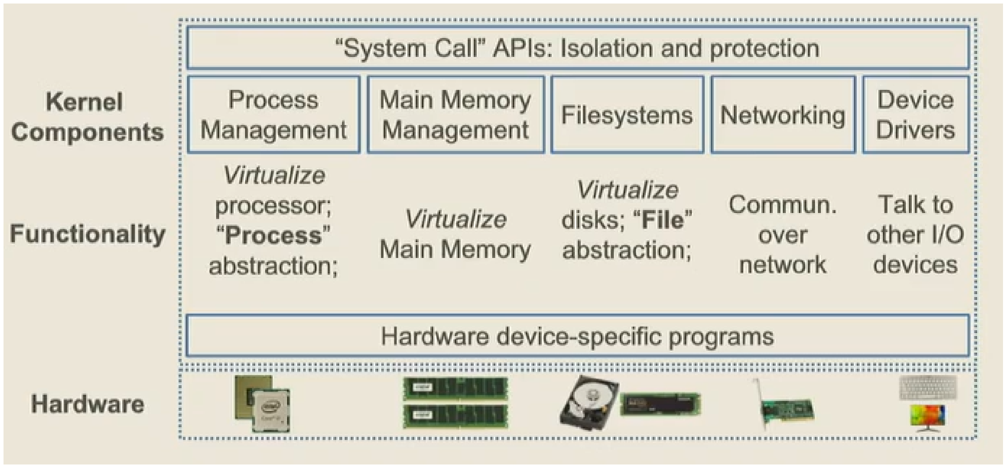
DSC-204A: Scalable Data Systems, Winter 2024

Basics: OS-1

Lecturer: Hao Zhang *Scribe: Arya Rahnama, Atulya Kumar*

1 Introduction to Operating Systems

An operating system (OS) acts as an intermediary between users and computer hardware. It provides an environment for the execution of programs and manages hardware resources. The OS is responsible for process management, memory management, file system management, and security.


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

1.1 OS High Level Design

1.1.1 OS v1: A Primitive OS

OS: interfaces above hw drivers
Hardware

In the initial iteration of "A Primitive OS v1," the operating system takes on a rudimentary form, serving primarily as a library of standard services without incorporating protective measures. This basic OS version operates under simplifying assumptions, envisioning a system where only one program runs at a time, and an absence of perceived risks from malicious users or programs. However, despite its simplicity, OS v1 faces challenges related to poor hardware utilization and inefficient use of human user time. Notably, hardware resources, such as the CPU, may remain underutilized as they idle during periods of waiting for disk operations. Simultaneously, the human user experiences delays, compelled to wait for each program to finish its execution before proceeding. These limitations underscore the need for advancements in subsequent OS versions to address issues of resource optimization and user efficiency.

1.1.2 OS v2: Multitasking OS

OS: support > 1 apps
Hardware

In the evolution from OS v1 to OS v2, a pivotal enhancement is introduced with the incorporation of multi-tasking capabilities. OS v2 extends its support to accommodate multiple applications (APPS) simultaneously. When one process encounters a blocking state, such as waiting for disk access, network communication, or user input, the operating system switches execution to another active process, thereby optimizing overall system efficiency. However, the introduction of multi-tasking brings forth new challenges related to ill-behaved processes. Processes that enter infinite loops and refuse to relinquish the CPU or attempt to tamper with the memory of other processes pose significant risks. To counteract these issues, OS v2 implements protective mechanisms. Preemption is introduced to forcefully reclaim the CPU from a looping process, preventing it from monopolizing resources indefinitely. Memory protection measures are also implemented to safeguard the memory space of one process from unauthorized access or modification by other processes, ensuring the stability and integrity of the overall operating system.

1.1.3 OS v3: Real OS

This operating system takes on a more comprehensive role, tasked with the management and allocation of hardware resources to various applications. The overarching goal is to ensure that, with an increasing number of users or applications (N), the system's performance does not degrade proportionally to N. To achieve this, OSv3 focuses on the efficient distribution of resources to users based on their actual needs. However, potential issues arise in this pursuit. One significant concern involves the possibility of one application interfering with the operations of another, necessitating the implementation of isolation measures. Additionally, users may exhibit resource-intensive behavior, such as excessive CPU usage, prompting the need for effective scheduling mechanisms to allocate resources judiciously. Memory management becomes crucial when the cumulative memory usage of all applications or users surpasses the available RAM. Furthermore, the shared nature of disks among applications and users requires proper organization through file systems to ensure coherent and secure data storage and retrieval. OSv3 represents a sophisticated evolution in operating systems, addressing these challenges to provide a robust and efficient environment for diverse applications and users.

