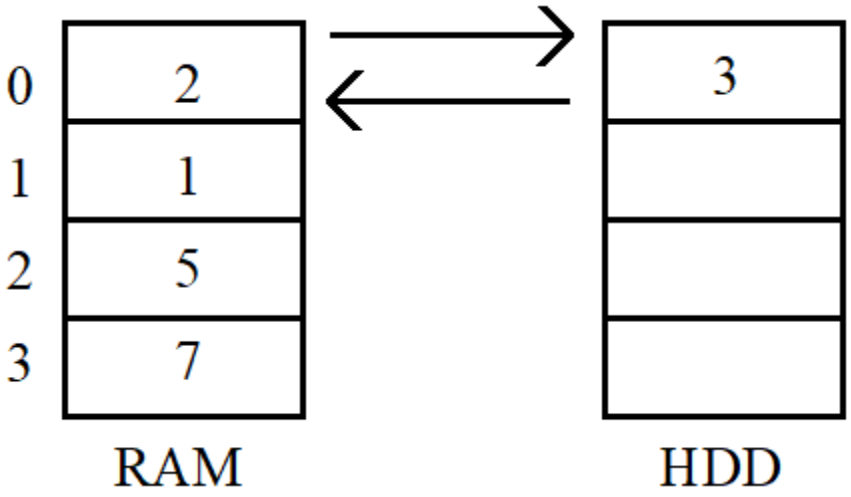
OPERATING SYSTEM

Topic:

- LRU Cache
- Concurrency & Parallelism
- Reader Writer Problem (→ Mutex)
- Dead Locks
    - Characteristics

Let's suppose, in an HDD,



If the HDD process has to come to the RAM, we have to swap it with a process in RAM. Virtual memory is also called swap memory. So, how should we remove processes from RAM? This can be done with Page Replacement Policy also called Eviction Policy.

Page Replacement or Eviction Policy

- Least frequently used →LFU Cache. Not used now-a-days.
- Least recently used → LRU Cache

LRU Cache:

A **Least Recently Used (LRU) Cache** organizes items in order of use, allowing you to quickly identify which item hasn't been used for the longest amount of time.

Picture a clothes rack, where clothes are always hung up on one side. To find the least-recently used item, look at the item on the other end of the rack.

Accessing and Evicting

Putting things together, here are the steps we'd run through each time an item was accessed:

- Look up the item in our hash map.
- If the item is in the hash table, then it's already in our cache—this is called a "**cache hit**"

1. Use the hash table to quickly find the corresponding linked list node.
2. Move the item's linked list node to the head of the linked list, since it's now the most recently used (so it shouldn't get evicted any time soon).
    - If the item isn't in the hash table, we have a **cache miss**. We need to **load** the item into the cache:

1. Is our cache full? If so, we need to **evict something** to make room:
    - Grab the least-recently used cache item—it'll be at the tail of the linked list.
    - Evict that item from the cache by removing it from the linked list and the hash map.
2. Create a new linked list node for the item. Insert it at the head of the linked list.
3. Add the item to our hash map, storing the newly-created linked list node as the value.

Ex:

We have the following process

2 5 1 3 2 7 5 8 9 4

LRU cache is like an array. LRU cache has a size of 4, which means it can take maximum of 4 processes.

We will push the first process 2 into the cache at time 1. The memory has 4 spaces, so process 5 is pushed in with time = 2. Process 1 is not present in the cache, so we will push it and as it is not present in the cache it is called "**Cache Miss**".

We will push process 3 into the cache at time = 4. The next element 2 is present in the cache, so it is called "**Cache Hit**". We will update the time, time = 5.

Now 7, the cache is filled so we have to remove the process with least time. So, 5 with time 2 is removed and 7 with time = 6 is pushed into the cache.

The next 5 is not present in the element so it is a **cache miss**. As there is no space, we will remove 1 with time = 3 and will push 5 with time = 7 into the cache. The next element 8 is pushed into the cache removing 3 with time = 4 as there is no element 8, it is a miss; "**cache miss**".

