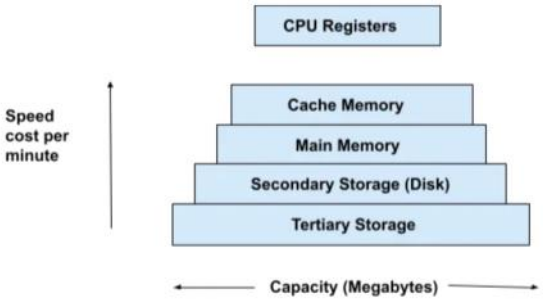
OS concepts - 1

Saturday, October 5, 2024 1:58 PM

→ **Cache**
- → **Cache** is fast because it is designed to store frequently accessed data in a way that allows quick retrieval, bypassing slower access methods.
- → Cache memory is nearest to the CPU and stores all the recent instructions. The term cache means a safe place for hiding and storing things.
- → Cache memory is a small fast memory that holds copies of recently accessed instructions and data.
- → The effectiveness of the cache mechanism is based on the property of locality of reference. It means many instructions for local areas of the program are executed repeatedly. The active segments of the program are placed in cache memory by reducing the total execution time.
- → The performance of cache memory is calculated in terms of hit ratio.
- → Hit ratio = hit/(hit + miss) = no. of hits/ total access.

→



Summary of Why Cache is Fast:

- → **Location**: Close to the CPU, often on the chip itself.
- → **Technology**: Uses SRAM, which is faster than DRAM used in main memory.
- → **Size**: Small, making it easier and quicker to search.
- → **Data Locality**: Takes advantage of temporal and spatial locality in data access patterns.
- → **Reduces I/O Operations**: Avoids the need to access slower memory or disk.
- → **Efficient Management**: Predictive algorithms keep the most relevant data in cache for faster access.

Summary of Why SRAM is Faster:

1. **No need for periodic refreshing**, unlike DRAM.
2. **Simpler and faster read/write operations** due to its transistor-based design.
3. **Closer proximity to the CPU** (used in cache) allows quicker access.
4. **Faster access time** due to optimized architecture (1–2 ns for SRAM vs. 10–100 ns for DRAM).
5. **No need for charge detection** in SRAM, unlike DRAM's capacitor-based storage.
6. **Parallel access** optimizations allow faster data throughput.

→ **Thrashing**

→

→ **What is Thrashing?**

Thrashing is a situation where the **system spends more time swapping pages in and out of memory (RAM and disk)** than executing actual processes. It's a kind of **performance collapse** caused by **excessive page faults**.

Why Does Thrashing Happen?

Thrashing usually occurs when:

- Many processes are running.
- Each process needs **more pages** than the system can keep in RAM.
- This leads to **constant page replacements**, meaning:
  - A page is loaded into memory…
  - Then immediately swapped out before it can be used much.

So the system just **keeps loading and unloading pages** (from disk), doing little useful work.

How to Handle or Avoid Thrashing

Solution	Description
Reduce multiprogramming	Limit number of active processes
Use better page replacement algorithms	Like LRU (Least Recently Used) or Working Set Model
Allocate more RAM	If possible, to hold more active pages
Use the Working Set Model	Keep only the pages that a process is actively using in RAM

How Thrashing Affects System Performance

When **thrashing occurs**, the system gets stuck in a loop of constantly **swapping pages** between **RAM and disk** instead of doing actual useful work.

Effects:

Problem	Impact on Performance
Excessive page faults	Wastes CPU time on memory management
High disk I/O	Slows down the system dramatically
Low CPU utilization	CPU waits for memory → becomes underused
Memory congestion	Not enough space to hold working sets
Application slowdown	Apps freeze, lag, or crash
System responsiveness drops	UI freezes, input/output delays

Key Terms:

- **Page**: A page is a fixed-size, contiguous block of a (process)virtual address space (i.e., virtual memory). Process is divided into pages.
- **Frame:** a fixed size of block of physical memory / RAM.
- **Paging**: Paging is a memory management technique where the operating system divides virtual memory into fixed-size pages and physical memory (RAM) into fixed-size frames, and maps pages to frames.
- **Thrashing**: When the operating system is overwhelmed by page faults (requests for pages not currently in RAM), leading to excessive paging operations and slowing down system performance.