2 Process

A process is a program in execution, an essential concept in OS. Processes need resources to accomplish their tasks. The OS must allocate, schedule, and manage these resources effectively. System calls provide a crucial interface for inter-process communication, allowing processes to request services from the OS.

2.1 Abstraction of OS

High-level steps OS takes to get a process going:

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

2.2 CPU Scheduling

The OS schedules processes on the CPU efficiently. Scheduling involves determining which processes run when there are multiple runnable processes. Different algorithms are used for scheduling tasks on the CPU. These include *First-Come, First-Served* (FCFS), *Shortest Job First* (SJF), *Priority Scheduling*, and *Round Robin* (RR).

2.3 Mechanisms and Policies

OS defines mechanisms for managing hardware and policies for deciding how to do so. Mechanisms implement functionalities, while policies decide what will be done. In operating systems, the distinction between mechanisms and policies is pivotal. Mechanisms are the methods used to implement various functionalities, such as context switching, while policies are the decision-making algorithms that guide these mechanisms. An example is the scheduling policy, which decides which process to run at a given time, potentially based on historical data and system performance goals. A critical mechanism in OS, *context switching*, allows the CPU to switch between processes. This involves saving the state of the current process and loading the state of the next process. A fundamental design principle in operating systems is the *separation of policy and mechanism*, which offers modularity and flexibility. This separation allows for changing policies according to system requirements without altering the underlying mechanisms, thus providing adaptability and ease of maintenance.

2.4 Process API

The Process API constitutes the set of operations that can be performed on processes by an OS.

2.4.1 Creation and Destruction

- **Create:** The OS must provide a method to create new processes, such as launching a program from the command line or double-clicking an application icon.
- **Destroy:** The OS also offers a way to forcibly terminate processes. While many processes exit on their own, sometimes it is necessary to manually end them.

2.4.2 Process Control

- **Wait:** An interface to wait for a process to stop running is often provided, which is useful in coordinating process execution.

- **Miscellaneous Control:** Other controls, like suspending and resuming processes, are also typically available.
- **Status:** Interfaces to query process status, such as runtime duration or current state, are commonly part of the Process API.

2.5 Process Creation in Detail

Understanding how an OS transforms a program into a process is crucial. The OS loads the program's code and static data into memory. Modern systems may use lazy loading, loading code and data as needed. The OS also sets up the process's runtime stack and heap. For instance, the stack is used for local variables and function calls, while the heap is for dynamic memory allocation.

- **Initialization:** The OS initializes the stack with arguments and allocates memory for the program's heap.
- **I/O Setup:** The OS prepares standard input, output, and error file descriptors for the process and performs other I/O-related setup tasks.

2.6 Process Management and Control

Managing processes efficiently is crucial in an operating system. The OS not only handles the creation and termination of processes but also oversees their overall lifecycle and resource allocation. This involves complex decision-making, balancing the needs of various processes while optimizing the overall system performance.

2.6.1 Process Creation and Termination

The creation of a process involves a series of steps: assigning a Process ID, allocating memory, loading code and data, setting up necessary inputs, and updating the process's state. Termination involves deallocating resources and updating system tables. The OS ensures that these steps are executed seamlessly, maintaining system stability and efficiency.

2.6.2 Process Control Block (PCB)

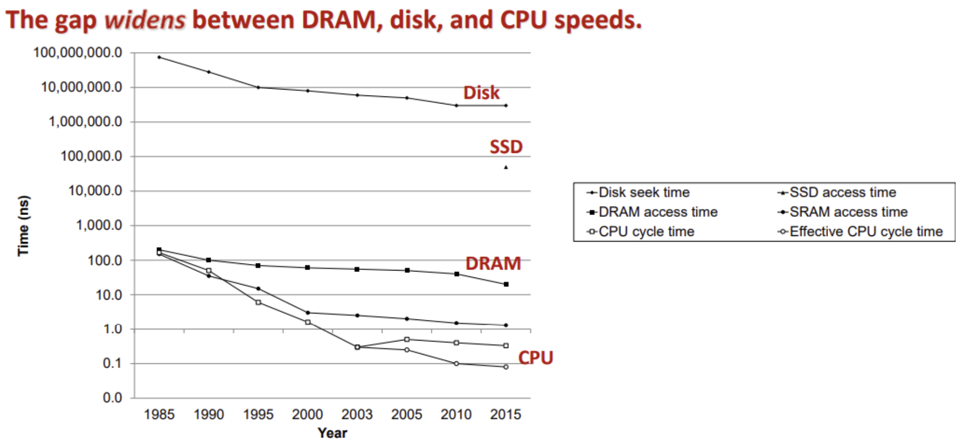
Each process has a PCB, which is a critical data structure containing vital information like the process state, program counter, CPU registers, memory limits, and scheduling information. The PCB is pivotal for context switching, allowing the OS to manage multiple processes effectively.

