DSC204A: Scaleable Data Systems Winter 2025

7: CPU-Virtualization

Instructor: Umesh Bellur *Scribe: Boyu Wang, Yihui Zhang*

1 Recap

In the previous lecture, we covered key concepts related to CPU architecture and memory systems. We began by understanding how an instruction executes on the CPU, specifically walking through the five-stage execution model: IF, ID, IE, MA and WB. We then discussed how to improve program performance, focusing on two main techniques: pipelining and caching. Pipelining helps improve throughput by overlapping the execution of instructions, though it can be affected by hazards such as pipeline flushes caused by control flow changes. Caching improves average memory access time by taking advantage of locality of reference—both spatial and temporal. The way data is accessed and stored heavily influences cache efficiency and, as a result, overall program performance. For example, using columnar databases instead of row-oriented storage can impact how well caching works. We also reviewed the memory hierarchy, noting the trade-offs between access time and storage capacity. Closest to the CPU are on-chip caches, followed by RAM, and then disk storage. Disk access, especially on spinning disks, is significantly slower and only used when memory is insufficient. Understanding this hierarchy is crucial because accessing slower memory levels can greatly impact performance.

2 Operating System Basics

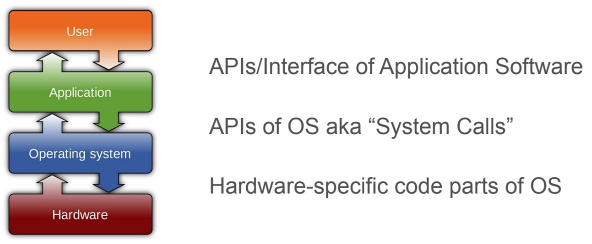
2.1 Popular Operating Systems

The three most commonly used operating systems today are Windows, macOS, and Linux. Windows and macOS are closed-source systems, developed and maintained by Microsoft and Apple, respectively. In contrast, Linux is open-source and widely used in both academic and industrial environments. It has many distributions (such as Ubuntu, Fedora, etc.), and there are even proprietary UNIX-based derivatives like HPUX. One of the earliest and most influential open-source operating systems was Berkeley Unix, which inspired many other variants.

Using Linux is recommended for gaining hands-on experience with operating system internals. Installing a dual-boot setup or running Linux in a virtual machine is a practical way to get started. More advanced users can compile the Linux kernel themselves and modify system-level code to deepen their understanding.

2.2 What is an Operating System

An operating system is a collection of software programs that serve as a bridge between hardware and user applications. Most operating systems are written in low-level languages like C to allow precise control over hardware. The OS abstracts away the complexities of hardware devices—including CPUs, memory, storage, network cards, and input/output devices—providing users and applications with a simpler, unified interface.


User
Application
Operating system
Hardware
APIs/Interface of Application Software
APIs of OS aka "System Calls"
Hardware-specific code parts of OS

Direct interaction with hardware is non-trivial, as it requires managing device drivers and vendor-specific configurations. The OS takes on the responsibility of interfacing with these components, so users do not have to. In multi-user environments, such as servers and cloud infrastructure, the OS ensures fair access to resources and enforces isolation between users to ensure stability and security.

2.3 Modularity and Abstraction in OS Design

- **Modularity** - Modern operating systems are designed with modularity in mind. Different subsystems handle different responsibilities, such as CPU scheduling or memory management. Developers can work on one part of the OS without needing to understand the entire codebase, which often spans millions of lines.

- **Abstraction** - Abstraction is another foundational principle. The OS presents high-level abstractions—like files, processes, and sockets—that hide lower-level details. These abstractions are accessed through well-defined APIs. For example, the processor exposes an instruction set as its API, while web applications use HTTP APIs (GET/POST) to communicate with servers. Similarly, the OS exposes a set of system calls that allow user programs to request services such as file access, memory allocation, and process management, while enforcing protection boundaries.

2.4 Goals of an OS

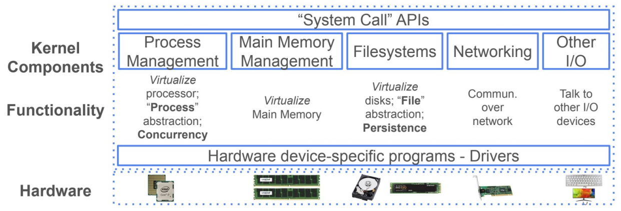
The design of an operating system aims to achieve three main goals: usability, efficiency, and protection. Usability ensures that developers and users can interact with the system easily. Efficiency ensures that programs run as close to hardware speed as possible, without significant overhead from the OS. Protection ensures that resources are used safely and that programs cannot interfere with each other or with the OS itself.

