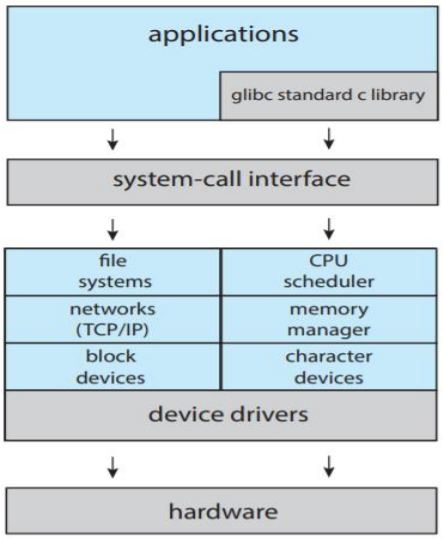


Figure 2.11 Linux system structure.

The Linux kernel is monolithic in that it runs entirely in kernel mode in a single address space, but it does have a modular design that allows the kernel to be modified during run time. Despite the apparent simplicity of monolithic kernels, they are difficult to implement and extend. Monolithic kernels do have a distinct performance advantage, however: there is very little overhead in the system-call interface, and communication within the kernel is fast. Therefore,

despite the drawbacks of monolithic kernels, their speed and efficiency explains why we still see evidence of this structure in the UNIX, Linux, and Windows operating systems.

Layered Approach

The monolithic approach is often known as a tightly coupled system because changes to one part of the system can have wide-ranging effects on other parts. Alternatively, we could design a loosely coupled system. Such a system is divided into separate, smaller components that have specific and limited functionality. All these components together comprise the kernel. The advantage of this modular approach is that changes in one component affect only that component, and no others, allowing system implementers more freedom in creating and changing the inner workings of the system.

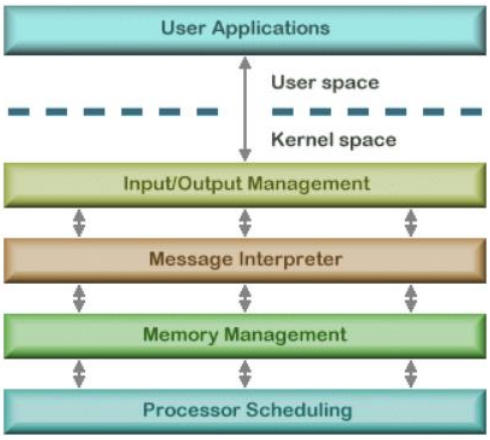


Figure 2.12 A layered OS architecture

As operating systems became larger and more complex, the monolithic approach was largely abandoned in favour of a modular approach which grouped components with similar functionality into layers to help operating system designers to manage the complexity of the system. In this kind of architecture, each layer communicates only with the layers immediately above and below it, and lower-level layers provide services to higher-level ones using an interface that hides their implementation.

The modularity of layered operating systems shown in figure 2.12 allows the implementation of each layer to be modified without requiring any modification to adjacent layers. Although this modular approach imposes structure and consistency on the operating system, simplifying debugging and modification, a service request from a user process may pass through many layers of system software before it is serviced and performance compares unfavourably to that of a monolithic kernel. Also, because all layers still have unrestricted access to the system, the

kernel is still susceptible to errant or malicious code. Many of today's operating systems, including Microsoft Windows and Linux, implement some level of layering.

Microkernels

This method structures the operating system by removing all nonessential components from the kernel and implementing them as userlevel programs that reside in separate address spaces. The result is a smaller kernel. There is little consensus regarding which services should remain in the kernel and which should be implemented in user space. Typically, however, microkernels provide minimal process and memory management, in addition to a communication facility. Figure 2.15 illustrates the architecture of a typical microkernel. The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space. For example, if the client program wishes to access a file, it must interact with the file server.

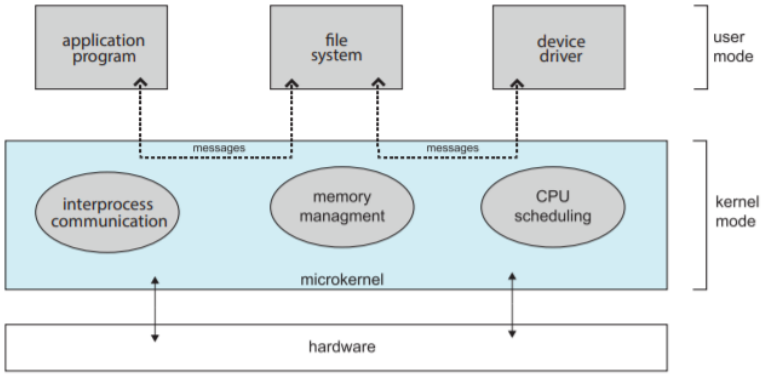


Figure 2.13 Architecture of a typical microkernel.

The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel. One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another. The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.

Perhaps the best-known illustration of a microkernel operating system is Darwin, the kernel component of the macOS and iOS operating systems. Another example is QNX, a real-time operating system for embedded systems. The QNX Neutrino microkernel provides services for message passing and process scheduling. It also handles low-level network communication and hardware interrupts. All other services in QNX are provided by standard processes that run outside the kernel in user mode.

Unfortunately, the performance of microkernels can suffer due to increased system-function overhead. When two user-level services must communicate, messages must be copied between the services, which reside in separate address spaces. In addition, the operating system may have to switch from one process to the next to exchange the messages. The overhead involved in copying messages and switching between processes has been the largest impediment to the growth of microkernel-based operating systems.

Modules

Perhaps the best current methodology for operating-system design involves using loadable kernel modules (LKMs). Here, the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time. This type of design is common in modern implementations of UNIX, such as Linux, macOS, and Solaris, as well as Windows. The idea of the design is for the kernel to provide core services, while other services are implemented dynamically, as the kernel is running. Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made. Thus, for example, we might build CPU scheduling and memory management algorithms directly into the kernel and then add support for different file systems by way of loadable modules. The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system, because any module can call any other module.

The approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate. Linux uses loadable kernel modules, primarily for supporting device drivers and file systems. LKMs can be "inserted" into the kernel as the system is started (or booted) or during run time, such as when a USB device is plugged into a running machine. If the Linux kernel does not have the necessary driver, it can be dynamically loaded. LKMs can be removed from the kernel during run time as well. For Linux, LKMs allow a dynamic and modular kernel, while maintaining the performance benefits of a monolithic system.

2.8 Process Management

A program can do nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process. A program such as a compiler is a process, and a word-processing program being run by an individual user on a PC is a process. Similarly, a social media app on a mobile device is a process.

It is possible to provide system calls that allow processes to create subprocesses to execute concurrently. A process needs certain resources—including CPU time, memory, files, and I/O devices— to accomplish its task. These resources are typically allocated to the process while it is running. In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (input) may be passed along. For example, consider a process

running a web browser whose function is to display the contents of a web page on a screen. The process will be given the URL as an input and will execute the appropriate instructions and system calls to obtain and display the desired information on the screen. When the process terminates, the operating system will reclaim any reusable resources. A program is a passive entity, like the contents of a file stored on disk, whereas a process is an active entity. A single-threaded process has one program counter specifying the next instruction to execute.

There are so many facets of and variations in process control that we next use two examples—one involving a single-tasking system and the other a multitasking system— to clarify these concepts. The Arduino is a simple hardware platform consisting of a microcontroller along with input sensors that respond to a variety of events, such as changes to light, temperature, and barometric pressure, to just name a few. To write a program for the Arduino, we first write the program on a PC and then upload the compiled program (known as a sketch) from the PC to the Arduino's flash memory via a USB connection. The standard Arduino platform does not provide an operating system; instead, a small piece of software known as a boot loaderloads the sketch into a specific region in the Arduino's memory (Figure 2.9).

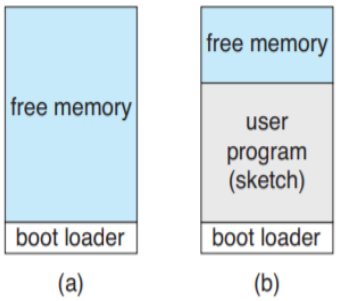


Figure 2.14 Arduino execution. (a) At system startup. (b) Running a sketch.

Once the sketch has been loaded, it begins running, waiting for the events that it is programmed to respond to. For example, if the Arduino's temperature sensor detects that the temperature has exceeded a certain threshold, the sketch may have the Arduino start the motor for a fan. An Arduino is considered a single-tasking system, as only one sketch can be present in memory at a time; if another sketch is loaded, it replaces the existing sketch. Furthermore, the Arduino provides no user interface beyond hardware input sensors. FreeBSD (derived from Berkeley UNIX) is an example of a multitasking system. When a user logs on to the system, the shell of the user's choice is run, awaiting commands and running programs the user requests. However, since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed (Figure 2.10).

To start a new process, the shell executes a fork() system call. Then, the selected program is loaded into memory via an exec() system call, and the program is executed. Depending on how the command was issued, the shell then either waits for the process to finish or runs the process "in the background." In the latter case, the shell immediately waits for another command to be entered. When a process is running in the background, it cannot receive input directly from the keyboard, because the shell is using this resource. I/O is therefore done through files or through a

GUI interface. Meanwhile, the user is free to ask the shell to run other programs, to monitor the progress of the running process, to change that program's priority, and so on. When the process is done, it executes an exit() system call to terminate, returning to the invoking process a status code of 0 or a nonzero error code. This status or error code is then available to the shell or other programs.

The status of the current activity of a process is represented by the value of the program counter and the contents of the processor's registers. The memory layout of a process is typically divided into multiple sections, and is shown in Figure 3.1.

These sections include:

• Text section— the executable code

• Data section—global variables

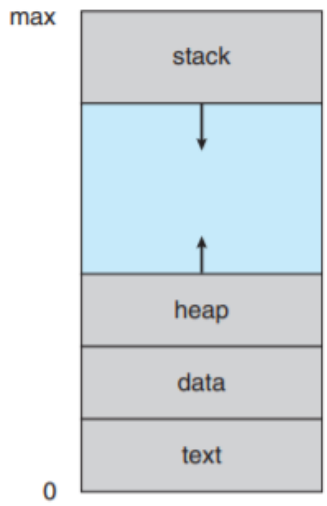


Figure 2.15 Layout of a process in memory

• Heap section—memory that is dynamically allocated during program run time

• Stack section— temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

2.8.1 Process State

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

- New. The process is being created.

• Running. Instructions are being executed.

• Waiting. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

• Ready. The process is waiting to be assigned to a processor.

• Terminated. The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be running on any processor core at any instant. Many processes may be ready and waiting, however. The state diagram corresponding to these states is presented in Figure 3.2.

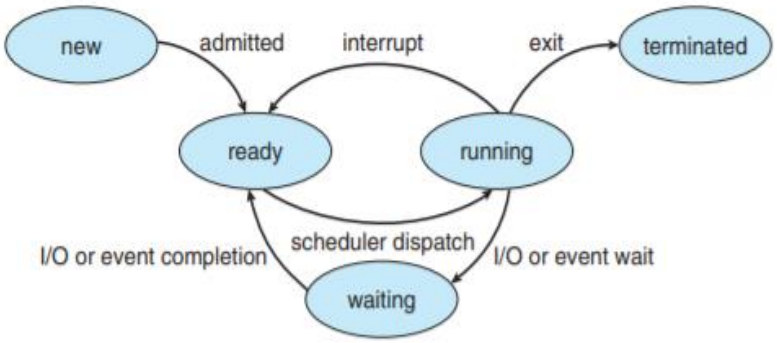


Figure 2.16 Diagram of process state.

2.8.2 Process Control Block

Each process is represented in the operating system by a process control block (PCB)—also called a task control block. A PCB is shown in Figure 3.3. It contains many pieces of information associated with a specific process, including these:

• Process state. The state may be new, ready, running, waiting, halted, and so on.

• Program counter. The counter indicates the address of the next instruction to be executed for this process.

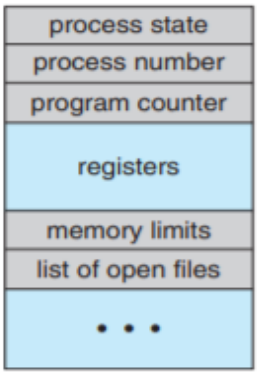


Figure 2.17 Process control block (PCB).

• CPU registers. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.

• CPU-scheduling information. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Chapter 5 describes process scheduling.)

• Memory-management information. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system .

• Accounting information. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

• I/O status information. This information includes the list of I/O devices allocated to the process, a list of open files, and so on. In brief, the PCB simply serves as the repository for all the data needed to start, or restart, a process, along with some accounting data.

2.8.3 Process creation and Termination

During the course of execution, a process may create several new processes. As mentioned earlier, the creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes. Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique process identifie (or pid), which is typically an integer number. The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.

In addition to supplying various physical and logical resources, the parent process may pass along initialization data (input) to the child process. For example, consider a process whose function is to display the contents of a file— say, hw1.c—on the screen of a terminal. When the process is created, it will get, as an input from its parent process, the name of the file hw1.c. Using that file name, it will open the file and write the contents out. It may also get the name of the output device. Alternatively, some operating systems pass resources to child processes. On such a system, the new process may get two open files, hw1.c and the terminal device, and may simply transfer the datum between the two. When a process creates a new process, two possibilities for execution exist: 1. The parent continues to execute concurrently with its children. 2. The parent waits until

some or all of its children have terminated. There are also two address-space possibilities for the new process:

- The child process is a duplicate of the parent process (it has the same program and data as the parent).
- The child process has a new program loaded into it.

To illustrate these differences, let's first consider the UNIX operating system. In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer. A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
   }

   return 0;
}
```

Figure 2.18 Creating a separate process using the UNIX fork() system call

As an alternative example, we next consider process creation in Windows. Processes are created in the Windows API using the CreateProcess() function, which is similar to fork() in that a parent creates a new child process. However, whereas fork() has the child process inheriting the address space of its parent, CreateProcess() requires loading a specified program into the address space of the child process at process creation. Furthermore, whereas fork() is passed no parameters, CreateProcess() expects no fewer than ten parameters.

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

   /* allocate memory */
   ZeroMemory(&si, sizeof(si));
   si.cb = sizeof(si);
   ZeroMemory(&pi, sizeof(pi));

   /* create child process */
   if (!CreateProcess(NULL, /* use command line */
    "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
    NULL, /* don't inherit process handle */
    NULL, /* don't inherit thread handle */
    FALSE, /* disable handle inheritance */
    0, /* no creation flags */
    NULL, /* use parent's environment block */
    NULL, /* use parent's existing directory */
    &si,
    &pi))
   {
      fprintf(stderr, "Create Process Failed");
      return -1;
   }
   /* parent will wait for the child to complete */
   WaitForSingleObject(pi.hProcess, INFINITE);
   printf("Child Complete");

   /* close handles */
   CloseHandle(pi.hProcess);
   CloseHandle(pi.hThread);
}
```

Figure 2.19 Creating a separate process using the Windows API.

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call. At that point, the process may return a status value (typically an integer) to its waiting parent process (via the wait() system call). All the resources of the process —including physical and virtual memory, open files, and I/O buffers— are deallocated and reclaimed by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, TerminateProcess() in Windows). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, a user— or a misbehaving application—could arbitrarily kill another user's processes. Note that a parent needs to know the identities of its children if it is to terminate them. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

2.8.4 Threads

The process model discussed so far has implied that a process is a program that performs a single thread of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time. Thus, the user cannot simultaneously type in characters and run

the spell checker. Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. This feature is especially beneficial on multicore systems, where multiple threads can run in parallel. A multithreaded word processor could, for example, assign one thread to manage user input while another thread runs the spell checker. On systems that support threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads.

2.9 Process Scheduling

The objective of multiprogramming is to have some process running at all times so as to maximize CPU utilization. The objective of time sharing is to switch a CPU core among processes so frequently that users can interact with each program while it is running. To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on a core. Each CPU core can run one process at a time. A process migrates among the ready queue and various wait queues throughout its lifetime. The role of the CPU scheduler is to select from among the processes that are in the ready queue and allocate a CPU core to one of them. The CPU scheduler must select a new process for the CPU frequently. An I/O-bound process may execute for only a few milliseconds before waiting for an I/O request. Although a CPU-bound process will require a CPU core for longer durations, the scheduler is unlikely to grant the core to a process for an extended period. Instead, it is likely designed to forcibly remove the CPU from a process and schedule another process to run. Therefore, the CPU scheduler executes at least once every 100 milliseconds, although typically much more frequently.

Basic Concepts of scheduling

- Almost all programs have some alternating cycle of CPU number crunching and waiting for I/O of some kind. (Even a simple fetch from memory takes a long time relative to CPU speeds.)
- In a simple system running a single process, the time spent waiting for I/O is wasted, and those CPU cycles are lost forever.
- A scheduling system allows one process to use the CPU while another is waiting for I/O, thereby making full use of otherwise lost CPU cycles.
- The challenge is to make the overall system as "efficient" and "fair" as possible, subject to varying and often dynamic conditions, and where "efficient" and "fair" are somewhat subjective terms, often subject to shifting priority policies.

CPU scheduler

- Whenever the CPU becomes idle, it is the job of the CPU Scheduler (a.k.a. the short-term scheduler) to select another process from the ready queue to run next.
- The storage structure for the ready queue and the algorithm used to select the next process are not necessarily a FIFO queue. There are several alternatives to choose from, as well as numerous adjustable parameters for each algorithm, which is the basic subject of this entire chapter.

Preemptive Scheduling

- CPU scheduling decisions take place under one of four conditions:
    1. When a process switches from the running state to the waiting state, such as for an I/O request or invocation of the wait( ) system call.
    2. When a process switches from the running state to the ready state, for example in response to an interrupt.
    3. When a process switches from the waiting state to the ready state, say at completion of I/O or a return from wait( ).
    4. When a process terminates.
- For conditions 1 and 4 there is no choice - A new process must be selected.
- For conditions 2 and 3 there is a choice - To either continue running the current process, or select a different one.
- If scheduling takes place only under conditions 1 and 4, the system is said to be ***non-preemptive***, or ***cooperative***. Under these conditions, once a process starts running it keeps running, until it either voluntarily blocks or until it finishes. Otherwise the system is said to be ***preemptive.***
- Windows used non-preemptive scheduling up to Windows 3.x, and started using pre-emptive scheduling with Win95. Macs used non-preemptive prior to OSX, and pre-emptive since then. Note that pre-emptive scheduling is only possible on hardware that supports a timer interrupt.
- Note that pre-emptive scheduling can cause problems when two processes share data, because one process may get interrupted in the middle of updating shared data structures. Chapter 6 will examine this issue in greater detail.
- Preemption can also be a problem if the kernel is busy implementing a system call (e.g. updating critical kernel data structures) when the preemption occurs. Most modern UNIXes deal with this problem by making the process wait until the system call has either completed or blocked before allowing the preemption Unfortunately this solution is problematic for real-time systems, as real-time response can no longer be guaranteed.
- Some critical sections of code protect themselves from concurrency problems by disabling interrupts before entering the critical section and re-enabling interrupts on exiting the section. Needless to say, this should only be done in rare situations, and only on very short pieces of code that will finish quickly, (usually just a few machine instructions.)

Dispatcher

- The **dispatcher** is the module that gives control of the CPU to the process selected by the scheduler. This function involves:
    - Switching context.
    - Switching to user mode.
    - Jumping to the proper location in the newly loaded program.
- The dispatcher needs to be as fast as possible, as it is run on every context switch. The time consumed by the dispatcher is known as **dispatch latency.**

Scheduling Criteria

- There are several different criteria to consider when trying to select the "best" scheduling algorithm for a particular situation and environment, including:
    - **CPU utilization** - Ideally the CPU would be busy 100% of the time, so as to waste 0 CPU cycles. On a real system CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)
    - **Throughput** - Number of processes completed per unit time. May range from 10 / second to 1 / hour depending on the specific processes.
    - **Turnaround time** - Time required for a particular process to complete, from submission time to completion. (Wall clock time.)
    - **Waiting time** - How much time processes spend in the ready queue waiting their turn to get on the CPU.
        - (**Load average** - The average number of processes sitting in the ready queue waiting their turn to get into the CPU. Reported in 1-minute, 5-minute, and 15-minute averages by "uptime" and "who".)
    - **Response time** - The time taken in an interactive program from the issuance of a command to the ***commence*** of a response to that command.
- In general one wants to optimize the average value of a criteria (Maximize CPU utilization and throughput, and minimize all the others.) However some times one wants to do something different, such as to minimize the maximum response time.
- Sometimes it is most desirable to minimize the ***variance*** of a criteria than the actual value. I.e. users are more accepting of a consistent predictable system than an inconsistent one, even if it is a little bit slower.

Scheduling Algorithms

The following subsections will explain several common scheduling strategies, looking at only a single CPU burst each for a small number of processes. Obviously real systems have to deal with a lot more simultaneous processes executing their CPU-I/O burst cycles.

First-Come First-Serve Scheduling, FCFS

- FCFS is very simple - Just a FIFO queue, like customers waiting in line at the bank or the post office or at a copying machine.
- Unfortunately, however, FCFS can yield some very long average wait times, particularly if the first process to get there takes a long time. For example, consider the following three processes:
    - In the first Gantt chart below, process P1 arrives first. The average waiting time for the three processes is (0 + 24 + 27) / 3 = 17.0 ms.
    - In the second Gantt chart below, the same three processes have an average wait time of (0 + 3 + 6) / 3 = 3.0 ms. The total run time for the three bursts is the same, but in the second case two of the three finish much quicker, and the other process is only delayed by a short amount.

Process	Burst Time
P1	24
P2	3
P3	3

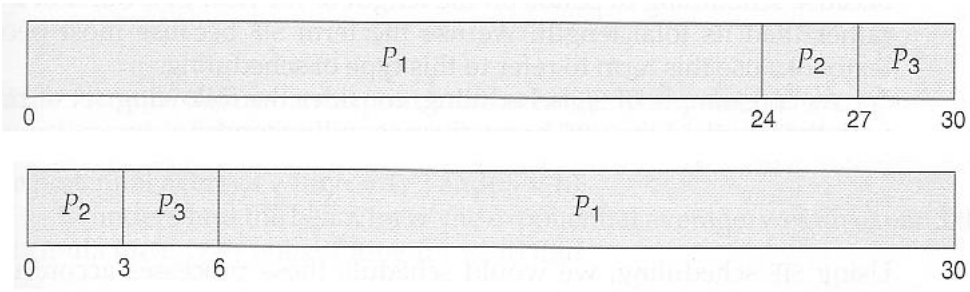


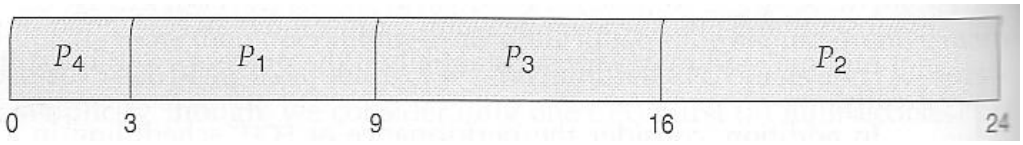
Figure 2.20 Gantt Chart or timing diagram of CPU schedule.

- FCFS can also block the system in a busy dynamic system in another way, known as the ***convoy effect***. When one CPU intensive process blocks the CPU, a number of I/O intensive processes can get backed up behind it, leaving the I/O devices idle. When the CPU hog finally relinquishes the CPU, then the I/O processes pass through the CPU quickly, leaving the CPU idle while everyone queues up for I/O, and then the cycle repeats itself when the CPU intensive process gets back to the ready queue.

