Advanced Operating System
Theory

Christian Rossi

Academic Year 2024-2025

Abstract

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.

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.

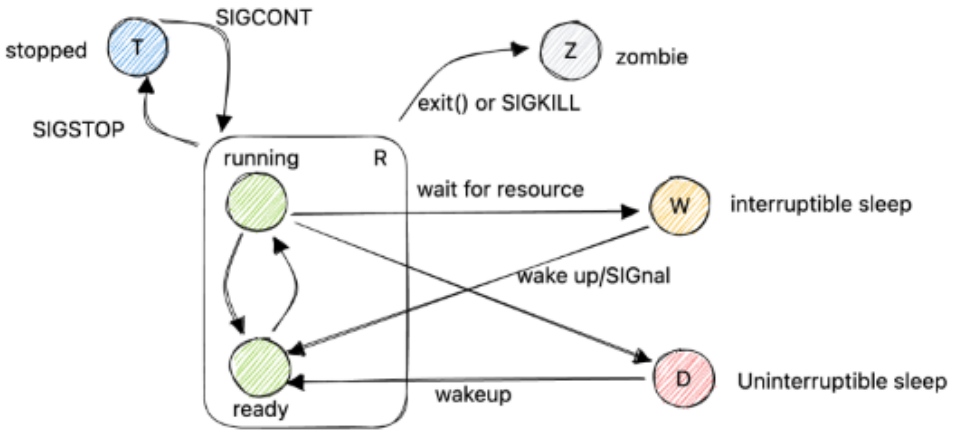

Figure 1.1: Process states

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 appropri-

ate 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.