What is a Page?

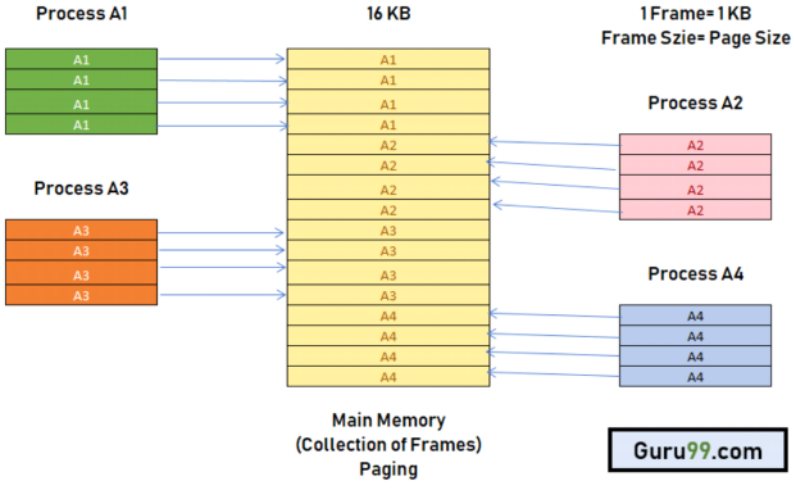
- A **page** is a **fixed-size block of virtual memory** or process is divided into fixed contiguous block of memory known as page.
- It is part of a **process's address space** (i.e., memory the process *thinks* it has).
- Managed by the **operating system**.
- Common size: **4 KB** (but can vary, e.g., 2 KB, 8 KB, etc.).

What is a Frame?

- A **frame** is a **fixed-size block of physical memory** (i.e., actual **RAM**).
- The size of a frame is **equal** to the size of a page.
- RAM is divided into multiple such frames.
- The OS uses these frames to **store pages**.

Relationship Between Page and Frame

- Pages (from **virtual memory**) are **mapped to frames** (in **physical memory**).
- This mapping is handled by the **page table**.
- When a process accesses a virtual address:
  - The OS finds the **page number**.
  - Checks where that page is loaded in RAM (i.e., which frame).
  - If not present, **page fault** → load from disk → into a free frame.



When a **process needs a page**:

The **process gives a virtual address** (e.g., Page 3, offset 100)

The **OS checks the page table**

- If Page 3 is in **RAM** → OS uses it ☑
- If not → **Page fault** occurs ❌

The OS then:
- Loads Page 3 from **disk (secondary storage)** into **RAM**
- Updates the **page table**
- Resumes the process

Virtual memory is a technique used by operating systems to allow a computer to run more processes or larger applications than its physical memory (RAM) can hold. It does this by using both **RAM** and **disk space** (typically a part of the hard drive called a **swap file** or **page file**) to simulate a larger pool of memory.
Virtual memory is a memory management technique used by the operating system to make a process think it has more memory than physically available in RAM.

Here's how virtual memory works:

1. **Virtual Address Space**: Each process running on your computer thinks it has access to a large, continuous block of memory (this is the **virtual address space**). For example, a process might think it has 4 GB of memory, even if your computer only has 8 GB of RAM for all processes combined.

2. **Paging**: The memory is divided into small chunks called **pages** (typically 4 KB). Some of these pages are stored in **RAM**, and others are temporarily stored on the **hard disk** in a swap file.

3. **Page Table**: The operating system uses a **page table** to keep track of where each page is: whether it's in **RAM** (physical memory) or on the **hard disk** (swap space). When a process needs to access data, it looks up the page in the page table to find where that data is.

4. **Page Fault**: If the data a process needs is not in RAM (i.e., it's been swapped out to disk), a **page fault** occurs. The operating system pauses the process, retrieves the page from the disk, loads it into RAM, and updates the page table. If RAM is full, it will swap out another page to make space.

