DSC-204A: Scalable Data Systems, Fall 2025

4: Basics: Computer Organization, Operating systems, Storage

Lecturer: Hao Zhang *Scribe: Ohm Rishabh Venkatachalam, Syed Mujtaba, Rishabh Thapliyal, Amirhossein Panahi*

1 Cache

1.1 Memory Locality

The below loop is memory friendly as we store data in a row-major format and in the loop we access elements sequentially.

```
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;
}
```

The below loop is not memory friendly as we store data in a column-major format and in the loop we do not access elements sequentially. This is because after arr[x][y] we access arr[x+1][y] which is present at N memory locations after arr[x][y].

```
int sum_array_rows(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;
}
```

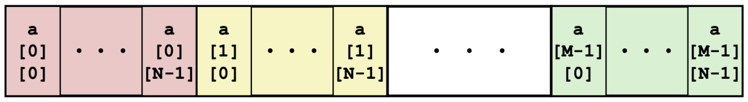


Figure 1: Row-major Format

1.2 Cache in Action

Cache: Smaller, faster memory that stores parts of the data from larger, slower, cheaper memory. Memory at level K in the hierarchy is cache to memory at level K+1. Works based on locality principle. We need to write programs and cache management algorithms that help us use cache more often.

Cache Hit: This is the case where data is present in the cache and we can return it back to our program. This is fast and we want to increase the amount of cache hits that we have. If cache has 100% hit rate we operate very fast at cache speeds.

Cache Miss: When the data we want is not present in the cache we have to fetch the data from lower memory tiers which is slow and delays our execution. When fetching data we need to decide where to place the data in our cache and if our cache is full we need to decide which data to evict. This is critical as we do not want to evict data that will likely be reused in the future. To use cache effectively we need good scheduling algorithms to ensure re-usability of data in cache. If cache has 100% miss rate we operate very slow at memory speeds.

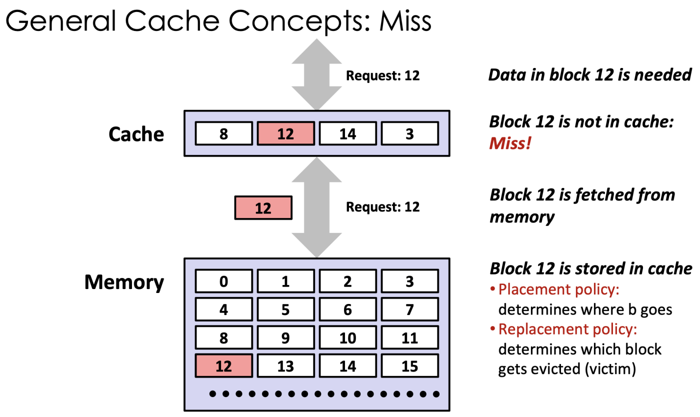

General Cache Concepts: Miss
Request: 12
Data in block 12 is needed
Cache
8
12
14
3
Block 12 is not in cache: Miss!
12
Request: 12
Block 12 is fetched from memory
Memory
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
Block 12 is stored in cache
• Placement policy: determines where b goes
• Replacement policy: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

Figure 2: Cache Miss

Cache in ChatGPT: Every time we need to output a single token we need to load all the weights (350GB). Cache sizes are very small (few MB) so cant store all weights in cache. To improve efficiency we have algorithms like Speculative Decoding that output more than one token for every read. This amortizes our read cost over generation.

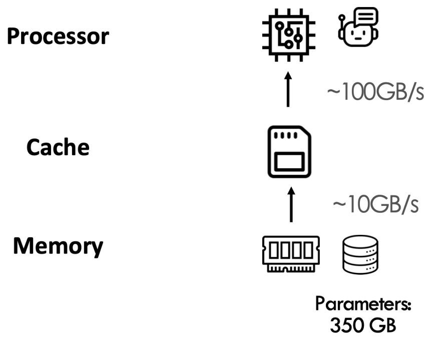


Figure 3: ChatGPT Generation

2 Operating System (OS)

What is OS:

- Layer between software and hardware
- Needed because we want to make hardware useful to programmers

- Provides abstraction for applications and hides details of hardware
- Accesses hardware through low level primitives
- Usually provides protection to prevent one application from obstructing another
- Manages resources for application

