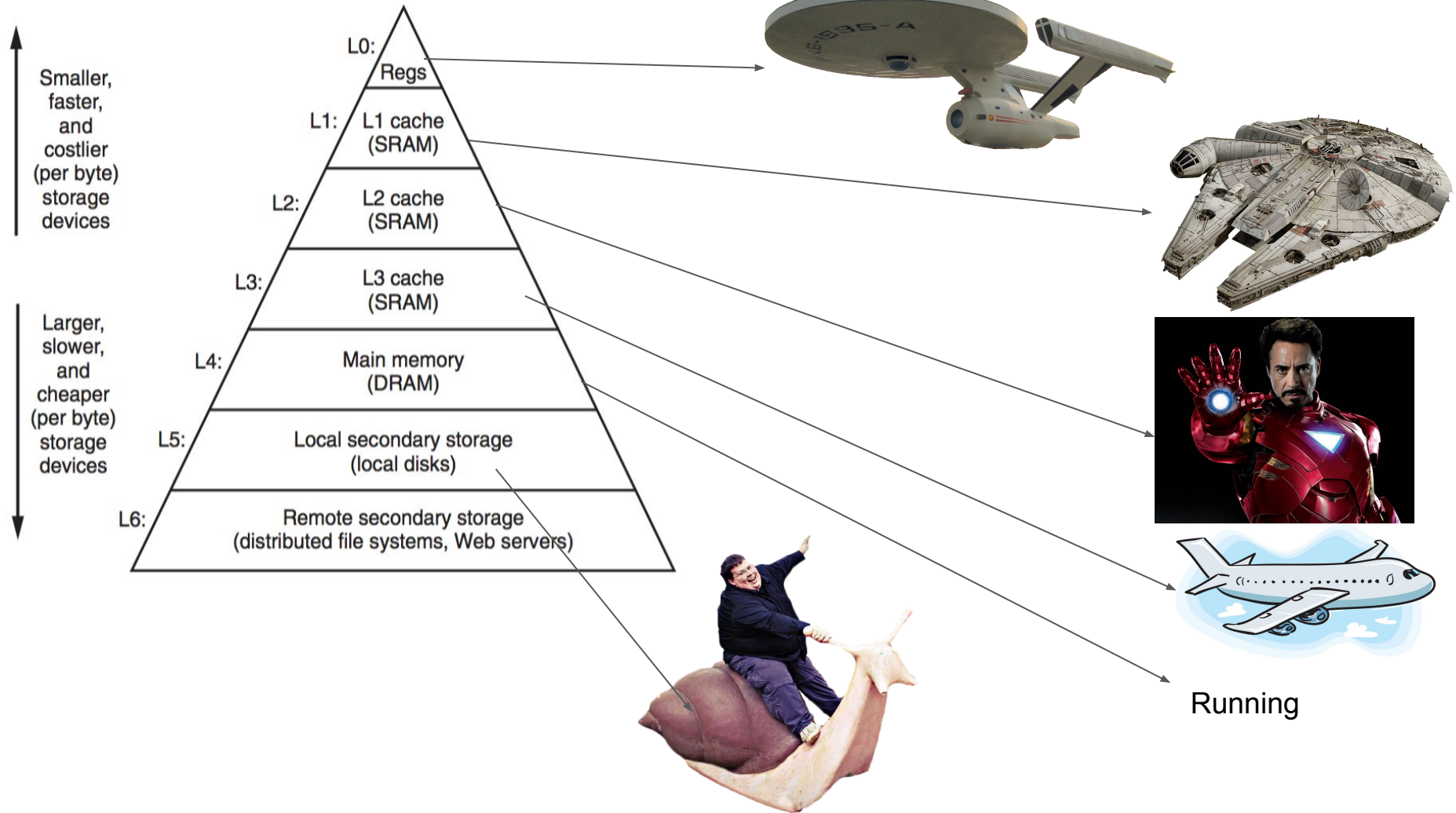
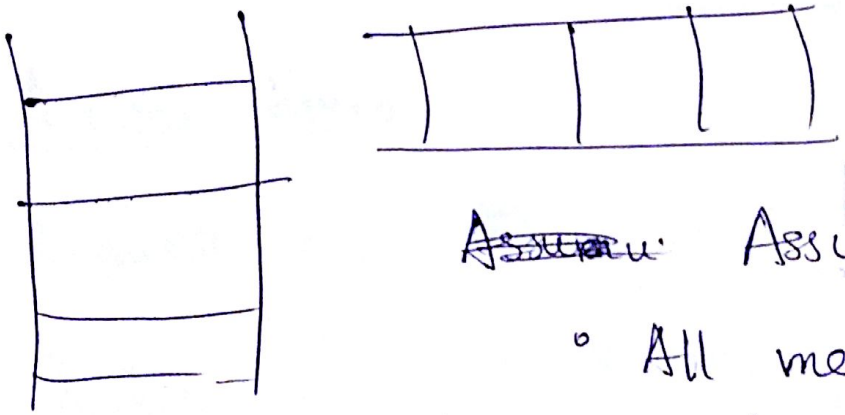

Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
L3 cache (SRAM)
L4:
Main memory (DRAM)
L5:
Local secondary storage (local disks)
L6:
Remote secondary storage (distributed file systems, Web servers)
Running

TODAY → Memory (Chapter 6).

- Locality
- Hierarchy
- Cache

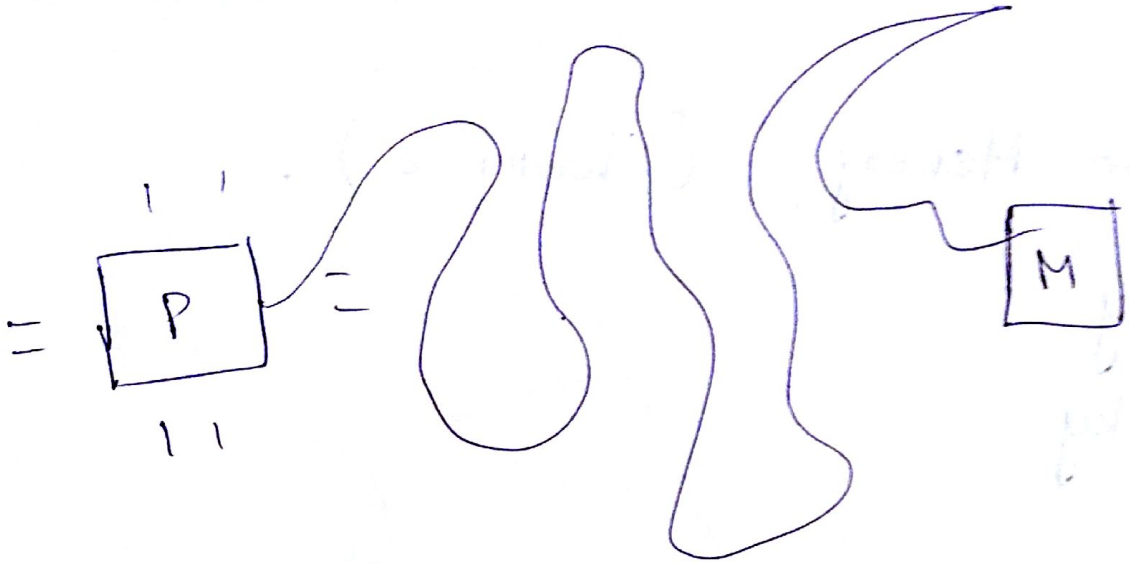
So far ⇒ simplistic memory



~~Assum~~ Assumption

- All memory accesses happend in a constant amount of time.

movl 4(%ebp), %eax



Physically - seperate

Faster !

