Advanced Operating Systems
Theory

Christian Rossi

Academic Year 2024-2025

Abstract

This course provides an overview of key topics related to operating systems, focusing on design patterns, resource management, and peripheral interaction. It begins by introducing the main goals of operating systems, detailing design patterns and mechanisms for mediating and regulating access to system resources. It also categorizes operating systems into different types, such as monolithic, microkernel, hybrid, and uni-kernel, with examples for each.

The section on resource management and concurrency covers support for multi-process and multi-threaded execution, CPU scheduling for both single and multiprocessors, and modern load-balancing techniques for NUMA systems. Topics such as memory consistency, multi-threaded synchronization, advanced locking techniques, lockless programming, and inter-process communication (IPC) primitives are discussed in detail. Additionally, asynchronous programming, deadlock, starvation, virtual memory management, and the basics of software and hardware virtualization, including containers, are explored.

In peripheral and persistence management, the document addresses low-level peripheral access, communication buses like PCI, programmable interrupt controllers, and interrupt management. Topics include character-based and block-based I/O, the development of device drivers, and an overview of modern storage devices (e.g., HDDs, SSDs) and file systems, emphasizing the filesystem-centric view of peripherals.

Lastly, run-time support is discussed, including boot loaders, kernel initialization, device discovery mechanisms (ACPI and device trees), C runtime support, the structure of dynamic libraries, and linking. Tools for the development, analysis, and profiling of embedded code are also reviewed.

Contents

CHAPTER 1

Operating Systems

1.1 Introduction

The primary objectives of an Operating System (OS) include:

- *Resource management*: the OS allows programs to be created and executed as though they each have dedicated resources, ensuring fair and efficient use of these resources. Common resources managed include the CPU, memory, and disk. For the CPU, time-sharing mechanisms are employed, while memory is often divided into multiple regions for better management.
- *Isolation and protection*: the OS ensures system reliability and security by controlling access to resources such as memory. This prevents conflicts, ensures that one application doesn't interfere with another or access sensitive data, enforces data access rights, and guarantees mutual exclusion when necessary.
- *Portability*: the OS uses interface and implementation abstractions to simplify hardware access and management for applications, effectively hiding the underlying complexity (using the facade pattern). Additionally, these abstractions allow the same applications to function across systems with different physical resources, facilitating compatibility, such as running older applications on newer systems.
- *Extensibility*: the OS employs interface and implementation abstractions to create uniform interfaces for lower layers, which enables the reuse of upper-layer components like device drivers. Additionally, these abstractions help hide the complexity associated with different hardware variants, such as different peripheral models, using patterns like the bridge pattern.

1.2 Resource management

Effective resource management can be achieved through CPU multiplexing, process control, and efficient scheduling.

1.2.1 CPU multiplexing

CPU multiplexing enhances CPU utilization by allowing the processor to switch between different processes, such as when a program is waiting for input and other processes need to run. To minimize the overhead of context switching, it is crucial to optimize latency. This can be achieved by quantizing the time allocated to each process, ensuring efficient use of the CPU's capabilities.

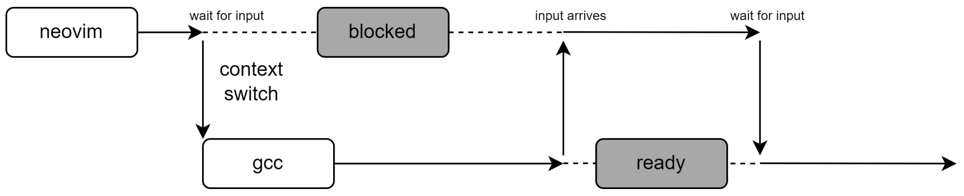


Figure 1.1: CPU multiplexing

1.2.2 Process control

The state of a process reflects its current condition and determines its capabilities, the resources it is using, and the conditions required to transition out of that state.

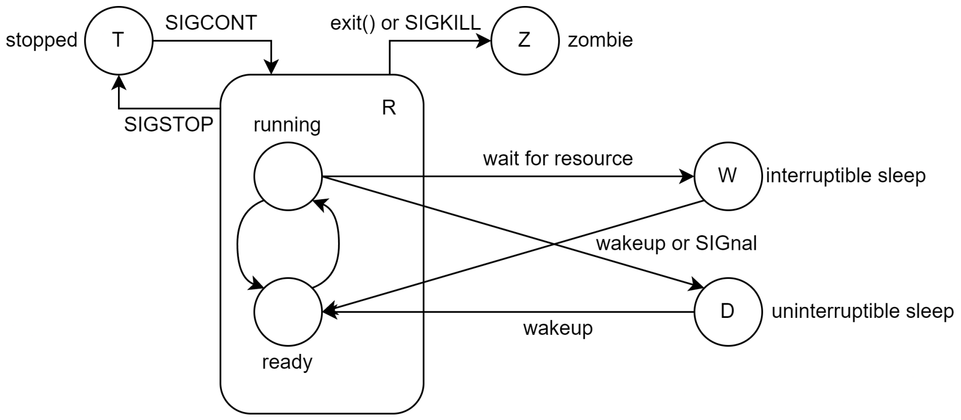


Figure 1.2: Process states

In Linux, each process is represented by a Process Control Block (PCB). The PCB stores vital information about the process, including its Process Identifier (PID), process context (architectural state), virtual memory mappings, open files (including memory-mapped files), credentials (user/group ID), signal handling information, controlling terminal, priority, accounting statistics, and more.