2.1 Building OS

2.1.1 A primitive OS v1

- Library of standard services (no protection) that serves as an interface above hardware drivers making it easier to program
- Simplifying assumptions (not good):
  - System runs one program at a time
  - No bad users or programs
- Problem: poor utilization
  - poor utilization of hardware (e.g., CPU idle while waiting for disk)
  - poor utilization of human user (must wait for each program to finish)

2.1.2 OS v2: Multitasking

- OS started supporting multi-tasking
- Basic efficiency strategy is employed: when one process becomes blocked (waiting for I/O like disk or network access), the OS runs another process.
- However, this introduces risks from ill-behaved processes, such as one process entering an infinite loop and never relinquishing the CPU, or maliciously scribbling over the memory of other processes, causing them to fail.
- The OS addresses these critical security and fairness issues by providing specific protection mechanisms: **Preemption**, which allows the OS to forcibly take the CPU away from a looping process, and **Memory Protection**, which ensures that one process's memory is isolated and safe from interference by another.

2.1.3 Summary of OS

The OS is fundamentally responsible for managing and assigning hardware resources to applications. Its primary goals are:

- **Goal 1**: Provide Convenience to Users.
- **Goal 2**: Efficiency (Resource Management): The system must manage compute, memory, and storage resources.
  - The aim is that when running N users or apps, the system should not become N times slower. This involves giving resources to users who actually need them.
  - The OS must also run N apps even when their total memory usage far exceeds the physical RAM cap.
- **Goal 3**: Provide Protection: Ensuring that one process cannot mess up the entire computer or interfere with other processes.

2.1.4 The Role of System Calls for Isolation

The OS provides essential protection and isolation using **System Calls**.

- A **System Call** is the layer responsible for **isolation**; it abstracts the hardware and provides APIs for programs to use.
- The **Kernel Components** (which include the System Call APIs) implement the core **functionality** of the OS:
  - **Process Management** virtualizes the processor and provides the "Process" abstraction.
  - **Main Memory Management** virtualizes main memory.
  - **Filesystems** virtualize disks and provide the File abstraction.
  - Other components manage networking (communication over the network) and device drivers (talking to other I/O devices).

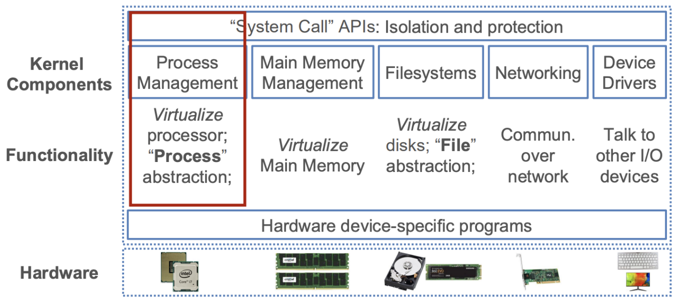

"System Call" APIs: Isolation and protection
Kernel Components
Process Management
Main Memory Management
Filesystems
Networking
Device Drivers
Functionality
Virtualize processor; "Process" abstraction;
Virtualize Main Memory
Virtualize disks; "File" abstraction;
Commun. over network
Talk to other I/O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

Figure 4: Isolation using System Calls

3 Processes: The Central Abstraction in Operating Systems

A **process** is the central abstraction provided by the operating system (OS). It differs from both a program and a processor.

Each process provides a running program with two key abstractions:

1. **Compute Resource:** Each program appears to have exclusive use of the CPU. This illusion is provided by the kernel mechanism called *context switching*.
2. **Memory Resource:** Each program appears to have exclusive access to the main memory. This is achieved using the kernel mechanism known as *virtual memory*.

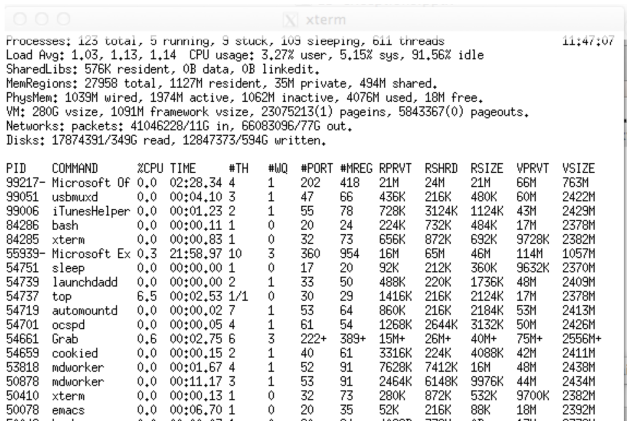
Example: The `top` command lists all running processes. Each process is identified by a unique **Process ID (PID)**.

