Module 22: The Linux System

- History
- Design Principles
- Kernel Modules
- Process Management
- Scheduling
- Memory Management
- File Systems
- Input and Output
- Interprocess Communication
- Network Structure
- Security

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.

The Linux Kernel

- Version 0.01 (May 1991) had no networking, ran only on 80386-compatible Intel processors and on PC hardware, had extremely limited device-driver support, and supported only the Minix file system.
- Linux 1.0 (March 1994) included these new features:
  - Support for 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.

Linux 2.0

- Released in June 1996, 2.0 added two major new capabilities:
  - Support for multiple architectures, including a fully 64-bit native Alpha port
  - Support for multiprocessor architectures
- Other new features included:
  - Improved memory-management code
  - Improved TCP/IP performance
  - Support for internal kernel threads, for handling dependencies between loadable modules, and for automatic loading of modules on demand.
  - Standardized configuration interface
- Available for Motorola 68000-series processors, Sun Sparc systems, and for PC and PowerMac systems.

The Linux System

- Linux uses many tools developed as part of Berkeley's BSD operating system, MIT's X Window System, and the Free Software Foundation's GNU project.
- The main system libraries were started by the GNU project, with improvements provided by the Linux community.
- Linux networking-administration tools were derived from 4.3BSD code; recent BSD derivatives such as FreeBSD have borrowed code from Linux in return.
- The Linux system is maintained by a loose network of developers collaborating over the Internet, with a small number of public ftp sites acting as de facto standard repositories.

Linux Distributions

- Standard, precompiled sets of packages, or *distributions*, include the basic Linux system, system installation and management utilities, and ready-to-install packages of common UNIX tools.
- The first distributions managed these packages by simply providing a means of unpacking all the files into the appropriate places; modern distributions include advanced package management.
- Early distributions included SLS and Slackware. *Red Hat* and *Debian* are popular distributions from commercial and noncommercial sources, respectively.
- The RPM package file format permits compatibility among the various Linux distributions.

Linux Licensing

- The Linux kernel is distributed under the GNU General Public License (GPL), the terms of which are set out by the Free Software Foundation.
- Anyone using Linux, or creating their own derivate of Linux, may not make the derived product proprietary; software released under the GPL may not be redistributed as a binary-only product.

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.
- Main design goals are speed, efficiency, and standardization.
- 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.

Components of a Linux System

System management programs	User processes	User utility programs	Compilers

System Shared Libraries

Linux Kernel

Loadable kernel modules

Components of a Linux System (cont'd)

- Like most UNIX implementations, Linux is composed of three main bodies of code; the most important distinction is between the kernel and all other components.
- The **kernel** is responsible for maintaining the important abstractions of the operating system.
  - Kernel code executes in *kernel mode* with full access to all the physical resources of the computer.
  - All kernel code and data structures are kept in the same single address space.

Components of a Linux System (cont'd)

- The **system libraries** define a standard set of functions through which applications interact with the kernel, and which implement much of the operating-system functionality that does not need the full privileges of kernel code.
- The **system utilities** perform individual specialized management tasks.

Kernel Modules

- Sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel.
- A kernel module may typically implement a device driver, a file system, or a networking protocol.
- The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL.
- Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in.
- Three components to Linux module support:
  - module management
  - driver registration
  - conflict resolution

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

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 *autoprobes* from interfering with existing device drivers
  - Resolve conflicts with multiple drivers trying to access the same hardware

Process Management

- 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 **execve**.
- 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.

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 one 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 refer to files by their index into this table.

Process Context (cont'd)

- 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 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.
- A distinction is only made when a new thread is created by the **clone** system call.
  - **fork** creates a new process with its own entirely new process context
  - **clone** creates a new process with its own identity, but that is allowed to share the data structures of its parent
- Using **clone** gives an application fine-grained control over exactly what is shared between two threads.

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.

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.

Kernel Synchronization (cont'd)

- Linux uses two techniques to protect critical sections:
  1. Normal kernel code is nonpreemptible
     - when a timer 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 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.

Kernel Synchronization (cont'd)

- 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

top-half interrupt handlers	increasing priority
bottom-half interrupt handlers	
kernel-system service routines (not preemptible)	
user-mode programs (preemptible)	decreasing priority

- 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.

Process Scheduling

- Linux uses two process-scheduling algorithms:
  - A time-sharing algorithm for fair preemptive scheduling between multiple processes
  - A real-time algorithm for tasks where absolute priorities are more important than fairness
- A process's scheduling class defines which algorithm to apply.
- For time-sharing processes, Linux uses a prioritized, *credit* based algorithm.
  - The crediting rule

    $$\text{credits} := \frac{\text{credits}}{2} + \text{priority}$$

    factors in both the process's history and its priority.
  - This crediting system automatically prioritizes interactive or I/O-bound processes.

Process Scheduling (cont'd)

- Linux implements the FIFO and round-robin real-time scheduling classes; in both cases, each process has a priority in addition to its scheduling class.
  - The scheduler runs the process with the highest priority; for equal-priority processes, it runs the longest-waiting one
  - FIFO processes continue to run until they either exit or block
  - A round-robin process will be preempted after a while and moved to the end of the scheduling queue, so that round-robin processes of equal priority automatically time-share between themselves.

