DSC-204A: Scalable Data Systems, Fall 2025

5: Memory, Cloud Computing

Lecturer: Hao Zhang *Scribe: John Driscoll, Jude Mariadas, Haoyun Wang, Hao Wang, Prabhleen Kaur*

1 Memory Management in Operating Systems

1.1 Motivation: Virtualizing CPU Resources

To support multiple processes efficiently, we need to virtualize CPU resources both temporally (time-sharing) and spatially (distributing across cores). This leads to the challenge of **concurrency**: multiple processors or cores running different or the same set of instructions simultaneously on different or shared data.

1.2 The Placement Problem

Placement refers to how we assign processes to different processors or cores to achieve optimal performance. Unlike scheduling (which determines *when* tasks run), placement determines *where* tasks run. The primary objective is Load Balancing: ensuring that different cores/processors remain roughly equally busy to reduce idle time and minimize overall completion time. This idle time is also referred to as *bubble* time. Below is an example of a Gantt chart which shows how 3 processes may be distributed across two CPUs in a way that minimizes idle time. Modern approaches use reinforcement learning algorithms to dynamically optimize

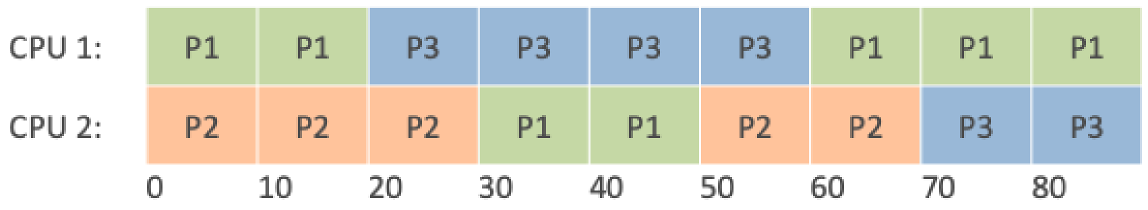


Figure 1: Gantt chart which shows how placement problem may be solved for 2 devices

placement strategies.

To illustrate the placement challenge in ML, Fig. 2 shows pipeline parallelism, which partitions a large model into stages across multiple GPUs. The grey areas are bubbles and this algorithm minimizes the bubbles.

1.3 Memory Management Challenge

Once we place processes on cores, a new problem emerges: How do we assign memory to each process? Particularly when total memory demand exceeds the physical memory capacity?

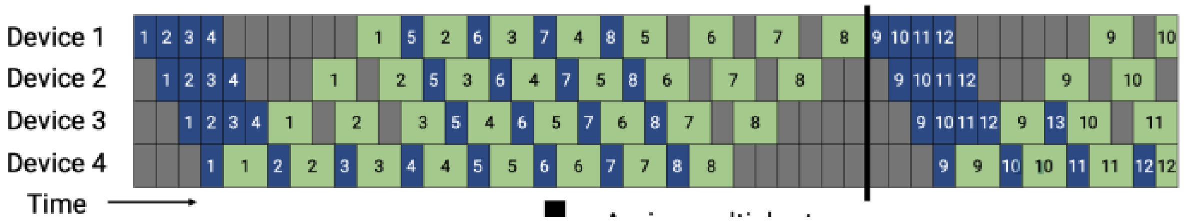


Figure 2: Pipeline parallelism across 4 GPUs.

1.4 Iterative Solutions to Memory Allocation

- V0: Sequential Hardware (Straw Man Solution) Given 3 processes with known memory requirements, allocate exactly the requested amount of memory to each process. Problem: If a process needs only part of a memory unit, the unused portion within that unit remains unused, resulting in internal fragmentation.
- V1: Smaller Chunks Reduce allocation granularity by dividing memory into smaller chunks (e.g., halves, quarters). Improvement: Reduces internal fragmentation. Maximum possible internal fragmentation: Still 1 unit (in the worst case). Still causes External Fragmentation: Wasted space *between* allocated blocks that cannot accommodate new processes requiring contiguous memory.

One potential solution is moving memory around to consolidate free space, but this is extremely expensive and inefficient.

1.5 The Dynamic Memory Problem

An even more fundamental issue: When we start a process, we don't know how much memory it will need. For example, opening Chrome results in memory usage that grows and shrinks dynamically based on:

- Number of tabs opened
- Content being loaded
- Active processes

1.6 The Problem with Reserving Memory

One approach is to predict memory needs and reserve accordingly (e.g., ask Chrome to reserve 2 GB upfront). Major problem: This suffers significant **Reservation Overhead**. Memory is reserved for a process that may not actually use it, while other processes that could use it immediately are blocked. Even if we know exact memory requirements in advance, reservation is **still inefficient**:

- Reserved memory remains unused until needed by the allocated process
- A smaller, faster job might arrive that could complete quickly but must wait because memory is already reserved

- This reduces overall system throughput

Conclusion: Always allocate memory on demand rather than reserving it in advance.

