Chapter – 14

Security

Topics:

- **User Authentication**
- **Program threats**
- **System threats**
- **Security system facilities**

Introduction to Security

Security refers to providing a protection system to computer system resources such as CPU, memory, disk, software programs and most importantly data/information stored in the computer system. If a computer program is run by an unauthorized user, then he/she may cause severe damage to computer or data stored in it. So a computer system must be protected against unauthorized access, malicious access to system memory, viruses, worms etc.

(14.1) User Authentication:

A major security problem for operating system is **authentication.** The protection system depends on an ability to identify the programs and processes currently executing.

- Authentication refers to identifying each user of the system and associating the executing programs with those users. It is the responsibility of the Operating System to create a protection system which ensures that a user who is running a particular program is authentic.

Operating Systems generally identifies/authenticates users using following **three ways**:

- User possession (a key or card)
- User knowledge (a user identifier and password),
- User attribute (fingerprint, retina pattern, or signature)
- **Username / Password** − User need to enter a registered username and password with Operating system to login into the system.
- The most common approach to authenticating a user identity is the use of passwords.
- If the password is correct, access is granted. Different passwords may be associated with different access rights.
- For example, different passwords may be used for each of the following file operations: reading, appending, and updating
- **Password Vulnerabilities-** Passwords are extremely common because they are easy to understand and use. Unfortunately, passwords can often be guessed, accidentally exposed, sniffed, or illegally transferred form an authorized user to an unauthorized one, as we show next.
- **Encrypted Passwords-** One problem with all these approaches is the difficulty of keeping the password secret within the computer. The Unix system uses encryption to avoid the necessity of keeping its password list secret.
- **One Time Passwords-**To avoid the problems of password sniffing and shoulder surfing, a system could use a set of **paired password**. When a session begins, the system randomly selects and presents one part of a password pair ,the user must supply the other parts.
- The user uses the keypad to enter the shared secret, also known as a **Personal identification number (PIN).** The display shows the one- time password.
- Another variation on one time passwords is the use of a **code book**, or **one-time pad**, which is a list of single -use password.

- **Biometrics-**There are many other variations on the use passwords for authentication. Palm or hand –readers are common to secure physical access, for example access to a data center. These readers match stored parameters against what is being read from their hand –reader pad.

One Time passwords

One-time passwords provide additional security along with normal authentication. In One-Time Password system, a unique password is required every time user tries to login into the system. Once a one-time password is used, then it cannot be used again. One-time password are implemented in various ways.

- **Random numbers** − Users are provided cards having numbers printed along with corresponding alphabets. System asks for numbers corresponding to few alphabets randomly chosen.
- **Secret key** − User are provided a hardware device which can create a secret id mapped with user id. System asks for such secret id which is to be generated every time prior to login.
- **Network password** − Some commercial applications send one-time passwords to user on registered mobile/ email which is required to be entered prior to login.

(14.2) Program Threats:

Operating system's processes and kernel do the designated task as instructed. If a user program made these process do malicious tasks, then it is known as **Program Threats**. One of the common example of program threat is a program installed in a computer which can store and send user credentials via network to some hacker. Following is the list of some well-known program threats.

- **Trojan Horse** − Such program traps user login credentials and stores them to send to malicious user who can later on login to computer and can access system resources.
- **Trap Door** − If a program which is designed to work as required, have a security hole in its code and perform illegal action without knowledge of user then it is called to have a trap door.
- **Stack and Buffer Overflow-** The stack or overflow attack is the most common way for an attacker outside of the system, on a network or dial up connection to gain unauthorized access to the target system.
    1. Overflow an input field, command line argument, or input buffer, for example, on a network daemon, until it writes into the stack.

2. Overflow the current return address on the stack with the address of the exploit code loaded.
3. Write a simple set of code for the next space in the stack that includes the commands that the attacker wishes to execute, for instance, spawn a shell.

(14.3) System Threats:

- System threats refers to misuse of system services and network connections to put user in trouble.
- System threats can be used to launch program threats on a complete network called as program attack.
- System threats creates such an environment that operating system resources/ user files are misused.
- Following is the list of some well-known system threats.

Worm – Worm is a process which can choked down a system performance by using system resources to extreme levels. A Worm process generates its multiple copies where each copy uses system resources, prevents all other processes to get required resources. Worm processes can even shut down an entire network.

- This worm consisted of two parts:
  1. A small program called a ***grappling hook,*** which was deposited on the target system through one of three vulnerabilities, and
  2. The main worm program, which was transferred onto the target system and launched by the grappling hook program.