Shortest-Job-First Scheduling, SJF

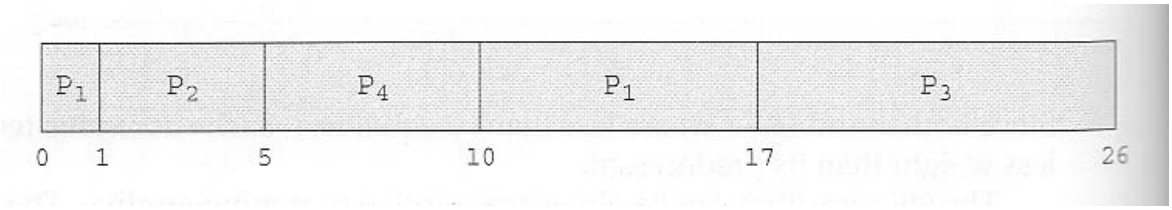
- The idea behind the SJF algorithm is to pick the quickest fastest little job that needs to be done, get it out of the way first, and then pick the next smallest fastest job to do next.
- (Technically this algorithm picks a process based on the next shortest **CPU burst**, not the overall process time.)
- For example, the Gantt chart below is based upon the following CPU burst times, (and the assumption that all jobs arrive at the same time.)
- In the case above the average wait time is (0 + 3 + 9 + 16) / 4 = 7.0 ms, (as opposed to 10.25 ms for FCFS for the same processes.)
- SJF can be proven to be the fastest scheduling algorithm, but it suffers from one important problem: How do you know how long the next CPU burst is going to be?
- SJF can be either preemptive or non-preemptive. Preemption occurs when a new process arrives in the ready queue that has a predicted burst time shorter than the time remaining in the process whose burst is currently on the CPU. Preemptive SJF is sometimes referred to as ***shortest remaining time first scheduling.***

Process	Burst Time
P1	6
P2	8
P3	7
P4	3



- For example, the following Gantt chart is based upon the following data:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
p4	3	5



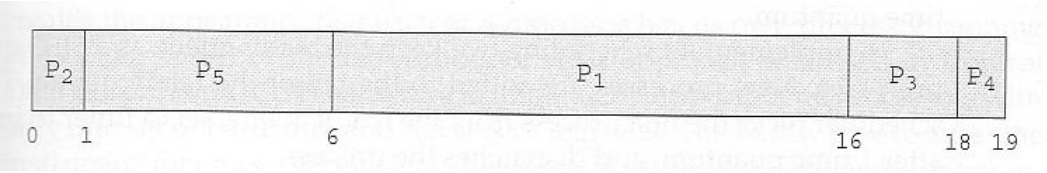
- The average wait time in this case is ((5 - 3) + (10 - 1) + (17 - 2)) / 4 = 26 / 4 = 6.5 ms. (As opposed to 7.75 ms for non-preemptive SJF or 8.75 for FCFS.)

Priority Scheduling

- Priority scheduling is a more general case of SJF, in which each job is assigned a priority and the job with the highest priority gets scheduled first. (SJF uses the inverse of the next expected burst time as its priority.(The smaller the expected burst, the higher the priority.)

- Note that in practice, priorities are implemented using integers within a fixed range, but there is no agreed-upon convention as to whether "high" priorities use large numbers or small numbers. This book uses low number for high priorities, with 0 being the highest possible priority.
- For example, the following Gantt chart is based upon these process burst times and priorities, and yields an average waiting time of 8.2 ms:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



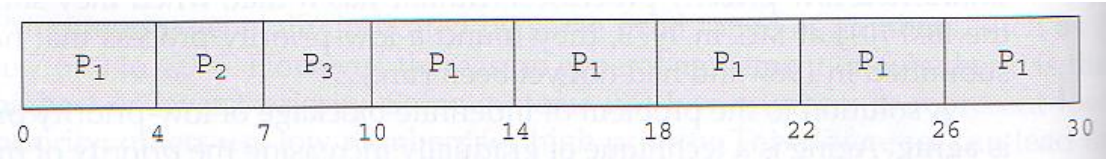
- Priorities can be assigned either internally or externally. Internal priorities are assigned by the OS using criteria such as average burst time, ratio of CPU to I/O activity, system resource use, and other factors available to the kernel. External priorities are assigned by users, based on the importance of the job, fees paid, politics, etc.
- Priority scheduling can be either preemptive or non-preemptive.
- Priority scheduling can suffer from a major problem known as ***indefinite blocking***, or ***starvation***, in which a low-priority task can wait forever because there are always some other jobs around that have higher priority.
    - If this problem is allowed to occur, then processes will either run eventually when the system load lightens (at say 2:00 a.m.), or will eventually get lost when the system is shut down or crashes. (There are rumors of jobs that have been stuck for years.)
    - One common solution to this problem is ***aging***, in which priorities of jobs increase the longer they wait. Under this scheme a low-priority job will eventually get its priority raised high enough that it gets run.

Round Robin Scheduling

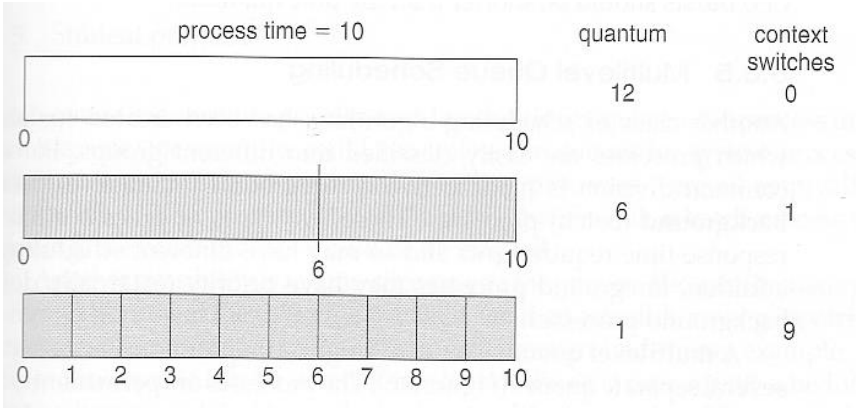
- Round robin scheduling is similar to FCFS scheduling, except that CPU bursts are assigned with limits called ***time quantum***.

- When a process is given the CPU, a timer is set for whatever value has been set for a time quantum.
    - If the process finishes its burst before the time quantum timer expires, then it is swapped out of the CPU just like the normal FCFS algorithm.
    - If the timer goes off first, then the process is swapped out of the CPU and moved to the back end of the ready queue.
- The ready queue is maintained as a circular queue, so when all processes have had a turn, then the scheduler gives the first process another turn, and so on.
- RR scheduling can give the effect of all processors sharing the CPU equally, although the average wait time can be longer than with other scheduling algorithms. In the following example the average wait time is 5.66 ms.

Process	Burst Time
P1	24
P2	3
P3	3



- The performance of RR is sensitive to the time quantum selected. If the quantum is large enough, then RR reduces to the FCFS algorithm; If it is very small, then each process gets 1/nth of the processor time and share the CPU equally.
- **BUT**, a real system invokes overhead for every context switch, and the smaller the time quantum the more context switches there are. (See Figure 5.4 below.) Most modern systems use time quantum between 10 and 100 milliseconds, and context switch times on the order of 10 microseconds, so the overhead is small relative to the time quantum.



Multilevel Queue Scheduling

- When processes can be readily categorized, then multiple separate queues can be established, each implementing whatever scheduling algorithm is most appropriate for that type of job, and/or with different parametric adjustments.
- Scheduling must also be done between queues, that is scheduling one queue to get time relative to other queues. Two common options are strict priority (no job in a lower priority queue runs until all higher priority queues are empty) and round-robin (each queue gets a time slice in turn, possibly of different sizes.)
- Note that under this algorithm jobs cannot switch from queue to queue - Once they are assigned a queue, that is their queue until they finish.

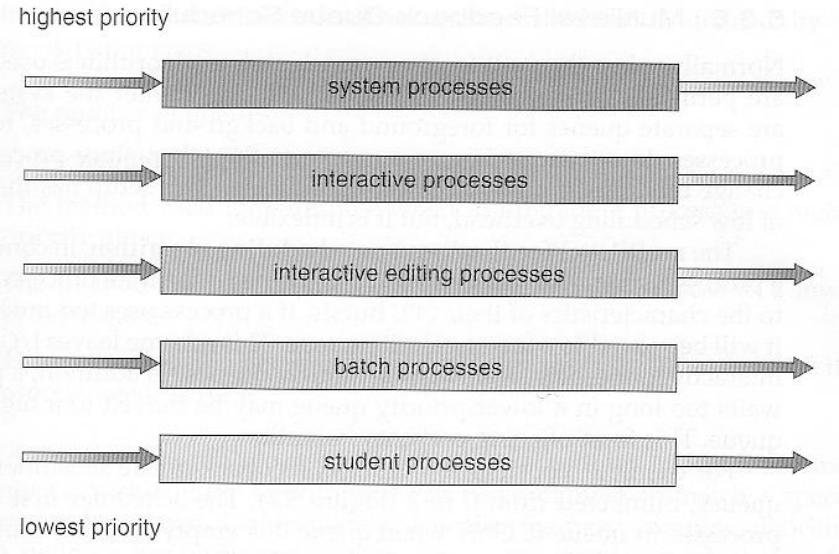


Figure 2.21 Multilevel queue scheduling

2.10 Context Switch

Sometimes interrupts cause the operating system to change a CPU core from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. When an interrupt occurs, the system needs to save the current context of the process running on the CPU core so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state (see Figure 3.2), and memory-management information. Generically, we perform a state save of the current state of the CPU core, be it in kernel or user mode, and then a state restore to resume operations. Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch and is illustrated in Figure 3.6.

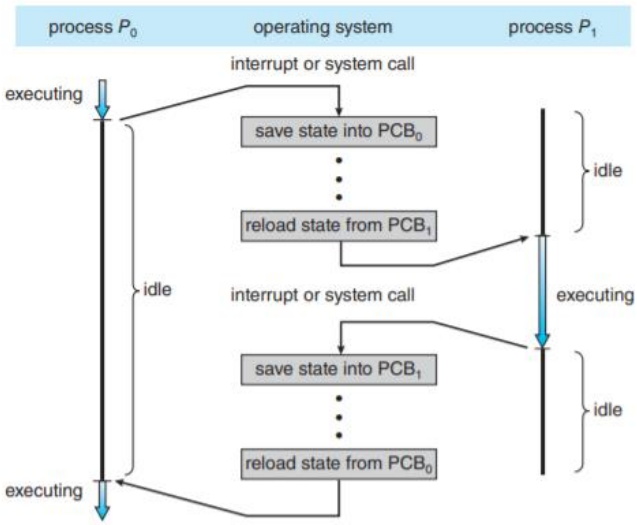


Figure 2.22 Diagram showing context switch from process to process.

When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context switch time is pure overhead, because the system does no useful work while switching. Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a several microseconds.

2.11 Interprocess Communication

There are two common models of interprocess communication: the message passing model and the shared-memory model. In the message-passing model, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network. Each computer in a network has a host name by which it is commonly known.

A host also has a network identifier, such as an IP address. Similarly, each process has a process name, and this name is translated into an identifier by which the operating system can refer to the process. The get hostid() and get processid() system calls do this translation. The identifiers are then passed to the general purpose open() and close() calls provided by the file system or to specific open connection() and close connection() system calls, depending on the system's model of communication. The recipient process usually must give its permission for communication to take place with an accept connection() call. Most processes that will be receiving connections are special-purpose daemons, which are system programs provided for that purpose. They execute a wait for connection() call and are awakened when a connection is made. The source of the communication, known as the client, and the receiving daemon, known as a server, then exchange

messages by using read message() and write message() system calls. The close connection() call terminates the communication.

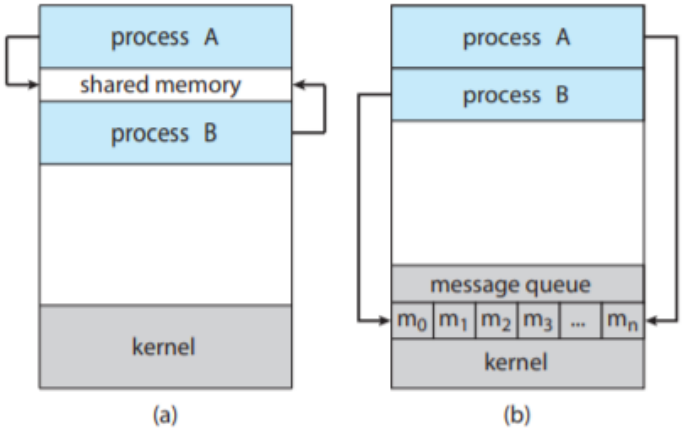


Figure 2.23 Communications models. (a) Shared memory. (b) Message passing.

In the shared-memory model, processes use shared memory create() and shared memory attach() system calls to create and gain access to regions of memory owned by other processes. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data is determined by the processes and is not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously. Such mechanisms are discussed in Chapter 6.

In Chapter 4, we look at a variation of the process scheme— threads—in which some memory is shared by default. Both of the models just discussed are common in operating systems, and most systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. It is also easier to implement than is shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication, since it can be done at memory transfer speeds when it takes place within a computer. Problems exist, however, in the areas of protection and synchronization between the processes sharing memory.

Shared Memory

We know that to communicate between two or more processes, we use shared memory but before using the shared memory what needs to be done with the system calls, let us see this −

- Create the shared memory segment or use an already created shared memory segment (shmget())
- Attach the process to the already created shared memory segment (shmat())
- Detach the process from the already attached shared memory segment (shmdt())
- Control operations on the shared memory segment (shmctl())

Let us look at a few details of the system calls related to shared memory.

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg)
```

The above system call creates or allocates a shared memory segment. The arguments that need to be passed are as follows −

The **first argument, key,** recognizes the shared memory segment. The key can be either an arbitrary value or one that can be derived from the library function ftok(). The key can also be IPC_PRIVATE, means, running processes as server and client (parent and child relationship) i.e., inter-related process communiation. If the client wants to use shared memory with this key, then it must be a child process of the server. Also, the child process needs to be created after the parent has obtained a shared memory.

The **second argument, size,** is the size of the shared memory segment rounded to multiple of PAGE_SIZE.

The **third argument, shmflg,** specifies the required shared memory flag/s such as IPC_CREAT (creating new segment) or IPC_EXCL (Used with IPC_CREAT to create new segment and the call fails, if the segment already exists). Need to pass the permissions as well.

Note − Refer earlier sections for details on permissions.

This call would return a valid shared memory identifier (used for further calls of shared memory) on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
#include <sys/types.h>
#include <sys/shm.h>
void * shmat(int shmid, const void *shmaddr, int shmflg)
```

The above system call performs shared memory operation for shared memory segment i.e., attaching a shared memory segment to the address space of the calling process. The arguments that need to be passed are as follows −

The first argument, shmid, is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.

The second argument, shmaddr, is to specify the attaching address. If shmaddr is NULL, the system by default chooses the suitable address to attach the segment. If shmaddr is not NULL and SHM_RND is specified in shmflg, the attach is equal to the address of the nearest multiple of SHMLBA (Lower Boundary Address). Otherwise, shmaddr must be a page aligned address at which the shared memory attachment occurs/starts.

The third argument, shmflg, specifies the required shared memory flag/s such as SHM_RND (rounding off address to SHMLBA) or SHM_EXEC (allows the contents of segment to be executed) or SHM_RDONLY (attaches the segment for read-only purpose, by default it is read-write) or SHM_REMAP (replaces the existing mapping in the range specified by shmaddr and continuing till the end of segment).

This call would return the address of attached shared memory segment on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
#include <sys/types.h>
#include <sys/shm.h>
int shmdt(const void *shmaddr)
```

The above system call performs shared memory operation for shared memory segment of detaching the shared memory segment from the address space of the calling process. The argument that needs to be passed is −

The argument, shmaddr, is the address of shared memory segment to be detached. The to-be-detached segment must be the address returned by the shmat() system call.

This call would return 0 on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

The above system call performs control operation for a System V shared memory segment. The following arguments needs to be passed −

The first argument, shmid, is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.

The second argument, cmd, is the command to perform the required control operation on the shared memory segment.

Valid values for cmd are −

- **IPC_STAT** − Copies the information of the current values of each member of struct shmid_ds to the passed structure pointed by buf. This command requires read permission to the shared memory segment.
- **IPC_SET** − Sets the user ID, group ID of the owner, permissions, etc. pointed to by structure buf.
- **IPC_RMID** − Marks the segment to be destroyed. The segment is destroyed only after the last process has detached it.
- **IPC_INFO** − Returns the information about the shared memory limits and parameters in the structure pointed by buf.
- **SHM_INFO** − Returns a shm_info structure containing information about the consumed system resources by the shared memory.

The third argument, buf, is a pointer to the shared memory structure named struct shmid_ds. The values of this structure would be used for either set or get as per cmd.

This call returns the value depending upon the passed command. Upon success of IPC_INFO and SHM_INFO or SHM_STAT returns the index or identifier of the shared memory segment or 0

for other operations and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

Message Passing

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.

A message-passing facility provides at least two operations:

send(message) and

receive(message)

Messages sent by a process can be either fixed or variable in size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler.

This is a common kind of tradeoff seen throughout operating-system design. If processes P and Q want to communicate, they must send messages to and receive messages from each other: a communication link must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network, which are covered in Chapter 19) but rather with its logical implementation.

Here are several methods for logically implementing a link and the send()/receive() operations:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication. Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:

- send(P, message)—Send a message to process P.
- receive(Q, message)—Receive a message from process Q.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate.
- The processes need to know only each other's identity to communicate.

•A link is associated with exactly two processes.

•Between each pair of processes, there exists exactly one link

With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. For example, POSIX message queues use an integer value to identify a mailbox. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox.

The send() and receive() primitives are defined as follows:

• send(A, message)—Send a message to mailbox A.

• receive(A, message)—Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

• A link is established between a pair of processes only if both members of the pair have a shared mailbox.

• A link may be associated with more than two processes.

• Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

Now suppose that processes P1, P2, and P3 all share mailbox A. Process P1 sends a message to A, while both P2 and P3 execute a receive() from A. Which process will receive the message sent by P1? The answer depends on which of the following methods we choose:

The mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists. In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process.

The operating system then must provide a mechanism that allows a process to do the following:

• Create a new mailbox.

• Send and receive messages through the mailbox.

• Delete a mailbox.

The process that creates a new mailbox is that mailbox’s owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

2.12 Process Synchronisation

Process Synchronization is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources. It is specially needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or data at the same time. This can lead to the inconsistency of shared data. So the change made by one process not necessarily reflected when other processes accessed the same shared data. To avoid this type of inconsistency of data, the processes need to be synchronized with each other.

Processes are categorized as one of the following two types:

- **Independent Process** : Execution of one process does not affects the execution of other processes.
- **Cooperative Process** : Execution of one process affects the execution of other processes.

Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

Race Condition

When more than one processes are executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing race to say that my output is correct this condition known as race condition. Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place.

Consider the following example where we have two processes and these processes are using the same variable "a". They are reading the variable and then updating the value of the variable and finally writing the data in the memory.

```
SomeProcess(){
   ...
   read(a) //instruction 1
   a = a + 5 //instruction 2
   write(a) //instruction 3
   ...
}
```

In the above, you can see that a process after doing some operations will have to read the value of "a", then increment the value of "a" by 5 and at last write the value of "a" in the memory. Now, we have two processes P1 and P2 that needs to be executed. Let's take the following two cases and also assume that the value of "a" is 10 initially.

1. In this case, process P1 will be executed fully (i.e. all the three instructions) and after that, the process P2 will be executed. So, the process P1 will first read the value of "a" to

be 10 and then increment the value by 5 and make it to 15. Lastly, this value will be updated in the memory. So, the current value of "a" is 15. Now, the process P2 will read the value i.e. 15, increment with 5(15+5 = 20) and finally write it to the memory i.e. the new value of "a" is 20. Here, in this case, the final value of "a" is 20.

2. In this case, let's assume that the process P1 starts executing. So, it reads the value of "a" from the memory and that value is 10(initial value of "a" is taken to be 10). Now, at this time, context switching happens between process P1 and P2(learn more about context switching from here). Now, P2 will be in the running state and P1 will be in the waiting state and the context of the P1 process will be saved. As the process P1 didn't change the value of "a", so, P2 will also read the value of "a" to be 10. It will then increment the value of "a" by 5 and make it to 15 and then save it to the memory. After the execution of the process P2, the process P1 will be resumed and the context of the P1 will be read. So, the process P1 is having the value of "a" as 10(because P1 has already executed the instruction 1). It will then increment the value of "a" by 5 and write the final value of "a" in the memory i.e. a = 15. Here, the final value of "a" is 15.

In the above two cases, after the execution of the two processes P1 and P2, the final value of "a" is different i.e. in 1st case it is 20 and in 2nd case, it is 15. What's the reason behind this?

The processes are using the same resource here i.e. the variable "a". In the first approach, the process P1 executes first and then the process P2 starts executing. But in the second case, the process P1 was stopped after executing one instruction and after that the process P2 starts executing. And here both the processes are dealing on the same resource i.e. variable "a" at the same time. Here, the order of execution of processes changes the output. All these processes are in a race to say that their output is correct. This is called a race condition.

The code in the above part is accessed by all the process and this can lead to data inconsistency. So, this code should be placed in the critical section. The critical section code can be accessed by only one process at a time and no other process can access that critical section code.

Critical Section Problem

Critical section is a code segment that can be accessed by only one process at a time. Critical section contains shared variables which need to be synchronized to maintain consistency of data variables.

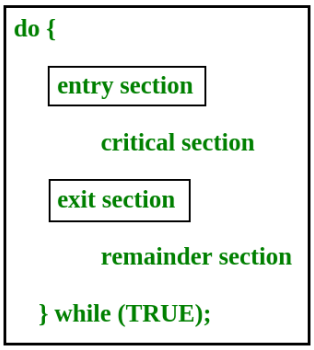


Figure 2.24 critical section scenario

In the entry section, the process requests for entry in the **Critical Section.** Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion**: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress** : If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection can not be postponed indefinitely.
- **Bounded Waiting** : A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

The following are the different solutions or algorithms used in synchronizing the different processes of any operating system:

1. Peterson's solution

Peterson's solution is one of the famous solutions to critical section problems. This algorithm is created by a computer scientist Peterson. Peterson's solution is solution to the critical section problem involving two processes. Peterson's solution states that when a process is executing in its critical state, then the other process executes the rest of code and vice versa. This insures that only one process is in the critical section at a particular instant of time.

In Peterson's solution, we have two shared variables:

- boolean flag[i] :Initialized to FALSE, initially no one is interested in entering the critical section
- int turn : The process whose turn is to enter the critical section.

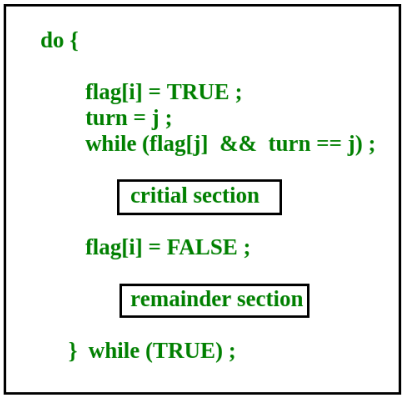


Figure 2.25 Peterson solution to critical section

Peterson's Solution preserves all three conditions :

- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

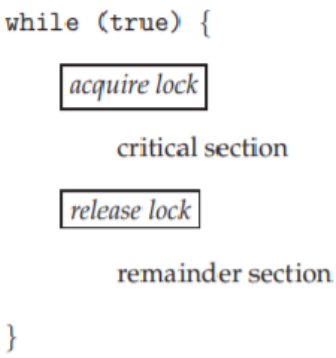
Disadvantages of Peterson's Solution

- It involves Busy waiting
- It is limited to 2 processes.

2. Mutex Lock solution

Locking solution is another solution to critical problems in which a process acquires a lock before entering its critical section. When a process finishes its executing process in the critical section, then it releases the lock. Then the lock is available for any other process that wants to execute its critical section. The locking mechanism also ensures that only one process is in the critical section at a particular instant of time.

A mutex lock has a boolean variable available whose value indicates if the lock is available or not. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released. The following example shows the solution to critical section problem with mutex lock.



The main disadvantage of the implementation given here is that it requires busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire(). This continual looping is clearly a problem in a real multiprogramming system, where a single CPU core is shared among many processes. Busy waiting also wastes CPU cycles that some other process might be able to use productively.

3. Semaphore solution

Semaphores are another solution to the critical section problem. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal(). Semaphores were introduced by the Dutch computer scientist Edsger Dijkstra, and such, the wait() operation was originally termed P (from the Dutchproberen, "to test"); signal() was originally called V (from verhogen, "to increment").

The definition of wait() is as follows:

```
wait(S)
{
while (S <= 0) ; // busy wait
S--;
}
```

The definition of signal() is as follows:

```
signal(S)
{
        S++;
}
```

All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed atomically. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of

wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption

Operating systems often distinguish between counting and binary semaphores. The value of a counting semaphore can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion. Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0.

In process P1, we insert the statements