1.7 Three Main Problems in Memory Allocation

- **Internal Fragmentation:** Wasted space *within* allocated memory blocks due to fixed-size allocation.
- **External Fragmentation:** Wasted space *between* allocated blocks that cannot be efficiently reused for contiguous allocation.
- **Reservation Overhead:** Memory is blocked for processes that may not use it immediately, creating opportunity costs.

1.8 Solution: Virtual Memory

V2: Virtual memory solves these problems by introducing a mapping between virtual addresses (used by processes) and physical memory (actual hardware) process perceives a large, contiguous block of memory even if it is physically fragmented or stored partially on disk This virtualization allows the OS to provide an abstraction of virtual hardware for each process to run on, enabling multiple processes to run concurrently without needing dedicated physical memory for each.

Key Concepts

- **Granularity:** Virtual memory uses the same granularity as physical memory to ensure one-to-one mapping.
- **Pages:** Fixed-size chunks of memory that form the basic unit of virtual memory.
- **Page Frames:** Slots in physical memory (DRAM) where pages are stored.
- Having too many small pages increases overhead due to more address translations, adding computational cost.

Benefits

- Allows scheduling of processes with memory requirements larger than physical memory by mapping overflow to disk.
- **Internal Fragmentation:** Reduced via small, uniform pages.
- **External Fragmentation:** Eliminated through continuous mapping in virtual memory.

2 ML Analogy: Virtual Memory in ChatGPT

Consider three prompts being processed by ChatGPT:

1. Summarize these readings.

2. Tell me a joke in 1,000 words.
3. Help me with my assignment.

If we reserved 100,000 tokens for each prompt (like memory reservation), resources would be underutilized. Instead, tokens are allocated dynamically as each prompt executes; similar to on-demand memory allocation in an OS.

Analogy Mapping

OS Concept	**ML / ChatGPT Analogy**
Process	Prompt
CPU Memory	GPU Memory
Memory Reservation	Token Preallocation
Virtual Memory Allocation	Dynamic GPU Computation

Computation in modern ML models (like ChatGPT) shifts from CPU-based to GPU-based tensor computation, which similarly benefits from efficient, on-demand memory allocation.

3 File Systems

3.1 Purpose

File systems **virtualize the disk** and manage persistent data storage. They provide:

- **Abstraction:** Hides hardware-level complexity.
- **Persistence:** Ensures data remains after shutdown or restart.

3.2 What is a File?

A **file** is a persistent, logically coherent **sequence of bytes** representing a digital object (e.g., `.gif`, `.pdf`, `.docx`, `.mp4`).

Files also contain **metadata**, such as:

- File size, compression type, creation or modification time, etc.
- For images (e.g., JPEG): compression method, dimensions, color depth.

3.3 Directories

Directories (folders) organize files and may contain subdirectories. In many OS implementations, a **directory is also a file**, storing a list of entries and metadata pointers.

3.4 File System Operations

The file system component of the OS manages file creation, deletion, modification, and access.

- **Logical Level:** Exposes file and data abstractions to applications (APIs, system calls).
- **Physical Level:** Interfaces with disk drivers and firmware to manage how and where bytes are stored.

3.5 File System Types

Common types include **ext2**, **ext3**, and **NTFS**. Each defines its own data structures, journaling, and fault recovery mechanisms.

4 Databases vs. File Systems

Feature	**File System**	**Database**
Structure	Unstructured (directory hierarchy)	Structured (tables, schemas)
Access	File-level APIs	Query language (SQL)
Serialization	Basic (file-by-file)	Optimized for storage efficiency
Use Case	General-purpose storage	Structured data management, analytics

Databases build on top of file systems but provide **structure, queryability**, and **efficient data access**.