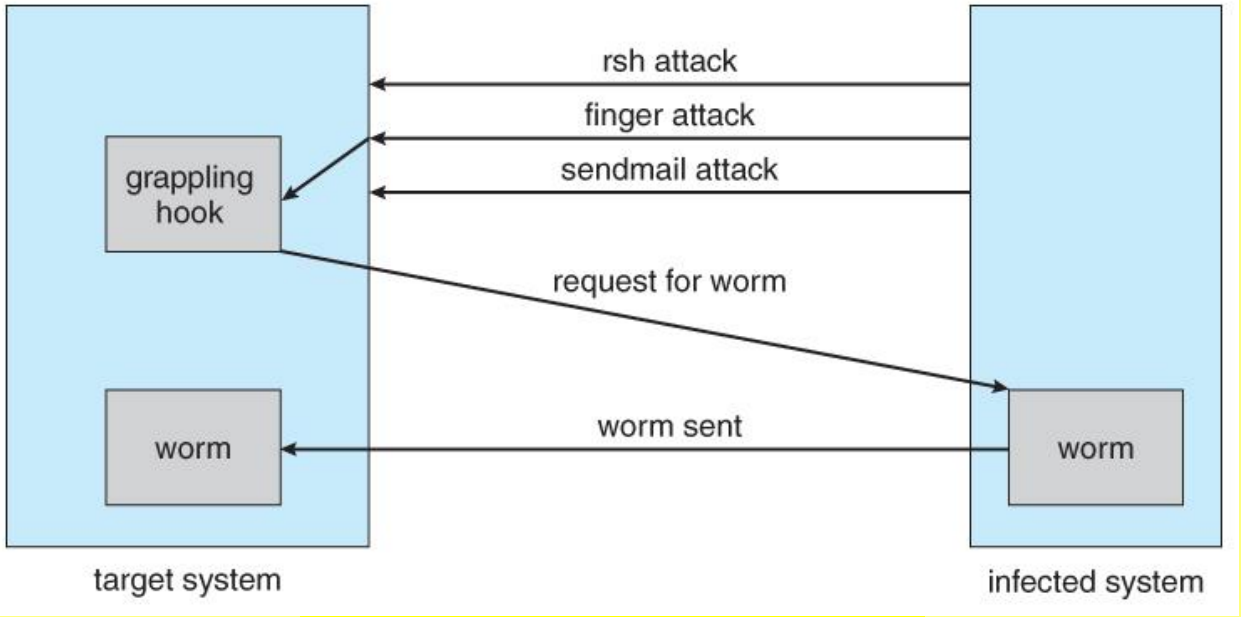


Figure 15.6 - The Morris Internet worm.

Virus – Virus as name suggest can replicate themselves on computer system. They are highly dangerous and can modify/delete user files, crash systems. A virus is generatlly a small code embedded in a program. As user accesses the program, the virus starts getting embedded in other files/ programs and can make system unusable for user

Denial of Service – The last attack category denial of service, does not involves gaining information or stealing resources, but rather disabling legitimate use of a system or facility. an intruder could delete all the files on a system.

For example. Most denial of service attacks involve system that the attacker has not penetrated. Indeed, launching an attack that prevents legitimate use is frequently easier than breaking into a machine or facility

(14.4) Securing Systems and Facilities:

Securing system and facilities is intimately linked to intrusion detection. Both techniques need to work together to assure that a system is secure and that, if a security breach happens, it is detected.

One method of improving system security is periodically to scan the system for security holes. These scans can be done when the computer has relatively little use, so they will have less effect than logging. Such a scan can check a variety of aspects of the system:

- Short or easy –to-guess passwords
- Unauthorized privileged programs, such as setuid programs
- Unauthorized programs in system directories
- Unexpected long-running processes
- Improper directory protections on both user and system directories
- Improper protections on system data files, such as the password file, device drivers, or even the operating-system kernel itself
- Dangerous entries in the program search path
- Changes to system programs detected with checksum values
- Unexpected or hidden network daemons

Any problems found by a security scan can either be fixed automatically or reported to the managers of the system.

- Firewalls are devices (or sometimes software) that sit on the border between two security domains and monitor/log activity between them, sometimes restricting the traffic that can pass between them based on certain criteria.
- For example a firewall router may allow HTTP: requests to pass through to a web server inside a company domain while not allowing telnet, ssh, or other traffic to pass through.

- A common architecture is to establish a de-militarized zone, DMZ, which sort of sits "between" the company domain and the outside world, as shown below.
- Company computers can reach either the DMZ or the outside world, but outside computers can only reach the DMZ.
- Perhaps most importantly, the DMZ cannot reach any of the other company computers, so even if the DMZ is breached, the attacker cannot get to the rest of the company network.

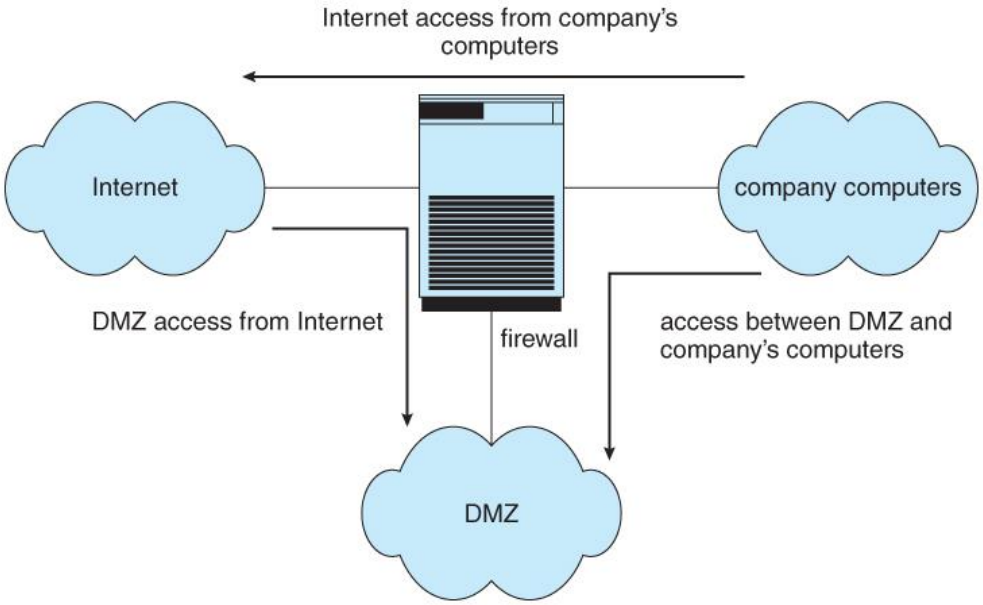


Figure - Domain separation via firewall.

Firewalls themselves need to be resistant to attacks, and unfortunately have several vulnerabilities:

- ***Tunneling,*** which involves encapsulating forbidden traffic inside of packets that are allowed.
- Denial of service attacks addressed at the firewall itself.
- Spoofing, in which an unauthorized host sends packets to the firewall with the return address of an authorized host.