5. **Demand Paging**: Only the pages of memory that are actively being used are kept in RAM. The rest stay on the hard drive until they are needed.

Virtual Memory in Simple Terms

1. **Illusion of Large Memory:**
   - Each process **thinks it has a large, contiguous memory**, even if physical RAM is smaller.
2. **Pages in RAM:**
   - At any time, **only some pages of a process are in RAM**. The rest can stay on the hard disk (swap space).
3. **Page Fault & Swapping:**
   - If a process tries to access a page **not in RAM**, the OS performs **page swapping**:
     - Moves a page from RAM to disk (if RAM is full).
     - Brings the needed page from disk to RAM.
4. **Bigger Processes:**
   - This allows RAM to handle processes **larger than its actual size**, because only a portion of the process needs to be in RAM at any moment.

Real-World Example:

Consider a scenario where you have a computer with 8 GB of RAM, but you need to run a video editing application that requires 12 GB. Without virtual memory, the application would fail or slow down significantly because it doesn't have enough RAM. With virtual memory, the operating system uses disk space to supplement RAM, allowing the application to run by swapping less frequently used data to disk, even though the physical memory isn't large enough.

Is Only a Small Amount of Data in RAM at a Given Time?

Yes, that's correct. At any given time, only a portion of the data from all the programs and processes running on your computer is loaded into **RAM**.

- **Active Pages in RAM**: When a program is running, it doesn't need to have all its data in RAM at once. Only the **most active or recently used data** (pages) are kept in RAM.
  - For example, if you're using a word processor and browsing the web, only the pages that your CPU needs immediately (like the text you're typing or the part of the web page you're viewing) are kept in RAM.
- **Inactive Pages in Swap**: Less frequently used data or background processes are moved to the **swap file** (on the hard disk or SSD). When you switch tasks (e.g., open another tab in your browser), the operating system may swap out less important pages to disk and bring the needed pages into RAM.

Does the CPU Access the Hard Disk or RAM?

You're right that the **CPU only directly accesses RAM** (main memory) and **not the hard disk**. Here's how this works:

1. **CPU Accesses RAM**: When the CPU needs data, it **accesses RAM**. If the required data is in RAM, it can be quickly fetched by the CPU.
2. **Page Fault and Disk Access**: If the data is not in RAM (because it has been swapped to the hard disk due to virtual memory), the CPU cannot directly access the hard disk. Instead, a **page fault** occurs, and the operating system takes over to handle the page fault:
   - The OS **pauses** the process.
   - It **retrieves** the necessary page from the hard disk or SSD (where the swap file is located) and loads it into **RAM**.
   - Once the data is in RAM, the CPU can access it.
     While this process happens, the CPU cannot access the data until the page is loaded back into RAM. This creates a delay, as a ccessing data from the hard disk is much slower than accessing it from RAM.
3. **Cache Memory**: To speed up data access even further, the CPU has small amounts of extremely fast memory called **cache** (L1, L2, L3 cache). The cache stores the most frequently accessed data. If the data the CPU needs is in the cache, it can access it even faster than from RAM. If not, it goes to RAM.

How Virtual Memory Creates the Illusion of More Memory:

1. **Abstraction of Memory**:
   - Each process thinks it has a large and continuous block of memory (its **virtual address space**), even if the physical RAM is limited. For example, a program might believe it has 4 GB of memory available.
2. **Limited Physical RAM**:
   - In reality, only a portion of that virtual memory space is loaded into **RAM** at any given time. The rest can reside on the hard disk or SSD in a **swap file**.
3. **Page Management**:
   - Memory is divided into fixed-size units called **pages**. Some of these pages are in RAM, while others are swapped out to disk when not actively needed.
4. **Swapping Pages**:
   - When a process needs to access data that is not currently in RAM, a **page fault** occurs. The operating system pauses the process, retrieves the required page from disk, and loads it into RAM.
   - If RAM is full, the OS will **swap out** another page (the least used or least recently accessed) to make room for the new page. This means pages can be constantly swapped in and out of RAM based on the process's needs.
5. **Efficient Use of Resources**:
   - Only the active pages (those currently in use) are kept in RAM, while inactive pages can be stored on the disk. This optimize s RAM usage and allows multiple processes to run simultaneously, even if their combined memory requirements exceed the physical RAM available .