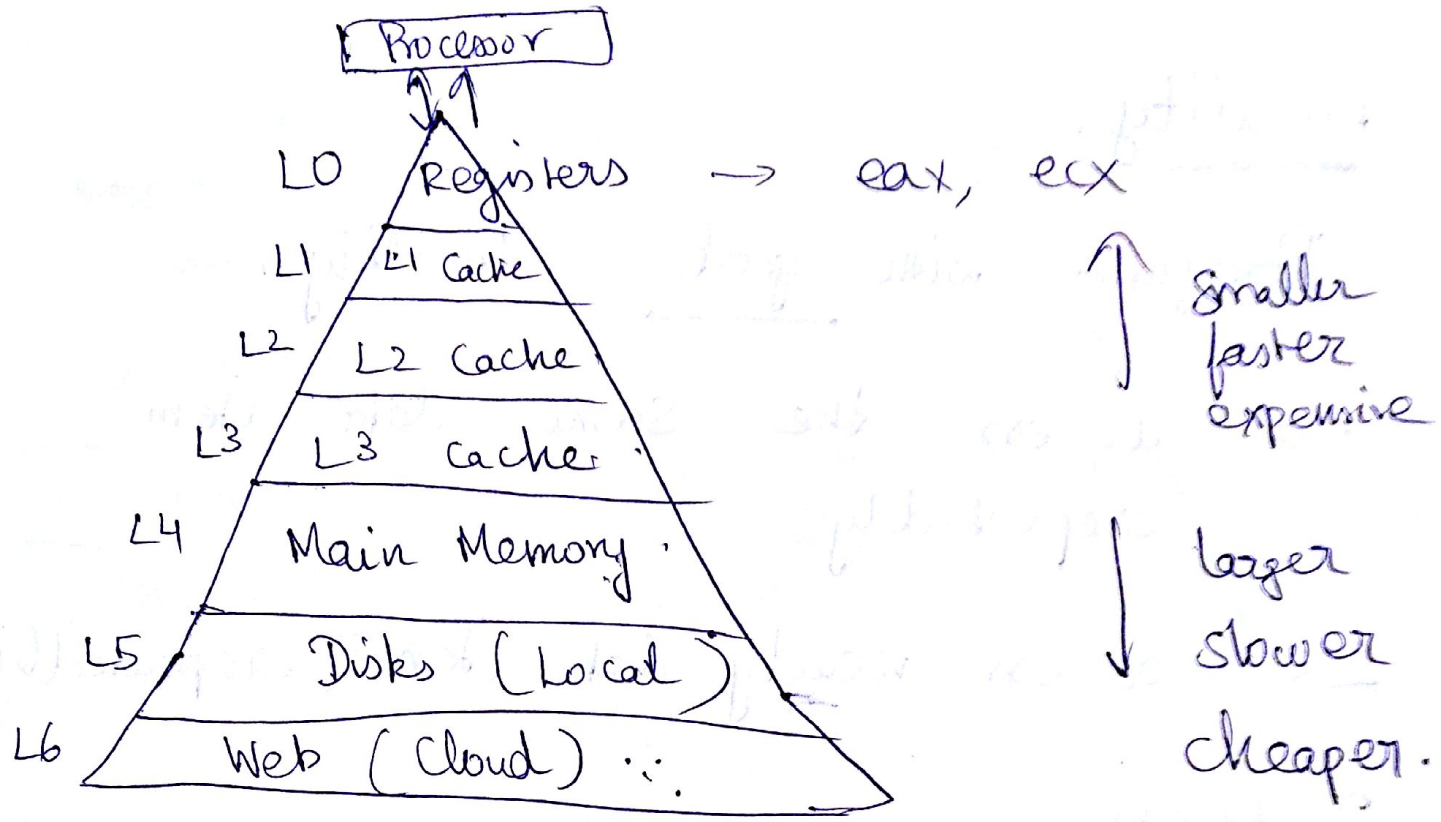
Drawback

→ Small ✓

→ expensive ✓

We have different levels of memory

We dont want extremities !



Access times

Register ~ 0 cycles.

Cache ~ 1 to 30 cycles.

Main Memory ~ 20 ~ 200 cycles.

Disk ~ 10's of millions of cycles.

The closer your data gets to the processor, the faster your program is going to execute.

Locality.

Programs with good locality.

→ access the same data item repeatedly

→ access nearby data items repeatedly.

