



Advanced C Workshop - Day 2

Overview of the Computer Architecture





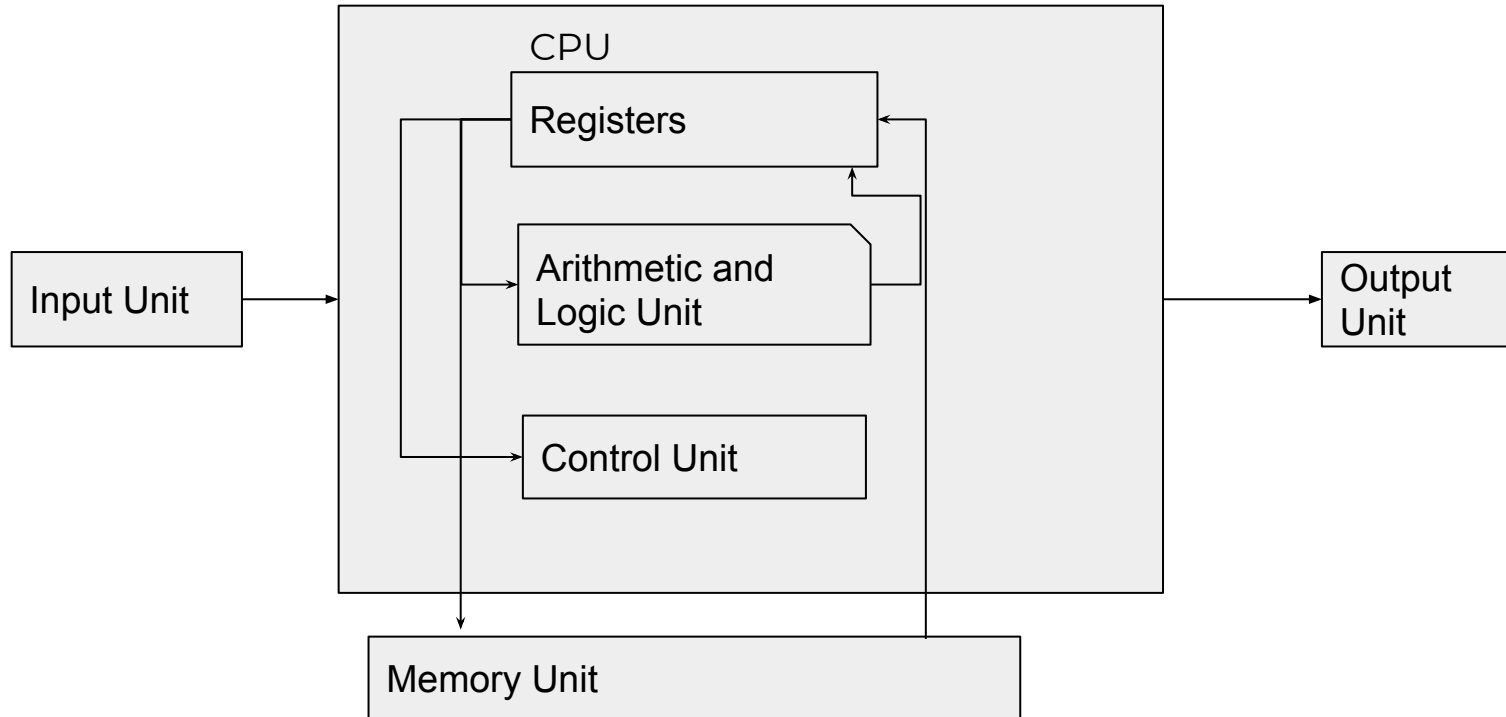
Why Are We Here Today?

We are here today for...

Knowledge of Computer Organization and Architecture will enable a C programmer to:

- Efficiently work on Embedded Systems(Microprocessors) which have limited resources.
 - In high-end graphics projects, determine which tasks to be delegated to GPU for the high performance.
 - Now you are equipped with OS as default, but a day may come, where you need to develop OS yourself.
-

What is a Computer?



Instruction Processing Stages

1. Fetching: Fetch instruction from memory.
2. Decoding: Decode the instruction by the Control Unit.
3. Calculate the (effective) address of the operand.
4. Fetching the operands from memory.
5. Executing the instruction.
6. Store the result in the proper place.