In preemptive OS, process switching occurs when the kernel regains control, typically through mechanisms such as interrupts and exceptions. During a context switch, the current process's context is saved into its PCB, and the PCB of the next process is loaded into the machine's state, enabling seamless switching between processes.

1.2.3 Scheduling

The operating system uses several criteria to determine which process should run next, aiming to balance multiple objectives. Fairness is a key consideration, ensuring that no process is starved of resources. Throughput is also important, as the system seeks to maintain good overall performance. Efficiency is crucial as well, minimizing the overhead introduced by the scheduler itself. Additionally, the system must account for priority, reflecting the relative importance of different processes, and deadlines, where certain tasks must be completed within specific time constraints.

There is no single universal scheduling policy because these goals often conflict with one another, such as the tension between meeting deadlines and ensuring fairness. The appropriate solution varies depending on the problem domain, with General-Purpose OS (GPOSes) requiring different approaches than Real-Time OS (RTOSes):

- *General-Purpose OS*: GPOSes prioritize fairness and throughput, although the specific definition of throughput can vary depending on the application. These systems typically implement a CPU timeslice mechanism, where tasks are preempted based on their allocated timeslice, unless they are blocked. Lower-priority tasks are allowed to consume their share of CPU resources, and the system is organized in a best-effort manner, meaning that while there are no guarantees, the OS strives to manage resources effectively.

- *Real-Time OS*: RTOSes focus more on meeting deadlines and prioritizing tasks efficiently. These systems assume that higher-priority threads do not always run continuously, but when a higher-priority thread becomes available, it is immediately granted control, without waiting for the current thread to complete its allocated processor time. In cases where meeting deadlines is critical, the OS is classified as Hard Real-Time (Hard RT). The emphasis in RTOS is on ensuring that high-priority tasks are executed promptly, reflecting their critical nature.

The main scheduling policies are the following.

Name	**Goal**	**Where it is used**
FIFO	Turnaround	Linux
Round robin	Response time	Linux
CFS	CPU fair share	Linux
EDF	Real-time	Linux
MLFQ	Response time	Solaris, Windows, macOS, BSD
SJF/SRTF/HRRN	Waiting time	Custom

1.3 Isolation and protection

To ensure isolation and protection, the operating system employs a Virtual Address Space (VAS), which encompasses all the memory locations a program can reference. This space is typically isolated from other processes, though certain portions may be shared in a protected manner. The VAS is constructed from various virtual memory areas, some of which are derived from the program's on-disk representation, others are dynamically created during execution, and some are entirely inaccessible, such as kernel space.

The usage of virtual address space does not directly correspond to the actual physical memory in use. Instead, it is fragmented to accommodate the most recently accessed portions. The remaining pages are stored in mass storage. For dynamically modified data, the swap area is utilized, while read-only data and executable code are retrieved from the original program on disk.

This system of indirection, known as paging, effectively cheats by allowing each process to operate as if it has access to a larger memory resource than physically available. As a result, a limited physical memory resource is perceived as abundant from the perspective of individual processes.

1.4 Portability and extendibility

Software design patterns are standardized solutions to common problems encountered in software design. They serve as reusable templates that can be adapted to address recurring challenges in code structure and implementation. In OS design, patterns play a significant role. Two of the most commonly employed patterns in OS design are the Facade and Bridge patterns, which help simplify and manage system complexity.

1.4.1 Facade

The Facade pattern provides a simplified interface to a complex subsystem, which may have many intricate components. While the facade typically offers limited functionality compared to interacting with the subsystem directly, it focuses on exposing only the features that are essential and relevant to the client.

For example, a system call to write to a file abstracts the complexity of interacting with the appropriate hardware device at the low level, making the interaction easier for the developer.

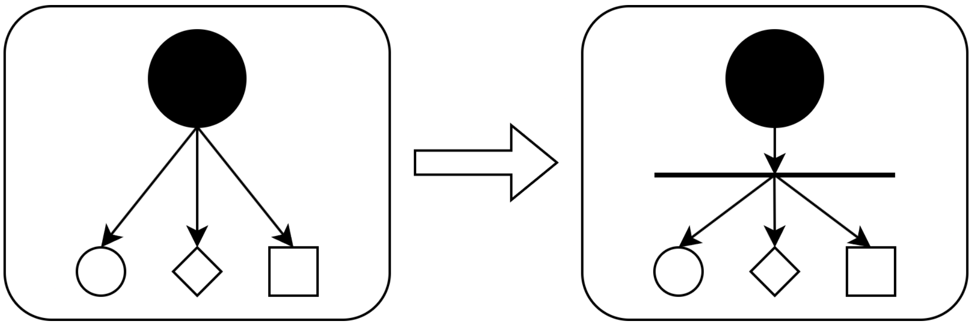


Figure 1.3: Facade pattern

The Facade pattern is useful when a simpler, higher-level interface to an underlying system is desired. It often exposes only high-level APIs and hides the lower-level complexities. This pattern is implemented using exceptions and is commonly seen in system calls.