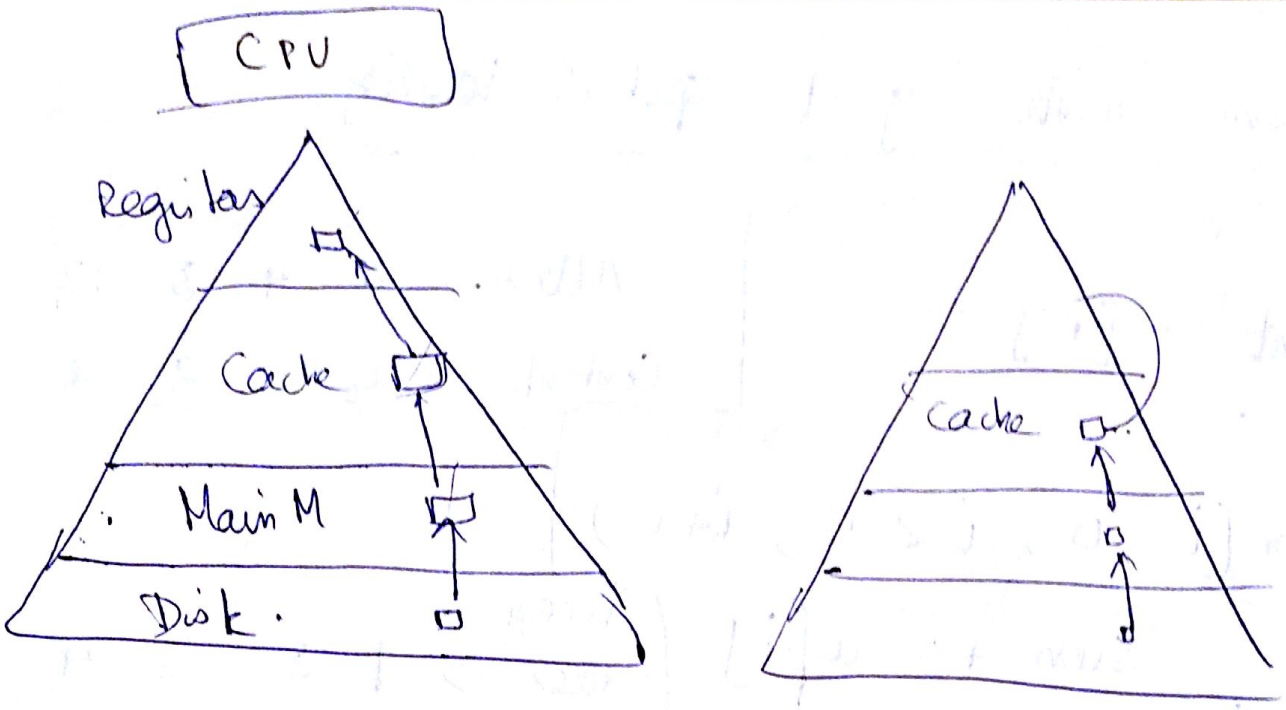
2 forms

① Temporal Locality

→ a memory location that is referenced once is likely to be referenced again in the near future.

② Spatial Locality

→ a program is likely to reference a memory location that is near a previously referenced location.



good temporal locality.

Real world example → Browsers!

Twitter → Twitter logo.

↓

Store this locally on some "cache" (Disk)

Instead of fetching from the web every time.

Program with good spatial locality

```
{
    int a[5];
    :
    for(i = 0; i < 5; i++)
        sum += a[i]
    :
}
```

Address →	0	4	8	12	16
Contents →	a_0	a_1	a_2	a_3	a_4
Access order →	1	2	3	4	5

i = 0
i = 1
i = 2

```
{
    int a[2][3];
    for (i=0; i < 2; i++)
        for (j=0; j < 3; j++)
            sum += a[i][j]
}
```

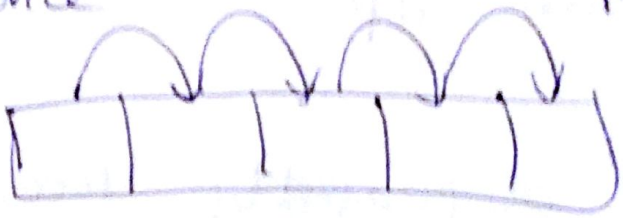
i = 0 j = 0
i = 0 j = 1
i = 0 j = 2

Addresses →	0	4	8	12	16	20
Contents →	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order →	1	2	3	4	5	6