3 Memory Management

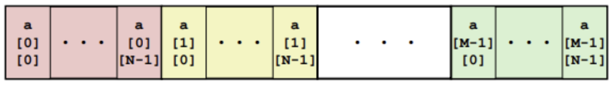
Effective memory management is key to the efficient operation of an OS.

3.1 Addressing the CPU-Memory Gap

The lecture emphasized the widening gap between CPU and memory speeds, highlighting the challenge of maintaining efficient data access and processing. The key to bridging this gap lies in the principle of locality, where programs tend to use data and instructions near those they have used recently. This concept divides into two types: temporal locality (recently referenced items are likely to be referenced again soon) and spatial locality (items with nearby addresses are referenced close together in time).



3.2 Locality



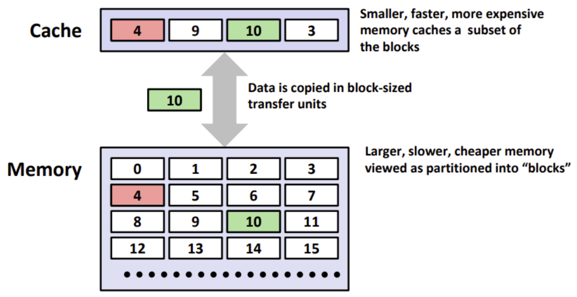
```c
// Row Major
int sum_array_rows(int a[M][N])
{
  int i, j, sum = 0;
  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      sum += a[i][j];
  return sum;
}
```

When traversing a row in a row-major order, consecutive elements are stored sequentially in memory, promoting spatial locality. As a result, when processing elements within a row, the likelihood of accessing nearby memory locations is higher, reducing cache misses and enhancing overall performance. In contrast, column-major ordering stores consecutive elements of a column contiguously, which may lead to scattered memory access patterns, potentially causing more cache misses and resulting in suboptimal data locality.

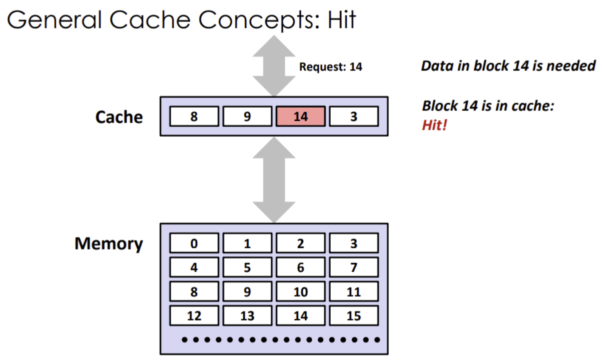
```c
//Column Major
int sum_array_cols(int a[M][N])
{
  int i, j, sum = 0;
  for (j = 0; j < N; j++)
    for (i = 0; i < M; i++)
      sum += a[i][j];
  return sum;
}
```

3.2.1 Locality into Practice: Caches

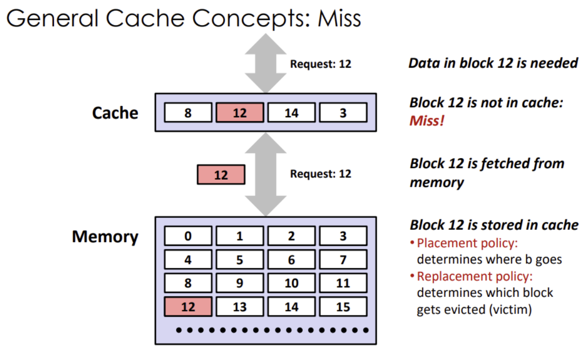
The concept of a memory hierarchy is rooted in the utilization of a cache, a smaller and faster storage device that operates as a staging area for a subset of data contained in a larger, slower device. The fundamental principle guiding this hierarchy is the organization of levels, where each level, denoted as k, involves a faster, smaller device serving as a cache for the larger, slower device at level $k + 1$. The effectiveness of memory hierarchies is attributed to the notion of locality, wherein programs exhibit a tendency to access data at level k more frequently than at level $k + 1$. This behavior allows the storage at level $k + 1$ to be slower, yet larger and more cost-effective per bit. Consequently, the memory hierarchy creates an extensive pool of storage that maintains a cost comparable to the inexpensive storage near the bottom, while delivering data to programs at the accelerated rate characteristic of the fast storage near the top. A cache is a smaller, faster storage device that serves as a staging area for data in a larger, slower device. Efficient cache utilization can significantly reduce data access times, enhancing overall system performance.