In addition to the common firewalls protecting a company internal network from the outside world, there are also some specialized forms of firewalls that have been recently developed:

- A ***personal firewall*** is a software layer that protects an individual computer. It may be a part of the operating system or a separate software package.
- An ***application proxy firewall*** understands the protocols of a particular service and acts as a stand-in (and relay) for the particular service.
- For example, and SMTP proxy firewall would accept SMTP requests from the outside world, examine them for security concerns, and forward only the "safe" ones on to the real SMTP server behind the firewall.
- ***XML firewalls*** examine XML packets only, and reject ill-formed packets. Similar firewalls exist for other specific protocols.
- ***System call firewalls*** guard the boundary between user mode and system mode, and reject any system calls that violate security policies.

Chapter -15

LINUX SYSTEM

TOPICS:

- **History**
- **Design Principles**
- **Kernel Modules**
- **Process Management**
- **Scheduling**
- **Memory Management**
- **File System**

History:

Linux is a modern, free operating system based on UNIX standards

- First developed as a small but self-contained kernel in 1991 by Linus Torvalds, with the major design goal of UNIX compatibility
- Its history has been one of collaboration by many users from all around the world, corresponding almost exclusively over the Internet
- It has been designed to run efficiently and reliably on common PC hardware, but also runs on a variety of other platforms
- The core Linux operating system kernel is entirely original, but it can run much existing free UNIX software, resulting in an entire UNIX-compatible operating system free from proprietary code
- Many, varying Linux Distributions including the kernel, applications, and management tools

Linux is one of popular version of UNIX operating System. It is open source as its source code is freely available. It is free to use. Linux was designed considering UNIX compatibility. Its functionality list is quite similar to that of UNIX.

The Linux Kernel:

Version 0.01 was released at May 1991

- no networking
- ran only on 80386-compatible Intel processors and on PC hardware
- extremely limited device-drive support • supported only the Minix file system
- Linux 1.0 (March 1994) included these new features:
- support UNIX's standard TCP/IP networking protocols
- BSD-compatible socket interface for networking programming
- device-driver support for running IP over an Ethernet
- enhanced file system
- support for a range of SCSI controllers for high-performance disk access

- extra hardware support

Version 1.2 (March 1995) was the final PC-only Linux kernel.

Version 2.0 was released in June 1996

- support for multiple architectures, including a fully 64-bit native Alpha port
- support for multiprocessor architectures
- improved memory-management code
- improved TCP/IP performance
- support for internal kernel threads
- standardized configuration interface

2.4 and 2.6 increased SMP support

- added journaling file system
- preemptive kernel
- 64-bit memory support

The Linux System

- Linux uses many free tools developed as part of
- Berkeley's BSD operating system • socket interface
- networking tools (e.g., traceroute…)MIT's X Window System
- Free Software Foundation's GNU project
- bin-utilities,
- Linux used to developed by individual, now also big cooperators
- IBM, Intel, Red hat, Marvell, Microsoft.
- Main Linux repository: www.kernel.org

Linux Distributions

- Standard, precompiled sets of packages, or distributions
- include the basic Linux system
- system installation and management utilities
- ready-to-install packages of common UNIX tools
- Popular Linux distributions

- Ubuntu, Fedora, Debian, Open Suse,

Linux Licensing •

- Linux kernel is distributed under GNU General Public License (GPL)
- GPL is defined by the Free Software Foundation
- GPL implications: •
- anyone using Linux, or creating their own derivative of Linux, may not make the derived (public) product proprietary •
- software released under GPL may not be redistributed as binary-only • LGPL: Lesser GPL •
- allow non-(L)GPL software to link to LGPL licensed software

Components of Linux System

Linux Operating System has primarily three components

- **Kernel** − Kernel is the core part of Linux. It is responsible for all major activities of this operating system. It consists of various modules and it interacts directly with the underlying hardware. Kernel provides the required abstraction to hide low level hardware details to system or application programs.
- **System Library** − System libraries are special functions or programs using which application programs or system utilities accesses Kernel's features. These libraries implement most of the functionalities of the operating system and do not requires kernel module's code access rights.
- **System Utility** − System Utility programs are responsible to do specialized, individual level tasks.

Kernel Mode vs User Mode

Kernel component code executes in a special privileged mode called **kernel mode** with full access to all resources of the computer. This code represents a single process, executes in single address space and do not require any context switch and hence is very efficient and fast. Kernel runs each processes and provides system services to processes, provides protected access to hardware to processes.

Support code which is not required to run in kernel mode is in System Library. User programs and other system programs works in **User Mode** which has no access to system hardware and kernel code. User programs/ utilities use System libraries to access Kernel functions to get system's low level tasks.

Basic Features

Following are some of the important features of Linux Operating System.

- **Portable** − Portability means software can works on different types of hardware in same way. Linux kernel and application programs supports their installation on any kind of hardware platform.
- **Open Source** − Linux source code is freely available and it is community based development project. Multiple teams work in collaboration to enhance the capability of Linux operating system and it is continuously evolving.
- **Multi-User** − Linux is a multiuser system means multiple users can access system resources like memory/ ram/ application programs at same time.
- **Multiprogramming** − Linux is a multiprogramming system means multiple applications can run at same time.
- **Hierarchical File System** − Linux provides a standard file structure in which system files/ user files are arranged.
- **Shell** − Linux provides a special interpreter program which can be used to execute commands of the operating system. It can be used to do various types of operations, call application programs. etc.
- **Security** − Linux provides user security using authentication features like password protection/ controlled access to specific files/ encryption of data.

Architecture

The following illustration shows the architecture of a Linux system −

The architecture of a Linux System consists of the following layers −

- **Hardware layer** − Hardware consists of all peripheral devices (RAM/ HDD/ CPU etc).
- **Kernel** − It is the core component of Operating System, interacts directly with hardware, provides low level services to upper layer components.
- **Shell** − An interface to kernel, hiding complexity of kernel's functions from users. The shell takes commands from the user and executes kernel's functions.