System calls A system call (syscall) is a mechanism that allows an application to request a privileged service from the operating system's kernel. Since applications run in an unprivileged mode, they rely on system calls to perform tasks that require higher privileges.

When a system call is invoked, a special instruction triggers an exception that transfers control to the kernel, allowing it to process and validate the request. This is similar to a library call but occurs in the more privileged kernel code.

System calls are identified by numbers listed in a syscall table. The system call number, along with its function parameters, is placed in designated CPU registers before the call is executed.

On Linux systems, you can observe the system calls made by a process using the `strace` tool in real time.

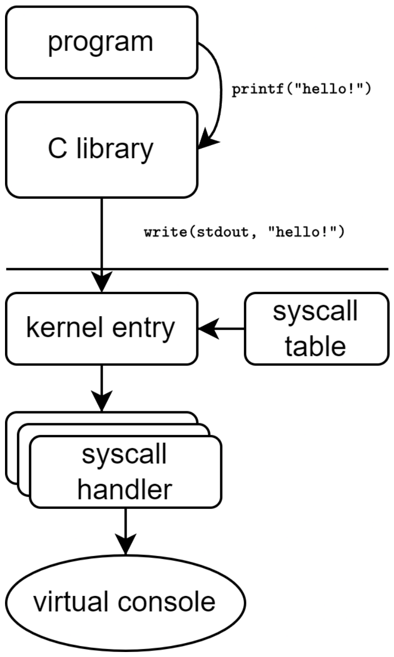


Figure 1.4: System call

1.4.2 Bridge

The Bridge pattern is designed to decouple an abstraction from its implementation, allowing both to be defined and extended independently. This separation enables flexibility, as the implementation can be selected or changed at runtime, rather than being fixed at compile time.

For example, in operating systems like Unix, the way the file system hierarchy is structured and exposed should remain independent of the actual file system being used (e.g., ext4, NTFS, FAT32). This allows different file systems to be mounted without modifying the core abstraction of the file system interface.

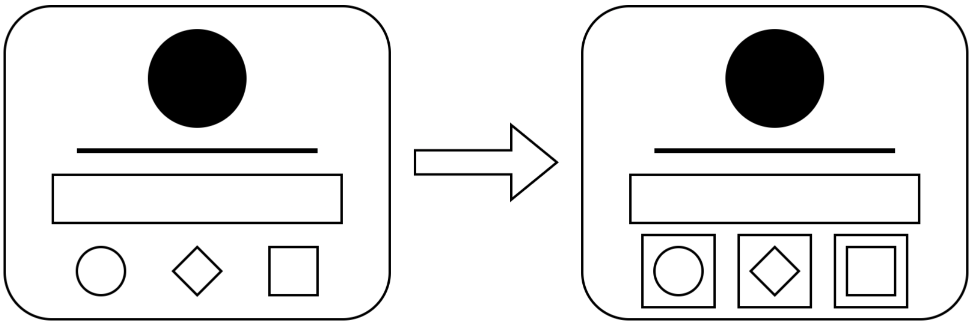


Figure 1.5: Bridge pattern

The Bridge pattern is particularly useful for organizing monolithic code that contains multiple variations of functionality. It divides the abstraction and its variants, improving modularity and maintainability. This pattern is commonly implemented using virtual classes and interfaces, and it is often used in file systems and peripheral drivers.

File system A file system is a set of mechanisms and policies used to manage access to persistent storage. It operates across multiple storage devices (such as hard drives, SSDs, RAM, and network storage) and supports various formats (e.g., FAT32, NTFS, ext2, ext3, ext4). The file system allows multiple processes to access storage concurrently while ensuring data integrity and security. The primary goals of a file system include providing:

- A common abstraction for storage devices.
- Efficient space management.
- Protection and security for stored data.