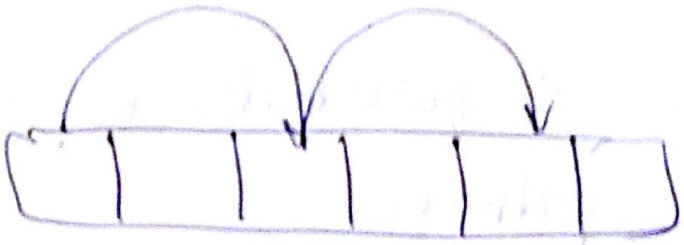
Sequential Reference Pattern

stride - 1 - reference pattern

=> Sequential reference pattern.

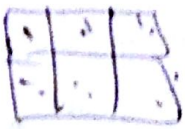


stride - 2 - reference pattern



stride - k - reference pattern.

lower k is better for spatial locality.



Bad spatial locality

```
for (j=0; j<3; j++)
  for (i=0; i<2; i++)
    sum += a[i][j]
```

Address →	0	4	8	12	16	20
content	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order →	1	3	5	2	4	6

j=0 i=0
j=0 i=1
j=1 i=0
j=1 i=1

Sum up

→ repeatedly access the same variables ⇒ good temporal locality

→ sequential reference pattern (stride-1-pattern) ⇒ good spatial locality.

CACHING

Generally a cache is a smaller, faster storage device that acts as a **staging area** for data stored in a larger, slower device.

The process of using a cache ⇒ caching.

Cache Hit

Say a program needs data object d from level k+1.

First → look in level k //

If d happens to be cached at level k itself, we have a cache hit. and we get d from k.

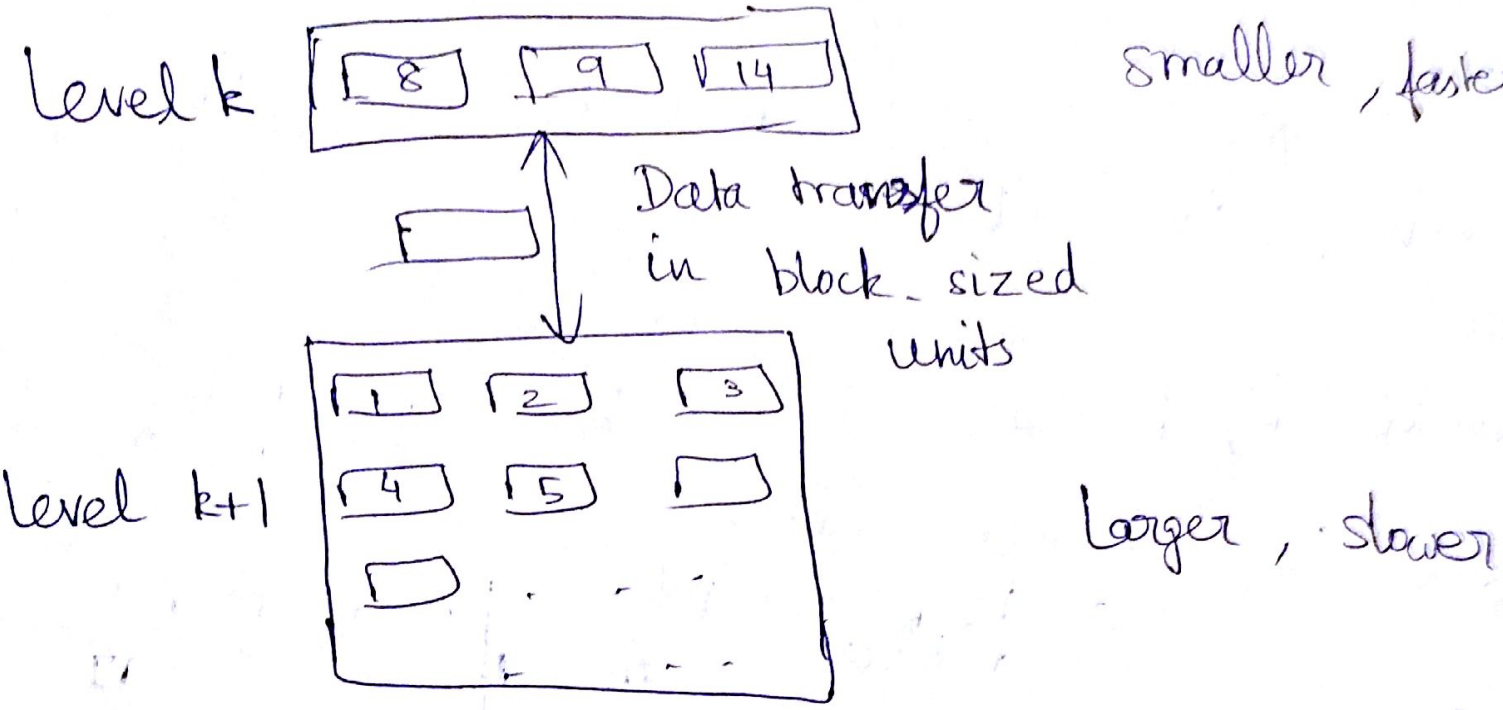
The more cache hits, the merrier!