- **Utilities** − Utility programs that provide the user most of the functionalities of an operating systems.

Linux History

- **Evolution of Computer:** In earlier days, computers were as big as houses or parks. So you can imagine how difficult it was to operate them.
- Moreover, every computer has a different operating system which made it completely worse to operate on them.
- Every software was designed for a specific purpose and was unable to operate on other computer.
- It was extremely costly and normal people neither can afford it nor can understand it.

- **Evolution of Unix**: In 1969, a team of developers of Bell Labs started a project to make a common software for all the computers and named it as 'Unix'.
- It was simple and elegant, used 'C' language instead of assembly language and its code was recyclable.
- As it was recyclable, a part of its code now commonly called 'kernel' was used to develop the operating system and other functions and could be used on different systems. Also its source code was open source.

Initially, Unix was only found in large organizations like government, university, or larger financial corporations with mainframes and minicomputers (PC is a microcomputer).

Unix Expansion:

- In eighties, many organizations like IBM, HP and dozen other companies started creating their own Unix. It result in a mess of Unix dialects.
- Then in 1983, Richard Stallman developed GNU project with the goal to make it freely available Unix like operating system and to be used by everyone.
- But his project failed in gaining popularity. Many other Unix like operating system came into existence but none of them was able to gain popularity.

Evolution of Linux:

- In 1991, Linus Torvalds a student at the university of Helsinki, Finland, thought to have a freely available academic version of Unix started writing its own code. Later this project became the Linux kernel.
- He wrote this program specially for his own PC as he wanted to use Unix 386 Intel computer but couldn't afford it.
- He did it on MINIX using GNU C compiler. GNU C compiler is still the main choice to compile Linux code but other compilers are also used like Intel C compiler.
- He started it just for fun but ended up with such a large project. Firstly he wanted to name it as 'Freax' but later it became 'Linux'.
- He published the Linux kernel under his own license and was restricted to use as commercially. Linux uses most of its tools from GNU software and are under GNU copyright. In 1992, he released the kernel under GNU General Public License.

- **Linux Today**
- Today, supercomputers, smart phones, desktop, web servers, tablet, laptops and home appliances like washing machines, DVD players, routers, modems, cars, refrigerators, etc use Linux OS.

- **Design Principles:**

- Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools.
- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model.
- Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification.
- The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior.
- As PCs became more powerful and as memory and hard disks became cheaper, the original, minimalist Linux kernels grew to implement more UNIX functionality.
- Speed and efficiency are still important design goals, but much recent and current work on Linux has concentrated on a third major design goal: standardization. One of the prices paid for the diversity of UNIX implementations currently available is that source code written for one may not necessarily compile or run correctly on another.

- Even when the same system calls are present on two different UNIX systems, they do not necessarily behave in exactly the same way.
- Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification.
- 1) Components of a Linux System
- The Linux system has three main bodies of code, in sequence with, most conventional UNIX implementations.
- **THE KERNEL**: "The kernel" is in charge for maintaining all the vital abstractions of the operating system, together with such things as virtual memory and processes. The Linux kernel forms the central part of Linux operating system. It provides all the functionality compulsory to run processes, and it also provides "system services" to give arbitrated and sheltered or protected access to hardware resources. The kernel implements every feature that is required to be eligible as an operating system.
- **THE SYSTEM LIBRARIES:** "the system libraries" describe a typical set of functions through which applications can interrelate through the kernel. And which apply much of the operating system functionality that does not require the full rights or privileges of kernel code.
- **THE SYSTEM UTILITIES:** "the system utilities" are the programs that execute individual, particular and specialized managing tasks.
- Some of the system utilities may be invoked just once to initialize and configure some features of the system; others (known as daemons in UNIX language) may run enduringly, conducting such tasks as responding to inward or incoming network connections, accepting logon requests terminals or updating log records and files.
- The whole kernel code executes in the privileged mode of processor along with the full access to all the physical resources of the computer. This privileged mode in Linux is referred as "kernel mode", equal to the monitor mode.
- In Linux user-mode code is not built into the kernel. Any operating-system-support code that does not require to execute in kernel mode is located into the system libraries as an alternative.
- Because all kernel code and data structures are kept in a single address space, no context switches are necessary when a process calls an operating-system function or when a hardware interrupt is delivered.

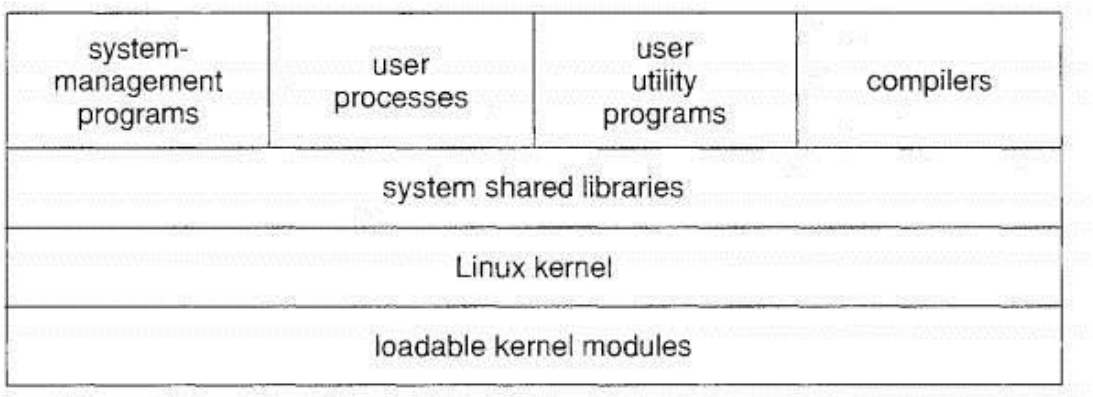


Fig: Components of the Linux system.