The next process 9 is not present in the cache, so it is a **cache miss**, and this will be removing the 2 with time = 5. Finally, 8, which is already present in the cache, so it will be **cache hit**.

2	5	1	3	2	7	5	8	9	4
CM	**CM**	**CM**	**CM**	**CH**	**CM**	**CM**	**CM**	**CM**	**CH**

CM – Cache Miss

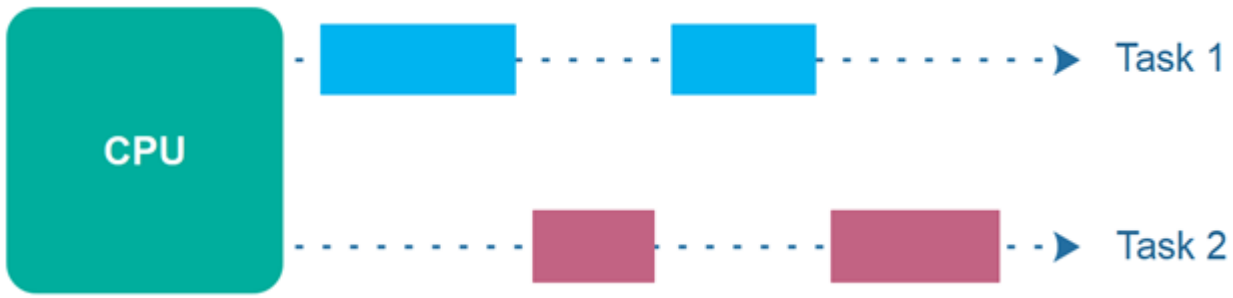
CH – Cache Hit

So, 8/10 is the miss rate and 2/10 is the hit rate of this cache.

Concurrency

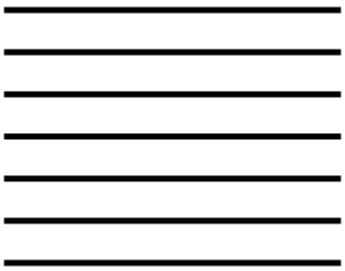
Concurrency means that an application is making progress on more than one task - at the same time or at least seemingly at the same time (concurrently).

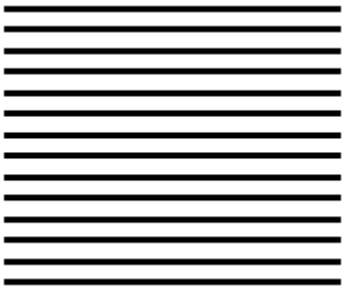
If the computer only has one CPU the application may not make progress on more than one task at exactly the same time, but more than one task is in progress at a time inside the application. To make progress on more than one task concurrently the CPU switches between the different tasks during execution. This is illustrated in the diagram below:



NOTE: Core is a computational unit. 1-core can only perform one task at time.

Ex:





CODE-1 CODE-2

The code runs from top to bottom. If we want code 1 to run in a concurrent manner i.e., to run as a separate entity and the other one as well. We want both the codes to run simultaneously. There is a race between first block and the second block. This is called a **Race Condition**. If we enclose these two codes in parallelism threads, there will be race condition regarding which one would run first.

In programming languages, if we want to run programs at same time or parallel with separate resources, is called '**thread**'.

