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.

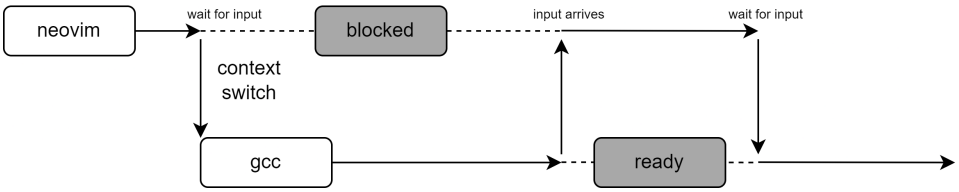

Figure 1.1: CPU multiplexing

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.

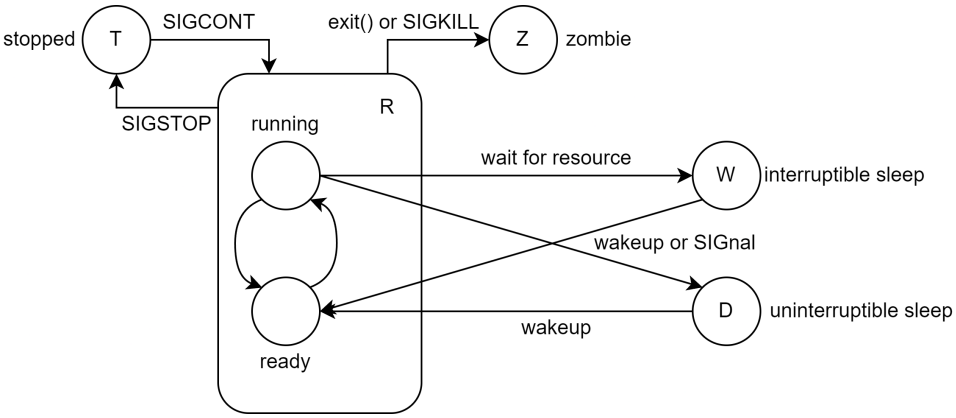

Figure 1.2: Process states

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.

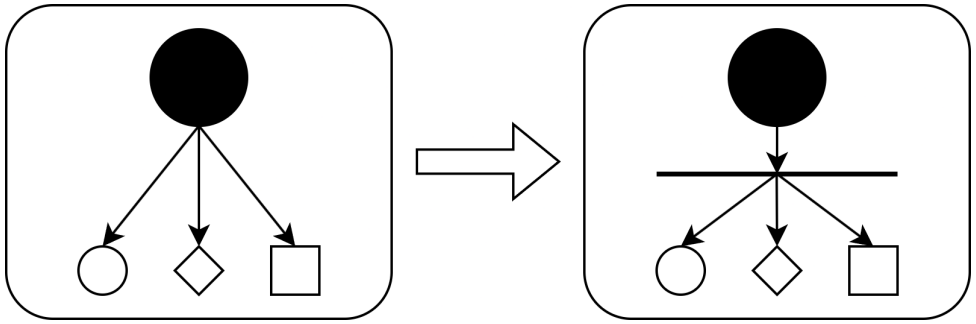

Figure 1.3: Facade pattern

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.

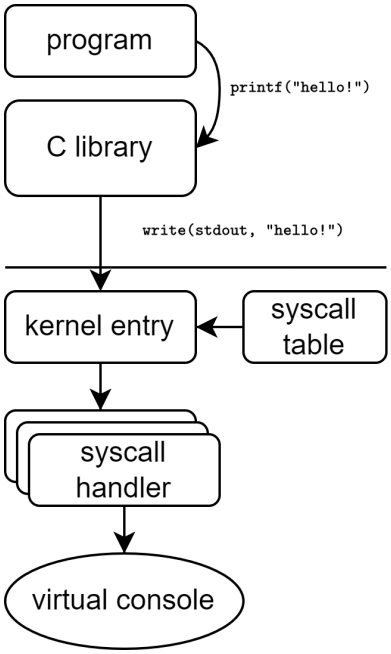

Figure 1.4: System call

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.

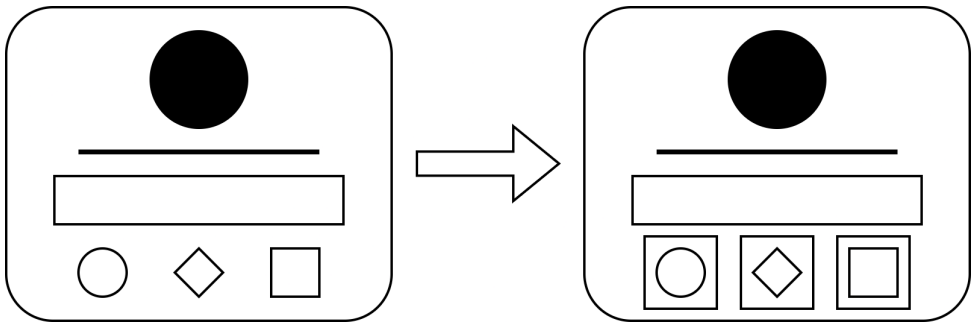

Figure 1.5: Bridge pattern

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.