```
S1;
signal(synch)
```

In process P2, we insert the statements

```
wait(synch);
S2;
```

Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed.

2.13 Memory Management

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

Process Address Space

The process address space is the set of logical addresses that a process references in its code. For example, when 32-bit addressing is in use, addresses can range from 0 to 0x7fffffff; that is, 2^31 possible numbers, for a total theoretical size of 2 gigabytes.

The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program. There are three types of addresses used in a program before and after memory is allocated.

Table 2.1 Memory Address Description

S.N.	Memory Addresses & Description
1	**Symbolic addresses** The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.
2	**Relative addresses** At the time of compilation, a compiler converts symbolic addresses into relative addresses.
3	**Physical addresses** The loader generates these addresses at the time when a program is loaded into main memory.

Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.

The set of all logical addresses generated by a program is referred to as a **logical address space**. The set of all physical addresses corresponding to these logical addresses is referred to as a **physical address space.**

The runtime mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses following mechanism to convert virtual address to physical address.

- The value in the base register is added to every address generated by a user process, which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.
- The user program deals with virtual addresses; it never sees the real physical addresses.

Static vs Dynamic Loading

The choice between Static or Dynamic Loading is to be made at the time of computer program being developed. If you have to load your program statically, then at the time of compilation, the complete programs will be compiled and linked without leaving any external program or module dependency. The linker combines the object program with other necessary object modules into an absolute program, which also includes logical addresses.

If you are writing a Dynamically loaded program, then your compiler will compile the program and for all the modules which you want to include dynamically, only references will be provided and rest of the work will be done at the time of execution. At the time of loading, with **static loading**, the absolute program (and data) is loaded into memory in order for execution to start. If you are using **dynamic loading**, dynamic routines of the library are stored on a disk in relocatable form and are loaded into memory only when they are needed by the program.

Static vs Dynamic Linking

As explained above, when static linking is used, the linker combines all other modules needed by a program into a single executable program to avoid any runtime dependency. When dynamic linking is used, it is not required to link the actual module or library with the program, rather a reference to the dynamic module is provided at the time of compilation and linking. Dynamic Link Libraries (DLL) in Windows and Shared Objects in Unix are good examples of dynamic libraries.

Swapping

Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.

Though performance is usually affected by swapping process but it helps in running multiple and big processes in parallel and that's the reason **Swapping is also known as a technique for memory compaction**. The total time taken by swapping process includes the time it takes to move the entire process to a secondary disk and then to copy the process back to memory, as well as the time the process takes to regain main memory. Let us assume that the user process is of size 2048KB and on a standard hard disk where swapping will take place has a data transfer rate around 1 MB per second. The actual transfer of the 1000K process to or from memory will take

2048KB / 1024KB per second
= 2 seconds
= 2000 milliseconds

Now considering in and out time, it will take complete 4000 milliseconds plus other overhead where the process competes to regain main memory.

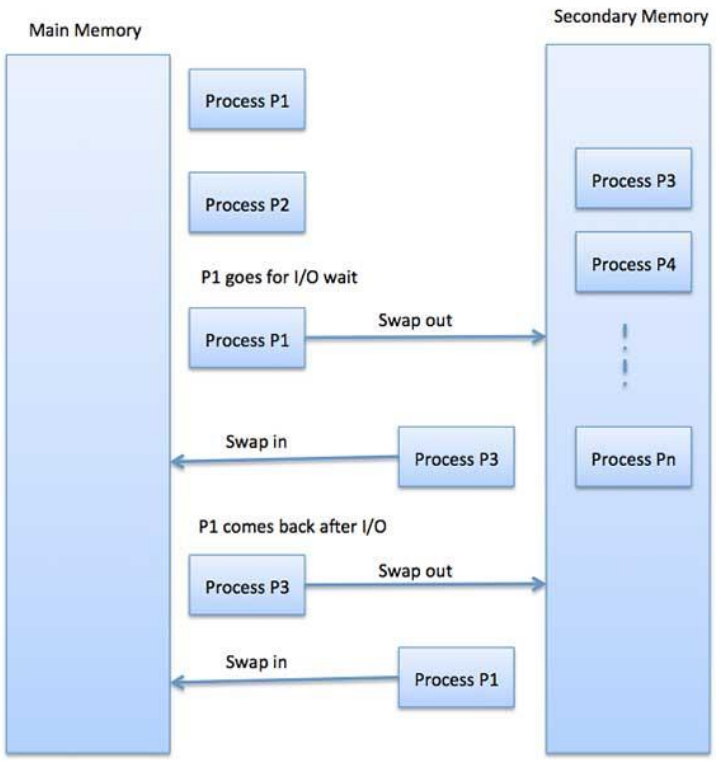


Figure 2.26 : Memory Allocation

Main memory usually has two partitions −

- **Low Memory** − Operating system resides in this memory.
- **High Memory** − User processes are held in high memory.

Operating system uses the following memory allocation mechanism.

S.N.	Memory Allocation & Description
1	**Single-partition allocation** In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register.
2	**Multiple-partition allocation** In this type of allocation, main memory is divided into a number of fixed-sized partitions where each partition should contain only one process. When a partition is free, a process is selected from the input queue and is loaded into the free

	partition. When the process terminates, the partition becomes available for another process.

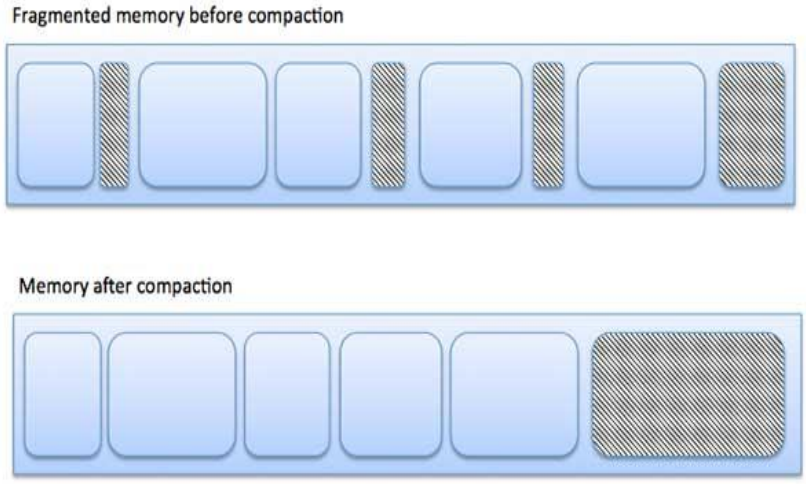
Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types –

S.N.	Fragmentation & Description
1	**External fragmentation** Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.
2	**Internal fragmentation** Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory −

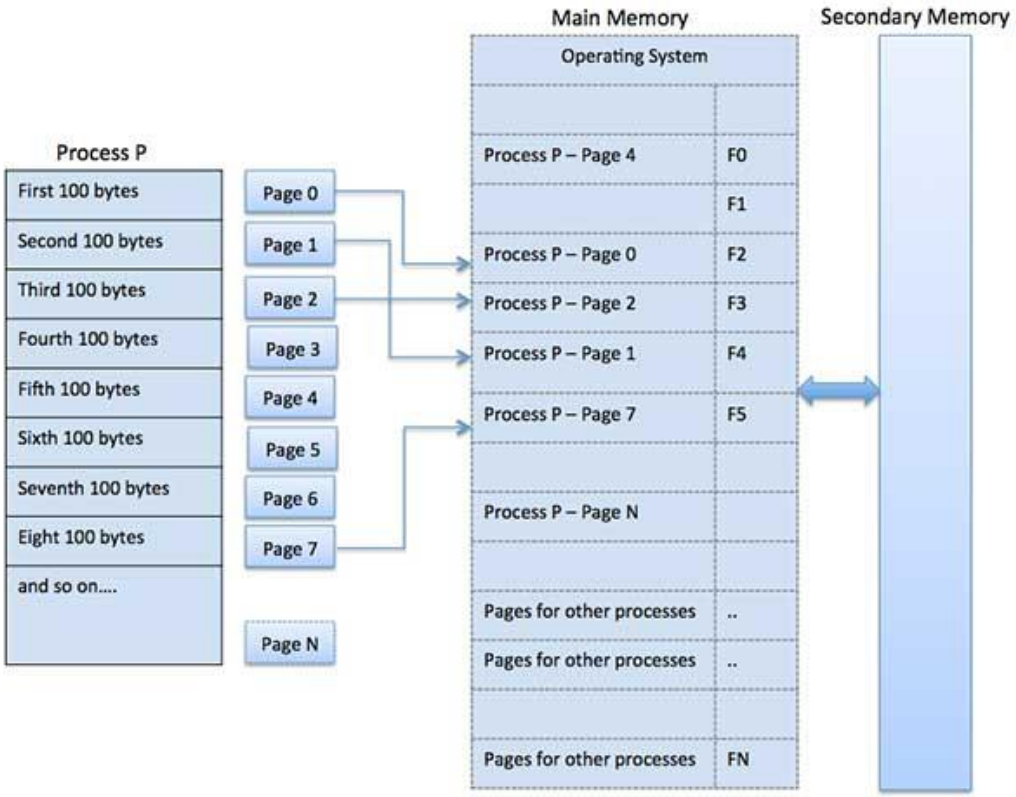


External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic. The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.

Paging

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory. Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.



Address Translation

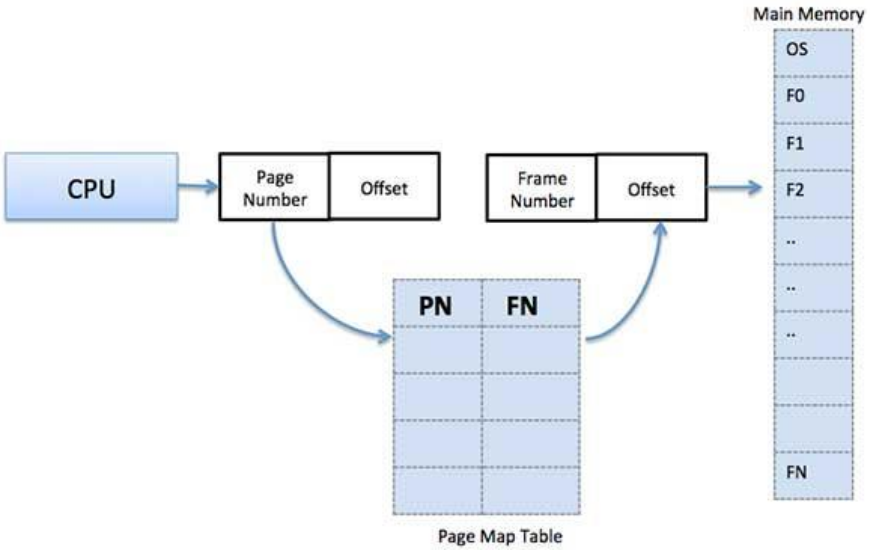
Page address is called **logical address** and represented by **page number** and the **offset**.

Logical Address = Page number + page offset

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

Physical Address = Frame number + page offset

A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.



When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program. When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program. This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

Advantages and Disadvantages of Paging

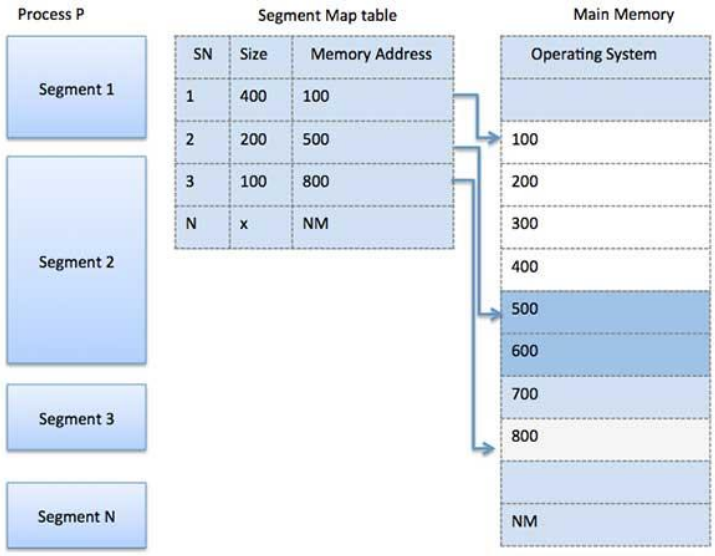
Here is a list of advantages and disadvantages of paging −

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.
- Page table requires extra memory space, so may not be good for a system having small RAM.

Segmentation

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program. When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.

Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size. A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a **segment map table** for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.



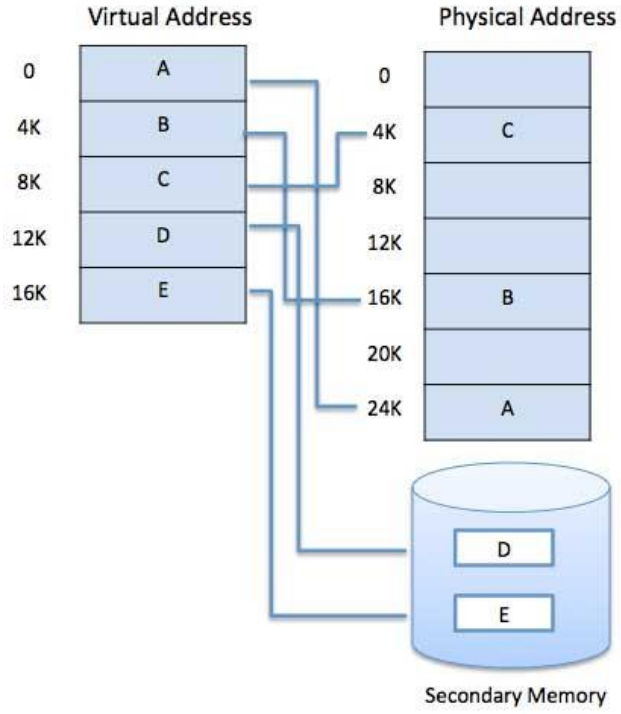
A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM.

The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occurred in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.

- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

Modern microprocessors intended for general-purpose use, a memory management unit, or MMU, is built into the hardware. The MMU's job is to translate virtual addresses into physical addresses. A basic example is given below −



Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

References:

1. Tammy Noergaard “Embedded Systems Architecture A Comprehensive Guide for Engineers and Programmers”, First Edition, Elsevier Inc., 2005.
2. Silberschatz, Abraham, Galvin Peter B, Gagne Greg, “Operating system concepts “, 10th edition, Wiley, 2018.

Exercise questions

1. What are the three main purposes of an operating system?
2. What is the main difficulty that a programmer must overcome in writing an operating system for a real-time environment?
3. How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security)?
4. What is the purpose of interrupts? How does an interrupt differ from a trap? Can traps be generated intentionally by a user program? If so, for what purpose?
5. Describe some of the challenges of designing operating systems for mobile devices compared with designing operating systems for traditional PCs.
6. What are some advantages of peer-to-peer systems over client–server systems? 1.26 Describe some distributed applications that would be appropriate for a peer-to-peer system.
7. Identify several advantages and several disadvantages of open-source operating systems. Identify the types of people who would find each aspect to be an advantage or a disadvantage.
8. Identify services provided by an operating system and illustrate how system calls are used to provide operating system services.
9. Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing operating systems.
10. Illustrate the process for booting an operating system.
11. Why Applications Are Operating-System Specific
12. What is the purpose of system calls?
13. What is the purpose of the command interpreter? Why is it usually separate from the kernel?
14. What system calls have to be executed by a command interpreter or shell in order to start a new process on a UNIX system?
15. What is the purpose of system programs?
16. What is the main advantage of the layered approach to system design? What are the disadvantages of the layered approach?
17. List five services provided by an operating system, and explain how each creates convenience for users. In which cases would it be impossible for user-level programs to provide these services? Explain your answer.
18. Why do some systems store the operating system in firmware, while others store it on disk?
19. How could a system be designed to allow a choice of operating systems from which to boot? What would the bootstrap program need to do?
20. What are the two models of interprocess communication? What are the strengths and weaknesses of the two approaches?
21. What are the advantages of using loadable kernel modules?

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – III - EMBEDDED OS AND DEVICE DRIVERS – SECA5204

3. EMBEDDED OPERATING SYSTEMS

3.1 Embedded OS overview

In the early days, most embedded systems were designed for special applications. An embedded system usually consists of a microcontroller and a few I/O devices, which is designed to monitor some input sensors and generate signals to control external devices, such as to turn on LEDs or activate some switches, etc. For this reason, control programs of early embedded systems are also very simple. They are usually written in the form of a super-loop or a simple event-driven program structure. However, as the computing power of embedded systems increases in recent years, embedded systems have undergone a tremendous leap in both complexity and areas of applications. As a result, the traditional approaches to software design for embedded systems are no longer adequate.

In order to cope with the ever increasing system complexity and demands for extra functionality, embedded systems need more powerful software. As of now, many embedded systems are in fact high-power computing machines with multicore processors, gigabytes memory and multi-gigabyte storage devices. Such systems are intended to run a wide range of application programs. In order to fully realize their potential, modern embedded systems need the support of multi-functional operating systems. A good example is the evolution of earlier cell phones to current smart phones. Whereas the former were designed to perform the simple task of placing or receiving phone calls only, the latter may use multicore processors and run adapted versions of Linux, such as Android, to perform multitasks. The current trend of embedded system software design is clearly moving in the direction of developing multi-functional operating systems suitable for future mobile environment.

An ***embedded operating system*** is an operating system for embedded computer systems. This type of operating system is typically designed to be resource-efficient and reliable. Resource efficiency comes at the cost of losing some functionality or granularity that larger computer operating systems provide, including functions which may not be used by the specialized applications they run.

The basic functions of an operating system are to manage the system's peripherals and schedule software tasks to ensure that each program gets some processor time. A file system is also part of a standard OS to store software modules and boot instructions. Another big benefit of an OS is to provide networking software and drivers for common hardware peripherals, eliminating constant reinvention. As shown in Figure 3.1, an OS sits over the hardware, either over the device driver layer or over a Board Support Package (BSP). However, an embedded OS is quite different from its desktop counterpart.

Desktop systems assume a keyboard, a mouse, a display, a hard disk, and plenty of memory. However, there is no such standardization in embedded products. One embedded system might have no hard disk and limited memory while another has no user I/O at all. An embedded OS must also be modular, allowing components to be added or removed to adjust the memory footprint. And moreover, an operating system (OS) is an optional part of an embedded device's system software stack, meaning that not all embedded systems have one.

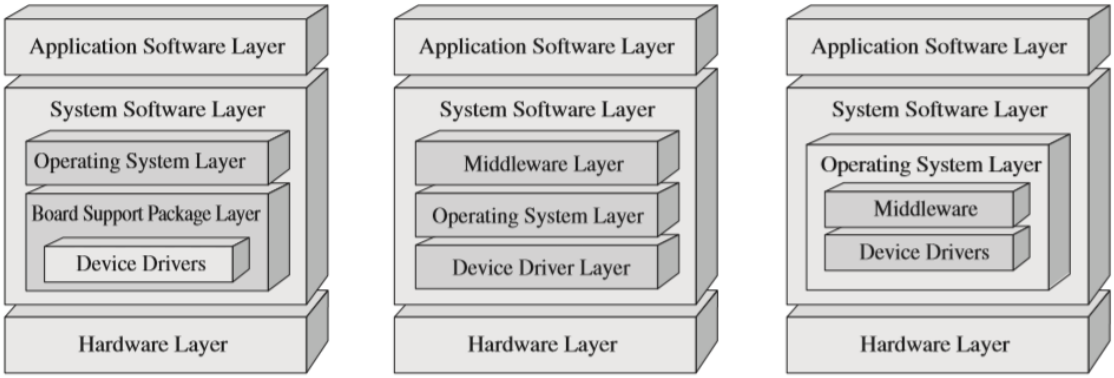


Figure 3.1. Hardware and Software layers in an Embedded Systems Model

3.3 Requirements and Features of Embedded OS

Operating systems for embedded system applications should have the following desired characteristics. The choice of embedded OS depends on the specific applications of the embedded system.

Size

The size is everytime considered in an embedded system. Because it has very limited resources like RAM, ROM and CPU power. So keep the embedded operating system small as possible to fit into given memory space.

Fast and Lightweight

As the embedded systems have small CPU with limited processing power. It should be customised perfectly so that it can execute fast. It the embedded system developer task to understand the existing hardware and remove the unwanted software modules at the time of compilation. This will give a lightweight operating system results a faster execution speed.

Configurability

As we know that embedded systems are designed as per the application requirement. And according to the hardware we need to customize the embedded operating system. So the operating system should be designed in such a way that an embedded developer can configure the operating system as per the need.
In some operating system facility of conditional compilation is available. Where the developer compiles the only required module from the overall modules. And it is best suitable if we are using the object-oriented approach.

I/O Device Flexibility

There is no generalise hardware that is suitable or adjustable for all operating system versions.

Direct Use of Interrupts

The embedded operating system provides the use of interrupts to give them more control over the peripheral. The general purpose operating system does not provide such kind

of facility to the user directly. In the embedded system we need more control on the individual hardware so there is the demand of the interrupts. The interrupt also has the priority. And according to that priority, the task is serviced by the CPU.

Real-Time Operation

As we know that real-time embedded systems have a time constrained to execute the task. This time is called a deadline. The soft-realtime system may vary the deadline. But the hard real-time system must complete the task in a given time frame.

Soft-Real-Time System

The example of the soft-realtime system could be our day to day lifer products like washing machine, microwave oven, printer and fax machine. Let’s suppose we are cooking something. We put some item to cook. We set a time and temperature. As soon as we press the start button of the oven it takes some random time to start to suppose 15sec. Even after a 15sec delay, it cooked perfectly, nothing went wrong in cooking. It missed the time by approx 15sec. This is generally happening in the soft real-time system.

Hard-Real-Time System

There is some application where the systems should act in a given time otherwise some went wrong or action is not acceptable. Like in a traffic light controller, the timing of different signals should be running in a mentioned time in the program. The variation in time is not acceptable because something wrong may happen. In essence, the deadline is fixed according to the system application. And to make the system time critical we pause other less important tasks so the main priority task can execute on time.

Reactive Operation

A system is called reactive if it acts on certain input by the user in the form of switch press or by some sensor. For example, a motion sensor security sensor triggers the alarm when someone comes in the rang of the sensor. Here system is reacting based on the sensor input.

Figure 3.2 shows us what's important to those who get a say in the choice of embedded OS. For most, real-time performance is the big issue. Right behind is processor compatibility. In other words, we can't use an OS that hasn't been ported to our preferred microprocessor, microcontroller, or DSP. That makes sense; compatibility is a pass/fail criterion for most developers. It's all the more significant, then, that a slight majority chose performance over compatibility as their hot-button issue.

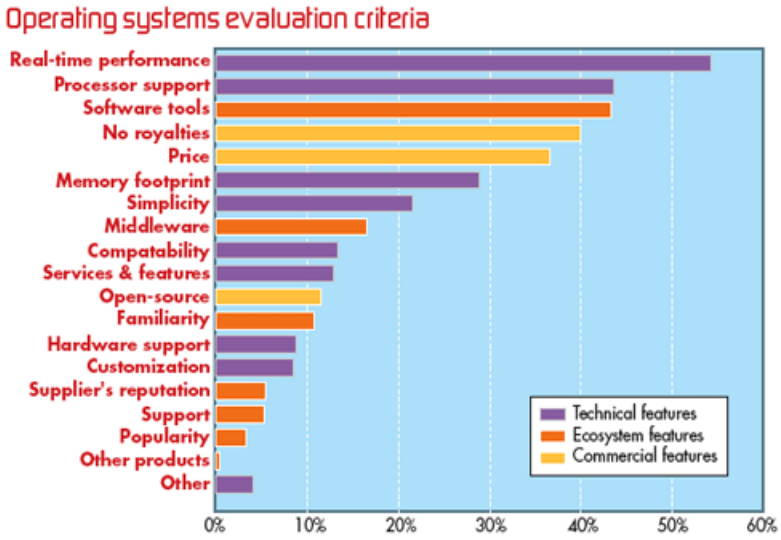


Figure 3.2. Selection factors for OS

Many developers have selected Linux for embedded applications, taking advantage of its open source distribution and the very large range of middleware and drivers that are available.

3.3 Internal Components of embedded OS

While embedded OSes vary in what components they possess, all embedded operating systems have a *kernel* at the very least. The kernel is a software component that contains the main functionality of the OS that includes Process Management, Memory Management and I/O System Management as shown in figure 3.3. I/O devices also need to be shared among the various processes and so, just as with memory, access and allocation of an I/O device need to be managed. Through I/O system management, file system management can also be provided as a method of storing and managing data in the forms of files.

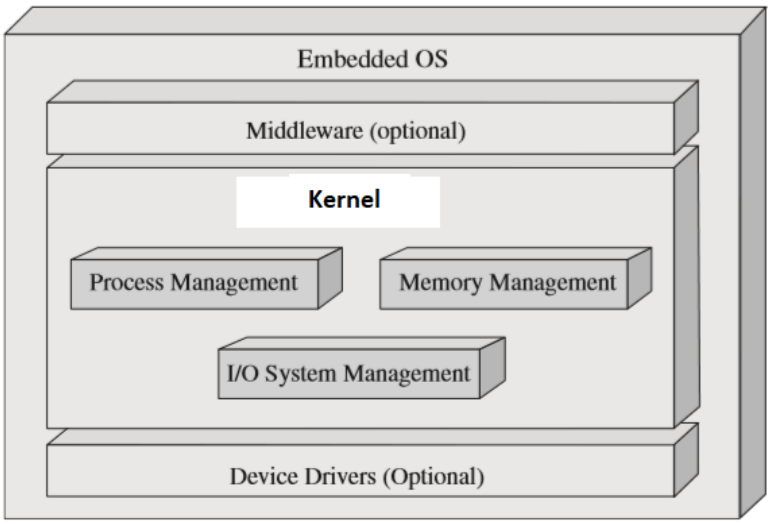


Figure 3.3. General embedded OS model

Device drivers and middleware are optional components of embedded operating systems. The dependencies among kernel components or kernel subsystems is illustrated in figure 3.4.

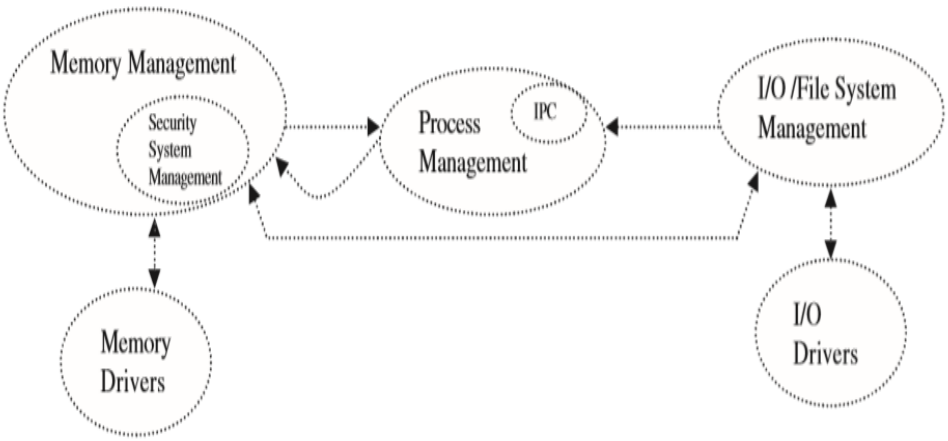


Figure 3.4. Kernel components dependencies

Because of the way in which an operating system manages the software in a system, using processes, the process management component is the most central subsystem in an OS. All other OS subsystems depend on the process management unit. Since all code must be loaded into main memory (RAM or cache) for the master CPU to execute, with boot code and data located in non-volatile memory (ROM, Flash, etc.), the process management subsystem is equally dependent on the memory management subsystem. I/O management, for example, could include networking I/O to interface with the memory manager in the case of a network file system (NFS).

Outside the kernel, the Memory Management and I/O Management subsystems then rely on the device drivers, and vice-versa, to access the hardware. Whether inside or outside an OS kernel, OSes also vary in what other system software components, such as device drivers and middleware, they incorporate (if any).

In fact, most embedded OSes are typically based upon one of three models, the monolithic, layered, or microkernel (client-server) design.

In general, these models differ according to the internal design of the OS's kernel, as well as what other system software has been incorporated into the OS. In a monolithic OS, middleware and device driver functionality is typically integrated into the OS along with the kernel. This type of OS is a single executable file containing all of these components. Monolithic OSes are usually more difficult to scale down, modify, or debug than their other OS architecture counterparts, because of their inherently large, integrated, cross-dependent nature. Thus, a more popular algorithm, based upon the monolithic design, called the monolithic-modularized algorithm, has been implemented in OSes to allow for easier debugging, scalability and better performance over the standard monolithic approach. In a

monolithic-modularized OS, the functionality is integrated into a single executable file that is made up of modules, separate pieces of code reflecting various OS functionality.

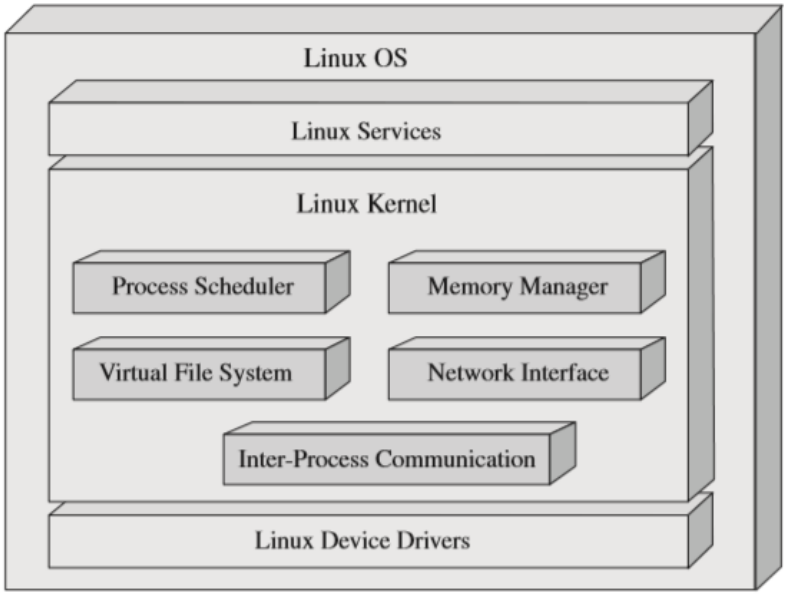


Figure 3.5. Linux OS block diagram

The embedded Linux operating system is an example of a monolithic-based OS, whose main modules are shown in figure 3.5. The Jbed RTOS, MicroC/OS-II, and PDOS are all examples of embedded monolithic OSes.

In the layered design, the OS is divided into hierarchical layers (0...N), where upper layers are dependent on the functionality provided by the lower layers. Like the monolithic design, layered OSes are a single large file that includes device drivers and middleware. While the layered OS can be simpler to develop and maintain than a monolithic design, the APIs provided at each layer create additional overhead that can impact size and performance. DOS-C(FreeDOS), DOS/eRTOS, and VRTX are all examples of a layered OS.

An OS that is stripped down to minimal functionality, commonly only process and memory management subunits as shown in Figure 9-6, is called a *client-server OS,* or a *microkernel.* (Note: a subclass of microkernels are stripped down even further to only process management functionality, and are commonly referred to as nanokernels.) The remaining functionality typical of other kernel algorithms is abstracted out of the kernel, while device drivers, for instance, are usually abstracted out of a microkernel entirely, as shown in Figure 3.6. A microkernel also typically differs in its process management implementation over other types of OSes.

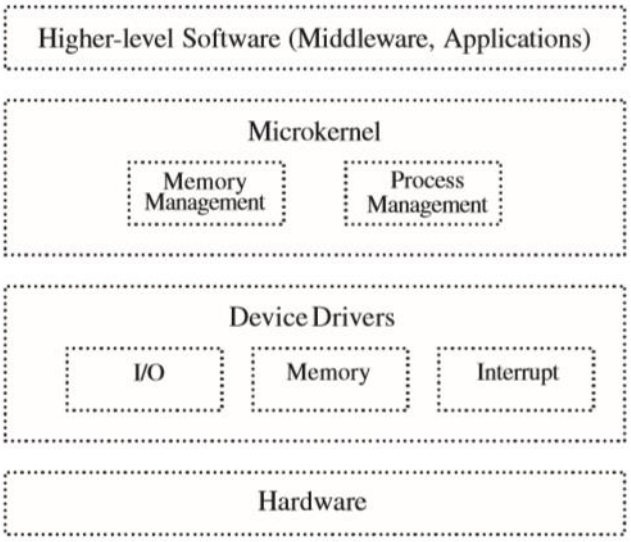


Figure 3.6. Microkernel-based OS block diagram

The microkernel OS is typically a more scalable (modular) and debuggable design, since additional components can be dynamically added in. It is also more secure since much of the functionality is now independent of the OS, and there is a separate memory space for client and server functionality. It is also easier to port to new architectures. However, this model may be slower than other OS architectures, such as the monolithic, because of the communication paradigm between the microkernel components and other "kernel-like" components. Overhead is also added when switching between the kernel and the other OS components and non-OS components (relative to layered and monolithic OS designs). Most of the off-theshelf embedded OSes—and there are at least a hundred of them—have kernels that fall under the microkernel category, including: OS-9, C Executive, vxWorks, CMX-RTX, Nucleus Plus, and QNX.

3.4 Common Operating Systems for Embedded Systems

Some popular operating systems used in embedded systems are briefly described in the following sections.

3.4.1 Embedded Linux

Almost everyone in the computer business knows the history of Linux - started in 1991 by Linus Torvalds as a simple hobby project, grown-up to a full-featured UNIX-like operating system. The name Linux is interchangeably used in reference to the Linux kernel, a Linux system or a Linux Distribution. Strictly speaking, Linux refers only to the kernel, but in colloquial language use of Linux means usually a Linux system. Such systems may be custom built (from the sources) or can be based on an already available binary distribution such as Ubuntu, Debian or Novell SUSE. Linux was developed as a (more or less) POSIX-conform general purpose operating system. There are still some issues that do not fully comply POSIX standards such as threading.

Linux is a multi-user system, which is suitable for any kind of application (multi-functional). Thus it is a big system that needs lots of resources in terms of memory and processing power and the scheduler is based on ”fairness” instead of real-time aspects. It seems to be the direct opposite of an operating system for embedded systems. A typical desktop Linux installation usually needs several hundreds of megabytes of disk space and at least 32 MB RAM. Embedded targets are often limited to very few megabytes of flash or ROM and only some megabytes of RAM. But due to the modularity and scalability of Linux it can be adapted to fit almost any embedded system. Much of the several hundred megabytes of the desktop distribution are composed of documentation, desktop utilities etc. and can be omitted as they are unnecessary for an embedded target.

It is absolutely possible to build a fully-functional Linux system needing less than 2 MB of non-volatile memory. Even the kernel itself is highly configurable and it is possible to remove unneeded kernel functionality with the assistance of several built-in frontends. Linux is available for almost every 32-bit architecture and many 64-bit architectures. For a list you may look in the directory arch/ in the Linux sources. Even some 16-bit x86-processors (e.g. 8086 and 286) are supported by a project called ELKS (http://elks.sourceforge.net/).

With Linux, all development tools and OS components (including the sources) are available free of charge and any royalties are prevented by the licenses.

Linux first became a viable choice for embedded devices around 1999. That was when Axis (www.axis.com) released their first Linux-powered network camera and Tivo (www.tivo.com) their first DVR (Digital video recorder). Since 1999, Linux has become ever more popular, to the point that today it is the operating system of choice for many classes of product. As of this writing, in 2015, there are about 2 billion devices running Linux. That includes a large number of smart phones running Android, set top boxes and smart TVs and WiFi routers. Not to mention a very diverse range of devices such as vehicle diagnostics, weighing scales, industrial devices and medical monitoring units that ship in smaller volumes. So, why does your TV run Linux? At first glance, the function of a TV is simple: it has to display a stream of video on a screen. Why is a complex Unix-based operating system like Linux necessary?

The simple answer is Moore’s Law: Gordon Moore, co-founder of Intel stated in 1965 that the density of components on a chip will double every 2 years. That applies to the devices that we design and use in our everyday lives just as much as it does to desktops, laptops and servers. A typical SoC (System on Chip) at the heart of current devices contains many function block and has a technical reference manual that stretches to thousands of pages. Your TV is not simply displaying a video stream as the old analog sets used to. The stream is digital, possibly encrypted, and it needs processing to create an image. Your TV is (or soon will be) connected to the Internet. It can receive content from smart phones, tablets and home media servers. It can be used to play games. And so on and so on. You need a full operating system to manage all that hardware.

Here are some points that drive the adoption of Linux:

- Linux has the functionality required. It has a good scheduler, a good network stack, support for many kinds of storage media, good support for multimedia devices, and so on. It ticks all the boxes.
- Linux has been ported to a wide range of processor architectures, including those important for embedded use: ARM, MIPS, x86 and PowerPC.
- Linux is open source. So you have the freedom to get the source code and modify it to meet your needs. You, or someone in the community, can create a board support package for your particular SoC, board or device. You can add protocols, features, technologies that may be missing from the mainline source code. Or, you can remove features that you don't need in order to reduce memory and storage requirements. Linux is flexible.
- Linux has an active community. In the case of the Linux kernel, very active. There is a new release of the kernel every 10 to 12 weeks, and each release contains code from around 1000 developers. An active community means that Linux is up to date and supports current hardware, protocols and standards.
- Open source licenses guarantee that you have access to the source code. There is no vendor tie-in.
- There is no vendor, no license fees, no restrictive NDAs, EULAs, and so on. Open source software is free in both senses: it gives you the freedom to adapt it for our own use and there is nothing to pay.

For these reasons, Linux is an ideal choice for complex devices. But there are a few caveats that should be mentioned here. Complexity makes it harder to understand. Coupled with the fast moving development process and the decentralized structures of open source, you have to put some effort into learning how to use it and to keep on re-learning as it changes.

3.4.1.1 Elements of embedded Linux

Every project begins by obtaining, customizing and deploying these four elements: Toolchain, Bootloader, Kernel, and Root filesystem.

Toolchain

The toolchain is the first element of embedded Linux and the starting point of your project. It should be constant throughout the project, in other words, once you have chosen your toolchain it is important to stick with it. Changing compilers and development libraries in an inconsistent way during a project will lead to subtle bugs.

Obtaining a toolchain can be as simple as downloading and installing a package. But, the toolchain itself is a complex thing. Linux toolchains are almost always based on components from the GNU project (http://www.gnu.org). It is becoming possible to create toolchains based on LLVM/Clang (http://llvm.org).

Bootloader

The bootloader is the second element of Embedded Linux. It is the part that starts the system up and loads the operating system kernel. When considering which bootloader to focus on, there is one that stands out: U-Boot. In an embedded Linux system the bootloader has two main jobs: to start the system running and to load a kernel. In fact the first job is in somewhat subsidiary to the second in that it is only necessary to get as much of the system working as is necessary to load the kernel.

Kernel

The kernel is the third element of Embedded Linux. It is the component that is responsible for managing resources and interfacing with hardware, and so affects almost every aspect of your final software build. Usually it is tailored to your particular hardware configuration.

The kernel has three main jobs to do: to manage resources, to interface to hardware, and to provide an API that offers a useful level of abstraction to user space

Root file system

The root filesystem is the fourth and final element of embedded Linux. The first objective is to create a minimal root file system that can give us a shell prompt. Then using that as a base we will add scripts to start other programs up, and to configure a network interface and user permissions. Knowing how to build the root file system from scratch is a useful skill.

3.4.2 Microsoft Windows Systems (Windows CE)

Windows CE (WinCE) is an operating system for minimalistic computers and embedded systems. It is not a smaller version of a desktop Windows, instead, it is a distinctly different kernel. It supports Intel x86 and compatibles, MIPS, ARM, and Hitachi SuperH processors.

Windows CE is optimized for devices that have minimal storage - a Windows CE kernel may run in under a megabyte of memory. Windows CE conforms to the definition of a real-time operating system, with deterministic interrupt latency. It supports 256 priority levels and uses priority inheritance for dealing with priority inversion. Similar to Linux, Windows CE forms only the kernel of the OS. By adding extra software such as a graphical user interface, it becomes a ”complete” operating system called e.g. Windows Mobile.

3.4.2.1 Windows CE Design Goals

The Windows CE design is based on the following goals:

- **Be compatible with Windows.** Windows CE supports the familiar Win32 programming model, and it exports a subset of the Win32 Application Programming Interface (API). Like Windows NT Workstation, Windows CE is a pre-emptive, multitasking operating system that can run many processes at the same time. The file formats for Windows CE executables and libraries are the same as those for Windows 95 and Windows NT. Although Windows CE cannot run arbitrary Windows-based applications, it is relatively easy to port most small Windows-based applications to Windows CE.
- **Provide the right system for many different devices.** Windows CE is flexible enough to accommodate a range of hardware and software features on a variety of devices. But, it does not overload each device by always including support for every possible feature. Instead, Windows CE is "componentized." Each Windows CE–based device contains only the pieces of the operating system that are absolutely essential. This "componentization" allows a system designer to pick and choose features at a very low level. For example, Windows CE systems could be built with no display, or with an alphanumeric LCD, or with a VGA monitor, with just the right amount of operating system support for each. The non-display system will be much smaller than the VGA system, all other things being equal.
- **Consume small amounts of RAM.** All of the components of Windows CE can execute in place in ROM, reducing the need for more expensive RAM. There is no requirement for flash memory or a disk drive (although both are supported).

- **Connect easily to the Internet and to PCs and servers running Windows 95 and Windows NT.** Windows CE communication components provide system designers with the ability to connect Windows CE–based devices to the Internet and corporate networks as well as to other Windows CE–based devices.
- **Leverage existing knowledge.** Development for Windows CE is done on a Windows NT based PC with the same Interactive Development Environment (IDE) and programming model used to develop for Windows. So, the hundreds of thousands of existing developers for Windows already know how to develop for the Windows CE operating system. The Windows CE IDE also adds an emulator and remote source level debugging tools to make device driver and application development easy.
- **Be processor and system hardware agnostic.** Windows CE works on a variety of 32-bit microprocessors, and it does not require a particular hardware architecture. An embedded system designer can adapt Windows CE to many different hardware products.

3.4.2.2 Windows CE 1.01 Architecture and Features

Windows CE is a modular operating system composed of several major software elements. There are well-defined, Win32-compatible interfaces between the elements. Each major element comprises many small feature-level components, and the embedded system designer has the ability to include or exclude feature level components as needed. The following figure shows the elements of the Windows CE 1.01 architecture:

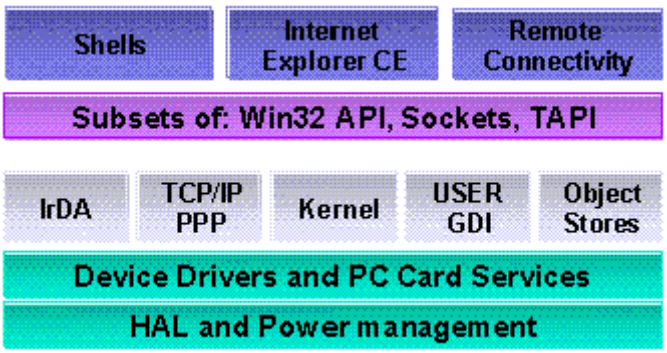


Figure 3.7 Windows CE architecture

Starting from the bottom, the principal elements that make up the Windows CE operating system are: the hardware abstraction layer (HAL), which includes power management; device drivers and PC card services; the Windows CE kernel, USER, GDI, file systems and databases; the IRDA and TCP/IP communications protocols; the APIs; the remote connectivity; the Microsoft Internet Explorer for Windows CE; and the shell(s). Many embedded systems will not have all of these elements, of course. The following descriptions of each Windows CE module highlight the critical features of the operating system.

HAL and Power Management

The HAL allows embedded systems designers to adapt Windows CE to their hardware platform and to provide hardware-specific power management functions. Windows CE does not require a standard memory map and interrupt structure, as the PC does. Instead, designers write small interrupt service routines in the Windows CE HAL that allow the

operating system to run in whatever hardware configuration is best for the device. Windows CE power management functions include "instant on," allowing Windows CE devices to be powered off and then turned on instantly if the device has non-volatile (battery backed) RAM.

Device Drivers and PC Card Services

Windows CE–based devices can contain two types of device drivers: built-in drivers for hardware that is always present in the device—like the keyboard on a handheld PC—and run-time installable drivers for plug-in peripherals. Windows CE 1.01 directly supports many types of devices, such as keyboards, mice, touch panels, serial ports, modems, displays, PC Card slots, audio processors, speakers, parallel ports, ATA disk or flash card drives, and the like. Embedded systems designers can easily add new device types.

For all supported device drivers, Windows CE has a well-defined set of Device Driver Interfaces (DDIs) to which designers write. Device drivers run as normal processes in the system, with access to all operating system services. This allows the interrupt service routines that typically "wake up" device drivers to be very simple and fast: the driver thread does almost all the work. Windows CE provides a subset of the Windows PC Card (previously known as PCMCIA) and Socket services, allowing PC Cards such as wireless or wireline modems and flash memory cards to be used. Not only can designers use these interfaces for their own cards, but other third-party hardware and software vendors can develop new add-in PCMCIA devices or write Windows CE drivers for their existing PC Card devices. This can include anything from VGA cards to GPS. A Windows CE Device Driver Kit (DDK) is available for this purpose. Using the same PC Card interfaces, Windows CE also supports other plug-in card form factors, such as the Minicard and Compact Flash (CF) standards.

Kernel

The Windows CE kernel was written specifically for non-PC devices. It implements the Win32 process, thread, and virtual memory model. Like Windows NT, it has a preemptive, priority-based scheduler, and it provides a rich variety of synchronization primitives, including semaphores, mutexes, and events. The Windows CE kernel supports execution of programs in place in ROM or RAM. It also implements demand paging for running applications that are stored compressed and/or that are stored on a media that does not support execution in place. The kernel has a low interrupt service routine and low thread latency (threads can be scheduled and switched to in less than 100 microseconds on handheld PCs running at 33 MHz). This allows Windows CE to be used in many types of real-time systems.

USER and GDI

The USER and GDI components of Windows CE provide the basic functionality for the user interface if there is one. In Windows CE 1.01, a grayscale display is supported by GDI. Windows CE USER exports the same primary Win32 features as are provided by the Windows version of USER: overlapping windows, event management, user interface controls, dialog boxes, interprocess communication, and so on. USER also provides the functions needed for internationalization: UNICODE character manipulation and locale NLS APIs. Windows CE 1.01 is fully usable for many different language applications. GDI and USER are both very flexible in terms of componentization: for example, a designer can

choose to keep the windowing system while excluding specific controls such as list views or buttons.

Object Stores

The Windows CE Object Store components provide persistent data and application storage. Persistent data is typically contained in non-volatile memory, such as battery-backed RAM or flash memory. When using RAM that is also used for running applications, the embedded system designer can adjust the amount of RAM used by the Object Store. (Users can also be provided the ability to do this adjustment.) The Object Store is made up of three classes of components: file systems, the registry, and databases. In Windows CE 1.01, there are three types of Windows CE file systems: a ROM-based file system, a RAM-based file system, and a FAT file system for disk drives, flash memory, and SRAM cards. These file systems all look like file systems on a Windows PC, and they are all accessed via the Win32 file system APIs. Similarly, the Windows CE registry exports the Win32 registry functions, and it is used by applications and the system to record and access run-time and other data. The Windows CE database functionality does not have a corollary on Windows NT or the Windows 95 operating system. The database provides object storage, access, and sorting. These were used initially in the HPC for such things as address books and appointments. One key feature of databases, the file system, and the registry is that they are all protected against unforeseen reset (such as caused by a main power interruption in systems like the handheld PC that have a backup battery). If a reset occurs during a write to the Object Store, Windows CE will ensure that the store is not corrupted by either completing the operation when the system restarts or by reverting to the known good state before the interruption.

TCP/IP, PPP, and IrDA

Windows CE 1.01 communications protocols provides connectivity to desktop Pcs running Windows, the Internet, and other Windows CE based devices. The primary communication protocol stack used for connections is the standard Internet protocol, TCP/IP, coupled with PPP. The TCP/IP and PPP protocols are used when directly connected to a Windows-based PC through the PC's Direct Cable Connection feature, as well as when a Windows CE-based device is communicating over a modem to the Internet or a corporate network. Windows CE also includes the standard infrared stack, IrDA.

APIs

Windows CE exports a subset of the Win32 API set. This includes over 500 of the most used Win32 APIs. Many major applications for Windows CE have already been written using this subset, including the pocket Office and PIM applications that ship with the handheld PC. The subset is not extremely limiting. Along with communications protocols, Windows CE provides several of the familiar Windows communications APIs, including Windows Sockets, TAPI, and Unimodem. The Sockets API is the application interface for TCP/IP and IrDA. TAPI and Unimodem provide the dial-out functionality needed by modem-based applications, like the handheld PC Inbox and Internet Explorer.

Remote Connectivity

To enable connectivity applications, Windows CE also exports a Remote Access API (RAPI) to a connected Windows-based PC over Sockets. An application on the PC, such as the handheld PC Explorer, uses RAPI to manipulate the connected Windows CE-based device's Object Store. Files can moved back and forth, databases can be updated, and the registry can be viewed and modified over the connection. By using TAPI functions to dial out, this connection can be done remotely over a phone line.