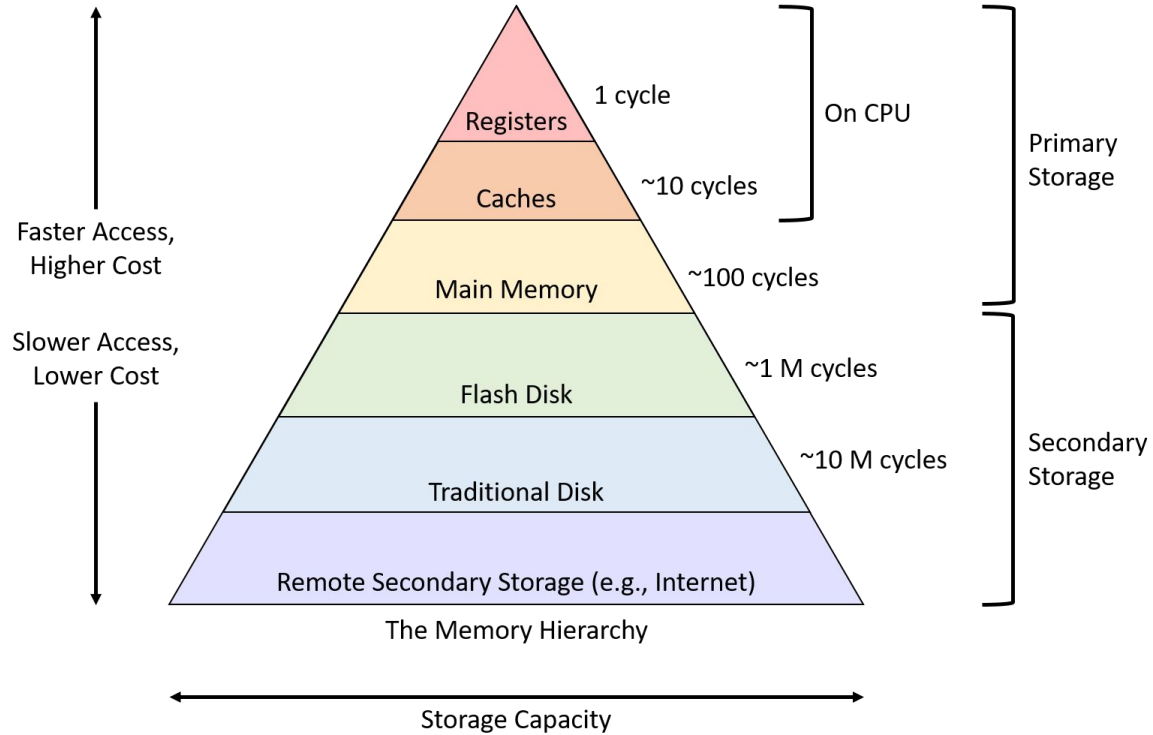
How can we make
computer faster?



Memory Hierarchy

Faster or Larger? Memory has a “wall”?





So how does this knowledge help me?

- Every upper level of memory will provide faster access time i.e. **better and faster programs** are those that frequently access the **upper levels of the hierarchy**.
- Consequently, if your program is “too slow” despite the code looking simple, chances are your program is not “memory-wise” (**too many file handling operations - a case study!**).

C too-many-file-handling.c > main()

```
21  #include <stdio.h>
20
19  // Program to -
18  // 1. Take a word say 'elephant'
17  // 2. Find all words within 'elephant', for eg - peel, pale, etc.
16  // 3. Score users on how much they score
15
14  int main(){
13      // Classic mistake - store different length words in different files
12      FILE *fp4, *fp5, *fp6, *fp7;
11
10      // Calculate permutation from words
9      char input_word[10];
8      function_to_calculate_permutations(input_word); // Spent huge time optimizing this part :facepalm:
7
6      // Write to file of 4 letter words
5      fp4 = fopen("4-letter-words-stored-here.txt", "w");
4
3      // Write to file of 5 letter words
2      fp5 = fopen("5-letter-words-stored-here.txt", "w");
1
22     // ... and so on.
1 }
```