Key Takeaway:

- **Virtual memory** allows systems to **efficiently manage memory** and create the illusion of having more memory available than physically exists. It does this by using a combination of **RAM** and **disk space** to swap pages in and out as needed, maintaining system performance and allowing for multitasking.

1. What is a deadlock?

- A deadlock is when two or more processes can't make any progress because each one is waiting for the other to release a resource. It's like a traffic jam where no car can move because they're blocking each other.

2. Example (Simple Real-Life Analogy):

Imagine two people:

- **Person A** has a fork and is waiting for a knife.
- **Person B** has a knife and is waiting for the fork.

Neither person can eat because they are both waiting for the other to give up what they need. This is a **deadlock**. Both people are stuck, and nothing can proceed unless someone lets go of the resource they have.

Key Conditions for Deadlock:

For a deadlock to occur, four conditions must hold simultaneously. These are known as **Coffman's conditions** or the **necessary conditions for deadlock**:

1. **Mutual Exclusion**:
   - At least one resource must be held in a non-sharable mode. Only one process can use the resource at a time, and if another process requests the resource, it must wait until the resource is released.
2. **Hold and Wait**:
   - A process is holding at least one resource and is waiting to acquire additional resources that are currently being held by ot her processes.
3. **No Preemption**:
   - A resource that has been allocated to a process cannot be forcibly taken away from that process. It can only be released volu ntarily by the process holding it.
4. **Circular Wait**:
   - There exists a circular chain of processes where each process is waiting for a resource that the next process in the chain ho lds. For example, Process A is waiting for a resource held by Process B, Process B is waiting for a resource held by Process C, and Process C i s waiting for a resource held by Process A.

If all four conditions are met, a deadlock can occur.

Real-World Analogy:

Think of deadlock as a traffic jam at a four-way intersection where:

- Each car is in a different lane (representing a different process).
- Each car needs to move forward (acquire a resource), but is blocked by another car (because it holds a resource needed by another car).
- No car can move because they are all waiting for one another, and no one is willing to back up (release the resource).

Example in Operating Systems:

Let's take an example involving two processes (P1 and P2) and two resources (R1 and R2):

1. **Process P1** acquires **resource R1** and then waits for **resource R2** to proceed.
2. **Process P2** acquires **resource R2** and then waits for **resource R1** to proceed.

In this situation, P1 is waiting for R2 (held by P2), and P2 is waiting for R1 (held by P1). Since neither process can release the resource it's holding (because it is waiting for the other process to release a different resource), both processes are stuck in a deadlock.

Methods to Handle Deadlock:

1. **Deadlock Prevention**:
   - Modify the system in such a way that at least one of the four deadlock conditions cannot hold. Examples:
     - **Mutual Exclusion**: Allow some resources to be shared among processes (not always possible, e.g., printers).
     - **Hold and Wait**: Require processes to request all resources they need at once, and only proceed when all requested resources are available.
     - **No Preemption**: Allow the operating system to forcibly take resources away from processes under certain conditions.
     - **Circular Wait**: Impose a total ordering of all resource types, and require that each process request resources in increasing order of enumeration.
2. **Deadlock Avoidance**:
   - Use algorithms like **Banker's Algorithm** to dynamically check whether allocating a requested resource will lead to a deadlock before actually

allocating it. If it does, the request is denied. This method requires the system to have prior knowledge of all resources a process may request during its execution.

3. **Deadlock Detection and Recovery**:
   - Allow the system to enter a deadlock state, but periodically check for deadlocks. If a deadlock is detected, take actions to recover from it, such as:
     - **Terminate processes**: One or more of the processes involved in the deadlock may be killed to break the cycle.
     - **Resource Preemption**: Resources can be forcibly taken away from some processes, and their execution can be rolled back to an earlier safe state.
4. **Ignoring Deadlock (Ostrich Algorithm)**:
   - In some systems, especially those where deadlocks are rare, the operating system may simply ignore the problem altogether. Th is is known as the **Ostrich Algorithm**, where the system assumes deadlocks won't occur often enough to warrant handling, and leaves it to the user to manually resolve deadlocks (e.g., by killing processes).