5 Why Cloud Computing

5.1 Need-Based Argument

The need for cloud computing arises when:

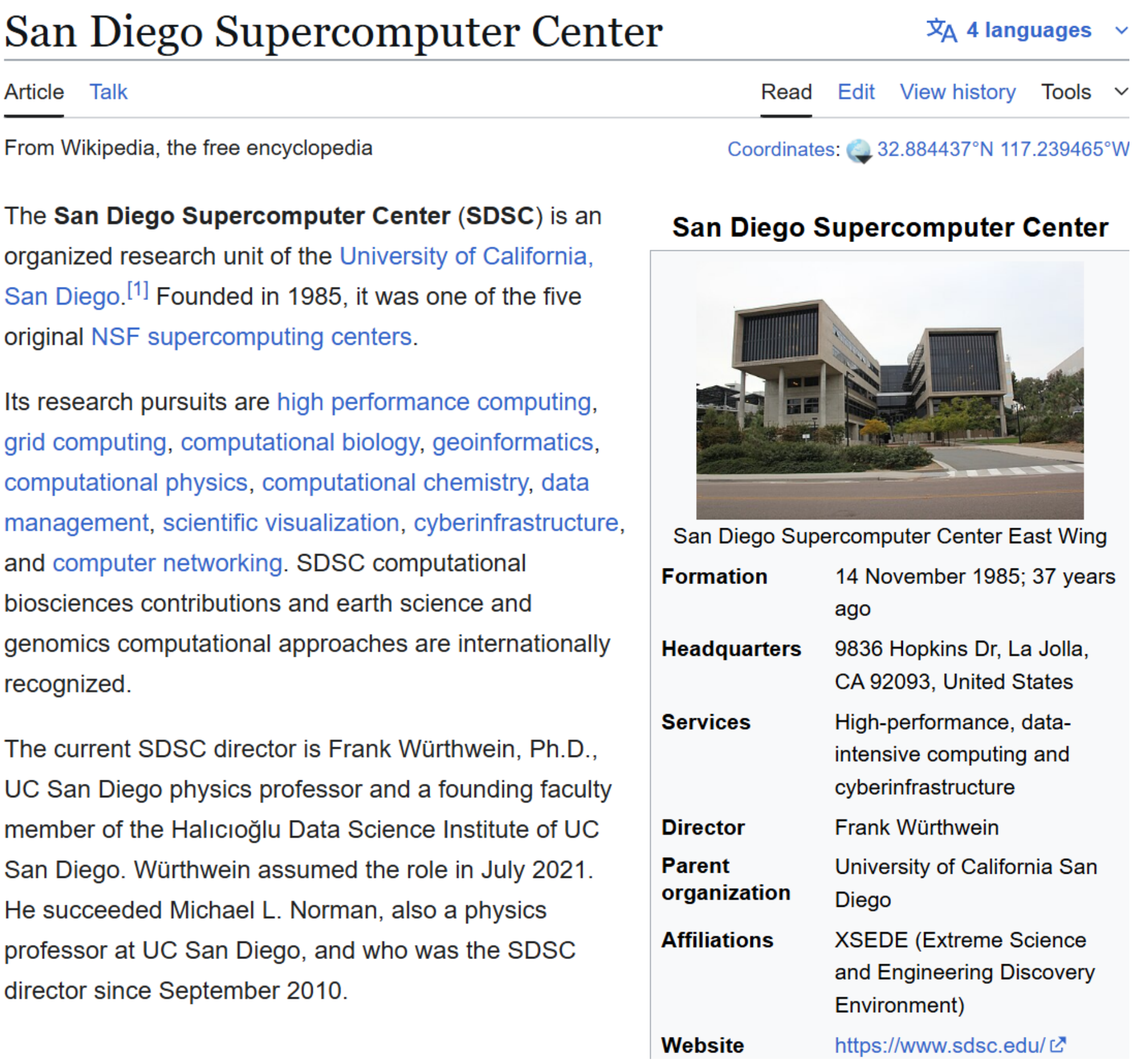
- Workloads outgrow single-machine capacity.
- Distributed computing becomes essential for scale and reliability.
- On-premise or HPC systems such as **UCSD SDSC's Expanse / Nova clusters**, while powerful, require manual scaling and maintenance.

Cloud offers elastic resources without the need for physical expansion.

5.2 Utility-Based Argument

Cloud computing follows the idea of public utilities such as **electricity**:

- Use more when needed; pay only for what you consume.
- No need to buy or install new hardware.



San Diego Supercomputer Center

From Wikipedia, the free encyclopedia

Coordinates: 32.884437°N 117.239465°W

The **San Diego Supercomputer Center** (**SDSC**) is an organized research unit of the University of California, San Diego.[1] Founded in 1985, it was one of the five original NSF supercomputing centers.

Its research pursuits are high performance computing, grid computing, computational biology, geoinformatics, computational physics, computational chemistry, data management, scientific visualization, cyberinfrastructure, and computer networking. SDSC computational biosciences contributions and earth science and genomics computational approaches are internationally recognized.