**"code
opteemizashion"**

**"++i is
better than i++"**

**understanding
memory
hlerarchy**

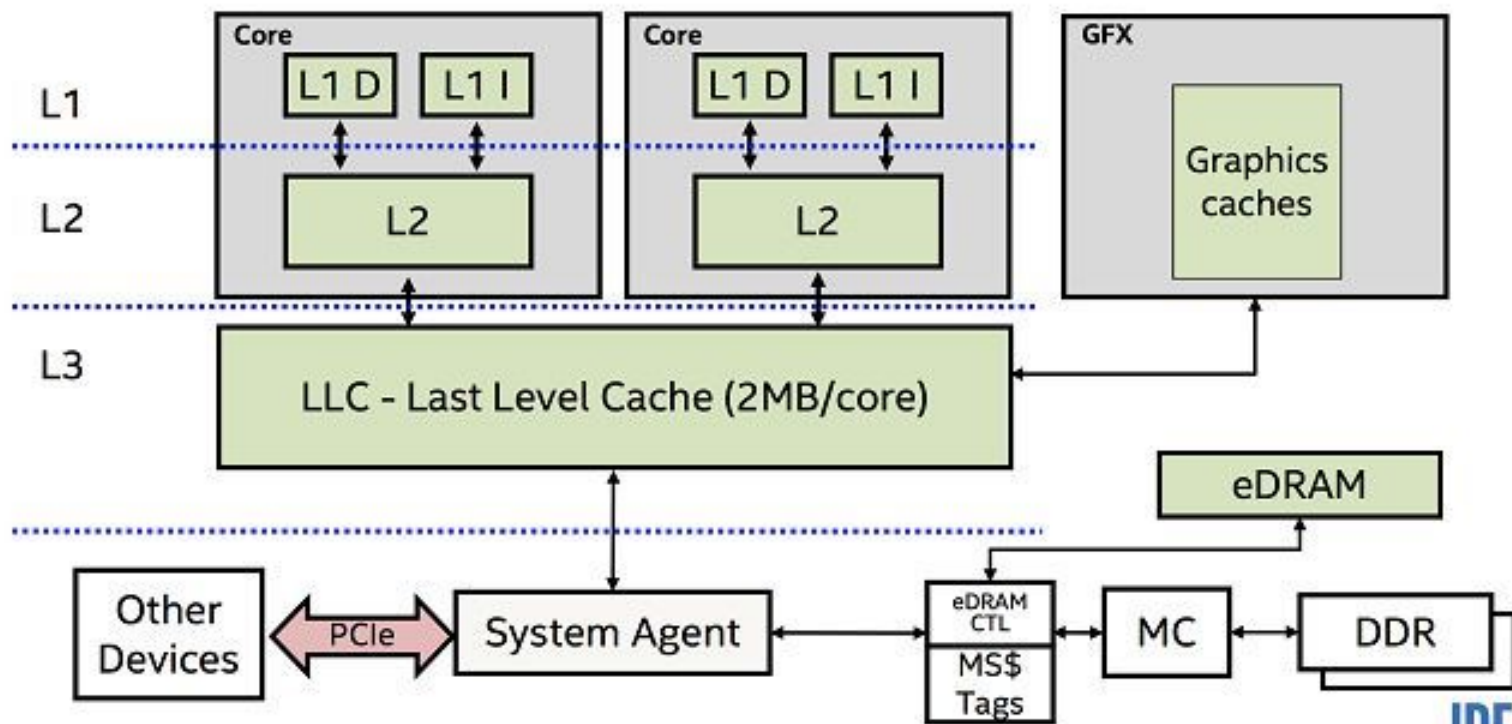


Cache

It's called "cash" NOT "catchi".

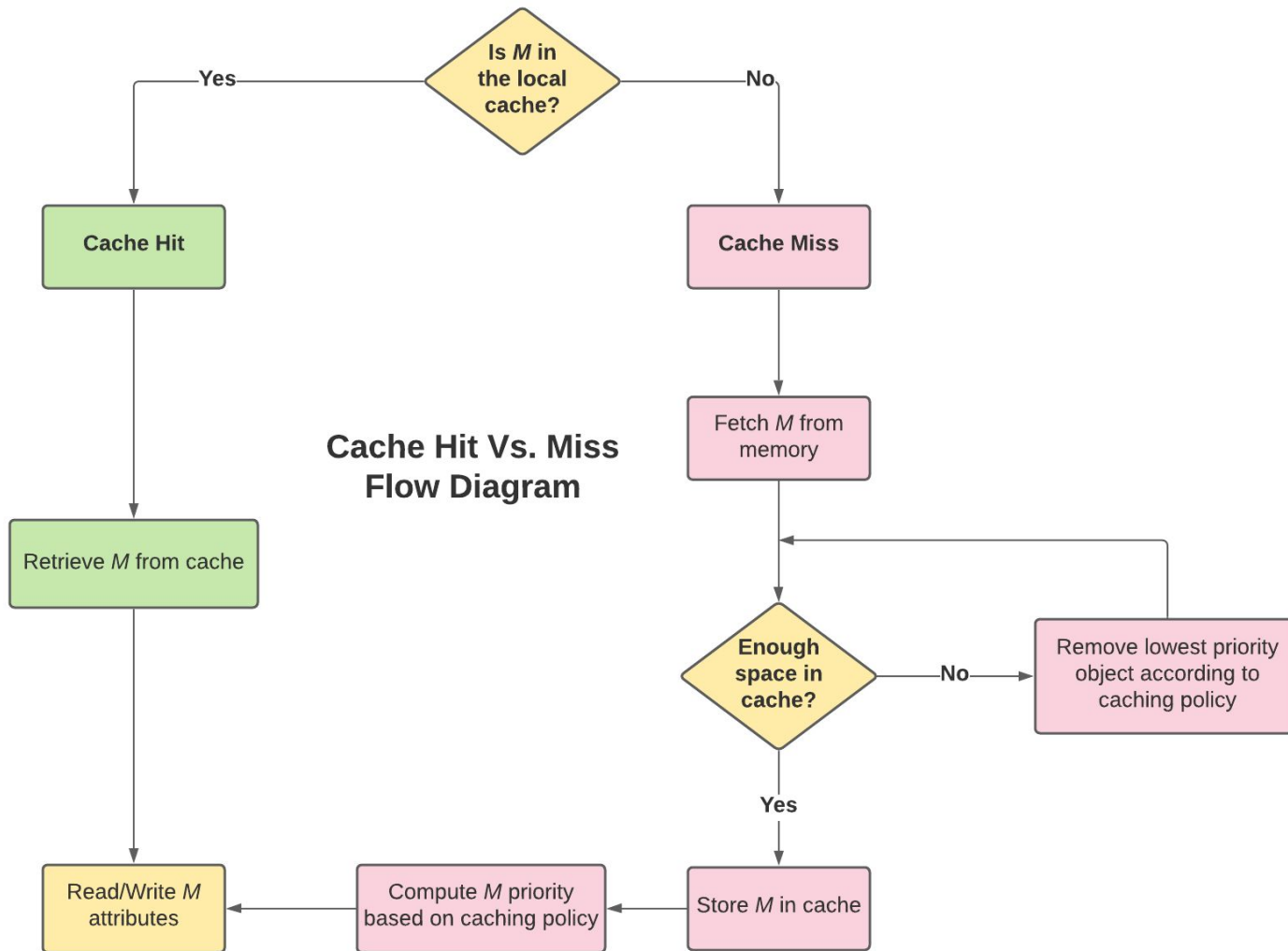
Introducing “caching”

