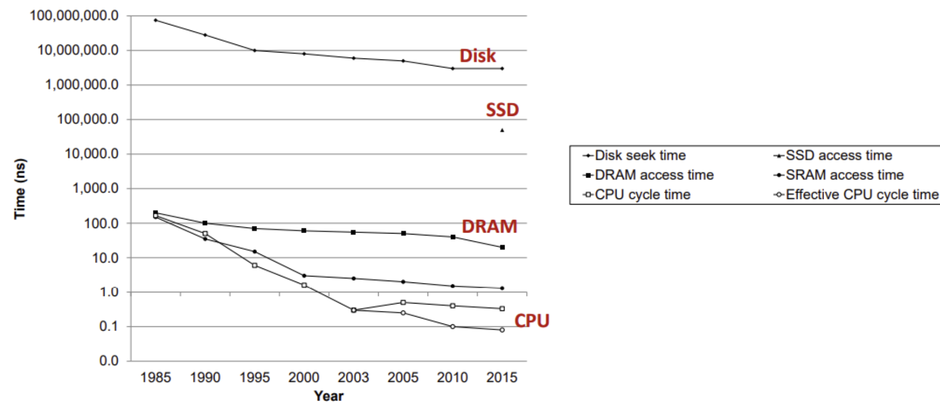
DSC-204A: Scalable Data Systems, Winter 2024

Basics: OS-1

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

1 Addressing the CPU Memory Gap

In the memory hierarchy, the CPU accesses data stored in memory at a relatively slower speed compared to accessing data from the faster memory cache. As we move down the hierarchy, the gap in speed increases, with disk storage being notably slower than both RAM and CPU cache. Memory exhibits a hierarchical structure, with each level offering progressively faster access speeds, GPU being the fastest.



1.1 Locality

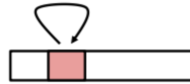
The widening gap between CPU and memory speeds, highlighting the challenge of maintaining efficient data access and processing. The key to bridging this gap lies in the principle of locality.

1.1.1 Principle of Locality

Many programs tend to use data and instructions with addresses near or equal to those they have use recently.

1.1.2 Temporal Locality

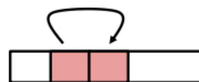
Recently referenced items are likely to be referenced again in the near future.



For example this memory block could just be code which the computer is repeatedly requesting for.

1.1.3 Spatial Locality

Items with nearby addresses tend to be referenced close together in time.



1.1.4 Locality Examples

This is a simple program to illustrate locality. A list is initialized from numbers 1 to 7 and we take sum of that list by looping through the list.

```
1 sum_list = [1,2,4,5,6,7]
2 sum = 0
3 for (x in num_list)
4     sum += x
5 return sum;
```

Data References

- References array elements in succession (stride-1 reference pattern). This is a spatial locality example as data is referenced sequentially.
- Reference variable sum each iteration. This is temporal, as the variable is being repeatedly accessed.

Instruction References

- Reference instructions in sequence. This is spatial as there is a sequence of code that is being accessed from first to last.
- Cycle through loop repeatedly. This is temporal as the loop is a single piece of program that is being accessed repeatedly.

Quick Question: Does this function have good locality with respect to array a?

```
1 int sum_array_rows(int a[M][N])
2 {
3   int i, j, sum = 0;
4   for (i = 0; i < M; i++)
5     for (j = 0; j < N; j++)
```

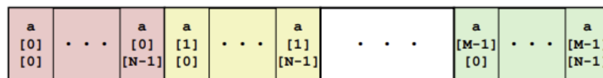
```
6         sum += a[i][j];
7     return sum;
8 }
```

It Depends on how the data is stored.

Row Major - Prioritizes on storing the rows together.

Column Major - Prioritizes on storing the columns together.

Assuming data is stored in row major. Yes, it does have good locality as it moves from left to right in the below memory block without any jumps. When there are jumps there is seek or lookup times. It executes in 4.3 ms.




```
1 //Column Major (Slower ~ 81.8ms)
2 int sum_array_cols(int a[M][N])
3 {
4     int i, j, sum = 0;
5     for (j = 0; j < N; j++)
6         for (i = 0; i < M; i++)
7             sum += a[i][j];
8     return sum;
9 }
```

For row major the above code is slower (81.8 ms) as it has to access elements from memory blocks that are spaced further apart from each other.

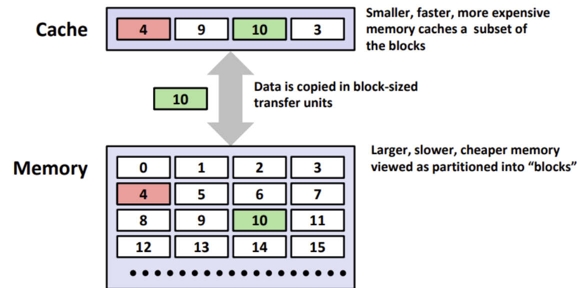
Example Exam Question: Can you permute the loops so that the function scans the 3-d array a with a stride-1 pattern (and thus has good spatial locality)?

```
1 int sum_array_3d(int a[M][N][N])
2 {
3     int i, j, k, sum=0;
4
5     for (i=0; i<N; i++)
6         for (j=0; j<N; j++)
7             for (j=0; k<M; k++)
8                 sum += a[k][i][j];
9
10     return sum;
11
12 }
```