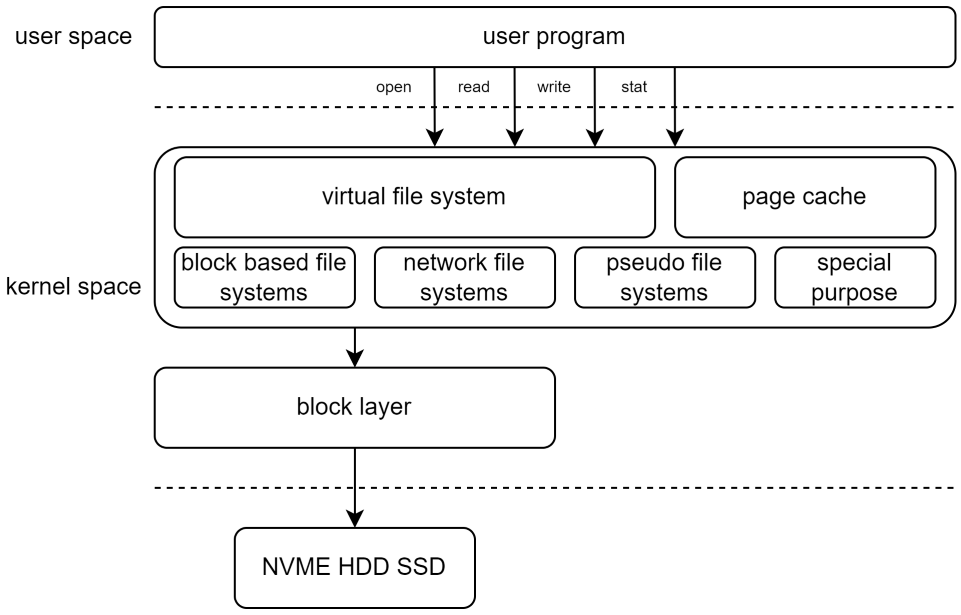


Figure 1.6: File system

1.4.3 Other patterns

Operating systems also utilize various behavioral design patterns to manage complex interactions and processes. Some of the key patterns used are:

- *Chain of Responsibility*: the Chain of Responsibility pattern allows a request to be passed along a chain of handlers. Each handler in the chain decides whether to process the request or pass it on to the next handler. This approach helps distribute the responsibilities and makes the system more flexible. For instance, in file systems, when serving data for a file, the request might first check if the data is available in a cache. If not, it passes the request to the disk for retrieval.

- *Command*: the Command pattern encapsulates a request as a stand-alone object that contains all necessary information about the request. This allows requests to be passed as method arguments, queued for later execution, or even undone if needed. It is particularly useful for decoupling the sender of the request from the receiver. For instance, block I/O requests in an operating system can be queued or delayed, allowing for more control over their execution. Additionally, undoable operations can be implemented using this pattern, enabling rollback in case of failure.

1.5 Operating systems architectures

The design of an operating system can adopt a hybrid of different architectural approaches, including: bare metal, monolithic (with modules), micro-kernel, hybrid, and library.

1.5.1 Bare metal

Bare metal programming is typically employed in scenarios where there is a single-purpose application that demands high control of the hardware along with strict timing requirements. This approach is also favored when low power consumption is essential, and there is no need for abstractions such as tasks.

1.5.2 Monolithic

In a monolithic architecture, there is a single large kernel binary. Device drivers and the kernel are part of the same executable and reside in the same memory area. Examples include Linux, Embedded Linux, AIX, HP-UX, Solaris, and *BSD.

Monolithic with modules This variation of the monolithic architecture includes only a subset of core components within the kernel. Additional services are implemented via external modules, which can be dynamically linked on demand at runtime.

1.5.3 Microkernel

All non-essential components of the kernel are implemented as processes in user space. A single small kernel provides minimal process and memory management, as well as communication facilities through message passing.

Due to its asynchronous nature, a crash of a system process does not necessarily result in a crash of the entire system.

Service invocation occurs between user-level client/server programs through message passing. Examples of such systems include SeL4, GNU Hurd, MINIX, and MkLinux.

1.5.4 Hybrid

Hybrid architectures are similar to microkernels but include some additional code in kernel space to enhance performance. Some services, such as the network stack or filesystem, run in kernel space, while device drivers operate in user space. Examples include Windows NT, 2000, XP, Vista, 7, 8, 8.1, 10, and macOS.

1.5.5 Library amd unikernels

In library operating systems, services such as networking are provided in the form of libraries that are compiled with the application and configuration code. A unikernel is a specialized, single address space machine image that can be deployed in cloud or embedded environments (RTOSes). Examples include FreeRTOS, IncludeOS, and MirageOS.

CHAPTER 2

Processes

2.1 Introduction

Definition (*Task*)**.** In the context of OS, a task is defined as a basic unit of work or a program that is scheduled for execution. It represents an instance of a program in execution, including its code, data, and context.

In Linux, a task is the common factor between a Unix process and a thread. Threads are tasks that shares the same address space, while a process is a single task.

2.1.1 Task control block

Each task in linux is represented into the memory with a structure called `task_struct`. The OS manages a list of `task_struct` to manage all the tasks and also an hash table to simplify certain operations.

For some hardware we need to add the `thread_info` structure at the end of the kernel mode stack to reach the `task_struct`. This is helpful when the number of registers is limited.

The main elements of `task_struct` and `thread_info` are:

- `thread_struct`: used for selecting the correct tasks when performing a context switch.
- `preempt_count`: avoid context swith if the current process is interacting directly with the kernel. The context swith is negated when this integer (in the active task) is greater than zero.
- `mm_struct`: this structure describes the memory layout of the task (only the process accessible space).
- `task_struct`: describes the task state.

The most important task states are shown in the following diagram.