Internet Explorer for Windows CE

As in Windows 95 and Windows NT operating systems, Internet connectivity is a module of the Windows CE operating system. The Windows CE Internet Explorer is both an HTML control and a user interface similar to that of Windows Internet Explorer. Although primarily intended for devices with a display, components of the Microsoft Internet Explorer can be used in embedded systems that need to get information from the Internet even if "browsing" or showing WWW pages to the user is not a requirement.

Shells

Windows CE 1.01 comes with a shell component similar to that of Windows 95, as shown in the figure below (taken from the handheld PC). However, many embedded systems that use Windows CE will not have this shell, or may have no shell at all (or even no display at all). For those systems needing a Windows look, the Windows CE shell provides a very similar experience to Windows 95 and Windows NT 4.0.

Together, the Windows CE operating system components provide a powerful Windows-compatible software platform in a small package.

3.4.3 Symbian

Symbian OS is the successor of 32-bit EPOC Platform from Psion. Symbian is currently owned by Ericsson (15.6%), Nokia (47.9%), Panasonic (10.5%), Samsung (4.5%), Siemens AG (8.4%), and Sony Ericsson (13.1%). All of the owners are (or were) manufacturers of mobile phones. [12] Symbian is structured like many desktop operating systems with pre-emptive multitasking, multithreading and memory protection. Its kernel is a microkernel architecture, which means that only the minimum necessary is within the kernel. Things like networking or file system support have to be provided by another layer called base layer. Between base layer and user software are system libraries. The most important user interfaces based on Symbian are S60 (Nokia) and UIQ (Sony Ericsson). The major advantage of this operating system is the fact that it was built for handheld devices with limited resources that may be running for months or years. It has programming idioms such as descriptors and a cleanup stack and other techniques in order to conserve RAM and avoid memory leaks. There are similar functions to save disk space (flash memory). All Symbian OS programming is event-based and the CPU is switched off when applications are not directly dealing with an event. This is achieved through a programming idiom called active objects. Correct use of these techniques helps ensure longer battery life. Symbian OS is solely employed in mobile phones (i.e. smartphones) and becoming obsolete now.

3.5 Real-Time Operating Systems

Real-time operating systems (RTOS) are operating systems intended for real-time applications. That is to say, a RTOS guarantees deadlines to be met generally (soft real-time) or deterministically (hard real-time). An RTOS facilitates the creation of a real-time system, but does not guarantee the final result will be real-time - this requires correct development of the software. An RTOS will typically use specialised scheduling algorithms in order to provide the real-time developer with the tools necessary to produce deterministic behaviour in the final system. There are numerous proprietary and free RTOSes available. See the following list for some examples.

• VxWorks is the most popular commercial RTOS. Like most other RTOSes it includes a multitasking kernel with pre-emptive scheduling and fast interrupt response, extensive inter-process communications, synchronization facilities and a file system. [14] Major distinguishing features of VxWorks include efficient POSIX-compliant memory management, multiprocessor facilities, a shell for user interface, symbolic and source level debugging capabilities and performance monitoring.

• QNX is a commercial POSIX-conform RTOS. It uses a microkernel, enabling the user (developer) to turn off any functionality he does not require. It offers features such as fault tolerance, pre-emptive multitasking and runtime memory protection. The system is quite fast and small, in a minimal fashion it fits on a single floppy disk.

• RTEMS is a free RTOS designed to support various open API standards including POSIX and uITRON. It was originally planned to be used for missiles and other military systems. RTEMS closely corresponds to POSIX Profile 52 which is ”single process, threads, file system” - it does not provide any form of memory management or processes. There is only a single process with multiple threads running on an RTEMS system. No services like memory mapping, process forking, or shared memory are offered.

• RTAI, Xenomai and RTLinux are extensions to the Linux kernel in order to allow Linux to meet real-time requirements. These systems run the Linux kernel as a low-priority task, so higher priority real-time tasks can interrupt the execution of the Linux kernel.

3.6 Operating systems for IoT applications

An operating system is the core program of IoT projects. Modern IoT operating system uses cloud computing technology to control IoT devices anywhere from the world. With a low memory footprint and higher efficiency, each operating system represented below can fulfill the user’s requirements.

3.6.1 Contiki

Invented in 2002, Contiki is an open-source IoT operating system particularly popular for low power microcontrollers and other IoT devices to run effectively using Internet protocol IPv6, and IPv4. These operating systems support wireless standard CoAP, 6lowpan, RPL. Mostly this IoT OS is very suitable for low powered internet connectivity.

Insight of Contiki

- Multitasking ability contains a built-in internet protocol suite.
- Only 10kb of RAM and 30 kb of ROM is needed to run this Operating system.
- The core language of this operating system is C language. Before the real-time deployment of IoT products, a simulator called Cooja test each IoT product.
- Both commercial and non-commercial purposes exist to use Contiki.

- Contiki programming model uses Protothread memory-efficient programming.
- Manageable by hardware platform, for example, TI MSP430x, Atmel AVR, Atmel Atmega128rfa1.

3.6.2 Android Things

Android Things is an IoT Operating System, and it is an invention of Google. As its previous name was Brillo, experts said that "Brillo is derived from android." It can run on low power and supports Bluetooth and WiFi technology. Android Things aims to remove all obstacles and simplify IoT development. If Android Things runs well in the market, we expect that Google will launch an IoT app store.

Insight of Android Things

- Android Things uses only 32-64 Kb of RAM as it is a lightweight operating system.
- Along with Android Things, Google announces it will provide a communication network protocol called Weave.
- As Android Things and Weave is connected, so it is possible to detect each IoT device by android smartphone.
- Developer kit can help to test, build, and debug each IoT solution.
- Android Things is an open-sourced technology and regularly updates every 6 weeks.
- As source code is not available, below is an example of how things are built for android things.

3.6.3 Riot

Riot is one of the free open source IoT operating systems built for IoT services. RioT has a huge development community, and it was released under an unclonable GNU Lesser General Public License. For these two reasons, RioT is called Linux of the IoT world. Academics, hobbyists, and different companies put their contribution together to develop Riot Operating System.

Insight of Riot

- With low power use capacity Riot is built upon microkernel architecture with C, C++ language.
- This open source IoT os supports full multithreading and SSL/TSL libraries, for example, wolfSSL.
- The processor of Riot is 8bit, 16bit and 32 bit.
- A port of this operating system makes it possible to run as Linux or macOS process.
- Provides content-centric networking and network protocols such as TCP, UDP, and CoAp.

3.6.4 Huawei LightOS

In 2015, the Chinese tech giant Huawei released an IoT operating system, and its name is LightOS. IoT OS of Huawei provides a standard API for the diverse IoT fields.

LightOS is a secure, interoperable, low-power operating system. LightOS uses middleware to remove the extra cost for the development of IoT devices. According to name, LightOS contains the smallest kernel (6kb) comparing with other operating systems.

Insight of Huawei LightOS

- Various network access protocols of LightOS supports diverse IoT products. For example, NB-IoT, Ethernet, Bluetooth, Wifi, Zigbee, and more.
- For security purposes, LightOS provides remote upgrades for terminals, two-factor authentication, and encrypted transmission.
- Suitable for operating system components such as queue, memory, time and task management, and more.
- According to a report, Huawei exports 50 million IoT devices, each containing LightOS.
- Accumulation of static function, low power consumption, and real-time data representation are the core features of the LightOS kernel.

3.6.5 TinyOS

TinyOS is a component-based open-source operating system. The core language of TinyOS is nesC which is a dialect of C language. TinyOS is popular among developers for its memory optimization characteristics. A component of TinyOS neutralizes some abstractions of IoT systems, for example, sensing, packet communication, routing, etc. The developer group of this IoT Operating System is TinyOS Alliance.

Insight of TinyOS

- ESTCube-1 is a space program that uses this operating system.
- Network protocols, sensor drivers, data acquisition tools are part of component libraries.
- Mostly use wireless sensor networks as its architecture designed in that way.
- Large scale use of this operating system contributes to simulate algorithms and protocols.

3.6.6 Windows IoT

Windows 10 IoT is a family of Windows 10 operating systems for the IoT sector. Besides Windows IoT divided into two-part. One is Windows 10 IoT core to support small embedded devices. Another one is Windows 10 IoT Enterprise for the industrial perspective.

Insight of Windows IoT

- IoT enterprise operating system runs on the ARM processor.
- It leverages IoT connectivity, cloud experience, and offers various organizations to connect with IoT devices.
- Windows IoT core provides manageability like Windows 10 operating system, although it acts like an app.
- Windows IoT core does not support Cortana and FileOpenPicker which is available in Windows 10.
- With the hybrid kernel, this is not an open-source operating system.

3.6.7 Raspbian

Raspberry Pi is one of the most used devices for IoT development, and Raspbian is its own operating system. Raspbian is highly flexible for Raspberry Pi lines CPUs. Raspbian provides a huge number of pre-installed IoT software for general use, experimental, educational purposes, etc. This is Debian based IoT Operating System for all models of Raspberry Pi.

Insight of Raspbian

- Active development of Raspbian is still going on as demand for this operating system is increasing.
- Raspbian Buster and Raspbian Strech are two versions of the Raspbian operating system.
- Main desktop environment is PIXEL which is PI improved x-window environment.
- Raspbian uses a computer algebra program "Mathematica" and a version of "Minecraft."
- The kernel is similar to the Unix kernel.

3.6.8 Amazon FreeRTOS

Amazon FreeRTOS is an open-source microcontroller-based operating system for IoT development invented by Amazon. Enriched software libraries make it easy to connect with small IoT devices. This IoT Operating System uses the cloud service of Amazon Web Service called AWS IoT Core to run the IoT applications. The memory footprint is only 6-15kb which makes it a more adaptable small powered microcontroller.

Insight of Amazon FreeRTOS

- Code modularity, task prioritization features help to meet the processing deadline with power optimization.
- The use of the standard generic access profile and generic attribute profile (GAP) via Bluetooth low energy makes it more effective.
- Amazon invested a lot of money behind the development of IoT data security.
- Users can maintain diverse architecture with this technology.
- IoT device tester ensures the possibility of IoT devices to integrate with cloud service.
- It has become a standard of the microcontroller-based operating systems in the last few years.

3.6.9 Mbed OS

For the development of IoT embedded products, Mbed operating system uses an ARM processor. This is a free, open-source operating system focusing on IoT projects. A significant number of connectivity options include Wifi, Bluetooth, 6LowPan, Ethernet, Cellular, RFID, NFC, Thread, and more. Multilayer security of this IoT operating system provides profound reliability to customers.

Insight of Mbed OS

- The developer can make a prototype of IoT applications with the use of ARM cortex M-based devices.
- From the rich library, required supporting updates automatically adds to IoT applications.
- Mbed OS API can keep your code clean and portable.
- Uses SSL and TSL security protocols for the security of the online application.
- It provides a large number of code examples to show how to integrate API on each application.

Open-source IoT operating systems are giving us a platform to check the functionality of IoT products in an easy manner. Those IoT operating system mentioned above is mostly open-source and comes free of charges. We hope that the modern IoT Operating System with all features will accelerate the changes in technology and bring some innovative IoT Trends which ultimately will shape our near future.

References

[1] Charles Crowley," Operating Systems: A Design-Oriented Approach", MGH, 1st Edition, 2001.

[2] Christopher Hallinan, "Embedded Linux Primer: A practical Real-World approach", Prentice Hall, 2nd Edition, 2011.

[3] Daniel P. Bovet and Marco Cesati, "Understanding the Linux Kernel, O'Reilly, 3rd Edition, 2005.

[4] John Madieu, "Linux Device Drivers Development: Develop customized drivers for embedded Linux", Packt Publishing, 1st Edition, 2017.

[5] Jonathan Corbet, Alessandro Rubini, Greg Kroah, "Linux Device Drivers", O'Reilly, 3rd Edition, 2005.

Exercise Questions

1. Explain the concepts of real-time operating system and its needs/necessity?
2. What do you understand by Real-Time task? Explain in detail?
3. What are the qualities of good RTOS? Why RTOS is the only solution when it comes to deal with critical task and minimal latency.
4. Differentiate Hard, soft and firm real time systems. State the one example of each?
5. Write the characteristics of real-time operating system.
6. Is it any difference between RTOS and EOS? If yes/no give valid reasons in support to your answer.
7. What are the different types of OS platform? Compare few of them.
8. How RTOS can be used for industrial and defense purpose?
9. How will you differentiate between latency and delay?
10. What is kernel? Write its characteristics. Also explain the difference between kernel and Operating System.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – IV - EMBEDDED OS AND DEVICE DRIVERS – SECA5204

4. INTRODUCTION TO DEVICE DRIVERS

Unix/Linux kernel fundamentals-Process Scheduling - Kernel Synchronization, I/O devices - Architecture - Character, Block Device handling, file systems - The Ext2 file System - The Virtual File System and peripheral devices.

4.1 Unix/Linux kernel fundamentals

LINUX is an operating system or a kernel distributed under an open-source license. Its functionality list is quite like UNIX. The kernel is a program at the heart of the Linux operating system that takes care of fundamental stuff, like letting hardware communicate with software. The main advantage of Linux was that programmers were able to use the Linux Kernel to design their own custom operating systems. With time, a new range of user-friendly OS's stormed the computer world.

Now, **Linux is one of the most popular and widely used Kernel,** and it is the backbone of popular operating systems like **Debian, Knoppix, Ubuntu, and Fedora**. Nevertheless, the list does not end here as there are thousands of Best versions of Linux OS based on the Linux Kernel available which offer a variety of functions to the users. Linux Kernel is normally used in combination of GNU project by Dr. Richard Stallman. All mordern distributions of Linux are actually distributions of Linux/GNU

4.2 Benefits of using Linux

Linux OS now enjoys popularity at its prime, and it's famous among programmers as well as regular computer users around the world. Its main benefits are:

- It offers a **free operating system**. You do not have to shell hundreds of dollars to get the OS like Windows.
- Being open-source, anyone with programming knowledge can modify it.
- It is easy to learn Linux for beginners
- The Linux operating systems now offer **millions of programs/applications and Linux softwares to choose from**, most of them are free!
- Once you have Linux installed you no longer need an antivirus! Linux is a highly secure system. More so, there is a global development community constantly looking at ways to enhance its security. With each upgrade, the OS becomes more secure and robust
- Linux freeware is the OS of choice for Server environments due to its stability and reliability (Mega-companies like Amazon, Facebook, and Google use Linux for their Servers). A Linux based server could run non-stop without a reboot for years on end.

- UNIX is called the mother of operating systems which laid out the foundation to Linux. Unix is designed mainly for mainframes and is in enterprises and universities. While Linux is fast becoming a household name for computer users, developers, and server environment. You may have to pay for a Unix kernel while in Linux it is free.

- But, the **commands used on both the operating systems are usually the same.** There is not much difference between UNIX and Linux. Though they might seem different, at the core, they are essentially the same. Since **Linux is a clone of UNIX**. So learning one is same as learning another.

4.3 Process scheduling in Linux

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy. Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

4.3.1 Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues −

- **Job queue** − This queue keeps all the processes in the system.

- **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.

- **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.

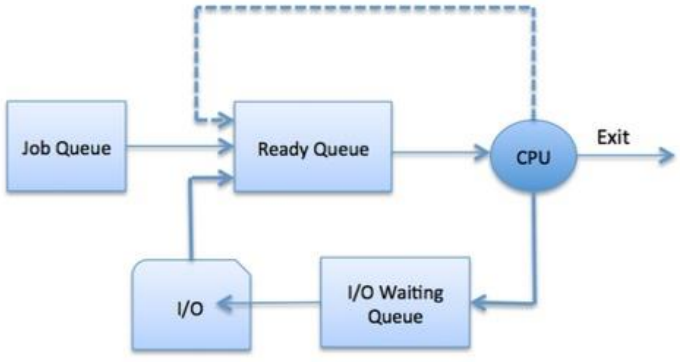


Figure 4.1: Process-scheduling queue

The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

4.4 Schedulers in Linux

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types −

- Long-Term Scheduler
- Short-Term Scheduler
- Medium-Term Scheduler

Long Term Scheduler

It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

Short Term Scheduler

It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them. Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

Medium Term Scheduler

Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.

A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to

the secondary storage. This process is called **swapping**, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

4.4.1 Comparison among Scheduler

The important features of the three types of schedulers in Linux is compared in table 4.1.

Table 4.1 : Schedulers in Linux

S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

4.5 Context Switch in Linux

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features. When the scheduler switches the CPU from executing one process to execute another, the

state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

- Program Counter
- Scheduling information
- Base and limit register value
- Currently used register
- Changed State
- I/O State information
- Accounting information

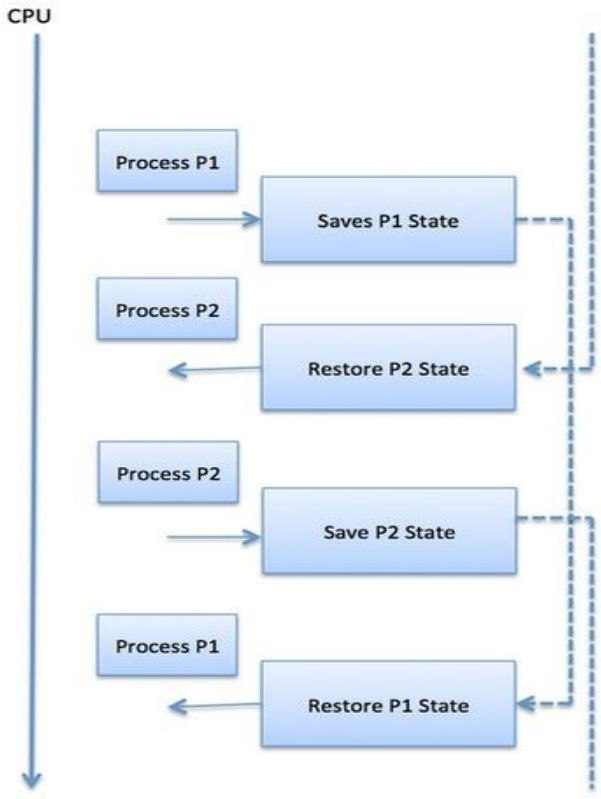


Figure 4.2 Context Switching in Linux

4.6 Kernel Synchronization

If a resource is being shared between multiple process at same time then we need to provide protection from concurrent access because if two or more thread will access and manipulate the data at same time then thread may overwrite each other's changes or access the data while it's in an inconsistent state. The process of maintaining multiple access to shared data at same in safe manner is called synchronization. In this chapter we will discuss what is synchronization , why it's needed, what is critical region and race conditions ,what is deadlock. In next chapter we will discuss various synchronization techniques available in Linux Kernel.

4.6.1 Critical Region and race conditions

Piece of code which access the shared data is called critical Region or critical section . If multiple process execute this code at same time then there might be chances of inconsistency of data. Hence System programmer need to ensure that two or more process should not execute critical section at same time. When two or more threads executing same critical section at same time is called **Race condition.**So to avoid race condition we need to use synchronization . Now let's understand with example that why we need protection.

Consider a global integer variable **i** and a simple critical section which tries to increment the variable. Let's assume two thread A and B tries to enter in critical section at same time and initial value of **i** is 5. Below is the expected operation according to the expected execution 5 is incremented twice and the result value of i is 7.

Thread 1	Thread 2
Get value of i(5)	
Increment value of i(5>>6)	
Write back value of i in memory.	
	Get value of i (6)
	Increment value of i(6>>7)
	Write back value of i in memory.

Figure 4.4 Critical section in Linux

As two threads are executing in critical section at same time then there might be chance of below execution.

Thread 1	Thread 2
Get value of i(5)	Get value of i (5)
Increment value of i(5>>6)	- - - - - - -
- - - - - - -	Increment value of i(5>>6)
Write back value of i in memory (6).	- - - - - - - -
- - - - - - - -	Write back value of i in memory(6).

Figure 4.5 Critical section leading to erroneous result

As we can see in above image thread 1 and 2 modified the same variable i at the same time, there might be outcome that result value of i will be 6. This is one of the simple example of a critical region and hence the outcome of it's also less harm full. But in kernel, we might come across very danger situation which can also effect performance of system. So to avoid these kinds of problem, programmer need to restrict simultaneous access of critical region by multiple process. One possible solution of above problem is to performing reading, incrementing and writing back value to memory in a single instruction as below.

Thread 1	Thread 2
Increment and store value of i(5>>6)	- - - - - - - - - -
- - - - - - - - - -	Increment and store value of i (6>>7)

Figure 4.6 Solution to Critical section problem

Atomic operation are special operation in which processor can simultaneously perform multiple operation like read and write etc in same bus operation . Atomic instructions are supported by all the processors. But by only usage of atomic operation , our problem of race condition doesn't get solve in every case. Let's see little complex example:

- Assume some kernel buffer which contains some important data. This buffer is modified or filled by some producer process A and data is read or taken by consumer thread B . If any time producer is updating or manipulating the buffer at the same time when consumer thread is reading the buffer then there are chances of inconsistency of data . By using the above discussed atomic operation solution we cannot avoid critical section in this case. We need to find out a different solution in which can restrict a processor to read and another processor to write at same time.

- One way is needed which make sure that only one thread can manipulate the data structure at a time. one mechanism for preventing access to a resource while any other is in critical section and accessing that resource is **lock.**

- A lock works like a lock on door and room behind the door is critical section . If a process tries to enter in the room the while room is empty then that process gets permission to enter in the room and lock the door from inside. Perform its task inside the room and after finishing the work , it leaves the door and unlocks the door. If another process tries to enter in the room while room is already occupied by some other process then it must wait for the thread inside to exit the room and unlock the door before it can enter.

- Above problem of buffer can be solved, by using of lock. Any thread, which wants to read or write the first it has to get the lock for accessing the buffer. If lock is granted then only process can access the buffer otherwise thread has to wait till the time when lock is available .Linux kernel provide different way of locking mechanism. we will discuss various kinds of mechanism provided in next tutorial.

4.6.2 Cause of concurrency

In linux kernel there are various reason for concurrency to occur:

Kernel Preemption

- As Kernel got preemptive after 2.6 version . It means one thread can preempt another task. The task which is newly scheduled might be executing in same critical region.

Interrupts

- As interrupt occurs asynchronously any time , which interrupt the execution of current task.

Softirqs and tasklets

- The can raise or schedule a softirq or tasklet any time, interrupting the currently executing process.

Symmetrical multiprocessing

- Two or more processor can execute same kernel code at same time which can cause race condition.

Sleeping

- A task in kernel can sleep any time and ask kernel schedule to schedule new process.
- Before writing code developer should indentify all the possible cause of concurrency . If all the reason of concurrency well known then its very easy for Kernel developer to avoid the race conditions.

4.6.3 Knowing Which data to need protection

Developer has to identify which part of data need protection . Generally its very easy to identify the data which needs protection. All the local variable which are specific to some particular thread they don't need any protection. But all the global variable which can be accessed by other processes needs to be protected. Below are the other possible data which needs to be protected

1) Data shared between process context and interrupt context.

2) Data shared between currently executing process and new scheduled process who preempted the currently executing task.

3) Function or code which can be executed by two or more processor at same time.

4.6.4 Deadlocks

Deadlock refers to a situation in which two or more threads are waiting for each process to release a resource. Because of which neither of process get chance to proceed further as the resource for which all the process are waiting will never be available. For

example, if a process tries to acquire a lock which is already hold will result in a deadlock. As process will never release the previously held lock as its waiting for other lock to acquire. There are various techniques of avoiding the deadlock.

4.7 I/O Devices

We now look at how the kernel communicates with the hardware of a secondary storage disk to give out read/write instructions and transfer data. That is, we will be studying the design of the kernel I/O subsystem as a whole. This subsystem consists of device drivers, and device-independent code. The device drivers understand the device hardware internals and communicate with the device in a language that it follows. For example, a disk device driver would know how to instruct the disk to fetch a certain block. The device independent part of the kernel code handles generic I/O functions, e.g., higher layers of the file system stack or the networking stack.

I/O devices are of two main types: **block devices** (e.g., hard disks), and **character devices** (e.g., keyboard, network interfaces). With block devices, data is stored and addressed on the device at the granularity of blocks. That is, the kernel can request the device to store or retrieve data at a specific block number. Character devices simply accept or deliver a stream of characters or bytes to the kernel; there is no notion of a permanent block of data that can be addressed. In other words, you can perform a seek on a block device to point to a specific block number to read from or write to, while with character devices, you simply read or write at the current location in the stream.

Irrespective of the type of device the kernel is communicating with, the user programs see a uniform interface consisting of the read and write system calls on a file descriptor to communicate with I/O devices. The system calls to initiate communication with the device, however, may differ slightly. For example, one has to open a file to communicate with the disk, while one has to use the socket system call to open a network connection. In addition, the seek system call can also be used to reposition the offset to read from or write to for block devices. Finally, the ioctl system call can be used to pass any other instruction to the device that is not a simple read/write/seek/open.

Communication with an I/O device can be blocking or non-blocking, depending on the options set for the particular I/O communication. When a process chooses to block on a read or write system call, until the I/O operation completes, such an I/O is referred to as synchronous I/O. In contrast, asynchronous I/O can refer to a process using non-blocking read/write operations (possibly with polling to check for status of completion), or special system calls or libraries to enable optimized event-driven communication with I/O devices. For disk operations, a blocking read makes the most sense, since the process cannot make much progress without receiving the disk data. The write to the disk, however, can be blocking on non-blocking, depending on whether the application needs to make sure that the write has reached the disk or not. Typically, applications use blocking writes for sensitive, important data to the disk, and non-blocking writes for regular data.

A note about networking-related system calls. The socket system call creates a communication socket, using which applications can send and receive data. Every socket has associated transmit and receive buffers which store application data. The read operation on a network 13 socket is blocking by default if there is no data in the receive buffer. A process that requests a read will block until data arrives from a remote host. When the data arrives via an interrupt, the TCP/IP network stack processing is performed on the packet as part of the interrupt handling, and the application data is placed into the receive buffer of a socket. The process blocked on the read is then woken up.

A non-blocking option can be used on a socket to not block on reads, in case the applications wants to do other things before data arrives. In such cases, the application must periodically check for data on the socket, either by polling, or by using event-driven system calls like select.

When a process calls write on a network socket, the data from the transmit buffer undergoes TCP/IP processing and is handed over to the appropriate device drivers. By default, network writes do not block until the data is transmitted, and only block until data is written into the send buffer of the socket. A transport layer protocol like TCP in the kernel is responsible for in-order reliable delivery of the data, and applications do not need to block to verify that the written data has reached the remote host. Writes can, however, block if there is insufficient space in the socket's transmit buffer due to previous data not being cleared (e.g., due to TCP's congestion control). Writes on non-blocking sockets will fail (not block) if there is insufficient space in the socket buffer, and the application must try later to write. Additional system calls connect, accept, bind, and listen are used to perform TCP handshake and connection setup on a socket, and must be performed prior to sending or receiving data on a socket.

Every devices connects to the computer system at a logical port and transfers data to and from the host system via a bus (e.g., PCI bus, SATA, SCSI). Ports and buses have rigidly specified protocols on how data is marshalled. Most I/O devices have a special hardware called the device controller to control the port and bus, and manage the communication with the main computer system. Device controllers typically have two parts, one on the host and the other on the device. The host controller or the host adapter plugs into the motherboard's system-wide bus (that connects the CPU, memory, and other device controllers), and provides the port abstraction by transferring instructions and data from the CPU onto the device bus. The part of the controller that resides on the I/O device is responsible for transferring instructions/data between the bus and the actual device hardware. Some device controllers also implement intelligent optimizations on top of the simpler hardware.

4.8 Linux File Systems

Linux file system is generally a built-in layer of a Linux operating system used to handle the data management of the storage. It helps to arrange the file on the disk storage. It manages the file name, file size, creation date, and much more information about a file. Linux

file system has a hierarchal file structure as it contains a root directory and its subdirectories. All other directories can be accessed from the root directory. A partition usually has only one file system, but it may have more than one file system.

A file system is designed in a way so that it can manage and provide space for non-volatile storage data. All file systems required a namespace that is a naming and organizational methodology. The namespace defines the naming process, length of the file name, or a subset of characters that can be used for the file name. It also defines the logical structure of files on a memory segment, such as the use of directories for organizing the specific files. Once a namespace is described, a Metadata description must be defined for that particular file.

The data structure needs to support a hierarchical directory structure; this structure is used to describe the available and used disk space for a particular block. It also has the other details about the files such as file size, date & time of creation, update, and last modified. Also, it stores advanced information about the section of the disk, such as partitions and volumes.

The advanced data and the structures that it represents contain the information about the file system stored on the drive; it is distinct and independent of the file system metadata. Linux file system contains two-part file system software implementation architecture. Consider the below image in figure 4.8.

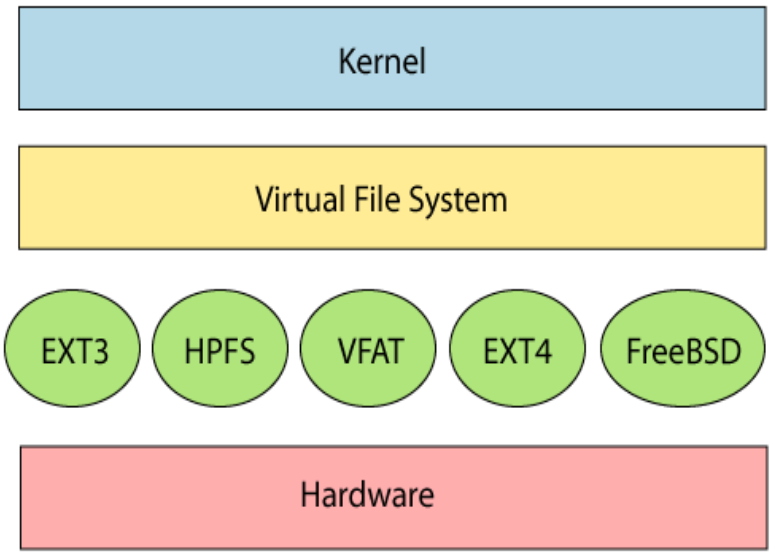


Figure 4.8: Linux File System

The file system requires an API (Application programming interface) to access the function calls to interact with file system components like files and directories. API facilitates tasks such as creating, deleting, and copying the files. It facilitates an algorithm that defines the arrangement of files on a file system. The first two parts of the given file system together called a **Linux virtual file system**. It provides a single set of commands for the kernel and

developers to access the file system. This virtual file system requires the specific system driver to give an interface to the file system.

4.8.1 Linux File System Features

In Linux, the file system creates a tree structure. All the files are arranged as a tree and its branches. The topmost directory called the **root (/) directory**. All other directories in Linux can be accessed from the root directory.

Some key features of Linux file system are as following:

- **Specifying paths:** Linux does not use the backslash (\) to separate the components; it uses forward slash (/) as an alternative. For example, as in Windows, the data may be stored in C:\ My Documents\ Work, whereas, in Linux, it would be stored in /home/ My Document/ Work.
- **Partition, Directories, and Drives:** Linux does not use drive letters to organize the drive as Windows does. In Linux, we cannot tell whether we are addressing a partition, a network device, or an "ordinary" directory and a Drive.
- **Case Sensitivity:** Linux file system is case sensitive. It distinguishes between lowercase and uppercase file names. Such as, there is a difference between test.txt and Test.txt in Linux. This rule is also applied for directories and Linux commands.
- **File Extensions:** In Linux, a file may have the extension '.txt,' but it is not necessary that a file should have a file extension. While working with Shell, it creates some problems for the beginners to differentiate between files and directories. If we use the graphical file manager, it symbolizes the files and folders.
- **Hidden files:** Linux distinguishes between standard files and hidden files, mostly the configuration files are hidden in Linux OS. Usually, we don't need to access or read the hidden files. The hidden files in Linux are represented by a dot (.) before the file name (e.g., .ignore). To access the files, we need to change the view in the file manager or need to use a specific command in the shell.

4.7.2 Types of Linux File System

When we install the Linux operating system, Linux offers many file systems such as **Ext, Ext2, Ext3, Ext4, JFS, ReiserFS, XFS, btrfs,** and **swap**. Let's understand each of these file systems in detail:

Ext, Ext2, Ext3 and Ext4 file system

The file system Ext stands for **Extended File System**. It was primarily developed for **MINIX OS**. The Ext file system is an older version, and is no longer used due to some limitations.

Ext2 is the first Linux file system that allows managing two terabytes of data. Ext3 is developed through Ext2; it is an upgraded version of Ext2 and contains backward

compatibility. The major drawback of Ext3 is that it does not support servers because this file system does not support file recovery and disk snapshot.

Ext4 file system is the faster file system among all the Ext file systems. It is a very compatible option for the SSD (solid-state drive) disks, and it is the default file system in Linux distribution.

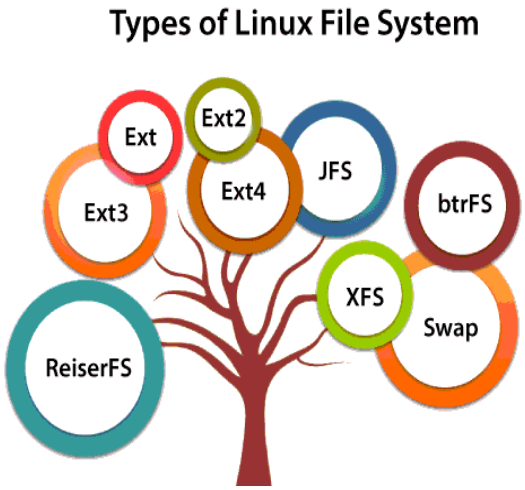


Figure 4.9 Types of Linux file systems

JFS File System

JFS stands for **Journaled File System**, and it is developed by **IBM for AIX Unix**. It is an alternative to the Ext file system. It can also be used in place of Ext4, where stability is needed with few resources. It is a handy file system when CPU power is limited.

ReiserFS File System

ReiserFS is an alternative to the Ext3 file system. It has improved performance and advanced features. In the earlier time, the ReiserFS was used as the default file system in SUSE Linux, but later it has changed some policies, so SUSE returned to Ext3. This file system dynamically supports the file extension, but it has some drawbacks in performance.

XFS File System

XFS file system was considered as high-speed JFS, which is developed for parallel I/O processing. NASA still using this file system with its high storage server (300+ Terabyte server).

Btrfs File System

Btrfs stands for the **B tree file system**. It is used for fault tolerance, repair system, fun administration, extensive storage configuration, and more. It is not a good suit for the production system.

6. Swap File System

The swap file system is used for memory paging in Linux operating system during the system hibernation. A system that never goes in hibernate state is required to have swap space equal to its RAM size. Linux file system considers everything as a file in Linux; whether it is text file images, partitions, compiled programs, directories, or hardware devices. If it is not a file, then it must be a process. To manage the data, it forms a tree structure.

4.8 Creating file in Linux

Linux files are case sensitive, so **test.txt** and **Test.txt** will be considered as two different files. There are multiple ways to create a file in Linux. Some conventional methods are as follows:

- using cat command
- using touch command
- using redirect '>' symbol
- using echo command
- using printf command
- using a different text editor like vim, nano, vi

The directories have specific purposes and generally hold the same types of information for easily locating files. Following are the directories that exist on the major versions of Unix/Linux.

Sr.No.	Directory & Description
1	**/** This is the root directory which should contain only the directories needed at the top level of the file structure
2	**/bin** This is where the executable files are located. These files are available to all users
3	**/dev** These are device drivers
4	**/etc** Supervisor directory commands, configuration files, disk configuration files, valid user lists, groups, ethernet, hosts, where to send critical messages

5	**/lib** Contains shared library files and sometimes other kernel-related files
6	**/boot** Contains files for booting the system
7	**/home** Contains the home directory for users and other accounts
8	**/mnt** Used to mount other temporary file systems, such as **cdrom** and **floppy** for the **CD-ROM** drive and **floppy diskette drive**, respectively
9	**/proc** Contains all processes marked as a file by **process number** or other information that is dynamic to the system
10	**/tmp** Holds temporary files used between system boots
11	**/usr** Used for miscellaneous purposes, and can be used by many users. Includes administrative commands, shared files, library files, and others
12	**/var** Typically contains variable-length files such as log and print files and any other type of file that may contain a variable amount of data
13	**/sbin** Contains binary (executable) files, usually for system administration. For example, ***fdisk*** and ***ifconfig*** utlities

Navigating the File System

Now that you understand the basics of the file system, you can begin navigating to the files you need. The following commands are used to navigate the system.

Sr.No.	Command & Description
1	**cat filename** Displays a filename
2	**cd dirname** Moves you to the identified directory
3	**cp file1 file2** Copies one file/directory to the specified location
4	**file filename** Identifies the file type (binary, text, etc)
5	**find filename dir** Finds a file/directory
6	**head filename** Shows the beginning of a file
7	**less filename** Browses through a file from the end or the beginning
8	**ls dirname** Shows the contents of the directory specified
9	**mkdir dirname** Creates the specified directory
10	**more filename** Browses through a file from the beginning to the end
11	**mv file1 file2** Moves the location of, or renames a file/directory
12	**pwd**

	Shows the current directory the user is in
13	**rm filename** Removes a file
14	**rmdir dirname** Removes a directory
15	**tail filename** Shows the end of a file
16	**touch filename** Creates a blank file or modifies an existing file or its attributes