What is Paging?

Paging is a memory management technique that breaks up **physical memory** (RAM) into fixed-size chunks called **pages** (in virtual memory) and **frames** (in physical memory). It allows the operating system to manage memory more efficiently and run programs that might require more memory than is physically available.

Key Concepts in Paging:

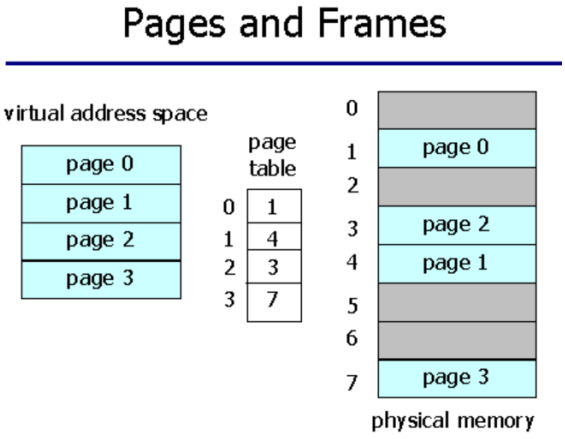
1. **Pages**:
   - The **virtual memory** (the address space a program "thinks" it has) is divided into equal-sized blocks called **pages**.
   - Pages typically have a size like 4 KB, 8 KB, etc.
2. **Frames**:
   - The **physical memory** (the actual RAM) is also divided into blocks of the same size as pages. These blocks are called **frames**.
   - For example, if pages are 4 KB in size, each frame in the physical memory will also be 4 KB.
3. **Page Table**:
   - The OS uses a **page table** to keep track of where each virtual page is stored in physical memory (which frame it is in).
   - Each entry in the page table maps a **virtual page number** to a **physical frame number**.
4. **Logical Address vs. Physical Address**:
   - The address a process uses to refer to memory is called a **logical address** (or **virtual address**). The operating system translates this logical address to a **physical address** (actual location in RAM).
   - Paging allows programs to use logical addresses, while the OS handles the actual physical addresses.

How Paging Works:

1. **Dividing Memory**:
   - Suppose you have a program that requires 20 KB of memory. This will be broken up into 5 pages (if the page size is 4 KB).
   - The operating system will place these pages into **available frames** in RAM. These frames may not be continuous; they can be scattered throughout the physical memory.
2. **Page Table Translation**:
   - When the program needs to access data, it uses a **virtual address**. The OS uses the **page table** to translate this virtual address into a **physical address**.
   - The virtual address consists of two parts:
     - **Page Number**: Used to index into the page table.
     - **Page Offset**: Used to locate the exact byte within the page.
   - The page table will map the page number to the correct frame number, and the offset will give the exact byte within that fram e.

Page Fault:

- **What happens if the page is not in memory (RAM)?**
  - Sometimes, a page might not be in physical memory when it's needed because it was swapped out to disk (in the **swap file** or **page file**) to make space for other programs. This is known as a **page fault**.
  - When a page fault occurs:
    1. The operating system pauses the program.
    2. It finds the required page on disk.
    3. Loads it into an available frame in RAM.
    4. Updates the page table to reflect the new location.
    5. Resumes the program.



Simple Explanation of Paging

1. **Virtual Memory**:
   - Think of this as a huge library where you can borrow books (data) that aren't all stored in your room (RAM) at the same time.
   - The library has a limit on how many books you can keep in your room (the RAM), but you can always swap them out with the libr ary's stock (the hard disk).
2. **Pages and Frames**:
   - **Pages**:
     - Imagine that each book in the library is divided into chapters (these are the pages). Each chapter is a small part of the entire book.
   - **Frames**:
     - Your room has bookshelves (these are the frames) where you can place the chapters you want to read right now. Each shelf can hold one chapter.
3. **How It Works**:
   - When you want to read a book (run a program), you don't bring the whole book to your room. Instead, you only bring the chapte rs you need (pages) and place them on your bookshelves (frames) in your room.
   - If you need to read a chapter that isn't currently on a shelf in your room, you have to go back to the library (hard disk) to get it. This is called a **page fault**.
4. **Page Table**:
   - This is like a list that tells you which chapters are on which shelves in your room and which ones are still in the library. It helps you know where to find everything.