The current SDSC director is Frank Würthwein, Ph.D., UC San Diego physics professor and a founding faculty member of the Halıcıoğlu Data Science Institute of UC San Diego. Würthwein assumed the role in July 2021. He succeeded Michael L. Norman, also a physics professor at UC San Diego, and who was the SDSC director since September 2010.

San Diego Supercomputer Center

San Diego Supercomputer Center East Wing

Formation	14 November 1985; 37 years ago
Headquarters	9836 Hopkins Dr, La Jolla, CA 92093, United States
Services	High-performance, data-intensive computing and cyberinfrastructure
Director	Frank Würthwein
Parent organization	University of California San Diego
Affiliations	XSEDE (Extreme Science and Engineering Discovery Environment)
Website	https://www.sdsc.edu/

Figure 3: UC San Diego Supercomputer Center (SDSC) as an example of large-scale on-premise infrastructure.

- Scaling up or down happens automatically.

This **utility computing** model provides on-demand access to compute and storage. (With exceptions like GPUs, where hardware supply can still limit instant scaling.)

5.3 Traditional vs Cloud Model

Aspect	**Traditional (On-Premise)**	**Cloud (Utility)**
Planning	Must estimate demand ahead of time	Scale dynamically
Cost	Pay for capacity (even if unused)	Pay per use
Responsibility	Security, power, maintenance	Managed by cloud provider

5.4 Evolution of Cloud Infrastructure

Cloud infrastructure has evolved through several stages:

- **Past Cloud 1.0:** Basic virtual machines and centralized data centers.
- **Current Cloud 2.0:** Containers, DevOps, and automation-driven deployment.
- **Future Cloud 3.0:** AI-powered, edge, and decentralized architectures.

This progression can be illustrated using a **car analogy**

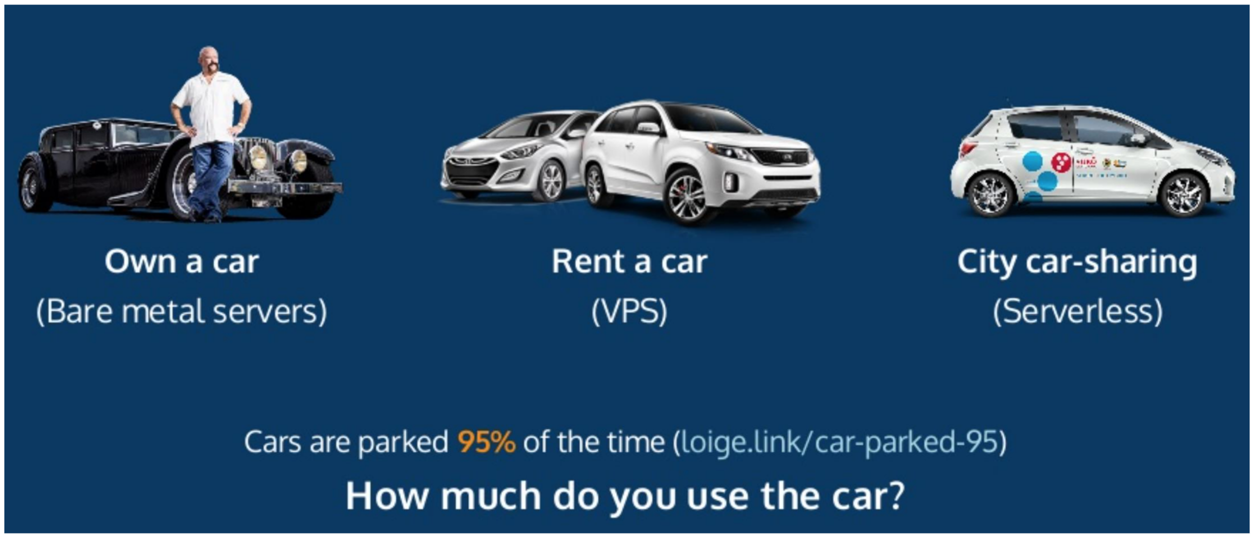

Own a car
(Bare metal servers)
Rent a car
(VPS)
City car-sharing
(Serverless)
Cars are parked 95% of the time (loige.link/car-parked-95)
How much do you use the car?

Figure 4: Car analogy illustrating different levels of cloud abstraction and resource sharing.

5.5 Open Question

Google has pioneered and created many distributed systems and technologies that shape today's cloud computing, but why have Amazon (and even Microsoft) won over Google Cloud (GCP) in cloud computing market shares?