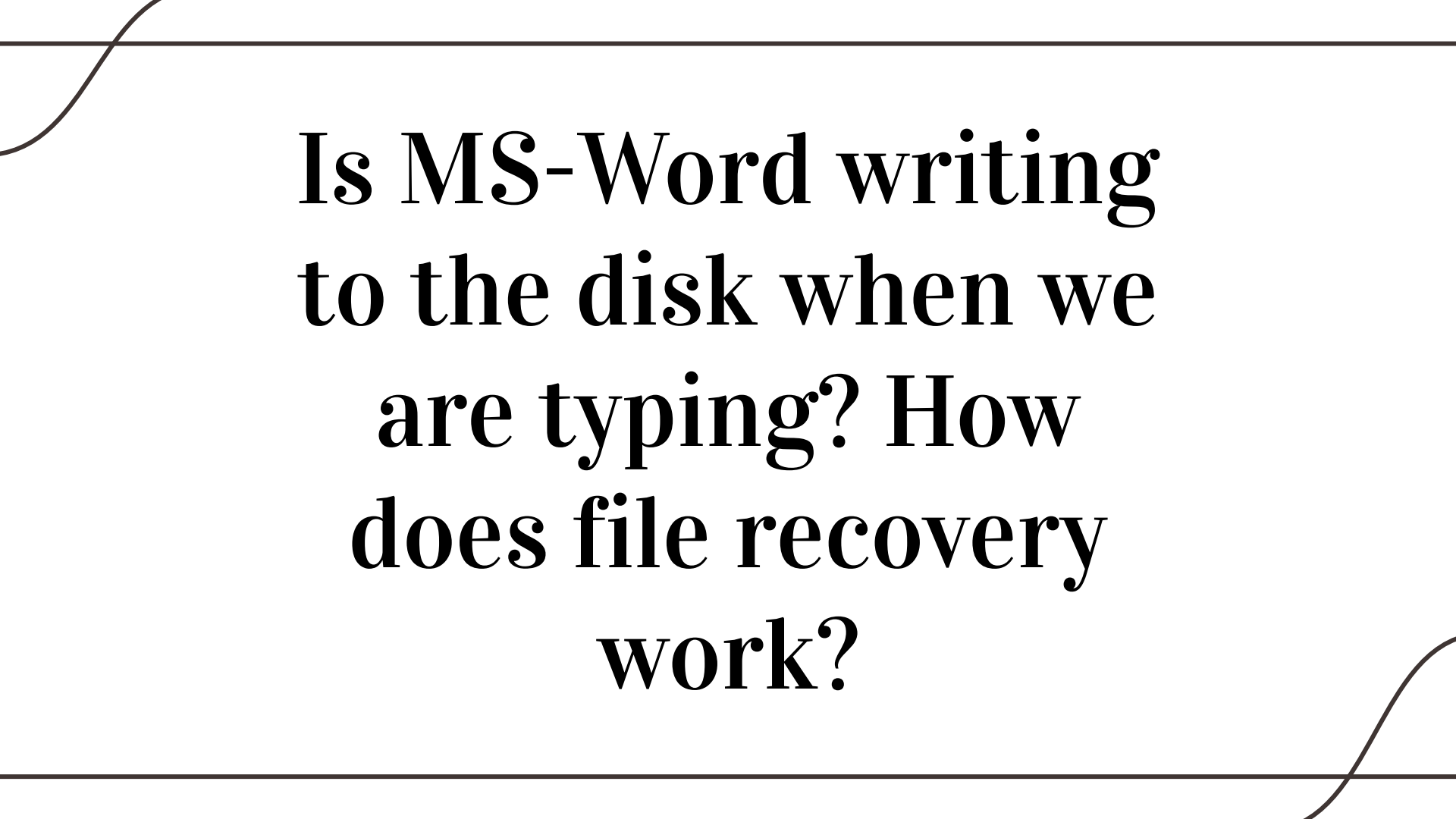
- To increase access speed to certain parts of repeatedly used memory, move it to the upper level of the memory hierarchy i.e. **size vs. access speed trade-off** (tap water vs. river water).
 - Technically speaking, **all memory levels in the upper tier act as caches for the lower tier** memory units in the hierarchy.
 - Caching is both software (web browsers) and hardware (SRAM, TLB) based.
 - **SRAM caches** are memory units especially dedicated to caching.
-