Key Takeaways:

- **Pages** are small parts of a program or data, like chapters of a book.
- **Frames** are where those pages are stored in RAM, like bookshelves in your room.
- Virtual memory lets you use more data than your RAM can hold by swapping pages in and out from the hard disk.

Example:

- If a program has 8 pages but your RAM can only hold 4 pages at a time:
  - You load pages 1 to 4 into the frames (shelves).
  - If you need page 5, you replace one of the loaded pages with page 5 from the hard disk (library).

Mutex (Mutual Exclusion)

What is Mutex?

- A **Mutex** ensures that only **one thread or process** can access a shared resource (e.g., a file, variable, or critical section) at a time.
- If one thread locks the mutex, all other threads that want to use the same resource must wait until the mutex is unlocked.

Real-World Analogy

Think of a **single-use restroom**:

- If one person enters the restroom and locks the door, no one else can enter until the door is unlocked.

1. **Definition**: A mutex is a lock that allows only one thread at a time to access a critical section (a piece of code that accesses shared resources).
2. **Key Properties**:
   - **Ownership**: A mutex can only be unlocked by the thread that locked it.
   - **Binary**: It is essentially a binary lock (locked/unlocked).
   - **Use Case**: Ensures **exclusive access** to a resource.
3. **How it Works**:
   - A thread **locks** the mutex before entering the critical section.
   - Other threads are **blocked** until the mutex is unlocked.
   - The first thread **unlocks** the mutex after completing its operation, allowing the next waiting thread to proceed.

Semaphore

What is Semaphore?

- A **Semaphore** is a more general synchronization tool that controls **access to a resource pool** with multiple available slots.
- It allows **multiple threads** to access the resource simultaneously, but only up to a **fixed limit** (like 3 threads at a time, for example).

Real-World Analogy

Think of a **parking lot with 5 parking spots**:

- If all 5 spots are occupied, new cars must wait until a spot is freed.

1. **Definition**: A semaphore is a signalling mechanism that allows a fixed number of threads or processes to access a resource simultaneously.
2. **Key Properties**:
   - **Counter**: A semaphore has a counter that represents the number of resources available.
   - **Permit System**: Threads acquire a permit to proceed. If no permits are available, threads are blocked.
   - **Non-Binary**: Unlike mutexes, semaphores can allow multiple threads to access a resource concurrently.
3. **Types**:
   - **Binary Semaphore**: Similar to a mutex but does not have ownership restrictions.
   - **Counting Semaphore**: Allows a specific number of threads to access a resource.
4. **How it Works**:
   - A thread **decrements** the semaphore (takes a permit) before entering the critical section.
   - If the counter is zero, the thread is **blocked** until another thread **increments** the semaphore (releases a permit).
   - When done, the thread **increments** the semaphore.

Key Differences Between Mutex and Semaphore

Aspect	Mutex	Semaphore
Purpose	Allows **only one thread** to access a resource.	Allows **multiple threads** to access a resource simultaneously (up to a limit).
Counter	Binary (0 or 1).	Can count (e.g., 0, 1, 2, ...).
Ownership	Only the locking thread can unlock it.	No ownership; any thread can release.
Blocking	Other threads are **blocked** until unlocked.	Threads are blocked if permits run out.
Use Case	Exclusive access (e.g., updating shared variables).	Shared resources with a limit (e.g., thread pools, database connections).,

Synchronization

What is Synchronization?

- **Synchronization** is the technique used to control the access of multiple threads/processes to shared resources/critical section.
- It ensures that **only one thread accesses a shared resource at a time**, preventing conflicts and inconsistent data.

Why is it needed?