- This single address space contains not only the core scheduling and virtual memory code but all kernel code, including all device drivers, file systems, and networking code.
- Even though all the kernel components share this same melting pot, there is still room for modularity.
- The Linux kernel forms the core of the Linux operating system.
- The system libraries provide many types of functionality. At the simplest level, they allow applications to make kernel-system service requests. Making a system call involves transferring control from unprivileged user mode to privileged kernel mode; the details of this transfer vary from architecture to architecture. The libraries take care of collecting the system-call arguments and, if necessary, arranging those arguments in the special form necessary to make the system call.
- The libraries may also provide more complex versions of the basic system calls. For example, the C language's buffered file-handling functions are all implemented in the system libraries, providing more advanced control of file I/ 0 than the basic kernel system calls.
- The LINUX system includes a wide variety of user-mode programs-both system utilities and user utilities.
- The system utilities include all the programs necessary to initialize the system, such as those to configure network devices and to load kernel modules. Continually running server programs also count as system utilities; such programs handle user login requests, incoming network connections, and the printer queues.

Kernel module:

A **kernel module** is a binary image containing code and data structures that runs in the UNIX kernel. It has the following characteristics:

- Is statically loaded as part of /vmunix or dynamically loaded into memory
- Runs in kernel mode
- Has a file name ending with the extension .mod
- Contains a well-defined routine that executes first to initialize the module
- May be a device driver when it performs any one of these additional tasks:
    - Handles interrupts from hardware devices
    - Accepts I/O requests from applications

The kernel contains many modules, some of which are device drivers. In this book, a kernel module is defined more broadly than a device driver because it can be used to perform a variety of functions, including:

- Management functions
- Common functions shared by other modules

- Kernel code that can be compiled, loaded, and unloaded independently
- it allows a Linux system to be set up with standard minimal kernel
- other components loaded as modules
- typically to implement device drivers, file systems, or networking protocols
- **Three components to Linux module support:**
- **module management** • load/unload the module
- resolve symbols (similar to a linker)
- **driver registration**
- kernel define an interface, module implement the interface
- module registers to the kernel, kernel maintain a list of loaded modules
- **conflict resolution**
- resource conflicts

- Tools to support kernel modules: lsmod, rmmod, modprobe

Module Management:

- Supports loading modules into memory and letting them talk to the rest of the kernel
- Module loading is split into two separate sections:
- Managing sections of module code in kernel memory
- Handling symbols that modules are allowed to reference
- The module requestor manages loading requested, but currently unloaded, modules; it also regularly queries the kernel to see whether a dynamically loaded module is still in use, and will unload it when it is no longer actively needed.

Driver Registration:

- Allows modules to tell the rest of the kernel that a new driver has become available
- The kernel maintains dynamic tables of all known drivers, and provides a set of routines to allow drivers to be added to or removed from these tables at any time
- Registration tables include the following items:
  - Device drivers
  - File systems
  - Network protocols
  - Binary format

Device drivers : These drivers include character devices (such as printers terminals, or mice), block devices (including all disk drivers), and network interface devices.

File system: The file system may be anything that implements Linux virtual-file –system calling routines.

Network protocol: A module may implement an entire networking protocol, such as IPX, or simply a new set of packet-filtering rules for a network firewall.

Binary format: This format specifies a way of recognizing, and loading, a new type of executable file.

Conflict Resolution :

- A mechanism that allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver.
- The conflict resolution module aims to:
- Prevent modules from clashing over access to hardware resources
- Prevent auto probes from interfering with existing device drivers
- Resolve conflicts with multiple drivers trying to access the same hardware:
- 1. Kernel maintains list of allocated HW resources
- 2. Driver reserves resources with kernel database first
- 3. Reservation request rejected if resource not available

Process Management :

- A process is the basic context within which all user-requested activity is serviced within the operating system.
- **The Fork/ Exec Process Model:**
  - UNIX process management separates the creation of processes and the running of a new program into two distinct operations.
  - The **fork()** system call creates a new process
  - A new program is run after a call to **exec()**
  - Under UNIX, a process encompasses all the information that the operating system must maintain to track the context of a single execution of a single program
  - Under Linux, process properties fall into three groups: the process ’s identity, environment, and context.

Process Identity:

Process ID (PID) - The unique identifier for the process; used to specify processes to the operating system when an application makes a system call to signal, modify, or wait for another process

Credentials - Each process must have an associated user ID and one or more group IDs that determine the process's rights to access system resources and files

Personality - Not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls.

Used primarily by emulation libraries to request that system calls be compatible with certain specific flavors of UNIX.

Namespace – Specific view of file system hierarchy.

- Most processes share common namespace and operate on a shared file-system hierarchy
- But each can have unique file-system hierarchy with its own root directory and set of mounted file systems.

Process Environment: The process's environment is inherited from its parent, and is composed of two null-terminated vectors:

- The **argument vector** lists the command-line arguments used to invoke the running program; conventionally starts with the name of the program itself.
- The **environment vector** is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values.
- Passing environment variables among processes and inheriting variables by a process's children are flexible means of passing information to components of the user-mode system software.
- The environment-variable mechanism provides a customization of the operating system that can be set on a per-process basis, rather than being configured for the system as a whole.

Process Context:

- The (constantly changing) state of a running program at any point in time
- The **scheduling context** is the most important part of the process context; it is the information that the scheduler needs to suspend and restart the process.
- The kernel maintains **accounting** information about the resources currently being consumed by each process, and the total resources consumed by the process in its lifetime so far.

- The **file table** is an array of pointers to kernel file structures.
- When making file I/O system calls, processes s refer to files by their index into this table, the **file descriptor (fd)**
- Whereas the file table lists the existing open files, the file-system context applies to requests to open new files.
- The current root and default directories to be used for new file searches are stored here
- The **signal-handler table** defines the routine in the process' s address space to be called when specific signals arrive.
- The **virtual-memory context** of a process describes the full contents of the its private address space.