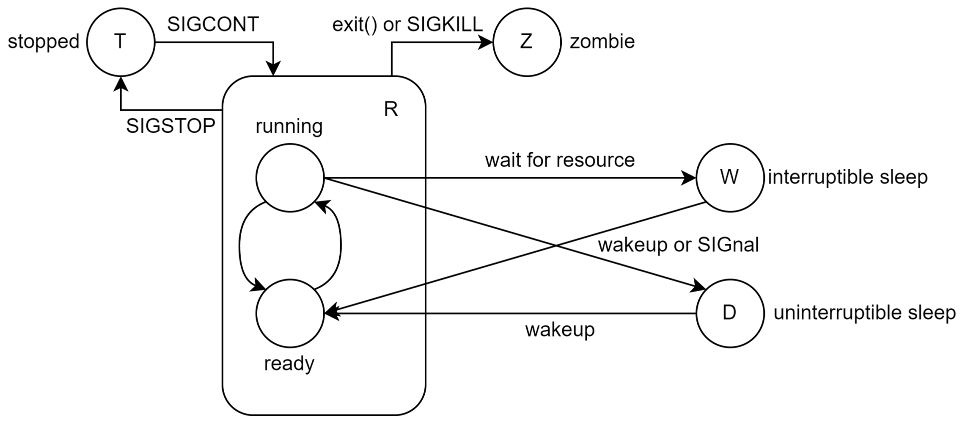


Figure 2.1: Tasks states

In kernel code, you typically enter the wait state and queue with `wait_event` and `wait_event_interrupt`
calls, respectively. The queue is used to schedule the tasks execution.

2.2 Task hierarchy

Cloning The `fork` system call invokes `sys_clone` to create a new copy of the current `task_struct` (the process descriptor). This new process, or child, is nearly identical to the parent but differs in a few key aspects:

- *PID*: the child is assigned a unique Process ID (PID).
- *PPID*: the Parent Process ID (PPID) of the child is set to the parent's PID.
- Certain resources, such as pending signals, are not inherited by the child.

Copy-on-Write Instead of duplicating the entire process address space during cloning, both the parent and child processes share the same memory pages. However, when either process attempts to modify the shared data, a copy of the data is created for that process, ensuring that each process has its own unique copy after a write operation.

2.2.1 Operating System initialization

The initialization of an operating system begins with the execution of a function called `start_kernel`. This function is responsible for performing the essential operations required by the system's architecture to boot the kernel and initialize the core components of the operating environment.

Once the basic elements of the kernel have been set up, the function `rest_init` is invoked. Its role is to create a new kernel thread, which calls the `kernel_init` function. This new thread is assigned a PID of 1 and is responsible for initializing all long-term services that are crucial to the system's ongoing operations.

Before proceeding with other tasks, the system calls the architecture-specific function `cpu_idle`, associated with PID 0. This function places the CPU in a low-power, idle state, waiting for other tasks to be scheduled. If no tasks are ready to execute, the CPU remains idle to conserve power. This idle process continues running for the entire lifespan of the kernel, ensuring that the CPU efficiently manages power when no active work is available.

System V In System V-based systems, initialization follows a structured approach where all files are scanned and organized into predefined run levels. Each run level represents a specific state of the system, controlling which services and processes should be running. The `init` process starts all services associated with the current run level in parallel, only moving to the next run level once the current one has fully initialized. The configuration files that define each run level are called `rc` scripts. However, due to the complexity and time-consuming nature of this sequential approach, a more modern system, known as System D, was developed.

System D In System D (also known as systemd), all initialization tasks are run in parallel, significantly speeding up the boot process. The core concept in systemd is the unit, which is a plain text file that defines the details of each service or task required during system startup. These units describe how and when services should be started, allowing for efficient, concurrent initialization.