- When multiple threads try to read/write the same data simultaneously, it can cause **data corruption** or **unexpected behavior**.
- Example: Two threads incrementing the same counter at the same time can produce wrong results.

```
mutex m; // a mutex lock

// Thread function
void increment() {
    m.lock();         // lo
    counter = counter + 1;
    m.unlock();       // un
}
```

🏦 What is Banker's Algorithm?

Banker's Algorithm is a **deadlock avoidance algorithm** that works **like a bank loan system**.

- Before giving a resource to a process, the system checks:
  "Will the system still be in a safe state if I do this?"

If yes, it **grants** the resource.
If no (might cause deadlock), it **waits**.

🤔 Why "Banker"?

Imagine a **banker** who gives out loans (resources) to customers (processes), but only if they are sure that all loans can eventually be repaid (processes can finish and release resources).

🧛 Example: 3 customers, 10 total dollars

- **You (Banker)**: $10 in hand
- **Customers**:
  - A might need $7 (currently has $0)
  - B might need $5 (has $3)
  - C might need $3 (has $2)

Now, **Customer A asks for $3**.

☑ Steps the Banker (algorithm) follows:

1. **Check**: Does A's request ($3) ≤ what A may still need AND ≤ what I (banker) have?
   - Yes: $3 ≤ A's remaining need ($7) and ≤ $10 available
2. **Pretend**: Suppose I give $3 to A → A now has $3, and I have $7 left
3. **Safety Check**: Can I still let **everyone** eventually finish?
   - A needs up to $4 more → I have $7 → ☑ can finish
   - B needs $2 more → I have $7 → ☑ can finish
   - C needs $1 more → I have $7 → ☑ can finish
4. ☑ Since **everyone can finish**, it's **safe**, so I give A the $3.

1️⃣ Process

- A **process** is an instance of a running program.
- It has its own **memory space**, **code**, **data**, and **resources**.
- Processes are **independent** and do not share memory by default.
- Switching between processes involves **context switching**, which is relatively heavy.

2️⃣ Thread

- A **thread** is the smallest unit of execution within a process.
- Multiple threads in the same process **share the same memory space** but have their own **stack and registers**.
- Threads are lightweight compared to processes.
- Threads can run concurrently and improve performance.

3️⃣ Critical Section Problem

- When multiple threads/processes access shared data/resources **at the same time**, it can cause **data inconsistency**.
- The part of code where shared resources are accessed is called the **critical section**.
- The **critical section problem** is how to ensure **only one thread/process accesses the critical section at a time** to prevent race conditions.

4️⃣ Solutions to the Critical Section Problem

🔒 **Mutex (Mutual Exclusion)**

- A **mutex** is a lock that allows **only one thread** to enter the critical section at a time.
- Other threads trying to enter wait until the mutex is unlocked.
- Basic operations:
  - lock() — acquire the mutex.
  - unlock() — release the mutex.
- Ensures **mutual exclusion** but can lead to **deadlocks** if not used carefully.

🔔 **Semaphore**

- A **semaphore** is a more general synchronization tool.
- It uses a counter to control access to shared resources.
- Types:
  - **Binary semaphore** (similar to mutex, counter is 0 or 1).
  - **Counting semaphore** (counter can be more than 1, allows multiple accesses).
- Operations:
  - wait() or P() — decrease the semaphore value, wait if zero.
  - signal() or V() — increase the semaphore value, release waiting threads.

☑️ **Shared Resource — Simple Definition:**
A shared resource is any data or object that multiple processes or threads can access — usually at the same time.

🔐 **What is a Critical Section?**
A **critical section** is a part of your program where **shared resources are accessed or modified** — and therefore, **only one thread/process should enter it at a time** to avoid problems like **race conditions** or **data corruption**.

⚠️ **What is a Race Condition?**
A **race condition** happens when **two or more threads/processes access shared data at the same time**, and **the final outcome depends on the timing** of how the threads are scheduled.

🔁 **In Simple Words:**
A **race condition** is a bug that happens when **the program "races" to use a shared resource**, and the **order of execution affects the result** — often in an **unexpected or wrong way**.