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.
- 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.

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.

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).

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 nonoverlapping 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 (cont'd)

- 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
  1. When a process runs a new program with the **exec** system call
  2. Upon creation of a new process by the **fork** system call

Virtual Memory (cont'd)

- 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.

Virtual Memory (cont'd)

- 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.

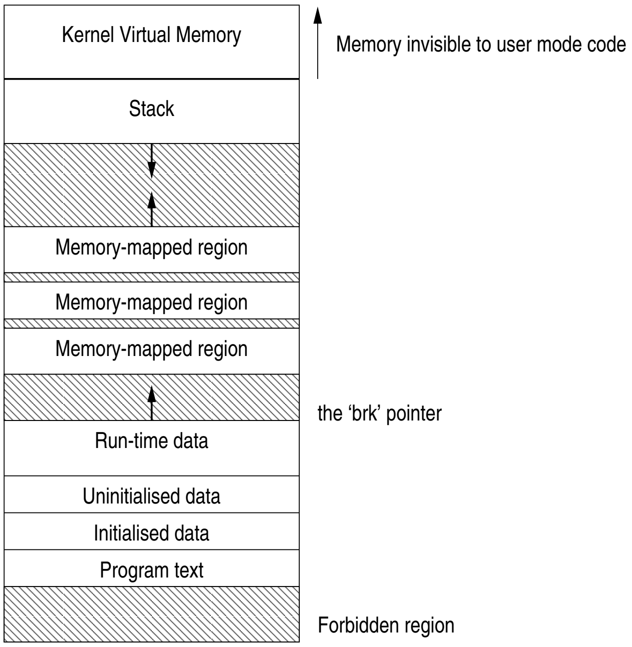
Virtual Memory (cont'd)

- 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 remainder 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.

Memory Layout for ELF Programs



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.

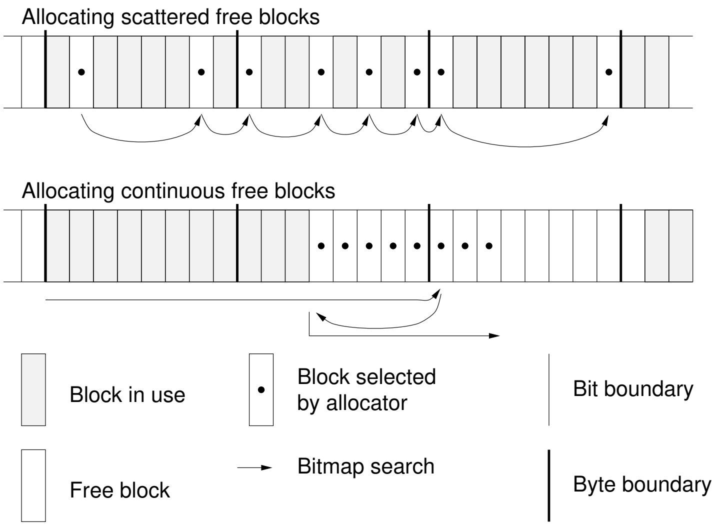
File Systems

- 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 two components:
  - A set of definitions that define what a file object is allowed to look like
    - the *inode-object* and the *file-object* structures represent individual files
    - the *file system object* represents an entire file system
  - A layer of software to manipulate those objects.

The Linux Ext2fs File System

- Ext2fs uses a mechanism similar to that of BSD Fast File System (ffs) for locating data blocks belonging to a specific file.
- 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.
  - Ext2fs does not use fragments; it performs its allocations in smaller units. The default block size on ext2fs is 1Kb, although 2Kb and 4Kb blocks are also supported.
- Ext2fs uses 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.

Ext2fs Block-Allocation Policies



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.

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.

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.

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 at 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 interprocess-communication mechanism.

Shared Memory Object

- The shared-memory object acts as a backing store for shared-memory regions in the same way as a file can act as backing store for a memory-mapped memory region.
- Shared-memory mappings direct page faults to map in pages from a persistent shared-memory object.
- Shared-memory objects remember their contents even if no processes are currently mapping them into virtual memory.

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 nonUNIX 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

Network Structure (cont'd)

- The 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.

Security

- The *pluggable authentication modules* (*PAM*) system is available under Linux.
- PAM is based on a shared library that can be used by any system component that needs to authenticate users.
- Access control under UNIX systems, including Linux, is performed through the use of unique numeric identifiers (**uid** and **gid**).
- Access control is performed by assigning objects a *protection mask*, which specifies which access modes—read, write, or execute—are to be granted to processes with owner, group, or world access.

Security (cont'd)

- Linux augments the standard UNIX **setuid** mechanism in two ways:
  - It implements the POSIX specification's *saved user-id* mechanism, which allows a process to repeatedly drop and reaquire its effective uid
  - It has added a process characteristic that grants just a subset of the rights of the effective uid.
- Linux provides another mechanism that allows a client to selectively pass access to a single file to some server process, without granting it any other privileges.