Cache Hits vs. Misses

- In simplest terms,
CPU finds the required data in cache - **Cache Hit**
CPU doesn't find the required data in cache - **Cache Miss**
 - **Cache Hit Ratio** = $(\text{Cache Hits}) / (\text{Cache Hits} + \text{Cache Misses})$
 - Higher the cache hit ratio, faster the memory access since the CPU doesn't have to "go looking" for data in the lower memory levels.
-






Is MS-Word writing
to the disk when we
are typing? How
does file recovery
work?




Virtual Memory

No, it's not VR.



**Can you run a 4GB
program on a
machine having 1GB
physical memory?**

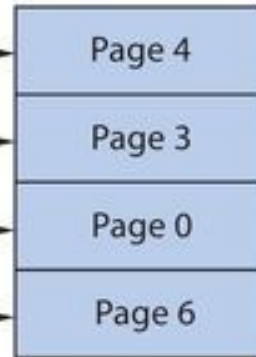
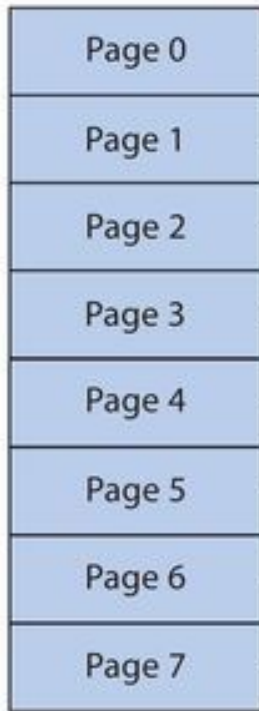


What is virtual memory?

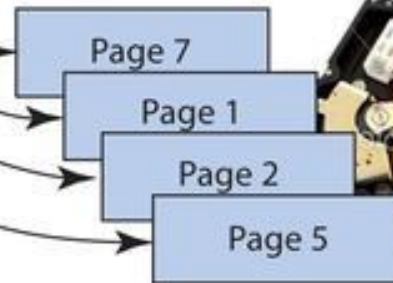
- Simply put, it is an **abstraction** of the available storage resources on a machine (anything below the physical/main memory in the hierarchy).
 - Nearly all virtual memory implementations today are based on **paging** i.e. virtual address space is **divided into fixed-size pages of contiguous memory**.
 - **Three important tasks** of virtual memory,
 1. Use main memory as cache for disk i.e. efficient memory management
 2. Provide each process with a uniform address space
 3. Protection of address space of one process from being corrupted by another
-

**"Virtual" memory
seen by the program**

Main memory



Disk storage



contains
"page
frames"

pages that
"fit into"
page
frames

This mapping is done by hardware
in the central processor, based on
tables in main memory

Page Table and TLB

- **Page table** contains entries (PTE) that contain information about which pages are already cached in the main memory.
 - **Translation Lookaside Buffer (TLB)** simply put is an on-chip cache for page table that contains PTEs that are most frequently/recently accessed.
 - Called “lookaside” as CPU can look for the PTE through the page table and the TLB simultaneously.
-



Writing Cache-friendly Code

Yay, finally some code...