Cache
4
9
10
3
Smaller, faster, more expensive memory caches a subset of the blocks
10
Data is copied in block-sized transfer units
Memory
Larger, slower, cheaper memory viewed as partitioned into "blocks"
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15

Cache Hit Scenario: A cache hit and a cache miss are fundamental concepts in the realm of computer memory systems, particularly in the context of memory hierarchies. A cache hit occurs when the processor or program accesses data that is already present in the cache. In other words, the requested data is found in the smaller, faster cache, resulting in a quick retrieval process. Cache hits are advantageous as they contribute to enhanced system performance, minimizing the latency associated with fetching data from slower, larger storage devices.



Cache Miss Scenario: A cache miss transpires when the processor seeks data that is not currently stored in the cache. This prompts the system to fetch the required data from the larger, slower main memory or storage. Cache misses introduce a delay in accessing the data, as the processor must wait for the data to be retrieved from the lower level of the memory hierarchy. Strategies such as prefetching and caching algorithms are employed to mitigate the impact of cache misses and optimize the efficiency of memory systems. The management of cache hits and misses is a crucial aspect of designing effective memory hierarchies to ensure that frequently accessed data is readily available in the faster cache, minimizing performance bottlenecks.


General Cache Concepts: Miss
Request: 12
Data in block 12 is needed
Cache
8 12 14 3
Block 12 is not in cache: Miss!
12
Request: 12
Block 12 is fetched from memory
Memory
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
Block 12 is stored in cache
• Placement policy: determines where b goes
• Replacement policy: determines which block gets evicted (victim)

Cache Placement is Important to reduce Seek times: The strategic placement of data within a cache is a critical aspect of optimizing memory systems, with a direct impact on reducing seek time and enhancing overall system efficiency. Placement strategies, including spatial locality and temporal locality considerations, play a pivotal role in optimizing cache performance and mitigating delays associated with seek time, thereby contributing to the seamless functioning of computer memory hierarchies.

3.2.2 Designing Cache for ChatGPT

ChatGPT Model Parameter size: 350GB

The utility of cache is nuanced and depends on various factors in ChatGPT. The process of loading and unloading the extensive parameters of ChatGPT, totaling 350GB, into and from the cache proves to be less efficient due to the sheer size of the model. While parallelism using cache could theoretically enhance performance, managing such a large cache becomes impractical for a language model of this scale. Opting for a reduced parameter size to fit into the cache might improve speed but at the expense of model accuracy. Additionally, the concept of speculative decoding, which involves outputting multiple words in a single cache movement to minimize storage transfers, is still an area of active research. Despite its challenges, the role of cache in optimizing ChatGPT's performance remains a dynamic area where trade-offs between speed, accuracy, and storage considerations continue to be explored and refined.

3.3 Virtual Memory

Virtual memory allows a program to execute without having all its code and data in physical memory, facilitating larger application memory space than physically available.

3.4 Paging and Segmentation

These techniques manage memory allocation. Paging divides memory into fixed-size blocks, while segmentation divides it into variable-sized segments based on logical divisions in the program.

4 Concurrency and Synchronization

Concurrency in an OS allows multiple processes to run in overlapping time periods, requiring mechanisms for synchronization. A race condition occurs when multiple processes access shared data concurrently, leading to unexpected results. Mechanisms like locks and semaphores are used to control access to shared resources, ensuring data integrity in concurrent environments.

4.1 Virtualization and Resource Management

The operating system virtualizes hardware resources, allowing multiple processes to share these resources efficiently. This includes virtualizing the processor, which enables process isolation and efficient multitasking. The OS uses mechanisms like context switching and virtual memory to manage these resources, ensuring that processes are executed fairly and efficiently.

4.2 Multiprocessing and Modern Computing Realities

Modern computing involves multicore processors, where multiple CPUs on a single chip share main memory and some caches. This architecture, coupled with OS-level virtualization, allows for effective multitasking and inter-process communication. The use of GPUs alongside CPUs poses unique challenges and opportunities. The lecture raised questions about the integration of GPUs into the computing process, especially concerning memory management and process execution efficiency.

5 File Systems and I/O Management

File systems organize and store data efficiently, while I/O management handles the input and output of data from various devices. The file system structure includes directories, inodes, and files. It manages space allocation, naming, directories, protection, and retrieval.

6 Security and Protection

An OS must provide security features to protect data and resources from unauthorized access and ensure system integrity. Authentication verifies user identity, while authorization controls user access to resources. Encryption protects data from unauthorized access. Firewalls monitor and control incoming and outgoing network traffic based on predetermined security rules.