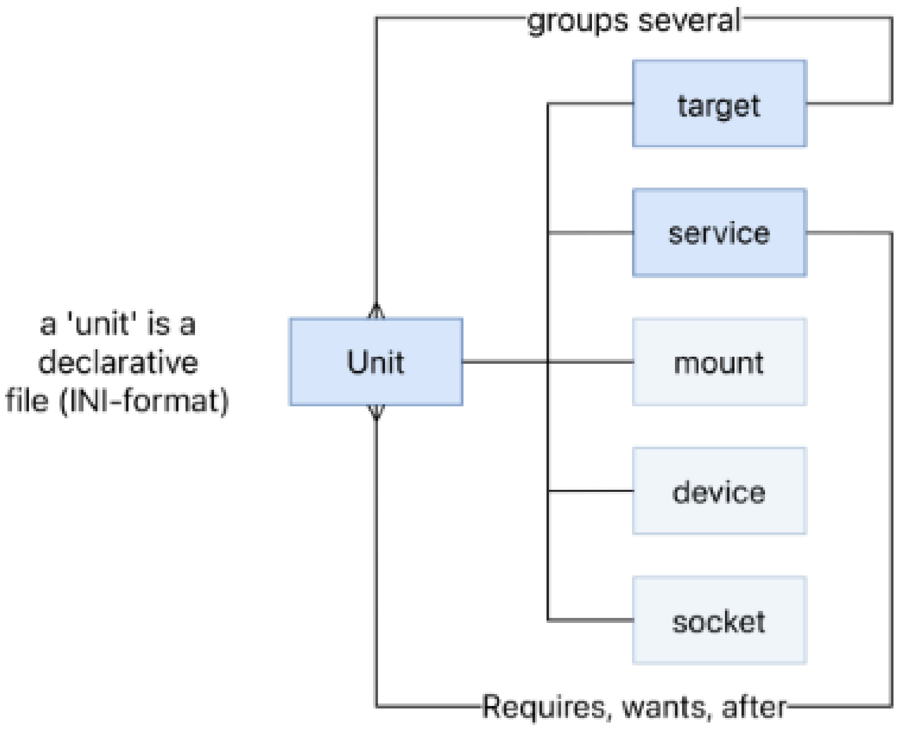


Figure 2.2: System D units

2.3 Task scheduling

A task switch occurs whenever the system needs to execute a task other than the currently running one. This switch is typically triggered by a system call or an interrupt.

When the new process enters kernel mode, it saves the current exception state and registers onto the kernel stack. If the process encounters a nested exception, this state is not added to the stack. Instead, only the `preempt_count` is incremented, simplifying the handling of nested tasks by avoiding unnecessary stack operations.

Upon task completion, the `handle` function asks the kernel if there are other tasks ready to run. It checks whether `preempt_count` is zero before selecting the next process according to a specific scheduling policy. Once the next process is chosen, the function `switch_to` is called.

This function saves the callee-saved registers of the outgoing process onto its stack and loads the corresponding registers of the incoming process, facilitating the switch between tasks.

A scheduling class is an API (a set of functions) that includes policy-specific code. It provides functions to:

- Update current task statistics (`task_tick`).
- Select the next task from the scheduling queue (`pick_next_task`).
- Choose the CPU core on which the task should be queued (`select_task_rq`).
- Place the task in the appropriate queue (`enqueue_task`).

This structure allows developers to implement custom thread schedulers without needing to rewrite generic scheduling code, helping to minimize bugs and improve system efficiency.

All processes have a priority level denoted a π. We can categorize processes as follows:

- *Real-time processes*: these have $\pi \in [0, 99]$ and belong to scheduling classes such as `SCHED_FIFO`, `SCHED_RR`, `SCHED_DEADLINE` (with $\pi = 0$ for deadline scheduling). In this case, the priority is referred to as `rt_priority`.
- *Non-realtime processes*: These belong to scheduling classes like `SCHED_OTHER`, `SCHED_BATCH` ($\pi \approx 139$), and `SCHED_IDLE` ($\pi \approx 139$). For these processes, the priority is called `static_prio`, which is determined by a nice value $v \in [-20, +19]$:

$$\pi(v) = 120 + v$$

2.3.1 Linux run queue

The core data structure used by the scheduler to manage processes waiting to run is the run queue, represented by the `rq` structure. Each CPU has its own `rq`, which reduces contention during task selection. Inside each `rq`, there are sub-queues like `cfs_rq`, `rt_rq`, and `dl_rq`, corresponding to different scheduling classes.

The `rq` structure tracks the number of running tasks, CPU state, and load metrics. The `cfs_rq` structure, used by the Completely Fair Scheduler (CFS), employs an `rb_tree` to manage tasks as `sched_entity` objects, which hold data related to CFS statistics.

The `sched_class` determines how tasks are scheduled, while the `thread_info` structure sets scheduling flags like `TIF_NEED_RESCHED`, which forces a context switch when set.

Multiple run queues can exist on a CPU through the use of task groups. By default, each CPU has a single task group known as the root group. When using control groups (cgroups), multiple task groups can be defined, each with its own set of scheduling entities and run queues on each CPU.

2.4 Completely Fair Scheduling

The Completely Fair Scheduler (CFS) was introduced as a solution to address limitations in the original $O(1)$ scheduler, which imposed rigid decisions regarding timeslices and priorities. The $O(1)$ scheduler struggled to balance CPU-bound and I/O-bound tasks effectively, leading to issues like excessive context switching, especially for low-priority, compute-heavy processes. CFS takes a more dynamic approach, adjusting timeslices based on system load and task priority.

The key features are the following:

1. *Dynamic timeslice assignment*: CFS assigns each process a timeslice proportional to the system load, ensuring efficient distribution of CPU resources. This approach minimizes context switching overhead for compute-bound processes while also accommodating I/O-bound processes more effectively.

2. *Fair CPU time allocation*: each process's timeslice (τ_p) is computed using the formula:

$$\tau_p = \max\left(\frac{\lambda_p \bar{\tau}}{\sum \lambda_i}, \mu\right)$$

   Here, λ_p is the weight of the process (derived from its priority), $\bar{\tau}$ is the configurable schedule latency (default: 6ms), and μ is the minimum granularity (default: 0.75ms). This formula ensures that processes receive a fair share of CPU time based on their priority, with the minimum granularity parameter guaranteeing that even low-priority tasks get some CPU time.

3. *Exponential weighting*: the process weight λ_i is calculated using an exponential formula:

$$\lambda_i = kb^{v_i}$$

   Here, $k = 1024$ and $b = 1.25$, and v_i is the process's nice value (priority). This gives finer control over process prioritization and CPU allocation.