Cache Pollution

```
#define N 100
#define CACHE_SIZE 4096 // size of our L1 cache
                        // in between 8 KB to 64 KB practically

int baka[CACHE_SIZE];
int mitai[CACHE_SIZE];

// ...initializations

void cache_polluting_function(){
    baka[0] = baka[0] + 1;
    for (int i = 0; i < CACHE_SIZE; i++){
        mitai[i] = mitai[i] + 1;
    }
    baka[0] = baka[0] + mitai[CACHE_SIZE - 1];
}

void cache_smart_function(){
    for (int i = 0; i < CACHE_SIZE; i++){
        mitai[i] = mitai[i] + 1;
    }
    baka[0] = baka[0] + 1;
    baka[0] = baka[0] + mitai[CACHE_SIZE - 1];
}
```

Locality of Reference

- **Temporal Locality:** Recently referenced locations are **more likely to be referenced in the near future** (eg - loop variables)
 - **Spatial Locality:** If a particular storage location is referenced at a particular time, then it is likely that **nearby memory locations will be referenced in the near future** (larger blocks of cache, eg - consecutive array elements). **Case study** - linked list.
 - Rule of thumb: Both types of locality must be leveraged for better performance.
-

Structure of Arrays vs. Array of Structures

```
#define N 100
```

```
// Structure of Arrays (SOA)
```

```
struct point_3D_SOA {  
    float x[100];  
    float y[100];  
    float z[100];  
};  
struct point_3D_SOA points;  
  
// ...initialization  
  
float offset_x(int offset) {  
    for (int i = 0; i < N; i++)  
        points.x[i] += offset;  
}
```

```
#define N 100
```

```
// Array of Structures (AOS)
```

```
struct point_3D_AOS {  
    float x;  
    float y;  
    float z;  
};  
struct point_3D_AOS points[N];  
  
// ...initialization  
  
float offset_x(int offset) {  
    for (int i = 0; i < N; i++) {  
        float sum_of_coordinates = points[i].x + points[i].y +  
                                     points[i].z;  
        // ...further calculations  
    }  
}
```

Re-visiting Matrix Multiplication

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} \times b_{11} + a_{12} \times b_{21} & a_{11} \times b_{12} + a_{12} \times b_{22} \\ a_{21} \times b_{11} + a_{22} \times b_{21} & a_{21} \times b_{12} + a_{22} \times b_{22} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

$$\begin{bmatrix} 1 \times 5 + 2 \times 7 & 1 \times 6 + 2 \times 8 \\ 3 \times 5 + 4 \times 7 & 3 \times 6 + 4 \times 8 \end{bmatrix} = \begin{bmatrix} 5 + 14 & 6 + 16 \\ 15 + 28 & 18 + 32 \end{bmatrix}$$

$$= \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Which loop order is the more cache-friendly?

```
void matrix_multiply_ijk_order(int **mat_A, int **mat_B, int **mat_res, int N) {  
    for (int i = 0; i < N; ++i) {  
        for (int j = 0; j < N; ++j) {  
            for (int k = 0; k < N; ++k) {  
                mat_res[i][j] += mat_A[i][k] * mat_B[k][j];  
            }  
        }  
    }  
}
```

```
void matrix_multiply_jki_order(int **mat_A, int **mat_B, int **mat_res, int N) {  
    for (int j = 0; j < N; ++j) {  
        for (int k = 0; k < N; ++k) {  
            for (int i = 0; i < N; ++i) {  
                mat_res[i][j] += mat_A[i][k] * mat_B[k][j];  
            }  
        }  
    }  
}
```

Cache-friendly Matrix Multiplication

Some assumptions we make,

- Both matrix A and matrix B to be multiplied are $N \times N$ matrices, where 'N' is an integer i.e. **we only multiply square matrices**.
- For each code run, the matrices to be multiplied will have the same elements in the same order i.e. **the seed to generate random matrices is constant**, with **upper random number limit being one less than a power of 2**.
- Matrices will be **allocated on the heap** since they are large.
- We measure **time taken for function call to return** as a measure of optimized code.
- The **code only runs on Linux** (bow down to Linux supremacy!).

The image features two thin, dark horizontal lines. The top line starts with a curved, wavy segment on the left side before becoming straight. The bottom line ends with a similar curved, wavy segment on the right side.

Demo



**Back to making
computers fast...**

Making Computer Fast?