3.1 Multiprocessing: A Strawman Solution

Suppose we want to run multiple (say, three) applications simultaneously:

- Assign each application one-third of the memory.
- Assign the CPU to work on one application until completion, then move to the next.

Problem: If we run N processes this way, execution becomes roughly N times slower. This approach is simple but neither convenient nor efficient.



```
Processes: 123 total, 5 running, 9 stuck, 109 sleeping, 611 threads                    11:47:07
Load Avg: 1.03, 1.13, 1.14  CPU usage: 3.27% user, 5.15% sys, 91.56% idle
SharedLibs: 576K resident, 0B data, 0B linkedit.
MemRegions: 27958 total, 1127M resident, 35M private, 494M shared.
PhysMem: 1039M wired, 1974M active, 1062M inactive, 4076M used, 18M free.
VM: 280G vsize, 1091M framework vsize, 23075213(1) pageins, 5843367(0) pageouts.
Networks: packets: 41046228/11G in, 66083096/77G out.
Disks: 17874391/349G read, 12847373/594G written.

PID    COMMAND      %CPU TIME     #TH   #WQ  #PORT #MREG RPRVT  RSHRD  RSIZE  VPRVT  VSIZE
99217- Microsoft Of 0.0  02:28.34 4     1    202   418   21M    24M    21M    66M    763M
99051  usbmuxd      0.0  00:04.10 3     1    47    66    436K   216K   480K   60M    2422M
99006  iTunesHelper 0.0  00:01.23 2     1    55    78    728K   3124K  1124K  43M    2429M
84286  bash         0.0  00:00.11 1     0    20    24    224K   732K   484K   17M    2378M
84285  xterm        0.0  00:00.83 1     0    32    73    656K   872K   692K   9728K  2382M
55939- Microsoft Ex 0.3  21:58.97 10    3    360   954   16M    65M    46M    114M   1057M
54751  sleep        0.0  00:00.00 1     0    17    20    92K    212K   360K   9632K  2370M
54739  launchdadd   0.0  00:00.00 2     1    33    50    488K   220K   1736K  48M    2409M
54737  top          6.5  00:02.53 1/1   0    30    29    1416K  216K   2124K  17M    2378M
54719  automountd   0.0  00:00.02 7     1    53    64    860K   216K   2184K  53M    2413M
54701  ocspd        0.0  00:00.05 4     1    61    54    1268K  2644K  3132K  50M    2426M
54661  Grab         0.6  00:02.75 6     3    222+  389+  15M+   26M+   40M+   75M+   2556M+
54659  cookied      0.0  00:00.15 2     1    40    61    3316K  224K   4088K  42M    2411M
53818  mdworker     0.0  00:01.67 4     1    52    91    7628K  7412K  16M    48M    2438M
50878  mdworker     0.0  00:11.17 3     1    53    91    2464K  6148K  9976K  44M    2434M
50410  xterm        0.0  00:00.13 1     0    32    73    280K   872K   532K   9700K  2382M
50078  emacs        0.0  00:06.70 1     0    20    35    52K    216K   88K    18M    2392M
```

Figure 5: top command output

3.2 Multiprocessing via Time Sharing

To improve efficiency, OSs use **time-sharing** of the CPU:

- The CPU time is divided into fixed-length intervals called **time slices**.
- Each process is assigned a time slice during which it can execute.
- When a time slice expires, the OS saves the current process's register state (context) in memory.
- The OS then loads the saved context of the next process and resumes its execution.

This mechanism of saving and restoring state is called a **context switch**.

With multiple CPU cores, scheduling can occur in parallel across cores, but each core typically runs one process at a time.