Processes and Threads

Linux uses the same internal representation for processes and threads; a thread is simply a new process that happens to share the same address space as its parent λ Both are called tasks by Linux ν A distinction is only made when a new thread is created by the clone() system call λ fork() creates a new task with its own entirely new task context λ clone() creates a new task with its own identity, but that is allowed to share the data structures of its parent ν Using clone() gives an application fine-grained control over exactly what is shared between two threads

flag	meaning
FS	is
VM	The same memory space is
GHAND	
LES	The set of open files is

Scheduling

- The job of allocating CPU time to different tasks within an operating system.
- While scheduling is normally thought of as the running and interrupting of processes, in Linux, scheduling also includes the running of the various kernel tasks
- Running kernel tasks encompasses both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver
- As of 2.5, new scheduling algorithm – preemptive, priority-based, known as *O*(1)
  - Real-time range
  - nice value
  - Had challenges with interactive performance
  - 2.6 introduced **Completely Fair Scheduler** (**CFS**)

CFS

- Eliminates traditional, common idea of time slice
- Instead all tasks allocated portion of processor's time
- CFS calculates how long a process should run as a function of total number of tasks
- *N* runnable tasks means each gets 1/*N* of processor's time
- Then weights each task with its nice value
- Smaller nice value -> higher weight (higher priority)
- Then each task run with for time proportional to task's weight divided by total weight of all runnable tasks
- Configurable variable **target latency** is desired interval during which each task should run at least once
- Consider simple case of 2 runnable tasks with equal weight and target latency of 10ms – each then runs for 5ms

- If 10 runnable tasks, each runs for 1ms
- **Minimum granularity** ensures each run has reasonable amount of time (which actually violates fairness idea)

Kernel Synchronization

- A request for kernel-mode execution can occur in two ways:
  - A running program may request an operating system service, either explicitly via a system call, or implicitly, for example, when a page fault occurs
  - A device driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt
- Kernel synchronization requires a framework that will allow the kernel's critical sections to run without interruption by another critical section
- Linux uses two techniques to protect critical sections:

1. Normal kernel code is nonpreemptible (until 2.6)
   - when a time interrupt is received while a process is executing a kernel system service routine, the kernel's **need_resched** flag is set so that the scheduler will run once the system call has completed and control is about to be returned to user mode
2. The second technique applies to critical sections that occur in an interrupt service routines

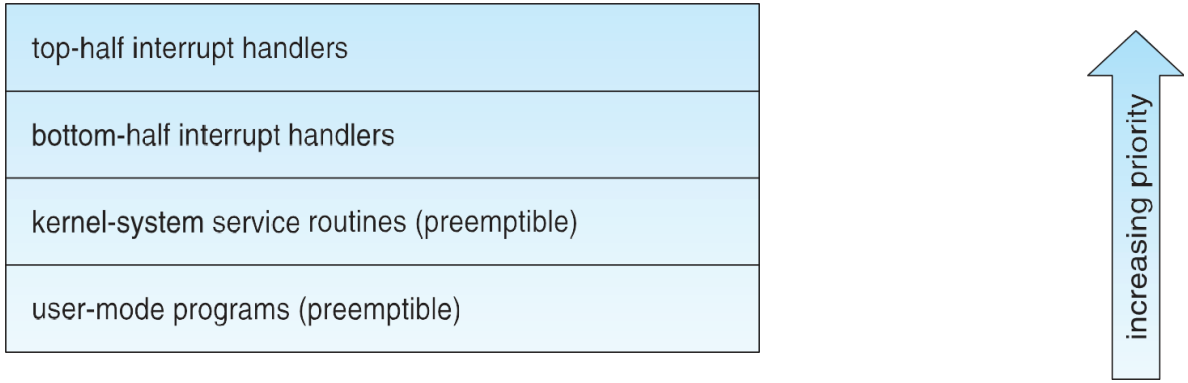
   - By using the processor's interrupt control hardware to disable interrupts during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access of shared data structures
   - Provides spin locks, semaphores, and reader-writer versions of both
     - Behavior modified if on single processor or multi:

single processor	multiple processors
Disable kernel preem	Acq re spin lock.
Enable kernel preem	spin lock.

- To avoid performance penalties, Linux's kernel uses a synchronization architecture that allows long critical sections to run without having interrupts disabled for the critical section's entire duration
- Interrupt service routines are separated into a *top half* and a *bottom half*
  - The top half is a normal interrupt service routine, and runs with recursive interrupts disabled
  - The bottom half is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves
  - This architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code

Interrupt Protection Levels

- each level may be inturrped by Each level may be interrupted by code running
- at a higher level, but will never be interrupted by code running at the same or a



lower level

- User processes can always be preempted by another process when a time-sharing scheduling interrupt occurs

Symmetric Multiprocessing

- Linux 2.0 was the first Linux kernel to support **SMP** hardware; separate processes or threads can execute in parallel on separate processors
- Until version 2.2, to preserve the kernel's nonpreemptible synchronization requirements, SMP imposes the restriction, via a single kernel spinlock, that only one processor at a time may execute kernel-mode code
- Later releases implement more scalability by splitting single spinlock into multiple locks, each protecting a small subsetof kernel data structures
- Version 3.0 adds even more fine-grained locking, processor affinity, and load-balancing

Memory Management

- Linux's physical memory-management system deals with allocating and freeing pages, groups of pages, and small blocks of memory
- It has additional mechanisms for handling virtual memory, memory mapped into the address space of running processes
- Splits memory into four different **zones** due to hardware characteristics
  - Architecture specific, for example on x86:

zone	physical memory
ZONE_DMA	< 6 MB
ZONE_NORMAL	6 – 896 MB
ZONE_HIGHMEM	> 896 MB