17	**whereis filename** Shows the location of a file
18	**which filename** Shows the location of a file if it is in your PATH

4.8 Virtual File System in Linux

The flexibility and extensibility of support for Linux file systems is a direct result of an abstracted set of interfaces. At the core of that set of interfaces is the virtual file system switch (VFS). **The *VFS provides a set of standard interfaces for upper-layer applications to perform file I/O over a diverse set of file systems***. And it does it in a way that supports multiple concurrent file systems over one or more underlying devices. Additionally, these file systems need not be static but may come and go with the transient nature of the storage devices.

You'll find VFS also defined as *virtual file system*, but *virtual file system switch* is a much more descriptive definition, as the layer switches (that is, multiplexes) requests across multiple file systems. The /proc file system adds even more confusion here, as it is commonly called a virtual file system.

For example, a typical Linux desktop supports an ext3 file system on the available hard disk, as well as the ISO 9660 file system on an available CD-ROM (otherwise called the *CD-ROM file system*, or CDFS). As CD-ROMs are inserted and removed, the Linux

kernel must adapt to these new file systems with different contents and structure. A remote file system can be accessed through the Network File System (NFS). At the same time, Linux can mount the NT File System (NTFS) partition of a Windows®/Linux dual-boot system from the local hard disk and read and write from it.

Finally, a removable USB flash drive (UFD) can be hot-plugged, providing yet another file system. All the while, the same set of file I/O interfaces can be used over these devices, permitting the underlying file system and physical device to be abstracted away from the user (see Figure 4.10).

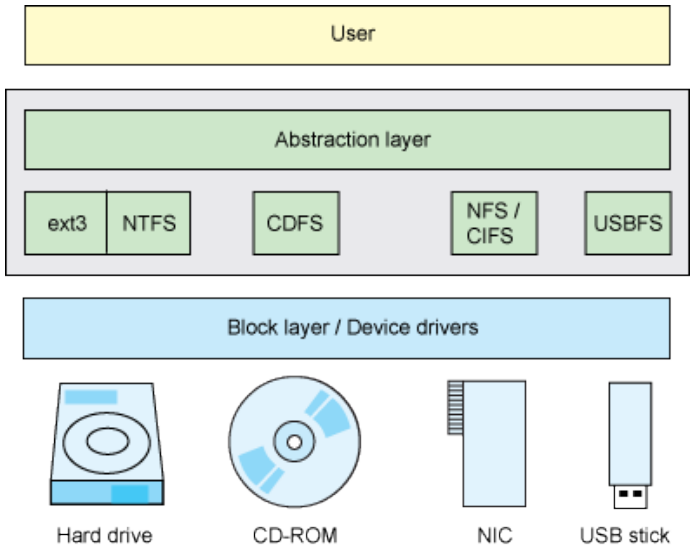


Figure 4.10 An abstraction layer providing a uniform interface over different file systems and storage devices

4.8.1 Layered abstractions

Now, let's add some concrete architecture to the abstract features that the Linux VFS provides. Figure 4.12 shows a high-level view of the Linux stack from the point of view of the VFS. Above the VFS is the standard kernel system-call interface (SCI). This interface allows calls from user-space to transition to the kernel (in different address spaces). In this domain, a user-space application invoking the POSIX open call passes through the GNU C library (glibc) into the kernel and into system call de-multiplexing. Eventually, the VFS is invoked using the call sys_open.

The VFS provides the abstraction layer, separating the POSIX API from the details of how a particular file system implements that behavior. The key here is that Open, Read, Write, or Close API system calls work the same regardless of whether the underlying file system is ext3 or Btrfs. VFS provides a common file model that the underlying file systems inherit (they must implement behaviors for the various POSIX API functions). A further abstraction, outside of the VFS, hides the underlying physical device (which could be a disk,

partition of a disk, networked storage entity, memory, or any other medium able to store information—even transiently).

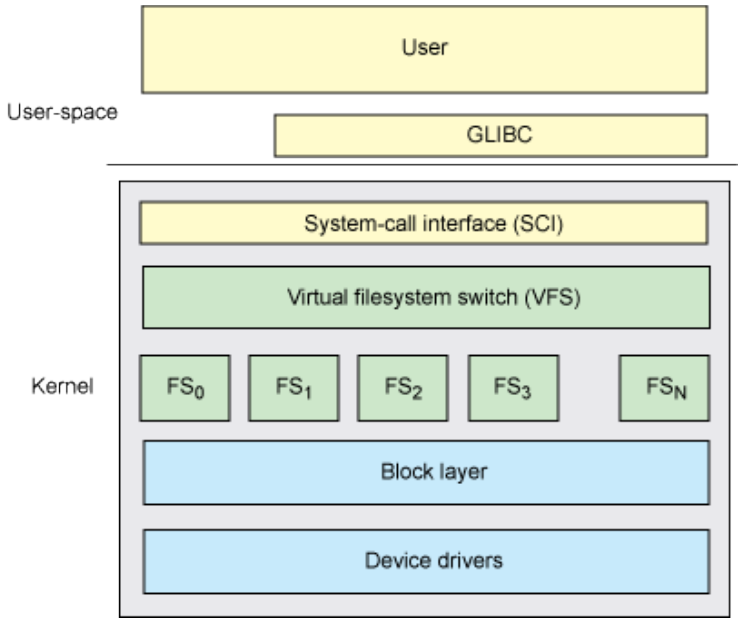


Figure 4.11 The layered architecture of the VFS

In addition to abstracting the details of file operations from the underlying file systems, VFS ties the underlying block devices to the available file systems. Let's now look at the internals of the VFS to see how this works.

4.8.2 VFS internals

Before looking at the overall architecture of the VFS subsystem, let's have a look at the major objects that are used. This section explores the superblock, the index node (or *inode*), the directory entry (or *dentry*), and finally, the file object. Some additional elements, such as caches, are also important here, and I explore these later in the overall architecture.

Superblock

The *superblock* is the container for high-level metadata about a file system. The superblock is a structure that exists on disk (actually, multiple places on disk for redundancy) and also in memory. It provides the basis for dealing with the on-disk file system, as it defines the file system's managing parameters (for example, total number of blocks, free blocks, root index node).

On disk, the superblock provides information to the kernel on the structure of the file system on disk. In memory, the superblock provides the necessary information and state to

manage the active (mounted) file system. Because Linux supports multiple concurrent file systems mounted at the same time, each super_block structure is maintained in a list (super_blocks, defined in ./linux/fs/super.c, with the structure defined in /linux/include/fs/fs.h).

The index node (inode)

Linux manages all objects in a file system through an object called an *inode* (short for *index node*). An inode can refer to a file or a directory or a symbolic link to another object. Note that because files are used to represent other types of objects, such as devices or memory, inodes are used to represent them also.

Note that the inode I refer to here is the VFS layer inode (in-memory inode). Each file system also includes an inode that lives on disk and provides details about the object specific to the particular file system.

Directory entry (dentry)

The hierarchical nature of a file system is managed by another object in VFS called a *dentry* object. A file system will have one root dentry (referenced in the superblock), this being the only dentry without a parent. All other dentries have parents, and some have children. For example, if a file is opened that's made up of /home/user/name, four dentry objects are created: one for the root /, one for the home entry of the root directory, one for the name entry of the user directory, and finally, one dentry for the name entry of the user directory. In this way, dentries map cleanly into the hierarchical file systems in use today.

The dentry object is defined by the dentry structure (in ./linux/include/fs/dcache.h). It consists of a number of elements that track the relationship of the entry to other entries in the file system as well as physical data (such as the file name).

File object

For each opened file in a Linux system, a file object exists. This object contains information specific to the open instance for a given user.

4.8.3 Object relationships

At the top is the open file object, which is referenced by a process's file descriptor list. The file object refers to a dentry object, which refers to an inode. Both the inode and dentry objects refer to the underlying super_block object. Multiple file objects may refer to the same dentry (as in the case of two users sharing the same file). Note also in Figure 4.12 that a dentry object refers to another dentry object. In this case, a directory refers to file, which in turn refers to the inode for the particular file.

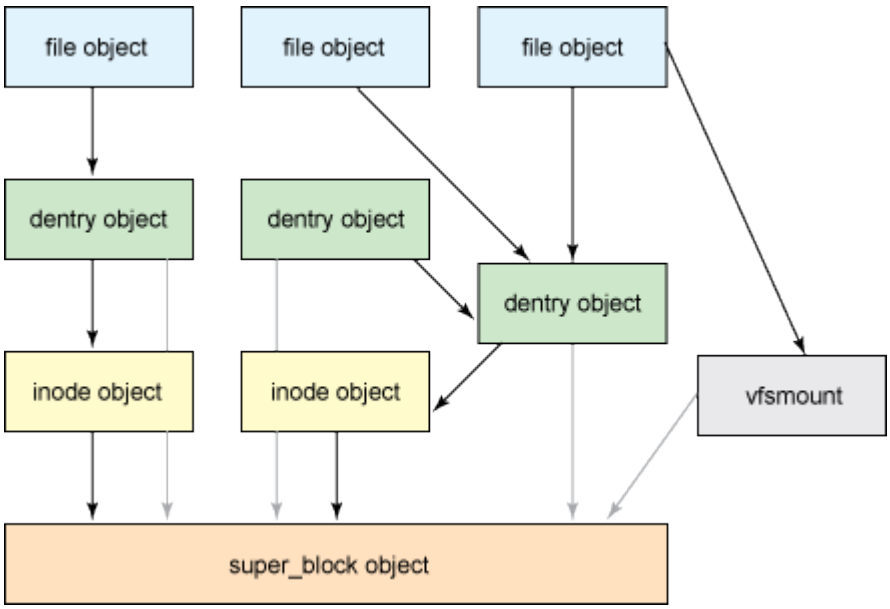


Figure 4.12. Relationships of major objects in the VFS

4.8.4 High-level view of the VFS layer

The internal architecture of the VFS is made up of a dispatching layer that provides the file system abstraction and a number of caches to improve the performance of file system operations. This section explores the internal architecture and how the major objects interact (see Figure 4.13).

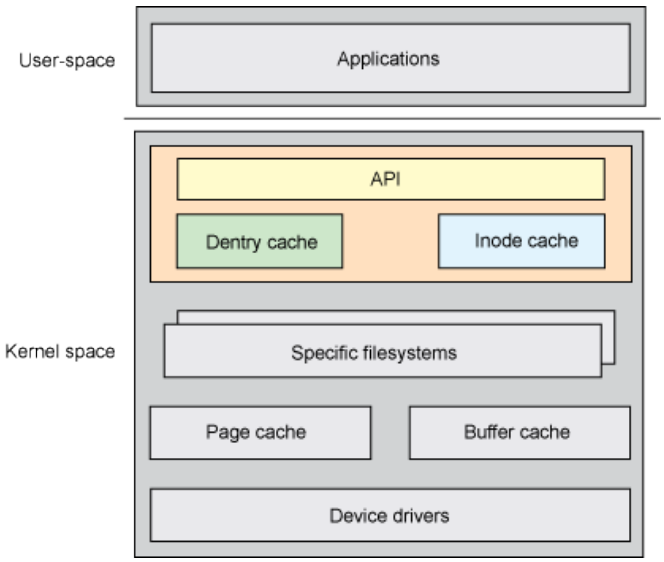


Figure 4.14 High-level view of the VFS layer

The two major objects that are dynamically managed in the VFS include the dentry and inode objects. These are cached to improve the performance of accesses to the underlying file systems. When a file is opened, the dentry cache is populated with entries representing the directory levels representing the path. An inode for the object is also created representing the file. The dentry cache is built using a hash table and is hashed by the name of the object. Entries for the dentry cache are allocated from the dentry_cache slab allocator and use a least-recently-used (LRU) algorithm to prune entries when memory pressure exists. You can find the functions associated with the dentry cache in ./linux/fs/dcache.c (and ./linux/include/linux/dcache.h).

The inode cache is implemented as two lists and a hash table for faster lookup. The first list defines the inodes that are currently in use; the second list defines the inodes that are unused. Those inodes in use are also stored in the hash table. Individual inode cache objects are allocated from the inode_cache slab allocator. You can find the functions associated with the inode cache in ./linux/fs/inode.c (and ./linux/include/fs.h). From the implementation today, the dentry cache is the master of the inode cache. When a dentry object exists, an inode object will also exist in the inode cache. Lookups are performed on the dentry cache, which result in an object in the inode cache.

References

[1] Charles Crowley," Operating Systems: A Design-Oriented Approach", MGH, 1st Edition, 2001.

[2] Christopher Hallinan, "Embedded Linux Primer: A practical Real-World approach", Prentice Hall, 2nd Edition, 2011.

[3] Daniel P. Bovet and Marco Cesati, "Understanding the Linux Kernel, O'Reilly, 3rd Edition, 2005.

[4] John Madieu, "Linux Device Drivers Development: Develop customized drivers for embedded Linux", Packt Publishing, 1st Edition, 2017.

[5] Jonathan Corbet, Alessandro Rubini, Greg Kroah, "Linux Device Drivers", O'Reilly, 3rd Edition, 2005.

Exercise questions

1. What is scheduling? Explain different scheduling method. Which one is more preferable?
2. What is user space and kernel space? Explain it with proper diagram if required.
3. How an EOS application starts executing. Explain it with related diagram if possible.
4. How do we practically apply FCFS scheduling policy? How closely do you observe the similarity between FCFS and Coo-operative scheduling?
5. Explain in brief about context switching between threads/tasks.
6. How time-slicing/pre-emption implemented in the EOS/RTOS setup?
7. Explain pre-emption vs blocking.
8. In a typical EOS/RTOS, how ready queue(s) are implemented and used?
9. What is the policy applied at a given level of Ready queue/Tasks/TCBs?
10. What do you understand by suspended state and terminated state?

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – V- EMBEDDED OS AND DEVICE DRIVERS – SECA5204

5. DEVICE DRIVER INTERNALS

In computing, a **device driver** (commonly referred to as a *driver*) is a computer program that operates or controls a particular type of device that is attached to a computer. A driver provides a software interface to hardware devices, enabling operating systems and other computer programs to access hardware functions without needing to know precise details of the hardware being used.

A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware connects. When a calling program invokes a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware-dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.

Device drivers simplify programming by acting as translator between a hardware device and the applications or operating systems that use it. Programmers can write the higher-level application code independently of whatever specific hardware the end-user is using.

For example, a high-level application for interacting with a serial port may simply have two functions for "send data" and "receive data." At a lower level, a device driver implementing these functions would communicate to the particular serial port controller installed on a user's computer. The commands needed to control a 16550 UART are much different from the commands needed to control an FTDI serial port converter, but each hardware-specific device driver abstracts these details into the same (or similar) software interface.

5.1 Development of Device Driver

Writing a device driver requires an in-depth understanding of how the hardware and the software works for a given platform function. Because drivers require low-level access to hardware functions in order to operate, drivers typically operate in a highly privileged environment and can cause system operational issues if something goes wrong. In contrast, most user-level software on modern operating systems can be stopped without greatly affecting the rest of the system. Even drivers executing in user mode can crash a system if the device is erroneously programmed. These factors make it more difficult and dangerous to diagnose problems.

The task of writing drivers thus usually falls to software engineers or computer engineers who work for hardware-development companies. This is because they have better information than most outsiders about the design of their hardware. Moreover, it was traditionally considered in the hardware manufacturer's interest to guarantee that their clients can use their hardware in an optimum way. Typically, the *logical device driver* (LDD) is written by the operating system vendor, while the *physical device driver* (PDD) is implemented by the device vendor. But in recent years non-vendors have written numerous

device drivers, mainly for use with free and open source operating systems. In such cases, it is important that the hardware manufacturer provides information on how the device communicates. Although this information can instead be learned by reverse engineering, this is much more difficult with hardware than it is with software.

5.2 Working of Device Driver

The kernel depends on individual pieces of software to control each individual piece of hardware, called device drivers. Device drivers contain instructions, like a manual for the kernel, on how to make the hardware perform a requested function. The OS calls the driver, and the driver "drives" the device. These software pieces exist for all hardware, and are often specialized for things like video cards, network adapters, input devices and sound cards. OSs typically use basic drivers that will simply make devices work, but not operate at their full potential. To fully use a device, the user should locate the latest available device driver (either from an included disc, or from the vendor's website).

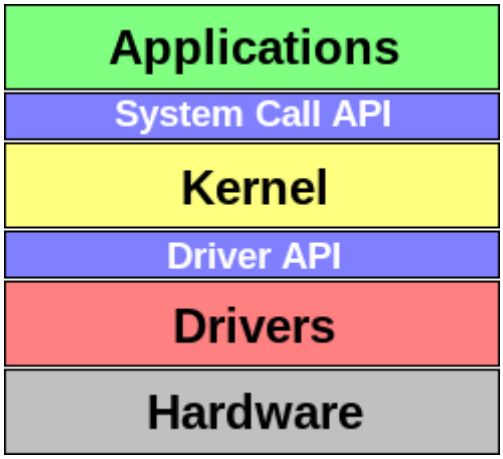


Figure 5.1 Role of Device Drivers

Device Drivers depend upon the Operating System's instruction to access the device and performing any particular action. After the action they also shows their reactions by delivering output or status/message from hardware device to the Operating system. For Example a printer driver tells the printer in which format to print after getting instruction from OS, similarly A sound card driver is there due to which 1's and 0's data of MP3 file is converted to audio signals and you enjoy the music. Card reader, controller, modem, network card, sound card, printer, video card, USB devices, RAM, Speakers etc need Device Drivers to operate

Installing a driver only makes the hardware installed in the computer function properly. If the correct driver is not installed, installing the latest driver for the hardware can take full advantage of the device. However, you cannot install a driver for hardware not installed in the computer and expect it to make your computer faster or more capable. In other words, installing video card drivers for a video card that's not installed in the computer does not give your computer all the capabilities of that video card. In this example, you'd need the video card hardware and the video card drivers to be installed.

5.3 Classification of Drivers According to Functionality

There are numerous driver types, differing in their functionality. This subsection briefly describes three of the most common driver types.

5.3.1 Monolithic Drivers

Monolithic drivers are device drivers that embody all the functionality needed to support a hardware device. A monolithic driver is accessed by one or more user applications, and directly drives a hardware device. The driver communicates with the application through I/O control commands (IOCTLs) and drives the hardware using calls to the different WDK, ETK, DDI/DKI functions.

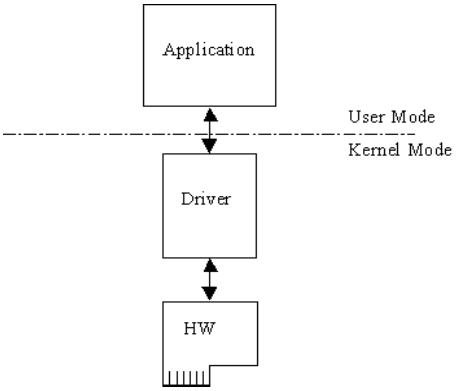


Figure 5.2 Monolithic Drivers

Monolithic drivers are supported in all operating systems including all Windows platforms and all Unix platforms.

5.3.2 Layered Drivers

Layered drivers are device drivers that are part of a stack of device drivers that together process an I/O request. An example of a layered driver is a driver that intercepts calls to the disk and encrypts/decrypts all data being transferred to/from the disk. In this example, a driver would be hooked on to the top of the existing driver and would only do the encryption/decryption. Layered drivers are sometimes also known as filter drivers, and are supported in all operating systems including all Windows platforms and all UNIX platforms.

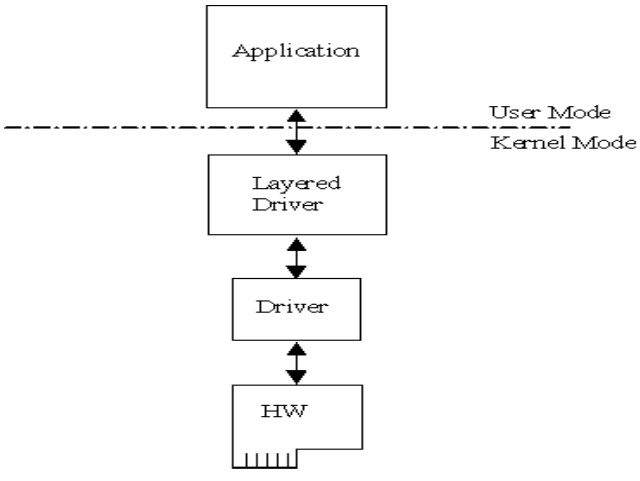


Figure 5.3 Layered Driver

5.3.3 Miniport Drivers

A Miniport driver is an add-on to a class driver that supports miniport drivers. It is used so the miniport driver does not have to implement all of the functions required of a driver for that class. The class driver provides the basic class functionality for the miniport driver. A class driver is a driver that supports a group of devices of common functionality, such as all HID devices or all network devices.

Miniport drivers are also called miniclass drivers or minidrivers, and are supported in the Windows NT (2000) family, namely Windows 7 / Vista / Server 2008 / Server 2003 / XP / 2000 / NT 4.0.

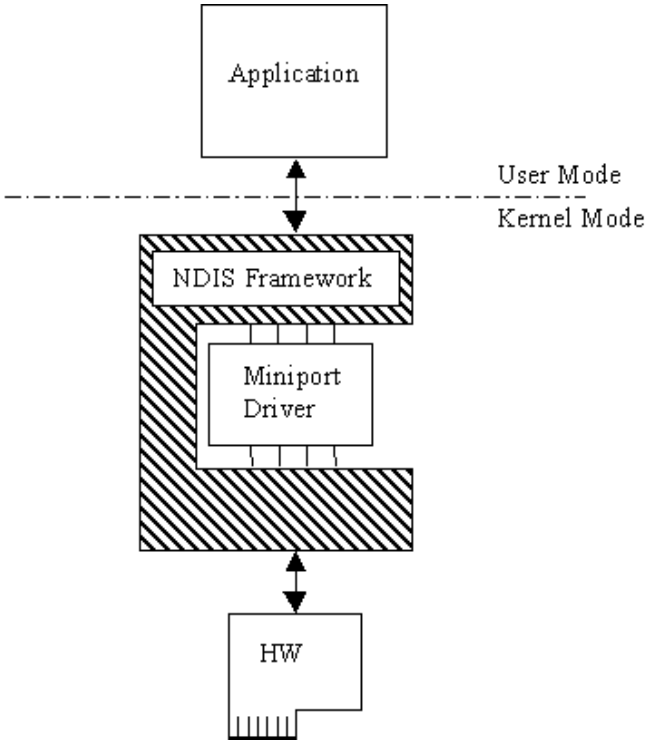


Figure 5.4 Miniport Drivers

Windows 7/Vista/Server 2008/Server 2003/XP/2000/NT 4.0 provide several driver classes (called ports) that handle the common functionality of their class. It is then up to the user to add only the functionality that has to do with the inner workings of the specific hardware. The NDIS miniport driver is one example of such a driver. The NDIS miniport framework is used to create network drivers that hook up to NT's communication stacks, and are therefore accessible to common communication calls used by applications. The Windows NT kernel provides drivers for the various communication stacks and other code that is common to communication cards. Due to the NDIS framework, the network card developer does not have to write all of this code, only the code that is specific to the network card he is developing.

5.4 Linux Device Drivers

Linux device drivers are based on the classic UNIX device driver mode. In addition, Linux introduces some new characteristics. Under Linux, a block device can be accessed like

a character device, as in Unix, but also has a block-oriented interface that is invisible to the user or application.

Traditionally, under Unix, device drivers are linked with the kernel, and the system is brought down and restarted after installing a new driver. Linux introduces the concept of a dynamically loadable driver called a module. Linux modules can be loaded or removed dynamically without requiring the system to be shut down. A Linux driver can be written so that it is statically linked or written in a modular form that allows it to be dynamically loaded. This makes Linux memory usage very efficient because modules can be written to probe for their own hardware and unload themselves if they cannot find the hardware they are looking for.

Like UNIX device drivers, Linux device drivers are either layered or monolithic drivers.

5.4.1 The Entry Point of the Driver

Every device driver must have one main entry point, like the main() function in a C console application. This entry point is called DriverEntry() in Windows and init_module() in Linux. When the operating system loads the device driver, this driver entry procedure is called.

There is some global initialization that every driver needs to perform only once when it is loaded for the first time. This global initialization is the responsibility of the DriverEntry()/init_module() routine. The entry function also registers which driver callbacks will be called by the operating system. These driver callbacks are operating system requests for services from the driver. In Windows, these callbacks are called *dispatch routines*, and in Linux they are called *file operations*. Each registered callback is called by the operating system as a result of some criteria, such as disconnection of hardware, for example.

Operating systems differ in the ways they associate a device with a specific driver.

In Windows, the hardware–driver association is performed via an INF file, which registers the device to work with the driver. This association is performed before the DriverEntry() routine is called. The operating system recognizes the device, checks its database to identify which INF file is associated with the device, and according to the INF file, calls the driver's entry point.

In Linux, the hardware–driver association is defined in the driver's init_module() routine. This routine includes a callback that indicates which hardware the driver is designated to handle. The operating system calls the driver's entry point, based on the definition in the code.

Communication between a user-mode application and the driver that drives the hardware, is implemented differently for each operating system, using the the custom OS Application Programming Interfaces (APIs).

On Windows, Windows CE, and Linux, the application can use the OS file-access API to open a handle to the driver (e.g., using the Windows CreateFile() function or using the Linux open() function), and then read and write from/to the device by passing the handle to the relevant OS file-access functions (e.g., the Windows ReadFile() and WriteFile() functions, or the Linux read() and write() functions).

The application sends requests to the driver via I/O control (IOCTL) calls, using the custom OS APIs provided for this purpose (e.g., the Windows DeviceIoControl() function, or the Linux ioctl() function).
The data passed between the driver and the application via the IOCTL calls is encapsulated using custom OS mechanisms. For example, on Windows the data is passed via an I/O Request Packet (IRP) structure, and is encapsulated by the I/O Manager.

5.5 Classes of Devices and Modules

The Linux way of looking at devices distinguishes between three fundamental device types. Each module usually implements one of these types, and thus is classifiable as a char module, a block module, or a network module. This division of modules into different types, or classes, is not a rigid one; the programmer can choose to build huge modules implementing different drivers in a single chunk of code. Good programmers, nonetheless, usually create a different module for each new functionality they implement, because decomposition is a key element of scalability and extendability. The three classes are:

5.5.1 Character devices

A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls. The text console (/dev/console) and the serial ports (/dev/ttyS0 and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as /dev/tty1 and /dev/lp0. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. There exist, nonetheless, char devices that look like data areas, and you can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using mmap or lseek.

- Character **devices** transmit the data **character** by **characters**, like a mouse or a keyboard.
- A **Character** ('c') **Device** is one with which the Driver communicates by sending and receiving single **characters** (bytes, octets).
- **Character driver** has only one position current one. It can't move back and forth.
- **Block driver** navigates back and forth between any location on media.
- The read() and write() calls do not return until the operation is complete.
- **Character devices** are those for which no buffering is performed, and

5.5.2 Block devices

Like char devices, block devices are accessed by filesystem nodes in the /dev directory. A block device is a device (e.g., a disk) that can host a filesystem. In most Unix systems, a block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of two) bytes in length. Linux, instead, allows the application to read and write a block device like a char device—it permits the transfer of any

number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node, and the difference between them is transparent to the user. Block drivers have a completely different interface to the kernel than char drivers.

- These **devices** transfer unit of data storage called a **block**, USB drives, hard drives, and CD ROMs
- A **Block** ('b') **Device** is one with which the Driver communicates by sending entire **blocks** of data. Examples for **Character Devices**: serial ports, parallel ports, sounds cards.
- A **block** driver provides access to **devices** that transfer randomly accessible data in fixed-size blocks—disk drives, primarily.
- The **Linux** kernel sees **block devices** as being fundamentally different from char **devices**; as a result, **block** drivers have a distinct interface and their own particular challenges.
- **block devices** are those which are accessed through a cache.
- **Block devices** must be random access, but **character devices** are not required to be, though some are. Filesystems can only be mounted if they are on **block devices**.

5.5.3 Network interfaces

Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Many network connections (especially those using TCP) are stream-oriented, but network devices are, usually, designed around the transmission and receipt of packets. A network driver knows nothing about individual connections; it only handles packets. Not being a stream-oriented device, a network interface isn't easily mapped to a node in the filesystem, as /dev/tty1 is. The Unix way to provide access to interfaces is still by assigning a unique name to them (such as eth0), but that name doesn't have a corresponding entry in the filesystem.

Communication between the kernel and a network device driver is completely different from that used with char and block drivers. Instead of read and write, the kernel calls functions related to packet transmission. There are other ways of classifying driver modules that are orthogonal to the above device types. In general, some types of drivers work with additional layers of kernel support functions for a given type of device. For example, one can talk of universal serial bus (USB) modules, serial modules, SCSI modules, and so on. Every USB device is driven by a USB module that works with the USB subsystem, but the device itself shows up in the system as a char device (a USB serial port, say), a block device (a USB memory card reader), or a network device (a USB Ethernet interface). Other classes of device drivers have been added to the kernel in recent times, including FireWire drivers

and I2O drivers. In the same way that they handled USB and SCSI drivers, kernel developers collected class-wide features and exported them to driver implementers to avoid duplicating work and bugs, thus simplifying and strengthening the process of writing such drivers.

5.6 Security Issues

Security is an increasingly important concern in modern times. We will discuss security-related issues as they come up throughout the book. There are a few general concepts, however, that are worth mentioning now. Any security check in the system is enforced by kernel code. If the kernel has security holes, then the system as a whole has holes. In the official kernel distribution, only an authorized user can load modules; the system call init_module checks if the invoking process is authorized to load a module into the kernel. Thus, when running an official kernel, only the superuser, or an intruder who has succeeded in becoming privileged, can exploit the power of privileged code. When possible, driver writers should avoid encoding security policy in their code. Security is a policy issue that is often best handled at higher levels within the kernel, under the control of the system administrator. There are always exceptions, however.

5.7 Polling and Interrupts

Management of I/O devices is a very important part of the operating system - so important and so varied that entire I/O subsystems are devoted to its operation. (Consider the range of devices on a modern computer, from mice, keyboards, disk drives, display adapters, USB devices, network connections, audio I/O, printers, special devices for the handicapped, and many special-purpose peripherals.)

- I/O Subsystems must contend with two (conflicting?) trends: (1) The gravitation towards standard interfaces for a wide range of devices, making it easier to add newly developed devices to existing systems, and (2) the development of entirely new types of devices, for which the existing standard interfaces are not always easy to apply.
- ***Device drivers*** are modules that can be plugged into an OS to handle a particular device or category of similar devices.

I/O Hardware

- I/O devices can be roughly categorized as storage, communications, user-interface, and other
- Devices communicate with the computer via signals sent over wires or through the air.
- Devices connect with the computer via ***ports***, e.g. a serial or parallel port.
- A common set of wires connecting multiple devices is termed a ***bus.***
    - Buses include rigid protocols for the types of messages that can be sent across the bus and the procedures for resolving contention issues.
    - Figure 13.1 below illustrates three of the four bus types commonly found in a modern PC:

1. The ***PCI bus*** connects high-speed high-bandwidth devices to the memory subsystem (and the CPU.)
2. The ***expansion bus*** connects slower low-bandwidth devices, which typically deliver data one character at a time (with buffering.)
3. The ***SCSI bus*** connects a number of SCSI devices to a common SCSI controller.
4. A ***daisy-chain bus,*** (not shown) is when a string of devices is connected to each other like beads on a chain, and only one of the devices is directly connected to the host.

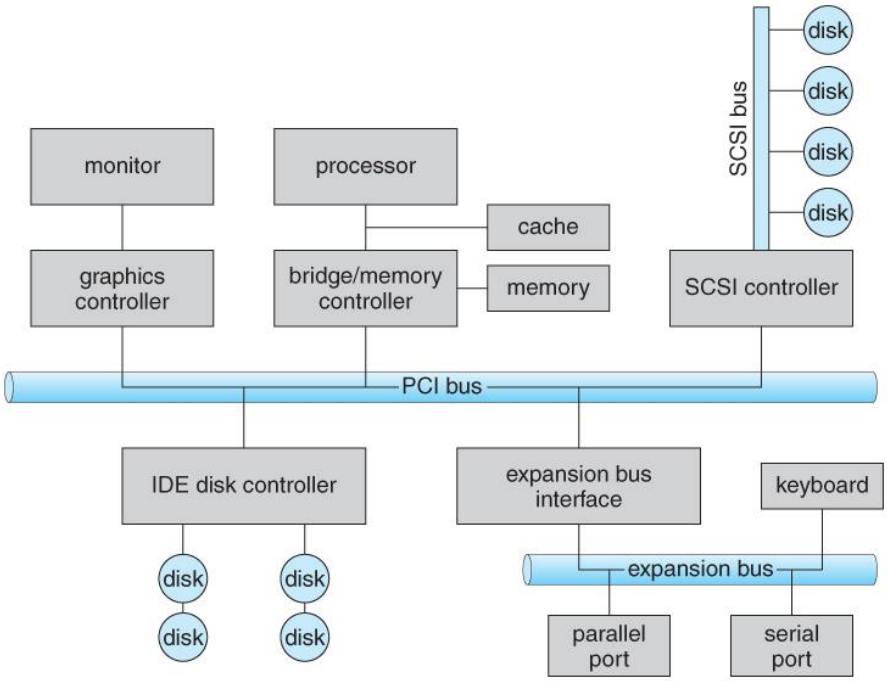


Figure 5.5 A typical PC bus structure.

- One way of communicating with devices is through ***registers*** associated with each port. Registers may be one to four bytes in size, and may typically include (a subset of) the following four:
    1. The ***data-in register*** is read by the host to get input from the device.
    2. The ***data-out register*** is written by the host to send output.
    3. The ***status register*** has bits read by the host to ascertain the status of the device, such as idle, ready for input, busy, error, transaction complete, etc.
    4. The ***control register*** has bits written by the host to issue commands or to change settings of the device such as parity checking, word length, or full- versus half-duplex operation.
- Figure 5.6 shows some of the most common I/O port address ranges.

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Figure 5.6 Device I/O port locations on PCs (partial).

- Another technique for communicating with devices is ***memory-mapped I/O.***
    - In this case a certain portion of the processor's address space is mapped to the device, and communications occur by reading and writing directly to/from those memory areas.
    - Memory-mapped I/O is suitable for devices which must move large quantities of data quickly, such as graphics cards.
    - Memory-mapped I/O can be used either instead of or more often in combination with traditional registers. For example, graphics cards still use registers for control information such as setting the video mode.
    - A potential problem exists with memory-mapped I/O, if a process is allowed to write directly to the address space used by a memory-mapped I/O device.
    - (Note: Memory-mapped I/O is not the same thing as direct memory access, DMA.)

5.7.1 Polling

One simple means of device ***handshaking*** involves polling:

1. The host repeatedly checks the ***busy bit*** on the device until it becomes clear.
2. The host writes a byte of data into the data-out register, and sets the ***write bit*** in the command register (in either order.)
3. The host sets the ***command ready bit*** in the command register to notify the device of the pending command.
4. When the device controller sees the command-ready bit set, it first sets the busy bit.
5. Then the device controller reads the command register, sees the write bit set, reads the byte of data from the data-out register, and outputs the byte of data.

6. The device controller then clears the ***error bit*** in the status register, the command-ready bit, and finally clears the busy bit, signaling the completion of the operation.

Polling can be very fast and efficient, if both the device and the controller are fast and if there is significant data to transfer. It becomes inefficient, however, if the host must wait a long time in the busy loop waiting for the device, or if frequent checks need to be made for data that is infrequently there.

5.7.2 Interrupts

Interrupts allow devices to notify the CPU when they have data to transfer or when an operation is complete, allowing the CPU to perform other duties when no I/O transfers need its immediate attention. The CPU has an ***interrupt-request line*** that is sensed after every instruction.

- A device's controller ***raises*** an interrupt by asserting a signal on the interrupt request line.
- The CPU then performs a state save, and transfers control to the ***interrupt handler*** routine at a fixed address in memory. (The CPU ***catches*** the interrupt and ***dispatches*** the interrupt handler.)
- The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a ***return from interrupt*** instruction to return control to the CPU. (The interrupt handler ***clears*** the interrupt by servicing the device.)
    - (Note that the state restored does not need to be the same state as the one that was saved when the interrupt went off. See below for an example involving time-slicing.)

- Figure 5.7 illustrates the interrupt-driven I/O procedure:

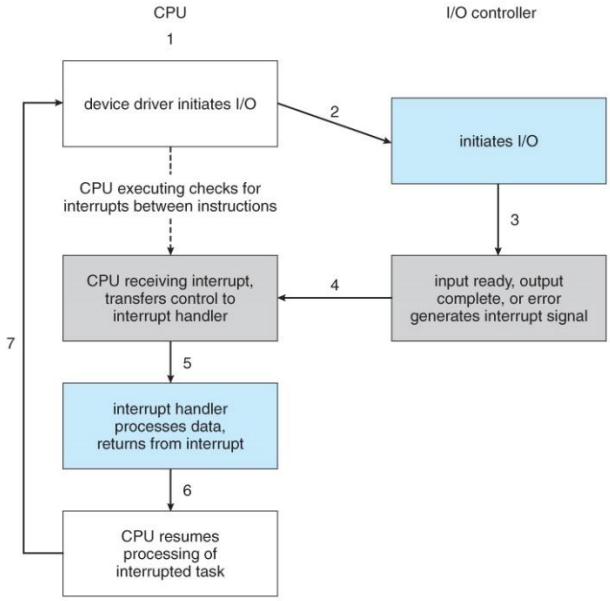


Figure 5.7 Interrupt-driven I/O cycle.

The above description is adequate for simple interrupt-driven I/O, but there are three needs in modern computing which complicate the picture:

1. The need to defer interrupt handling during critical processing,
2. The need to determine ***which*** interrupt handler to invoke, without having to poll all devices to see which one needs attention, and
3. The need for multi-level interrupts, so the system can differentiate between high- and low-priority interrupts for proper response.

These issues are handled in modern computer architectures with ***interrupt-controller*** hardware.

- Most CPUs now have two interrupt-request lines: One that is ***non-maskable*** for critical error conditions and one that is ***maskable,*** that the CPU can temporarily ignore during critical processing.
- The interrupt mechanism accepts an ***address,*** which is usually one of a small set of numbers for an offset into a table called the ***interrupt vector.*** This table (usually located at physical address zero ?) holds the addresses of routines prepared to process specific interrupts.
- The number of possible interrupt handlers still exceeds the range of defined interrupt numbers, so multiple handlers can be ***interrupt chained***. Effectively the addresses held in the interrupt vectors are the head pointers for linked-lists of interrupt handlers.
- Figure 13.4 shows the Intel Pentium interrupt vector. Interrupts 0 to 31 are non-maskable and reserved for serious hardware and other errors. Maskable interrupts, including normal device I/O interrupts begin at interrupt 32.
- Modern interrupt hardware also supports ***interrupt priority levels***, allowing systems to mask off only lower-priority interrupts while servicing a high-priority interrupt, or conversely to allow a high-priority signal to interrupt the processing of a low-priority one.

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Figure 5.9 Intel Pentium processor event-vector table.

- At boot time the system determines which devices are present, and loads the appropriate handler addresses into the interrupt table.
- During operation, devices signal errors or the completion of commands via interrupts.
- Exceptions, such as dividing by zero, invalid memory accesses, or attempts to access kernel mode instructions can be signaled via interrupts.
- Time slicing and context switches can also be implemented using the interrupt mechanism.
    - The scheduler sets a hardware timer before transferring control over to a user process.
    - When the timer raises the interrupt request line, the CPU performs a state-save, and transfers control over to the proper interrupt handler, which in turn runs the scheduler.
    - The scheduler does a state-restore of a ***different*** process before resetting the timer and issuing the return-from-interrupt instruction.
- A similar example involves the paging system for virtual memory - A page fault causes an interrupt, which in turn issues an I/O request and a context switch as described above, moving the interrupted process into the wait queue and selecting a different process to run. When the I/O request has completed (i.e. when the requested page has been loaded up into physical memory), then the device interrupts, and the interrupt handler moves the process from the wait queue into the ready queue, (or depending on scheduling algorithms and policies, may go ahead and context switch it back onto the CPU.)
- System calls are implemented via ***software interrupts,*** a.k.a. ***traps.*** When a (library) program needs work performed in kernel mode, it sets command information and possibly data addresses in certain registers, and then raises a software interrupt. (E.g. 21 hex in DOS.) The system does a state save and then calls on the proper interrupt handler to process the request in kernel mode. Software interrupts generally have low priority, as they are not as urgent as devices with limited buffering space.
- Interrupts are also used to control kernel operations, and to schedule activities for optimal performance. For example, the completion of a disk read operation involves **two** interrupts:
    - A high-priority interrupt acknowledges the device completion, and issues the next disk request so that the hardware does not sit idle.
    - A lower-priority interrupt transfers the data from the kernel memory space to the user space, and then transfers the process from the waiting queue to the ready queue.
- The Solaris OS uses a multi-threaded kernel and priority threads to assign different threads to different interrupt handlers. This allows for the "simultaneous" handling of

multiple interrupts, and the assurance that high-priority interrupts will take precedence over low-priority ones and over user processes.

5.7.4 Direct Memory Access

For devices that transfer large quantities of data (such as disk controllers), it is wasteful to tie up the CPU transferring data in and out of registers one byte at a time. Instead, this work can be off-loaded to a special processor, known as the ***Direct Memory Access, DMA, Controller.*** *It works as outlined below.*

- The host issues a command to the DMA controller, indicating the location where the data is located, the location where the data is to be transferred to, and the number of bytes of data to transfer. The DMA controller handles the data transfer, and then interrupts the CPU when the transfer is complete.
- A simple DMA controller is a standard component in modern PCs, and many ***bus-mastering*** I/O cards contain their own DMA hardware.
- Handshaking between DMA controllers and their devices is accomplished through two wires called the DMA-request and DMA-acknowledge wires.
- While the DMA transfer is going on the CPU does not have access to the PCI bus (including main memory), but it does have access to its internal registers and primary and secondary caches.
- DMA can be done in terms of either physical addresses or virtual addresses that are mapped to physical addresses. The latter approach is known as ***Direct Virtual Memory Access, DVMA,*** and allows direct data transfer from one memory-mapped device to another without using the main memory chips.
- Direct DMA access by user processes can speed up operations, but is generally forbidden by modern systems for security and protection reasons. (I.e. DMA is a kernel-mode operation.)
- Figure 5.10 below illustrates the DMA process.

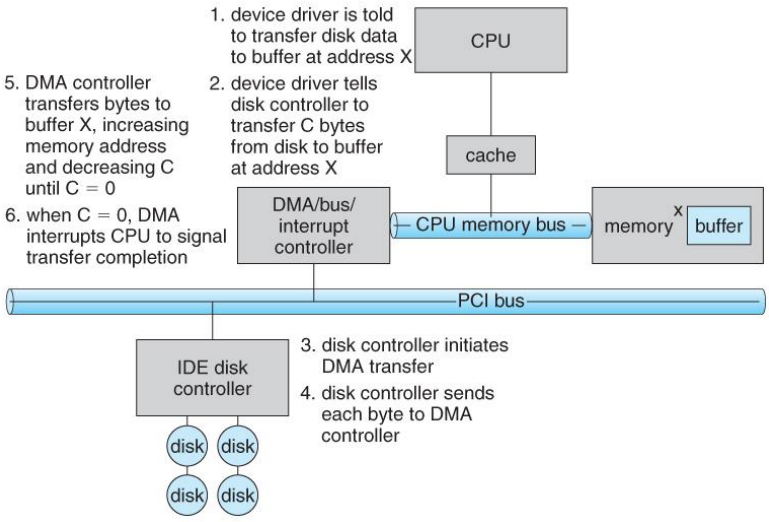


Figure 5.10 Steps in a DMA transfer.

5.7.5 Application I/O Interface

User application access to a wide variety of different devices is accomplished through layering, and through encapsulating all of the device-specific code into ***device drivers***, while application layers are presented with a common interface for all (or at least large general categories of) devices.

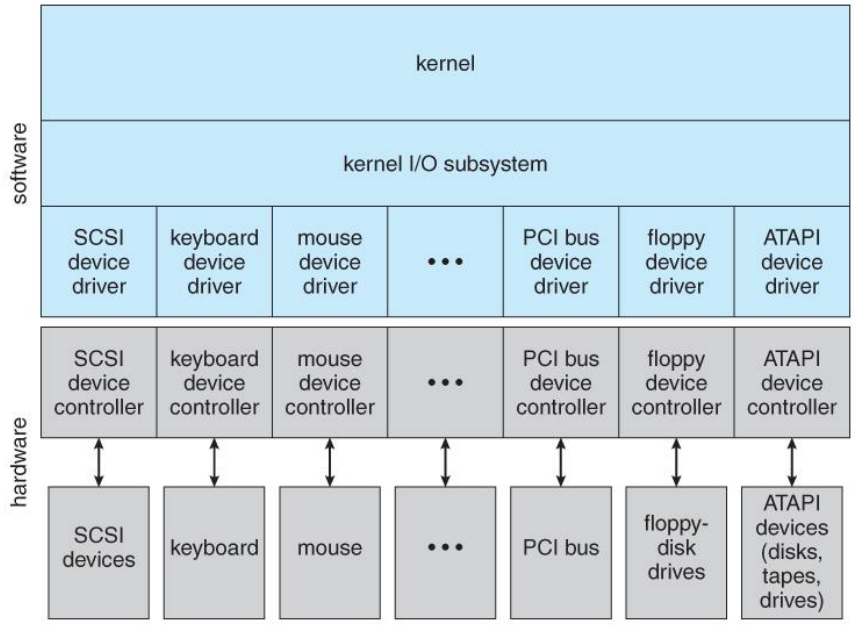


Figure 5.11 A kernel I/O structure.

Devices differ on many different dimensions, as outlined in Figure 5.12

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Figure 5.12 Characteristics of I/O devices.

- Most devices can be characterized as either block I/O, character I/O, memory mapped file access, or network sockets. A few devices are special, such as time-of-day clock and the system timer.
- Most OSes also have an ***escape,*** or ***back door,*** which allows applications to send commands directly to device drivers if needed. In UNIX this is the ***ioctl( )*** system call

(I/O Control). Ioctl( ) takes three arguments - The file descriptor for the device driver being accessed, an integer indicating the desired function to be performed, and an address used for communicating or transferring additional information.

5.8 Device Enumeration and Configuration

Enumeration is the process whereby the Host detects the presence of a Device and takes the necessary steps to ensure that the Device endpoints are added to the list of endpoints serviced by the Host. Device enumerator manager process is used by operating systems to detect all known hardware devices on the system and to start the appropriate drivers and managers. It's called by the /etc/rc.d/rc.devices script, which /etc/system/sysinit invokes.

The enum-devices manager uses a series of configuration files to specify actions to take when the system detects specific hardware devices. After it reads the configuration file(s), enum-devices queries its various enumerators to discover what devices are on the system. It then matches these devices against the device IDs listed in the configuration files. If the device matches, the action clauses associated with the device are executed. You can find the enumerator configuration files in the /etc/system/enum directory.

For example, the /etc/system/enum/devices/net file includes commands to detect network devices, start the appropriate drivers, and then start netmanager to configure the TCP/IP parameters, using the settings in /etc/net.cfg.

Here's some sample code from a configuration file:

```
device(pci, ven=2222, dev=1111)
   uniq(sernum, devc-ser, 1)
driver(devc-ser8250,  "-u$(sernum) $(ioport1),$(irq)" )
```

This code directs the enumerator to do the following when it detects device 1111 from vender 2222:

1. Set sernum to the next unique serial device number, starting at 1.
2. Start the devc-ser8250 driver with the provided options (the device enumerator sets the ioport and irq variables).

To detect new hardware or specify any additional options, you can extend the enumerator configuration files in the following ways:

- an oem file or directory
- an overrides file or directory
- a host-specific set of enumeration files

as described below.

The enumerator reads and concatenates the contents of all configuration files under the chosen directory before it starts processing.

5.8.1 USB Enumeration and Configuration Process

Enumeration is the process whereby the Host detects the presence of a Device and takes the necessary steps to ensure that the Device endpoints are added to the list of endpoints serviced by the Host. The various states of the enumeration and configuration process is illustrated in figure below.

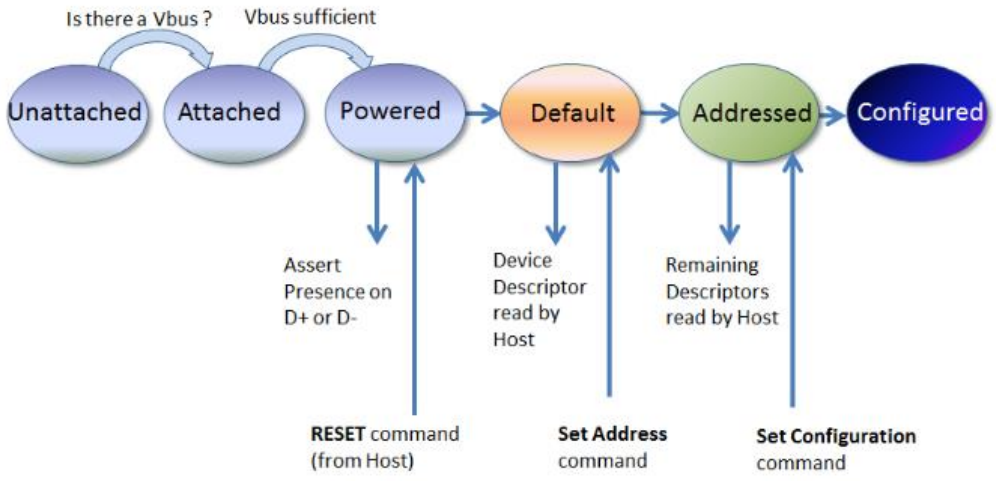


Figure 5.13 Device Enumeration States

5.8.2 Descriptors

Each Universal Serial Bus (USB) device has a set of descriptors. The descriptors are read by the Host during enumeration. Descriptors inform the Host of the following information about a device:

- The version of USB supported by the device
- Who made the device
- How many ways the device can be configured by the Host
- The power consumed by each device configuration
- The number and length of endpoints on the device
- What type of transfer method is to be used to communicate with endpoints
- How often the endpoints are to be serviced
- What text to display if the Host operating systems accept text descriptions

Descriptor Types

The most commonly used descriptors include:

- Device Descriptor
- Configuration Descriptor
- Interface Descriptor
- Endpoint Descriptor

- String Descriptor

Every USB device must have one Device Descriptor and at least one each of the Configuration, Interface, and Endpoint Descriptors.

Device Descriptor

The Device Descriptor is the first descriptor read by the Host during enumeration. The purpose of the Device Descriptor is to let the Host know what specification of USB the device complies with and how many possible configurations are available on the device. Upon successful processing of the Device Descriptor, the Host will read all the Configuration Descriptors. Figure 5.14 illustrates different type of file descriptors used by hardware devices.

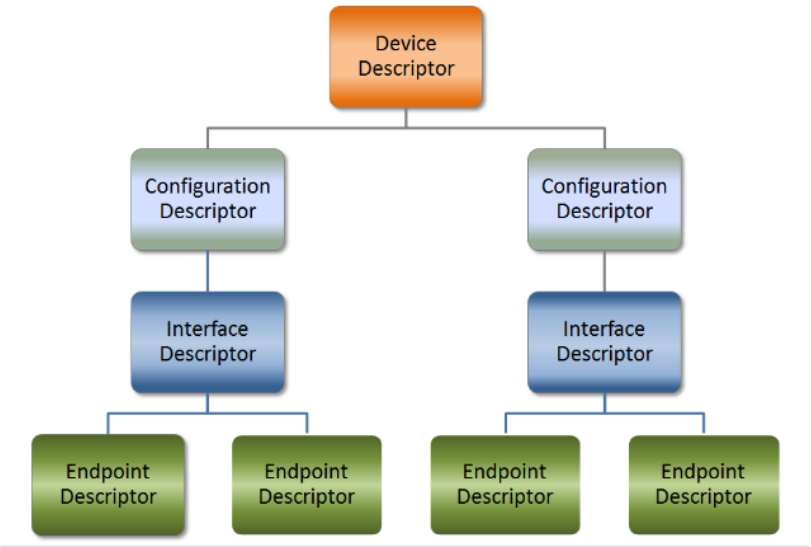


Figure 5.14 Type of file descriptors

Device Detection

The presence of a newly installed *Full Speed*, *High Speed*, or *Low Speed Device* is recognized by changes in the D- or D+ signal. A low-speed device places 5 V on D-, high- and full-speed devices assert 5 V on D+. The connection signals are detected by the Hub and reported to the Host. Once a Device is detected, the Host issues a RESET command to the Device.

Default State

When a RESET control signal sequence is received, the Device will manage its load, per specification, to enumerate. If the attached Device is a High-Speed device, a "chirp" will be returned and the High-Speed detection process will be completed. Once the speed has been settled, the Host reads the Device descriptor and assigns an address.

Addressed State

After setting the address, the Host reads all remaining descriptor tables for the device. If a Host determines it can service the Device's interface endpoints and provide sufficient power, the Host issues a command informing the Device which of its configurations to activate.

Configured State

After receiving notification from the Host regarding which configuration to activate, the Device is ready to run using the active configuration.

5.8.3 Descriptor Structures

There are several types of descriptors for USB devices arranged in a logical hierarchy. The following diagram illustrates the hierarchy of a descriptor for a single device with two possible configurations for the Host to activate. Each of these configurations has a single interface with two endpoints.

Structure of a Device Descriptor