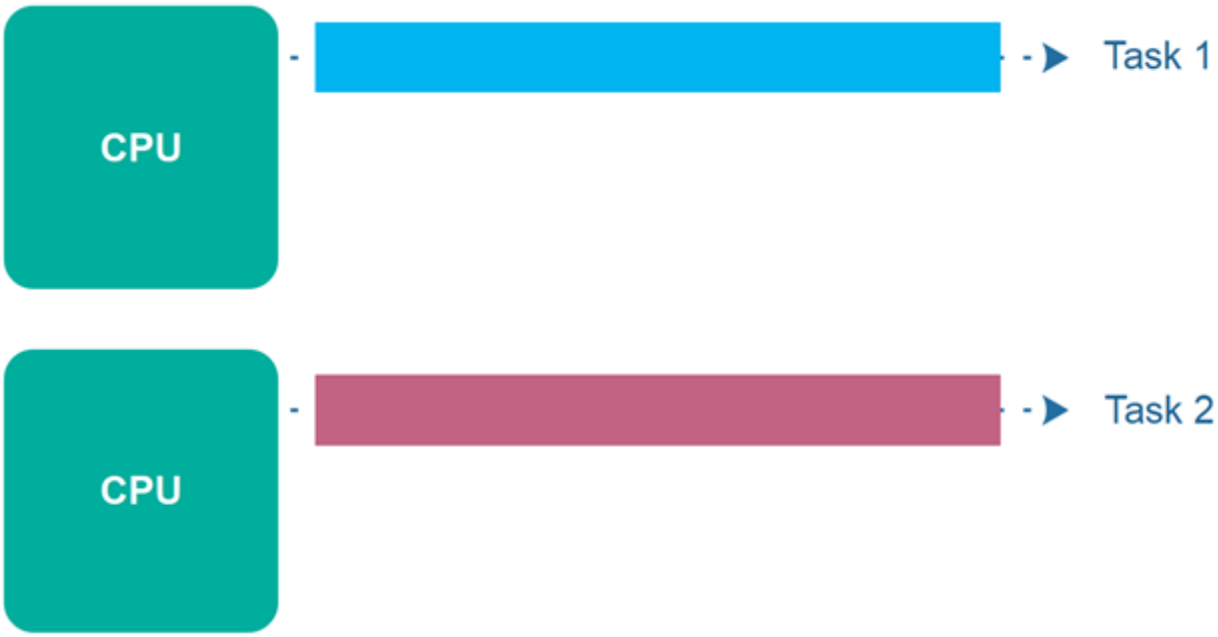
A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

If a particular file is downloaded in 30 mins, using 1-core. If we break the file into 4 parts and all the download is being done in 4 separate parts, then the time is also being reduced by $1/4^{th}$. When we are supposed to get something done fast, then we use multi-processing or multi-programming.

Parallel Execution

Parallel execution is when a computer has more than one CPU or CPU core, and makes progress on more than one task simultaneously. However, *parallel execution* is not referring to the same phenomenon as *parallelism*. I will get back to parallelism later. Parallel execution is illustrated below:



We read our code from top to bottom, but if we want a part of a code to run as a separate entity, and the next section to be run separately,

MAP REDUCE:

MapReduce is a processing technique and a program model for distributed computing based on java. The MapReduce algorithm contains two important tasks, namely Map and Reduce. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job.

The major advantage of MapReduce is that it is easy to scale data processing over multiple computing nodes. Under the MapReduce model, the data processing primitives are called mappers and reducers. Decomposing a data processing application into mappers and reducers is sometimes nontrivial. But, once we write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change. This simple scalability is what has attracted many programmers to use the MapReduce model.

READER WRITE PROBLEM:

The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e., they only want to read the data from the object and some of the processes are writers i.e., they want to write into the object.

The readers-writers problem is used to manage synchronization so that there are no problems with the object data. For example - If two readers access the object at the same time there is no problem. However, if two writers or a reader and writer access the object at the same time, there may be problems.

To solve this situation, a writer should get exclusive access to an object i.e., when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.

Reader is a person who can read item at a time and will not read if buffer is empty.

Writer is a person who can write data and produce data, and will not write if buffer is full.

PRODUCER CONSUMER PROBLEM:

The producer consumer problem is a synchronization problem. There is a fixed size buffer and the producer produces items and enters them into the buffer. The consumer removes the items from the buffer and consumes them.

A producer should not produce items into the buffer when the consumer is consuming an item from the buffer and vice versa. So the buffer should only be accessed by the producer or consumer at a time.