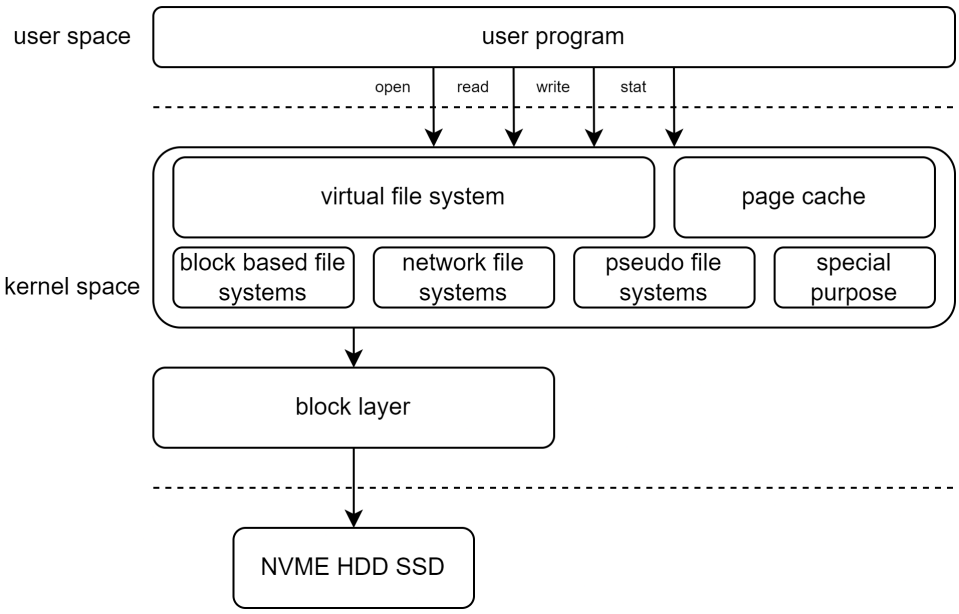

Figure 1.6: File system

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.

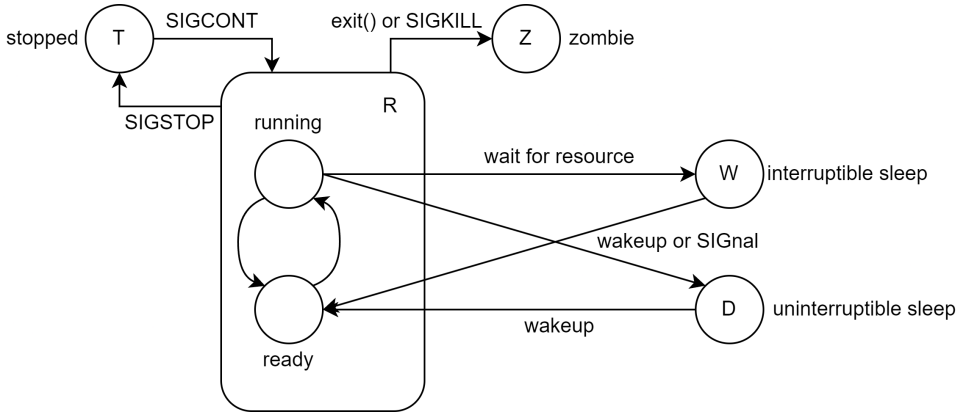

Figure 2.1: Tasks states

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.

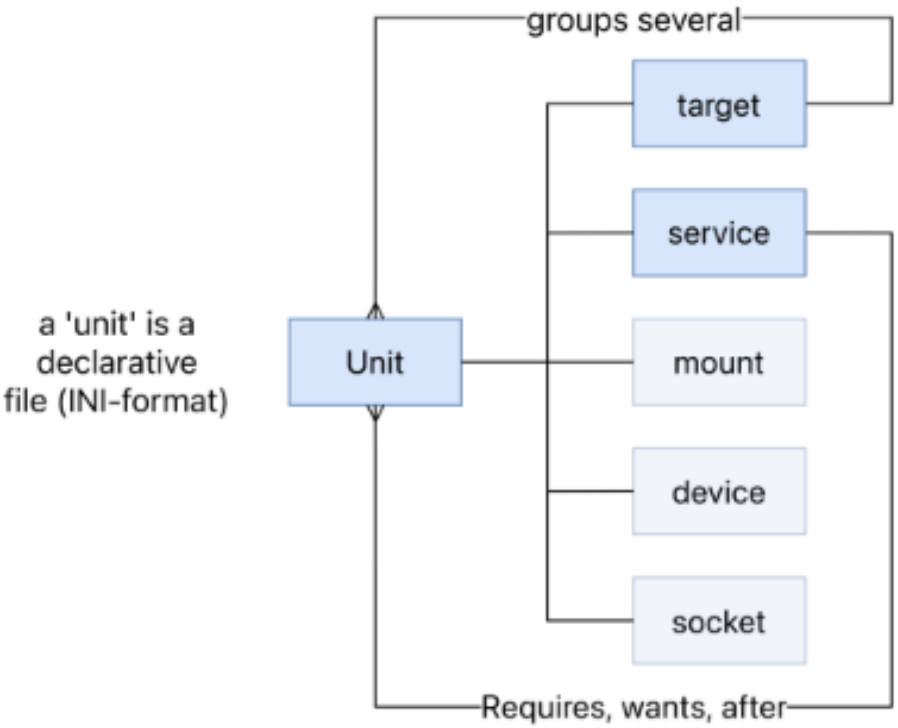

Figure 2.2: System D units

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.

2.3.1 Scheduling classes

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.

CHAPTER 3

Concurrency

CHAPTER 4

Memory management

CHAPTER 5

Drivers

CHAPTER 6

Boot

CHAPTER 7

Virtualization