```
typedef struct _USB_DEVICE_DESCRIPTOR {
  UCHAR  bLength;
  UCHAR  bDescriptorType;
  USHORT bcdUSB;
  UCHAR  bDeviceClass;
  UCHAR  bDeviceSubClass;
  UCHAR  bDeviceProtocol;
  UCHAR  bMaxPacketSize0;
  USHORT idVendor;
  USHORT idProduct;
  USHORT bcdDevice;
  UCHAR  iManufacturer;
  UCHAR  iProduct;
  UCHAR  iSerialNumber;
  UCHAR  bNumConfigurations;
} USB_DEVICE_DESCRIPTOR, *PUSB_DEVICE_DESCRIPTOR;
```

Key Elements of a Device Descriptor

bcdUSB	Informs the Host of what version of USB the device supports
bDeviceClass	00 - The device class is defined in the Interface Descriptor FF - the device class is Vendor class any other number is the specification for the class of this device

idVendor	16-bit number assigned by USB.org to the product's manufacturer
idProduct	16-bit product model ID assigned by the Vendor to this product
bNumConfigurations	How many different configurations are available for this device

Configuration Descriptor

A device may have more than one configuration. Each device configuration is assigned a number. The Configuration Descriptor serves two purposes:

1. Informs the Host as to how many interfaces (i.e., virtual devices) are in the configuration. While it is common for a configuration to offer only one interface, Devices that appear like two or more products have more than one interface.
2. How much power the device will consume if this configuration is activated by the Host. If the device is capable of controlling its power consumption, it may offer more than one configuration. Each configuration will advertise how much power would be consumed if the configuration were to be activated.

Structure of a Configuration Descriptor