Pseudo Code:	**Consumer Code:**
cnt = 0 ***buffer = [0] * 10*** ***def producer:*** ***if buffer is full:*** ***continue*** ***buffer[cnt] = data*** ***cnt += 1***	***def consumer:*** ***while buffer is empty:*** ***continue*** ***buffer pop()*** ***cnt -= 1***

The ‘cnt’ in both producer and consumer are called atomic. It is actually a combination.

In producer: cnt += 1 is a combination of 3 statements,

```
register = cnt
register = register + 1
cnt = register
```

In consumer: cnt -= 1 is a combination of 3 statements,

```
register = cnt
register = register - 1
cnt = register
```

NOTE: In parallelism, we cannot have shared variable, but in concurrency we can access common variable.

Common code which two sections access at the same time.

Locking Mechanism:

If A has Rs. 1000/- and so does B. If A remove 50 and B adds 500, then

1000 → 950 → 1450 BUT

If 50 is not removed as removing 50 and add 500 happened at same time, then

1000 → 1500.

In this Locking Mechanism is used. In this, after every transaction, only water balance is updated, the lock is released so that we can do the next transaction.

Python has a library mutex, **`mutex.lock(function, object)`**.

DEAD LOCK

Deadlock is a situation that occurs in OS when any process enters a waiting state because another waiting process is holding the demanded resource. Deadlock is a common problem in multi-processing where several processes share a specific type of mutually exclusive resource known as a soft lock or software.

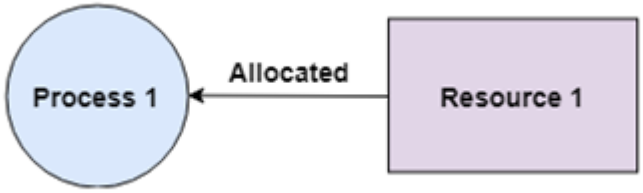
Coffman Conditions

A deadlock occurs if the four Coffman conditions hold true. But these conditions are not mutually exclusive.

The Coffman conditions are given as follows −

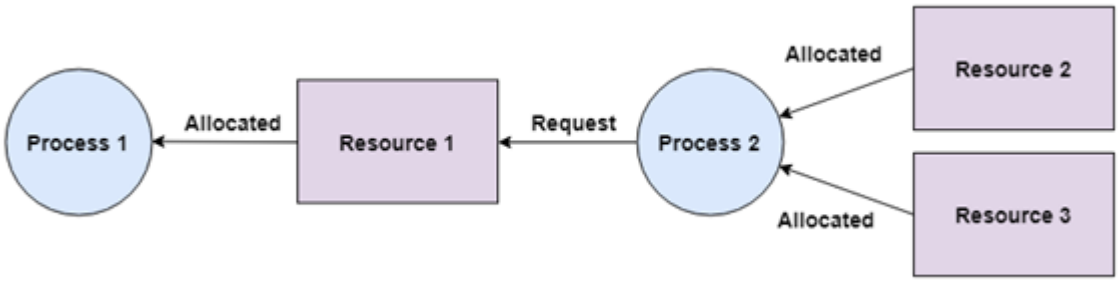
Mutual Exclusion

There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



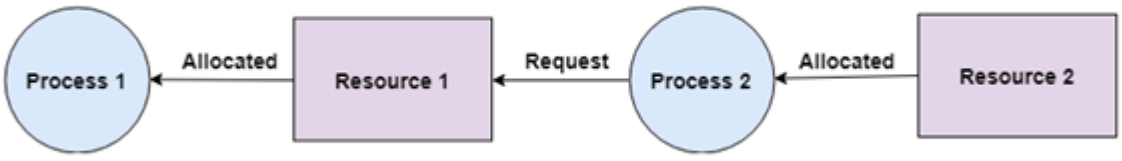
Hold and Wait

A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



No Preemption

A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



Circular Wait

A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.