To meet these goals, the OS implements mechanisms for process scheduling, memory management, and I/O control. However, these mechanisms must be guided by policies that define how and when actions are taken. For instance, when multiple processes are ready to run, the scheduling policy determines which one gets CPU time next—based on rules such as "first come," "shortest job," or "highest priority."

2.5 Key Components of OS

The kernel is the central component of the OS responsible for managing hardware and system-level abstractions. It handles four primary functions:

- **Process Management**: Abstracts and schedules CPU usage by creating and managing processes.


"System Call" APIs
Kernel Components
Process Management
Main Memory Management
Filesystems
Networking
Other I/O
Functionality
Virtualize processor; "**Process**" abstraction; **Concurrency**
Virtualize Main Memory
Virtualize disks; "**File**" abstraction; **Persistence**
Commun. over network
Talk to other I/O devices
Hardware device-specific programs - Drivers
Hardware

- **Memory Management**: Provides virtual memory and manages physical memory allocation.
- **File System**: Abstracts storage into files and directories, offering structure and persistence.
- **Networking and I/O**: Manages communication with devices and external networks.

Other components, such as graphical interfaces or shell programs (like Bash or Windows Command Prompt), are peripheral and operate outside the kernel. These are user-facing tools built on top of the kernel's core functionality.

3 The Abstraction of a Process

Writing a program is just the starting point. A Python script saved as a `.py` file, for example, is a static entity—just a file on disk. It does not run by itself. It is **persistent** because it resides on the file system, but it is **inactive** until a user takes action to execute it. Running a program can be done in various ways, such as:

- Executing via terminal: `python3 script.py`
- Running it inside an interpreter session

Upon execution, the operating system creates a process, which is an active instance of the program. This process includes memory, access to CPU time, and other system resources needed for execution. The OS handles this transition from static code to a running process.

Every user-initiated application—whether opening a browser, launching a terminal, or starting a background task—corresponds to a process. Modern systems are designed to support many concurrent processes. Running the command `top` on Unix-like systems (e.g., macOS or Linux) will typically show hundreds of active processes.

The OS is responsible for:

- **Tracking** all running processes
- **Managing life cycles** of processes (creation, execution, suspension, termination)
- **Providing interfaces** for user or programmatic process control

Users can control processes in various ways:

- **GUI actions**: Clicking the "X" on a window to terminate the associated process

- **Terminal signals**:
  - `Ctrl+C`: Sends an interrupt signal to stop the current process
  - `Ctrl+D`: Ends input to the terminal, causing it to exit
- **System APIs**: Using system calls to create, terminate, or manage processes

Each of these actions sends a signal to the OS, which interprets and handles it accordingly. The OS maintains an internal inventory of processes and exposes APIs to manage them, ensuring proper scheduling, protection, and cleanup.

Understanding the concept of a process is essential in contexts where multiple tasks run in parallel, such as:

- Data preprocessing and transformation pipelines
- Parallel model training
- Resource sharing in distributed systems

Thus, even in data science, knowledge of process-level operations helps in diagnosing performance issues, optimizing workloads, and building scalable systems.

One of the reasons why we care about process management in Data Science is because there are a lot of occupations that lead to a process. For example, a query in SQL is a program that becomes a process. Therefore, the process is commonly occurring. However, the cost of starting a process is expensive. That is why combining multiple queries into one is a smart idea.

Another thing we need to pay attention to is that data system, such as Dask and Spark, typically abstracts away process management because user specifies the queries or processes in system's API.

3.1 Physical Manifestation of a Process

There are two ways of writing a program after setting up the connection:

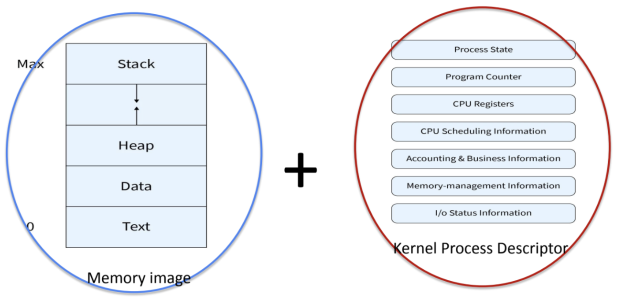
1. Store the connection, do the query, throw the connection away, and shut down the connection.
2. Keep the connection open, do all the work which can be multiple queries, and close the connection afterwards.