```
typedef struct _USB_CONFIGURATION_DESCRIPTOR {
  UCHAR  bLength;
  UCHAR  bDescriptorType;
  USHORT wTotalLength;
  UCHAR  bNumInterfaces;
  UCHAR  bConfigurationValue;
  UCHAR  iConfiguration;
  UCHAR  bmAttributes;
  UCHAR  MaxPower;
} USB_CONFIGURATION_DESCRIPTOR, *PUSB_CONFIGURATION_DESCRIPTOR;
```

Key Elements of a Configuration Descriptor

bNuminterfaces	Number of Interface Descriptor tables available

MaxPower	Power load of this device if the Host activates this configuration

Interface Descriptor

An Interface Descriptor describes the details of the function of the product. Key elements include the number of endpoints on the device and which USB device class is implemented by the endpoints. For example, if the device were a keyboard, the specified device class would be Human Interface Device (HID) and the number of endpoints would be two. See the *"USB Device Classes"* page for details on how device classes are implemented in USB.

Structure of an Interface Descriptor

```
typedef struct _USB_INTERFACE_DESCRIPTOR {
  UCHAR bLength;
  UCHAR bDescriptorType;
  UCHAR bInterfaceNumber;
  UCHAR bAlternateSetting;
  UCHAR bNumEndpoints;
  UCHAR bInterfaceClass;
  UCHAR bInterfaceSubClass;
  UCHAR bInterfaceProtocol;
  UCHAR iInterface;
} USB_INTERFACE_DESCRIPTOR, *PUSB_INTERFACE_DESCRIPTOR;
```

Key Elements of an Interface Descriptor

bNumEndpoints	Number of endpoints in the interface
bInterfaceClass	USB device class used to set transfer types for the endpoints

Only one configuration can be active at any time. When a configuration is active, all of its interfaces and endpoints are available to the Host. Devices that have multiple interfaces are referred to as Composite Devices. One physical product with one available USB connector would appear to the Host as two separate devices. A keyboard with an integrated mouse (or trackball) is an example of a composite device.

Endpoint Descriptor

Each endpoint on a device has its own descriptor. The descriptor provides the endpoint address (i.e., endpoint number), the size of the endpoint, and the data transfer type used to access the endpoint.

```
typedef struct _USB_ENDPOINT_DESCRIPTOR {
  UCHAR  bLength;
  UCHAR  bDescriptorType;
  UCHAR  bEndpointAddress;
  UCHAR  bmAttributes;
  USHORT wMaxPacketSize;
  UCHAR  bInterval;
} USB_ENDPOINT_DESCRIPTOR, *PUSB_ENDPOINT_DESCRIPTOR;
```

Key Elements of Endpoint Descriptors

bEndpointAddress	The address of the endpoint (i.e. endpoint number)
wMaxPacketSize	Length of the endpoint
bInterval	How often in frames is this endpoint to be serviced by the Host

5.8 Plug n Play Device Working Sequence

The following steps are followed by the software modules in the host to detect and read data from a plug and play device.

- Plug in device
- Detect Connection
- Set address
- Get device info
- Choose a device driver
- Choose configuration
- Choose drivers for interfaces
- Use it

==========

References

[1] Charles Crowley," Operating Systems: A Design-Oriented Approach", MGH, 1st Edition, 2001.

[2] Christopher Hallinan, "Embedded Linux Primer: A practical Real-World approach", Prentice Hall, 2nd Edition, 2011.

[3] Daniel P. Bovet and Marco Cesati, "Understanding the Linux Kernel, O'Reilly, 3rd Edition, 2005.

[4] John Madieu, "Linux Device Drivers Development: Develop customized drivers for embedded Linux", Packt Publishing, 1st Edition, 2017.

[5] Jonathan Corbet, Alessandro Rubini, Greg Kroah, "Linux Device Drivers", O'Reilly, 3rd Edition, 2005.

Exercise Questions

1. What happens if an h/w interrupt event is generated when a task in privileged mode is executing in an EOS/RTOS?
2. What is Task? How task is created? Explain Task-Management in detail.
3. What is interrupt and polling? How it can affect your application in real-time. How you will use it for your application?
4. Explain the task states cycle in your RTOS. Explain the working of TCB.
5. What do you understand by Time –slicing? What is difference between blocking API and yielding API?
6. What do you understand by the term Task modeling?
7. Explain the difference between vTaskDelay() and vTaskDelayUntil().
8. What do you understand by Co-operative scheduling?
9. What is the application of pendsv? When and how it is triggered?
10. What is difference between system API and system-call API?