4. *vruntime (ρ) for fairness*: CFS tracks a virtual runtime (ρ) for each process, which reflects how much CPU time a process has used relative to its weight. Lower ρ values indicate higher priority, and tasks with higher weights typically have lower ρ. The goal is to balance ρ values across all processes, ensuring that each task receives its fair share of CPU time, even when priorities differ. The relationship is described by:

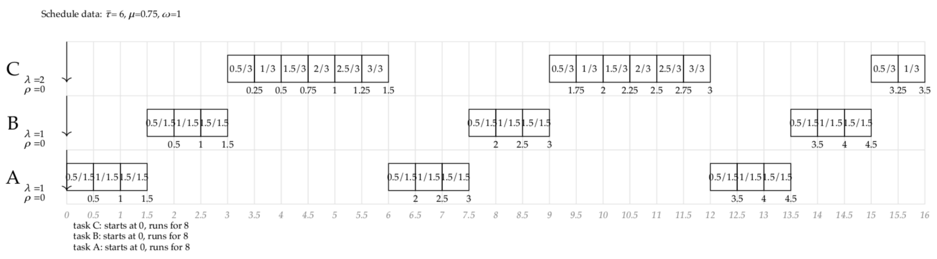
$$\Delta\rho_p = \frac{\tau_p}{\lambda_p} = \frac{\bar{\tau}}{\sum \lambda_i}$$

   This formula highlights that the increase in a process's ρ over time depends only on the total weight of all runnable processes, allowing for equitable CPU distribution.

The purpose of CFS is to ensure fair CPU time allocation among all processes. It achieves this by scaling task priorities using the nice value, which adjusts the proportion of CPU time a process receives based on its priority. This design helps balance the needs of different processes, promoting fairness across the system.

However, CFS has some limitations. One major drawback is its latency sensitivity. CFS struggles to handle processes that require quick access to the CPU but only need short bursts of processing time. It lacks a dedicated mechanism to account for such latency-sensitive tasks. Additionally, while real-time scheduling classes can address latency concerns, they are privileged and can disrupt the fairness of the system by monopolizing CPU resources, making them an imperfect solution.

Example:
The diagram below illustrates a time chart for the Completely Fair Scheduler (CFS) with three tasks, labeled A, B, and C, each running for eight time units. Task A and Task B both have a weight (λ) of 1, while Task C has a higher weight of 2. At the beginning, none of the tasks have any priority adjustments, indicated by the initial virtual runtime (ρ) being 0.



At each timer interrupt, CFS updates the virtual runtime (ρ_i) and the total execution time (ϵ_i) for each task based on the elapsed time (Δt) since the last update. The update formula, as shown in the previous slide, governs how these values change over time.

When a task either blocks or finishes its allocated timeslice (τ_i), CFS selects the next task with the smallest ρ from a red-black tree. This tree efficiently manages tasks, keeping them sorted by their virtual runtime and allowing quick insertion and extraction of the minimum runtime task, both with logarithmic time complexity.

In summary, CFS strives to balance fairness and responsiveness by dynamically allocating CPU resources based on system load and process priority. However, its design does not fully accommodate tasks with specific latency requirements, leaving room for further refinement.

2.4.1 Control Groups

The CFS alone is insufficient to fully optimize CPU usage, especially in multi-user or multi-threaded environments. To better manage CPU resource distribution, Control Groups (CGroups) are used to limit, throttle, and account for the CPU usage of tasks. CGroups provide a more granular and fair way of allocating CPU resources, especially when dealing with multiple users or processes.

Example:
Consider two users: User A has 2 threads, while User B has 98 threads. Without CGroups, CFS would allocate only 2% of the CPU time to User A, which is unfair since both users should ideally receive an equal share of the CPU, regardless of the number of threads. With CGroups, the CPU is divided equally between users first, and then their respective shares are distributed among their threads. This prevents users with more threads from receiving an unfairly large portion of the CPU time.

To achieve this, the system treats each user as if they were a single task in a root run queue. Each user is assigned their own specific run queue, where their threads take turns consuming the user's allocated timeslice. This grouping of tasks into user-specific run queues is managed through CGroups.

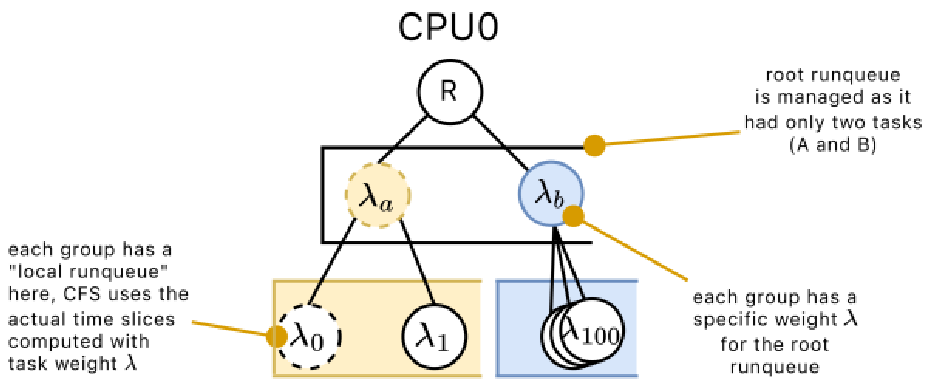

CPU0
R
root runqueue is managed as it had only two tasks (A and B)
λ_a
λ_b
each group has a "local runqueue" here, CFS uses the actual time slices computed with task weight λ
λ_0
λ_1
λ_{100}
each group has a specific weight λ for the root runqueue