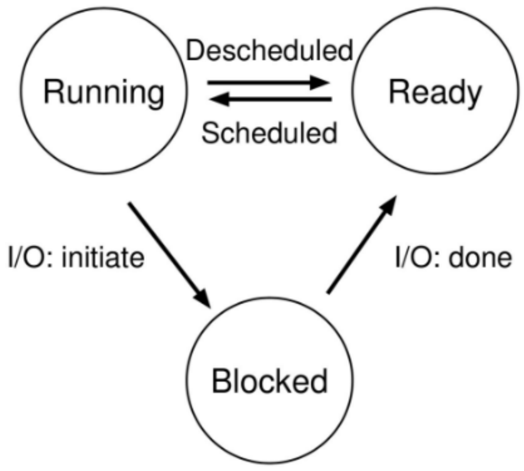
If the connection did not get closed, the process will hanging around which does nothing and takes up resources at the server site. In addition, each process is limited in terms of the number of resources it can actually access.

In general there is some state associated with the physical manifestation of a Process, and the state is something that is a memory event which the example of below in blue circle.

Start from simple, the memory event contains the code. When the process is rotted, the memory is allocated to run. Next any static data have been allocated in the program. In addition, there is also space for dynamic data allocation which is the heap. The text of the program is static which is where the memory points to.

Outside of memory event, we have kernel process (showed on the graph below in red cricle) which contains a lot of process stages such as CPU Register, I/O Status Information, and Memory-management Information.


Max
Stack
Heap
Data
0
Text
Memory image
+
Process State
Program Counter
CPU Registers
CPU Scheduling Information
Accounting & Business Information
Memory-management Information
I/o Status Information
Kernel Process Descriptor



3.2 State of Process

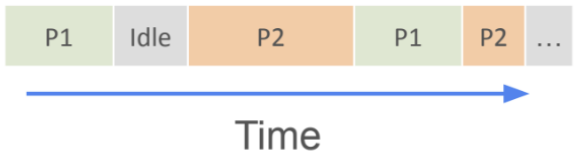
A process can be in one of the 3 states in (also shows above):

1. Running: When the CPU is free, it runs the process which make process present in this state.
2. Ready: When the CPU is no free, the process waits in the ready state.
3. Blocked: If a process is no longer running, it stays in this state.

For example, if there is a browser and email open which means they are in ready state. You are currently typing some email and hitting send. Now the email is running on the CPU. However, if you click browser window after sent, the browser process has now come to the foreground and run which moves the email to the block state.

3.3 Steps of Process start from a Program

1. Create a process (get Process ID; add to Process List).
2. Assign part of DRAM to process, aka its Address Space.
3. Load code and static data (if applicable) to that space.



4. Set up the inputs needed to run program's main().
5. Update process' State to Ready.
6. When process is scheduled (Running), OS temporarily hands off control to process to run the show!
7. Eventually, process finishes or run Destroy.

Some interesting points in the Gantt chart above which shows what process runs is that:

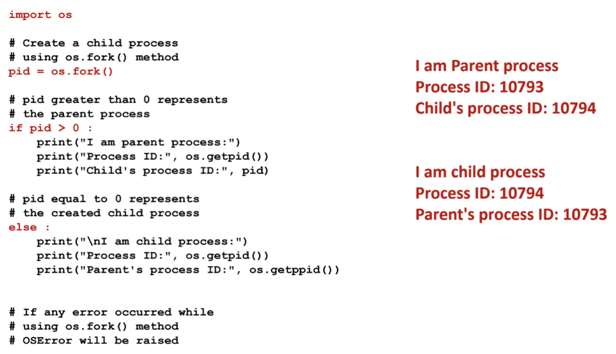
1. The process runs in order which starts from left to right
2. Some item such as P2 appears more than once because at the first time the P2 might not be completed at that point.

3.4 Process API

There are 4 major steps:

1. CREATE – fork(): Creates an identical copy of the process as a child
2. WAIT – wait(): Waits for a process to complete execution
3. EXECUTE – execvp*(): Starts a program programmatically!
4. STOP – kill(): Sends a signal to a process

Here is an example of code and expect output:



```
import os

# Create a child process
# using os.fork() method
pid = os.fork()

# pid greater than 0 represents
# the parent process
if pid > 0 :
    print("I am parent process:")
    print("Process ID:", os.getpid())
    print("Child's process ID:", pid)

# pid equal to 0 represents
# the created child process
else :
    print("\nI am child process:")
    print("Process ID:", os.getpid())
    print("Parent's process ID:", os.getppid())


# If any error occurred while
# using os.fork() method
# OSError will be raised
```

```
I am Parent process
Process ID: 10793
Child's process ID: 10794

I am child process
Process ID: 10794
Parent's process ID: 10793
```