HEAP MANAGEMENT

- The Memory Manager
- The Memory Hierarchy of a computer
- Locality in Programs

What is Heap?

- It is a portion of the store that is used for data that lives indefinitely, or until the program explicitly deletes it.
- All dynamically allocated data and pointers to data are stored in Heap memory

The Memory Manager

It keeps track of all the free space in heap storage at all times.

Basic functionalities are:

1. Allocation
2. Deallocation

Allocation: When a program requests memory it produces a chuck of contiguous heap memory of requested size.

Two cases/situations faced during allocation:

1. If possible, it satisfies an allocation request using free space in heap.

2. If no chunk of needed size is available, it increase the heap storage space by getting consecutive bytes of virtual memory from OS.

Deallocation: Memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests.

Memory manager do not return memory to the operating system, even if the program's heap usage drops.

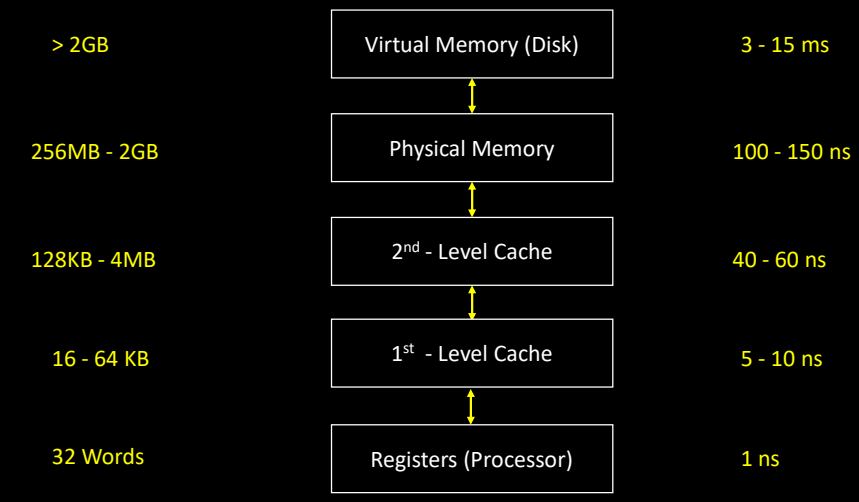
Properties we desire for memory managers

Space Efficiency: Achieved by minimizing fragmentation

Program Efficiency: Achieved by placement of objects in memory

Low Overhead: Achieved by minimizing the overhead for allocation and deallocation request. (Because these are very frequent operations in any programs)

The Memory Hierarchy of a computer



Locality in Programs

Locality: The tendency of a processor to access the same set of memory locations repetitively over a short period of time

There are two kinds of localities:

1. Temporal locality
2. Spatial locality

Temporal locality: If the memory locations it access are likely to be accessed again within a short period of time.

Spatial locality: If the memory locations close to the locations accessed are likely also to be accessed within a short period of time

The typical program spends most of its time executing innermost loops and tight recursive cycles in a program

Locality helps us to keep the most common instructions and data in the fast-but-small storage, while leaving the rest in the slow-but-large storage, with this we can lower the average memory-access time of a program.

Sometimes we may have sections of code will be heavily used, all the code is unable to keep in the fast storage so we have to dynamically adjust the content in the fast-but-small storages.