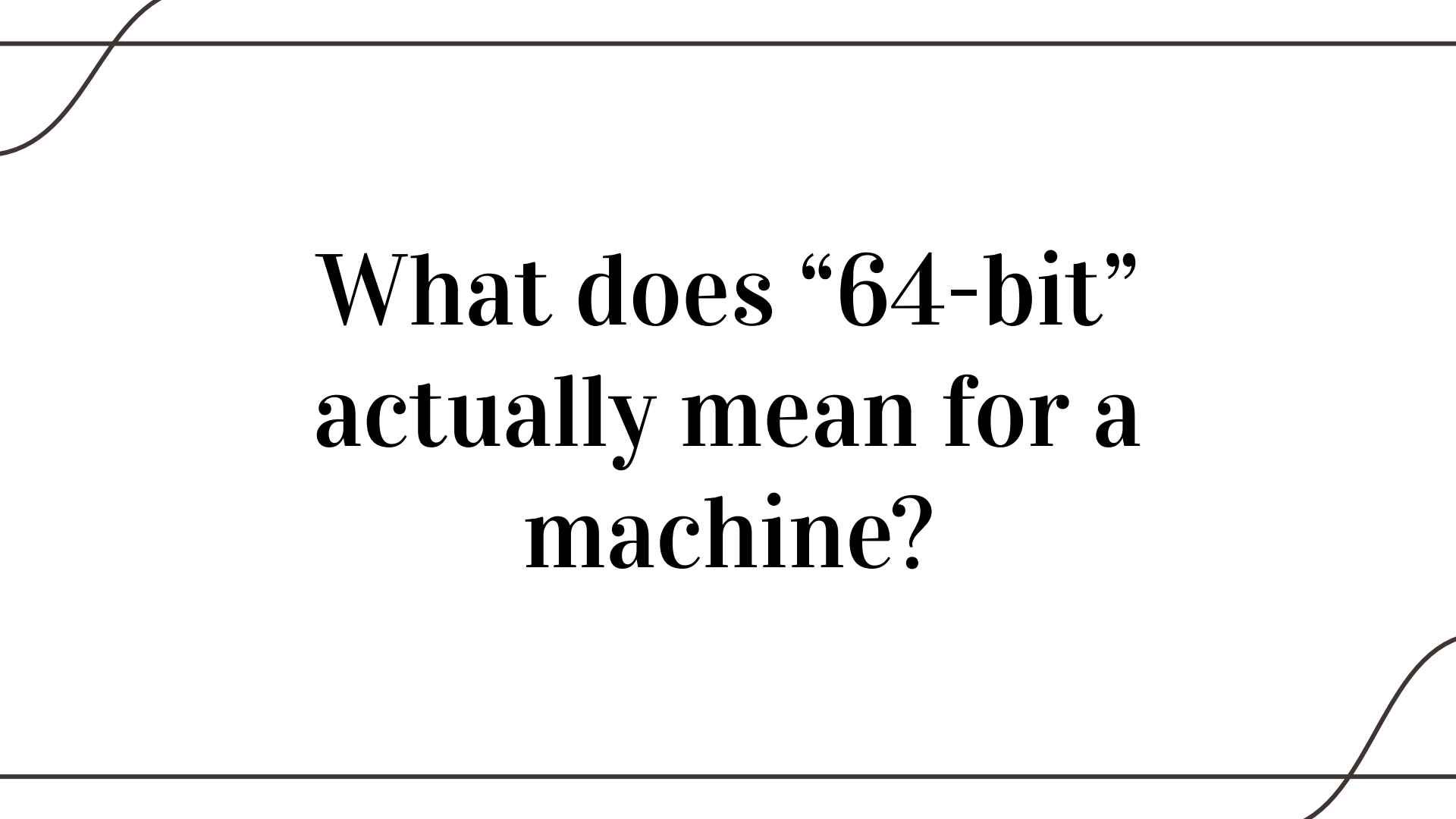
- Increasing the speed of clock cycle?
- Parallelism

Parallel Computing

- Parallel computing is a type of computation in which many calculations or processes are carried out simultaneously.

Three types:

- Bit Level Parallelism (n-bit processors)
 - Instruction Level Parallelism (Pipelining)
 - Data Level Parallelism (GPUs)
 - Task Level Parallelism (Multiprocessing)
-



**What does “64-bit”
actually mean for a
machine?**

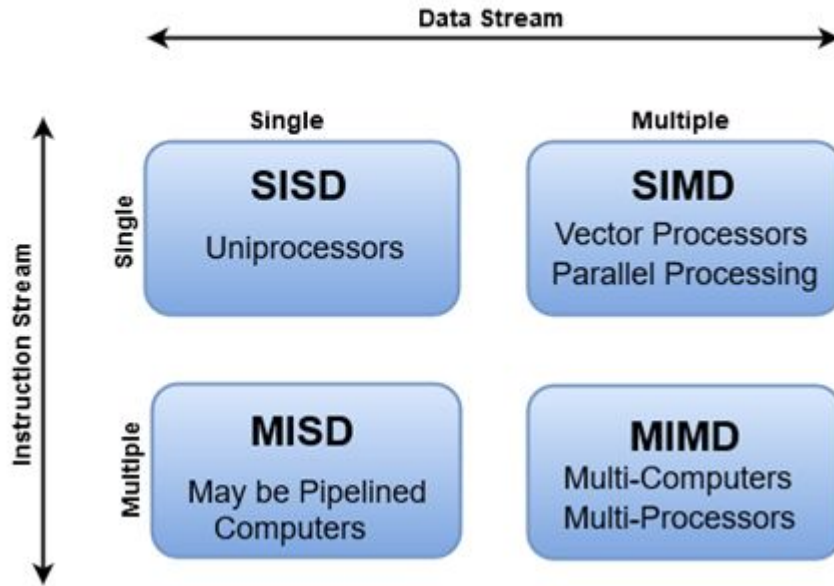
Flynn's Classification

- Two terms: **Instruction Stream** and **Data Stream**
- Instruction Stream: The sequence of instructions read from memory.
- Data Stream: The operations performed on the data in the processor.