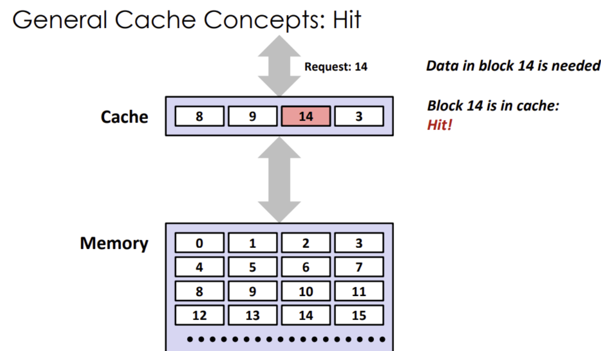
1.1.5 Locality into Practice: Caches

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

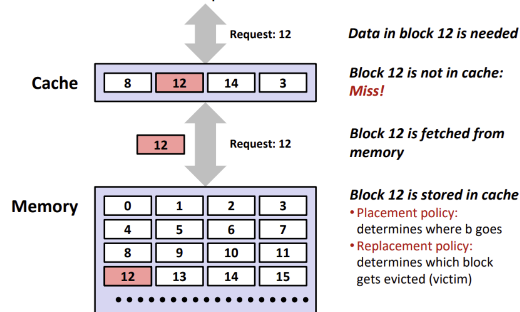
principle guiding this hierarchy is the organization of levels, where each level, denoted as k, involves a faster, smaller device serving as a cache for the larger, slower device at level $k+1$. The effectiveness of memory hierarchies is attributed to the notion of locality, wherein programs exhibit 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 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 pre-fetching 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.



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.

1.1.6 Open Question: 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.

2 Introduction to Operating Systems

Operating System Responsibilities:

1. Process, Scheduling, Concurrency
2. Memory Management
3. File 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. It also usually provides abstraction for applications, manages and hides details of hardware and accesses hardware through low/level interfaces unavailable to applications. It will prevent bad software to destroy or misuse hardware resources.

2.1 OS High Level Design

2.1.1 OS v1: A Primitive OS

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.

2.1.2 OS v2: Multitasking OS

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.

2.1.3 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

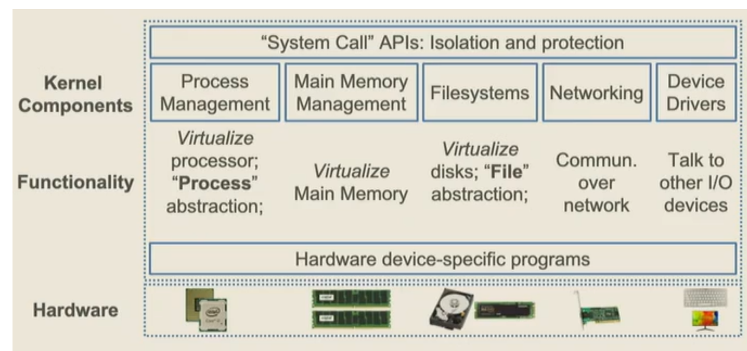
secure data storage and retrieval. OSv3 represents a sophisticated evolution in operating systems, addressing these challenges to provide a robust and efficient environment for diverse applications and users.

Question: What is a real Operating System?

Operating system is a manager that assigns hardware resources to apps. The goal is to not make the system N times slower when N users/apps are running on the system.

1. **Isolation** needs to be implemented such that apps do not interfere with other apps.
2. **Scheduling** needs to be implemented so that users using too much CPU can be managed.
3. **Memory Management** needs to be implemented so that all apps/users are limited to available machine RAM (Cache population and eviction).
4. **File Systems** is required as disks are shared across apps/users and must be arranged properly.

Modules

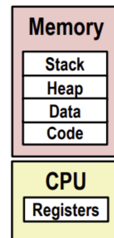


1. **System Call -** For Isolation, it abstracts the hardware and APIs for program to use. Example - File open, close or read and write. The apps can only access limited types of system calls thereby maintaining protection for hardware as well.
2. **Process Management -** Runs N processes but not N times slower. Virtualizes all the application of process and tries to manage them.
3. **Main Memory Management -** Virtualizes all the available hardware memory and allocates it to the software for use. It ensures that there is no interference between processes. It also ensures that if all existing processes exceed the memory available, it can switch between processes and support the processes.
4. **File System -** The disks store bits which are interpreted by this system as files, ensuring efficient storage and GUI access.
5. **Networking -** This manages how one computer communicates with another computer.

3 Process

A process is an instance of a running program. Example - A simple Hello world python program is a process. The python interpreter will execute the statements and then return a "Process finished with exit code 0" which is system call indicating that all processes executed successfully.

3.1 Process - The central abstraction in OS



Process provides each program with two key abstractions (for resources):

- **Compute Resource** - Each program seems to have exclusive use of the CPU. The program is not aware of other programs and acts like it has exclusive access to CPU cycles. It also has a kernel mechanism called as context switching - this enables switching from one program to another and shall be explained in future lectures.
- **Memory Resource** - Each program seems to have exclusive use of main memory. It does not need to worry about other processes. The OS will provide this type of virtual attention by kernel mechanism called Virtual Memory.

3.2 Abstraction of a Process

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

1. Create a process (get Process ID; add to Process List) - Process ID is a unique identifier for process launch. The OS will then need to allocate memory and CPU to the process.
2. Assign part of DRAM to process, aka its Address Space
3. Load code and static data (if applicable) to that space (cache).
4. Set up the inputs needed to run the program's `main()`
5. Update process' State to Ready. The OS has a table of processes where it stores the process ID and state. Ready state means it's ready to execute as there is CPU and RAM available to finish the processes. Process Management has a scheduler which finds ready processes and then schedule the process.
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`. It shall give information if the process fails and then recycle the CPU cycles and memory for other processes.

Question: But is it not risky/foolish for OS to hand off control of hardware to process (random user-written program)?!

OS has mechanisms to regain control so it's safe. The virtualization of hardware, i.e. treating hardware as a virtual entity that OS can divvy up among processes in a controlled way.