Figure 2.3: Control Groups implementation

The diagram illustrates how CFS integrates with CGroups to manage tasks. In the root run queue (e.g., CPU0), the system handles user task groups, treating them as schedulable entities similar to individual tasks. For example, User A and User B, although they represent task groups with multiple threads, are treated as two tasks in the root queue. Each user task group has its own local run queue, where tasks are scheduled independently, with CFS determining the time slices based on the weights (denoted as λ) of the tasks.

At the root level, each task group is assigned a specific weight (λ), which determines how much CPU time the group receives. This ensures fairness between different groups. CGroups can be nested in a hierarchical structure, allowing task groups to include other groups, all the way up to the root task group.

CGroups are particularly useful for isolating core workloads from background processes, ensuring that critical tasks are not delayed by less important jobs.

CGroup creation The steps to create a CGroup are:

1. Create the appropriate entries in `/sys/fs/cgroup/<group name>`.
2. Add the task's process ID (PID) to the `cgroup.procs` file.

By managing resources through CGroups, you can fine-tune the system to ensure critical workloads receive the necessary resources without interference from background processes, leading to improved system efficiency and responsiveness.

2.4.2 Load balancing

In systems with more than one CPU, ensuring equal distribution of work, or load balancing, can become quite complex, especially when considering tasks with different priorities. Basic strategies like balancing based on the number of threads or the total load of each run queue are often ineffective.

For instance, balancing based solely on thread count can result in high-priority threads receiving the same CPU time as low-priority threads, which defeats the purpose of prioritization. Similarly, balancing based on the total load of a run queue may seem appropriate initially, but if one queue contains a high-priority thread that frequently sleeps, the CPU managing that queue may end up idle. This CPU would then have to steal tasks from other, more loaded CPUs, leading to inefficiency and performance degradation.

A more effective approach is to balance based on total weighted load, where both CPU usage and thread priority are considered. By defining a process's CPU usage and computing its weighted load, a more balanced and fair distribution of CPU time is achieved. This method ensures that high-priority threads receive more appropriate CPU time without overwhelming the system with low-priority threads.

Let's define $\gamma_i q$ as the CPU usage of process i on run queue q, and Ω^p as the total weighted load, calculated as:

$$\Omega^p = \sum_i \lambda_{i,q}\gamma_{i,q}$$

Here, $\lambda_{i,q}$ represents the weight (priority) of process i on run queue q. Balancing the system on the total weighted load, Ω^p, ensures a more efficient load distribution that takes task priority into account.

To understand load balancing in Linux, it's important to consider the hierarchical layout of processor cores, caches, and memory, particularly in systems with Non-Uniform Memory Access (NUMA). In NUMA architectures, the memory access cost varies depending on the distance between the processor and the memory it accesses. If a thread is moved across different memory banks on separate dies, it incurs significant performance penalties. Hence, the Linux load balancer must adopt a hierarchical approach, ensuring that threads stay close to their memory and cache domains whenever possible.

To minimize these penalties, the Linux load balancer works to keep threads local to their assigned cores, caches, and memory domains, reducing latency and optimizing performance.

Algorithm The load balancing algorithm in Linux revolves around the concept of a designated core, which is the least loaded core in the system. This core periodically attempts to pull tasks from the busiest cores to distribute the load more evenly. Key steps of the algorithm:

- *Schedule tick*: load balancing is triggered periodically or when a core becomes idle.
- *Walk the memory hierarchy*: the algorithm walks up the hierarchy of scheduling domains.
- *Task stealing*: the designated core attempts to pull tasks from the busiest cores within the same scheduling domain.

At boot time, the Linux kernel gathers hardware information to understand the system's topology, such as which cores share caches and the layout of the NUMA interconnect. This information helps build a model of scheduling domains, which group processing units that share resources. Higher-level domains may span multiple cores or sockets, but moving threads across these domains incurs greater penalties.

Load balancing must also take into account the worst-case execution time of tasks, particularly for time-sensitive processes. In constant bandwidth scheduling, tasks are guaranteed a certain amount of CPU time. If a task exceeds its allocated time, it is throttled to prevent it from interfering with other tasks. However, greedy reclaiming allows tasks to use more CPU time if available, provided it doesn't violate scheduling guarantees. By considering both CPU usage and thread priority, and ensuring tasks remain close to their memory and cache domains, Linux's load balancing mechanism maximizes performance while maintaining fairness across CPUs.

CHAPTER 3

Concurrency

CHAPTER 4

Memory management

CHAPTER 5

Drivers

CHAPTER 6

Boot

CHAPTER 7

Virtualization