Managing Physical Memory

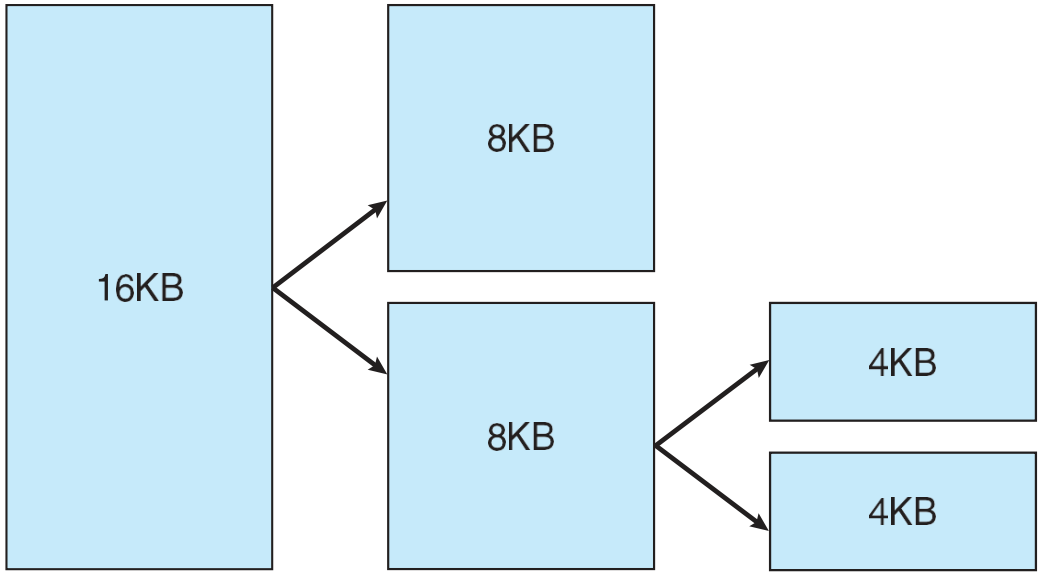
- The page allocator allocates and frees all physical pages; it can allocate ranges of physically-contiguous pages on request
- The allocator uses a buddy-heap algorithm to keep track of available physical pages
- Each allocatable memory region is paired with an adjacent partner
- Whenever two allocated partner regions are both freed up they are combined to form a larger region
- If a small memory request cannot be satisfied by allocating an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request

- Memory allocations in the Linux kernel occur either statically (drivers reserve a contiguous area of memory during system boot time) or dynamically (via the page allocator)
  - Also uses **slab allocator** for kernel memory
  - **Page cache** and virtual memory system also manage

Physical memory

- Page cache is kernel's main cache for files and main mechanism for I/O to block devices
- Page cache stores entire pages of file contents for local and network file I/O

Splitting of Memory in a Buddy Heap



Virtual Memory

- The VM system maintains the address space visible to each process: It creates pages of virtual memory on demand, and manages the loading of those pages from disk or their swapping back out to disk as required.
- The VM manager maintains two separate views of a process's address space:
- A logical view describing instructions concerning the layout of the address space
- The address space consists of a set of non-overlapping regions, each representing a continuous, page-aligned subset of the address space
- A physical view of each address space which is stored in the hardware page tables for the process
- Virtual memory regions are characterized by:
- The backing store, which describes from where the pages for a region come; regions are usually backed by a file or by nothing (**demand-zero memory**)
- The region's reaction to writes (page sharing or copy-on- write
- The kernel creates a new virtual address space
- When a process runs a new program with the exec() system call
- Upon creation of a new process by the fork()system call
- On executing a new program, the process is given a new, completely empty virtual-address space; the program-loading routines populate the address space with virtual-memory regions
- Creating a new process with fork() involves creating a complete copy of the existing process's virtual address space
- The kernel copies the parent process's VMA descriptors, then creates a new set of page tables for the child
- The parent's page tables are copied directly into the child's, with the reference count of each page covered being incremented
- After the fork, the parent and child share the same physical pages of memory in their address spaces

Swapping and Paging

The VM paging system relocates pages of memory from physical memory out to disk when the memory is needed for something else

The VM paging system can be divided into two sections:

- The **pageout-policy** algorithm decides which pages to write out to disk, and when
- The **paging mechanism** actually carries out the transfer,and pages data back into physical memory as needed
- Can page out to either swap device or normal files

- Bitmap used to track used blocks in swap space kept in physical memory
- Allocator uses next-fit algorithm to try to write contiguous runs

Kernel Virtual Memory:

- The Linux kernel reserves a constant, architecture-dependent region of the virtual address space of every process for its own internal use.
- This kernel virtual-memory area contains two regions:
- A static area that contains page table references to every available physical page of memory in the system, so that there is a simple translation from physical to virtual addresses when running kernel code.
- The reminder of the reserved section is not reserved for any specific purpose; its page-table entries can be modified to point to any other areas of memory.

Executing and Loading User Programs

- Linux maintains a table of functions for loading programs; it gives each function the opportunity to try loading the given file when an exec system call is made
- The registration of multiple loader routines allows Linux to support both the ELF and a out binary formats
- Initially, binary-file pages are mapped into virtual memory
- Only when a program tries to access a given page will a page fault result in that page being loaded into physical memory
- An ELF-format binary file consists of a header followed by several page-aligned sections
- The ELF loader works by reading the header and mapping the sections of the file into separate regions of virtual memory

Static and Dynamic Linking:

- A program whose necessary library functions are embedded directly in the program's executable binary file is statically linked to its libraries
- The main disadvantage of static linkage is that every program generated must contain copies of exactly the same common system library functions
- Dynamic linking is more efficient in terms of both physical memory and disk-space usage because it loads the system libraries into memory only once
- Linux implements dynamic linking in user mode through special linker library
- Every dynamically linked program contains small statically linked function called when process starts
- Maps the link library into memory
- Link library determines dynamic libraries required by process and names of variables and functions needed
- Maps libraries into middle of virtual memory and resolves references to symbols contained in the libraries
- Shared libraries compiled to be position-independent code (PIC) so can be loaded anywhere
- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics
- Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the virtual file system (VFS)
- The Linux VFS is designed around object-oriented principles and is composed of four components:
- A set of definitions that define what a file object is allowed to look like The inode object structure represent an individual file
- The file object represents an open file
- The superblock object represents an entire file system
- A dentry object represents an individual directory entry
- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics
- Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the virtual file system (VFS)
- The Linux VFS is designed around object-oriented principles and layer of software to manipulate those objects with a set of operations on the objects

- ❖ For example for the file object operations include (from struct file_operations in /usr/include/linux/fs.h

  int open(. . .) — Open a file
  ssize t read(. . .) — Read from a file
  ssize t write(. . .) — Write to a file
  int mmap(. . .) — Memory-map a file

The Linux ext3 File System:

- ext3 is standard on disk file system for Linux
- Uses a mechanism similar to that of BSD Fast File System (FFS) for locating data blocks belonging to a specific file
- Supersedes older extfs, ext2 file systems
- Work underway on ext4 adding features like extents
- Of course, many other file system choices with Linux distros.
- The main differences between ext2fs and FFS concern their disk allocation policies
- In ffs, the disk is allocated to files in blocks of 8Kb, with blocks being subdivided into fragments of 1Kb to store small files or partially filled blocks at the end of a file
- ext3 does not use fragments; it performs its allocations in smaller units
- The default block size on ext3 varies as a function of total size of file system with support for 1, 2, 4 and 8 KB blocks
- ext3 uses cluster allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation on a block group
- Maintains bit map of free blocks in a block group, searches for free byte to allocate at least 8 blocks at a time

Journaling:ext3 implements journaling, with file system updates first written to a log file in the form of transactions Once in log file, considered committed Over time, log file transactions replayed over file system to put changes in place On system crash, some transactions might be in journal but not yet placed into file system Must be completed once system recovers No other consistency checking is needed after a crash (much faster than older methods) Improves write performance on hard disks by turning random I/O into sequential I/O.

The Linux Proc File System:

The proc file system does not store data, rather, its contents are computed on demand according to user file I/O requests

proc must implement a directory structure, and the file contents within; it must then define a unique and persistent inode number for each directory and files it contains

It uses this inode number to identify just what operation is required when a user tries to read from a particular file inode or perform a lookup in a particular directory inode

When data is read from one of these files, proc collects the appropriate information, formats it into text form and places it into the requesting process's read buffer

Input and Output:

The Linux device-oriented file system accesses disk storage through two caches: Data is cached in the page cache, which is unified with the virtual memory system

- Metadata is cached in the buffer cache, a separate cache indexed by the physical disk block
- Linux splits all devices into three classes:
- block devices allow random access to completely independent, fixed size blocks of data
- character devices include most other devices; they don't need to support the functionality of regular files
- network devices are interfaced via the kernel's networking subsystem

  Block Devices: Provide the main interface to all disk devices in a system
  - The block buffer cache serves two main purposes
  - it acts as a pool of buffers for active I/O. it serves as a cache for completed I/O.
  - The request manager manages the reading and writing of buffer contents to and from a block device driver .
  - Kernel 2.6 introduced Completely Fair Queueing (CFQ)
  - Now the default scheduler.Fundamentally different from elevator algorithms .Maintains set of lists, one for each process by default.
  - Uses C-SCAN algorithm, with round robin between all outstanding I/O from all processes . Four blocks from each process put on at once

Device-Driver Block Structure:

Character Devices: A device driver which does not offer random access to fixed blocks of data .

A character device driver must register a set of functions which implement the driver's various file I/O operations . The kernel performs almost no preprocessing of a file read or write request to a character device, but simply passes on the request to the device .The main exception to this rule is the special subset of character device drivers which implement terminal devices, for which the kernel maintains a standard interface.

Line discipline is an interpreter for the information from the terminal device . The most common line discipline is tty discipline, which glues the terminal's data stream onto standard input and output streams of user's running processes, allowing processes to communicate directly with the user's terminal .

Several processes may be running simultaneously, tty line discipline responsible for attaching and detaching terminal's input and output from various processes connected to it as processes are suspended or awakened by user. Other line disciplines also are implemented have nothing to do with I/O to user process – i.e. PPP and SLIP networking protocols.

Interprocess Communication: Like UNIX, Linux informs processes that an event has occurred via signals .There is a limited number of signals, and they cannot carry information: Only the fact that a signal occurred is available to a process .The Linux kernel does not use signals to communicate with processes with are running in kernel mode, rather, communication within the kernel is accomplished via scheduling states and wait_queue structures .Also implements System V Unix semaphores . Process can wait for a signal or a semaphore .Semaphores scale better.Operations on multiple semaphores can be atomic.

Passing Data Between Processes:

The pipe mechanism allows a child process to inherit a communication channel to its parent, data written to one end of the pipe can be read a the other .

Shared memory offers an extremely fast way of communicating; any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space .

To obtain synchronization, however, shared memory must be used in conjunction with another Inter process communication mechanism.

Network Structure:

Networking is a key area of functionality for Linux .It supports the standard Internet protocols for UNIX to UNIX communications .It also implements protocols native to non-UNIX operating systems, in particular, protocols used on PC networks, such as

Appletalk and IPX . Internally, networking in the Linux kernel is implemented by three layers of software:

The socket interface Protocol drivers Network device drivers Most important set of protocols in the Linux networking system is the internet protocol suite .

It implements routing between different hosts anywhere on the network . On top of the routing protocol are built the UDP, TCP and ICMP protocols Packets also pass to firewall management for filtering based on firewall chains of rules.