Classified as:

Flynn's Classification

Flynn's Classification of Computers



Courtesy: JavatPoint

Multiprocessors

A multiprocessor is a computer system having two or more processing units (multiple processors) each sharing main memory and peripherals, in order to simultaneously process programs.

Classified as Multiple Instruction Stream, Multiple Data Stream

Mostly tightly coupled system

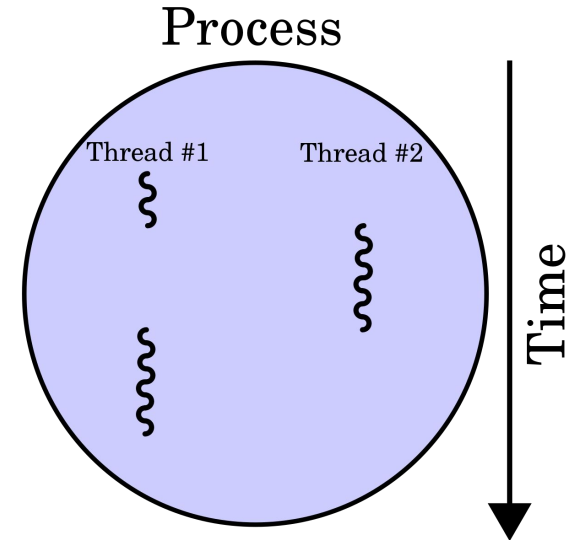
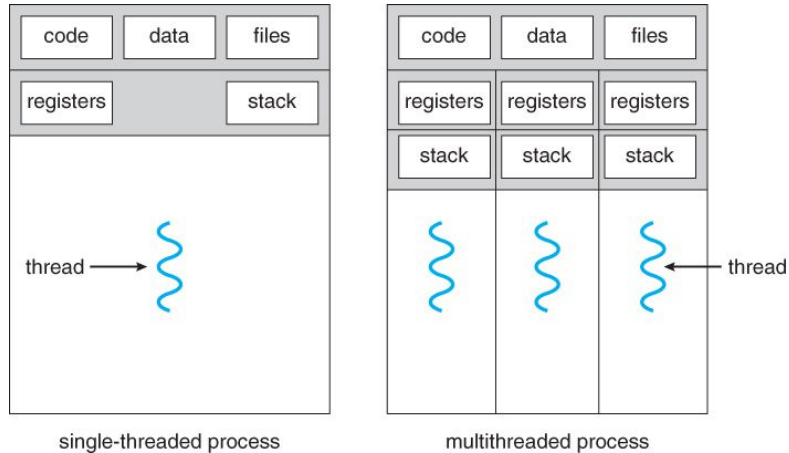
Multiprocessing

From a programmer's or operating system point of view, Multiprocessing is execution of multiple concurrent processes in a system.

Where each process running on a separate CPU or core, as opposed to a single process at any one instant.

Threads

- A basic unit of CPU utilization.
- Includes Program Counter, Register Set, Stack.
- Examples:
 - o Server accepting different requests.



Types of threads

- Hardware Threads (HyperThreading)[Symmetrical Multi-threads]
- Logical Threads (Threads managed by Operating System)



Why Threads over processes?



Context Switching

In computing, a context switch is the process of storing the **state of a process or thread**, so that it can be restored and resume execution at a later point.

This process is performed by an operating system.

Advantages of threads over process

1. Context switches between threads are faster than between processes. That is, it's quicker for the OS to stop one thread and start running another than do the same with two processes.
2. Inter-thread communication (sharing data etc.) is significantly simpler to program than inter-process communication.

Things you can see for additional knowledge

- [8085 Simulator](#) - For Observing Registers In Action on a simpler CPU
 - [Branchless Programming](#) - Some More Exploration Towards Optimization
 - [Godbolt Online Compiler](#) - For C to Assembly Inspection
 - [MIT 6.172 - Performance Engineering of Software Systems](#) - First lecture has more explanation on the matrix multiplication optimization
 - [Cachegrind - Valgrind Tool](#) - To simulate cache hits and misses for your program
 - Additionally you may search for topics such as clock synchronization in multiprocessor systems, cache lines, virtual memory and paging and more. At your own risk, fellow journeyman.
-



**All
The Best !!**