6 File System

6.1 Definition

A **file system** is a persistent sequence of bytes that stores logically coherent digital objects for applications. It is part of the operating system (OS) that provides the mechanisms for creating, organizing, accessing, and managing files.

Core component:

- **File:** A collection of related data identified by a name. It is interpreted by an application as structured content (e.g., text, image, or executable).
- **Directory (Folder):** A cataloging structure that organizes files hierarchically. It contains references to files or subdirectories.

6.2 Logical vs. Physical Levels

- **Logical Level:** The abstraction provided to users and programs—file names, directories, and file paths.
- **Physical Level:** The organization of data on storage devices (blocks, inodes, sectors, etc.).

6.3 Examples of File Systems

- Linux: `ext2`, `ext3`, `ext4`
- Windows: `NTFS`
- macOS: `APFS`
- Cross-platform: `FAT`, `exFAT`

A file system can also be **mounted** onto an OS, making its files accessible under a specific directory path.

6.4 File Systems vs. Databases

Aspect	**File System**	**Database**
Structure	Unstructured or semi-structured (byte streams)	Structured (tables, schemas, indexes)
Access	File-based (open/read/write)	Query-based (SQL, APIs)
Concurrency	Limited (file locks)	Built-in (ACID transactions)
Use Case	Documents, binaries, media files	Analytical or transactional data

A database builds on top of a file system, providing organization, indexing, transactions, and query capability.

7 Cloud and Distributed Systems

7.1 Historical Context

In the lecture the professor offered a economical aspect of the history of Cloud and distributed system. Early computing relied on increasingly powerful single machines, following **Moore's Law**—transistor density doubling every 18–24 months. As Moore's Law plateaued (due to physical and economic limits), computing performance stopped scaling exponentially, while data volumes continued to grow.

To keep up, engineers began **scaling out** instead of scaling up, combining many commodity machines into clusters—leading to **distributed systems** and later **cloud computing**. This is the **need-base** argument for the development of cloud computing.

7.2 Distributed Systems

A **distributed system** is a loosely coupled collection of computers communicating through message passing (usually over a network) to achieve a common goal.

Key Challenges

- **Partial Failures:** Some nodes or network links may fail while others continue operating.
- **Asynchrony:** Communication delays and timing differences make synchronization difficult.

Relationship between distributed computing and parallel computing:

$$DistributedComputing = ParallelComputing + PartialFailures$$

7.3 Historical Distributed Programming Models

- **MPI (Message Passing Interface):** High-performance computing; explicit message passing.
- **DSM (Distributed Shared Memory):** Provides a shared-memory abstraction across nodes.
- **RPC (Remote Procedure Calls):** Allows calling remote functions as if local.
- **Modern Frameworks:** Ray, Dask, Spark, and others simplify distributed computation with higher-level APIs.

Remark: Distributed programming is very hard even with the simplification with mordern frameworks.

7.4 Datacenter as the "New Computer"

If an entire datacenter is treated as one large computer, then its **Operating System** comprises distributed resource management frameworks, with distributed storage and disk, distributed and shared Memory, distributed CPU and GPU to process data, and networks to retrieve data from remote.

7.5 Core Components in Cloud/Distributed Infrastructure

Category	Examples	Purpose
Data Storage	GFS, HDFS, S3, Ceph	Reliable distributed file systems
Data Sharing / KV Stores	Bigtable, DynamoDB, Cassandra	Low-latency distributed data access
Programming Abstractions	MapReduce, Hive, Pig, Spark, Ray	Distributed data processing
Resource Management	YARN, Kubernetes, Mesos	Scheduling and resource orchestration
Coordination Services	ZooKeeper, BookKeeper, etcd	Metadata management and synchronization

7.6 Milestones

- **Google File System (GFS, 2003):** Foundation for HDFS.
- **Google MapReduce (2004):** Inspired Hadoop MapReduce.
- **Google Bigtable (2006):** Inspired HBase and Cassandra.

These works established the core ideas behind modern cloud data processing.

7.7 Utility argument for cloud computing

There are two major solution for cloud computing for IT companies:

- **On-premise/supercomputer :** Buying and maintaining physical servers in their own company.
- **Cloud sevice:** Renting compute, storage, and networking resources on demand. This model provides elasticity, scalability, and global accessibility.

Somehow, computation is like utility such as electricity, network, we use everyday, and uneven in different time (on demand). Managing and storing computers spent a lot of cost, which might not be practical for smaller companies. This is the **utility argument** for developing massive cloud services.