4 Temporal Abstraction and Process States

The OS maintains each process in one of several states and transitions between them as needed:

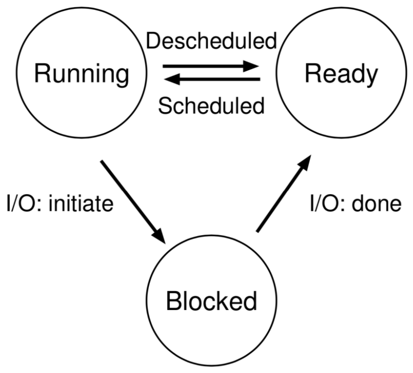


Figure 6: Process states

- **Running:** Currently executing on the CPU.
- **Ready:** Waiting to be scheduled on the CPU.

- **Blocked:** Waiting for an I/O operation to complete.

A **Gantt Chart** is a visualization showing which process runs on the CPU at each point in time:

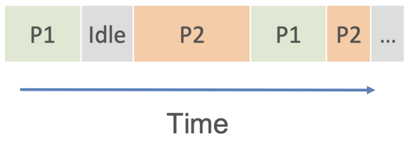


Figure 7: ChatGPT Generation

5 Scheduling Policies and Algorithms

Scheduling refers to deciding which process runs on a CPU at any given time.

- The **schedule** is a record of what process runs when.
- The **policy** is the algorithm controlling how the OS time-shares CPUs among processes.

5.1 Key Terminology

- **Arrival Time (A):** Time when a process is created.
- **Job Length (L):** Total time required for completion.
- **Start Time (S):** Time when a process first starts running.
- **Completion Time (C):** Time when a process finishes.
- **Response Time:** $S - A$
- **Turnaround Time (TAT):** $C - A$

Turnaround time is a key metric for user-perceived performance.

5.2 Scheduling Policies

1. FIFO / FCFS (First-In-First-Out or First-Come-First-Serve):

- Processes are scheduled in order of arrival.
- No preemption: once a job starts, it runs to completion.

2. SJF (Shortest Job First):

- Jobs are ranked by total job length.
- No preemption.
- Problem: Long jobs can experience **starvation**.
- Requires knowledge of job lengths in advance.

3. Round Robin (RR):

- Each process receives a fixed time quantum (e.g., 2ms).
- After its quantum expires, the process is preempted and moved to the end of the ready queue.
- Does not require knowledge of job lengths.
- Very fair but can increase average turnaround time.

4. STCF (Shortest Time to Completion First):

- A preemptive version of SJF.
- If a new job arrives with a shorter remaining time, it can preempt the running process.
- Useful when arrival times differ.

5.3 Performance and Fairness

There is an inherent tension between:

- **Workload performance** $\rightarrow$ finishing jobs quickly, maximizing throughput
- **Allocation fairness** $\rightarrow$ giving all jobs fair access to resources.

Scheduling aims to strike a balance between these two.

5.4 Metrics

Performance Metric: Average Turnaround Time

$$\text{Turnaround Time} = \text{Completion Time} - \text{Arrival Time}$$

Fairness Metric: Measures how evenly CPU time is shared,
e.g., Jain's Fairness Index (1 = perfectly fair).

5.5 Scheduling Algorithms

Common ones: **FIFO**, **SJF**, **STCF**, **Round Robin**, **Random**.

5.5.1 Variations

- **Preemptive vs Non-preemptive**
- **Multi-level feedback queue** (complex hybrid approaches)
- **Machine Learning-based schedulers** increasingly popular

5.5.2 Mapping the Problem

ChatGPT faces a similar scheduling challenge when handling multiple user requests simultaneously. Each user request behaves like a separate "job" competing for GPU resources.

Request	**Example**	**Expected Length**
S1	"Help me with assignments."	Medium
S2	"Summarize the readings."	Short
S3	"Tell a 1000-word joke."	Long

5.5.3 Scheduling Questions

When multiple prompts arrive, the system must consider:

- What is the **response time**? (Time until first token)
- What is the **turnaround time**? (Time until full completion)
- What is **fairness**? (Are all users treated equally?)
- Can we estimate **job length** from the prompt?
- Can jobs run **in parallel**?

6 Additional Notes

- In Transformer or LLM models, job length is often unknown (generation continues token by token until an end token). This presents similar challenges to CPU scheduling.