Why? Accessing data from level k is faster than accessing from level k+1

Cache miss.

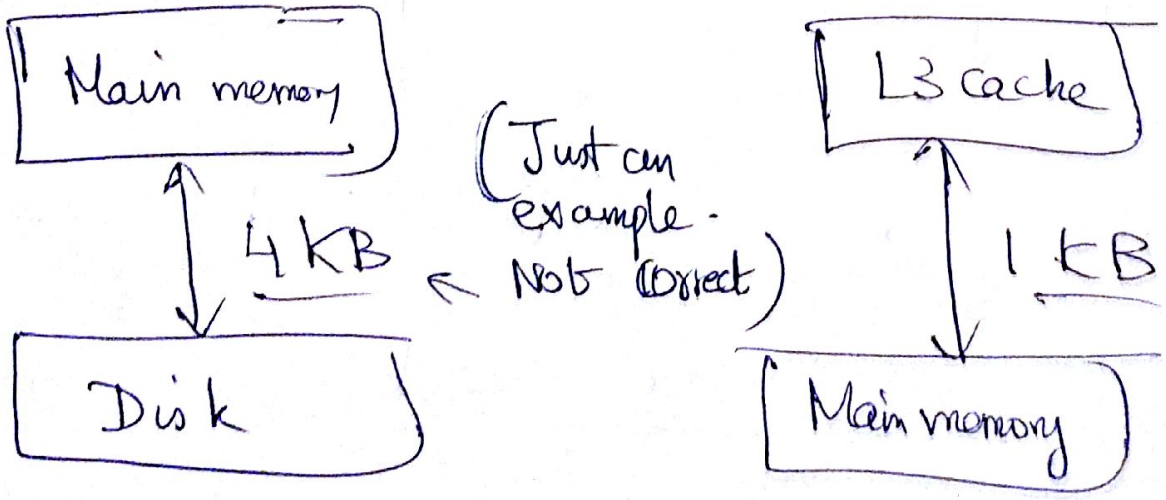
What if d is not in level k?

We have a cache miss!



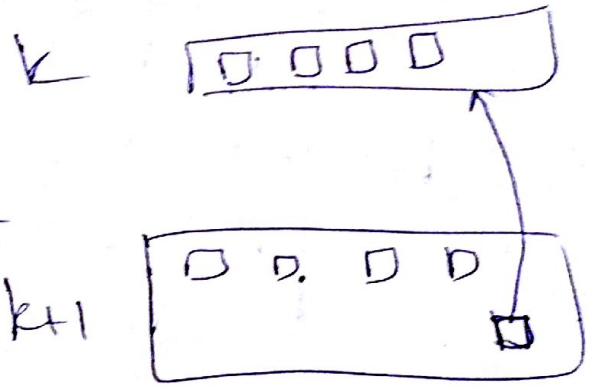
Block size is fixed (but only between two levels).

It can be different for other level pairs



Then → level k fetches d from level k+1. and it will store it.

What if k is full ? Remove some old block and put d there



Cache Replacement Policy

- Random
- Least Recently Used Policy.

Block to be replaced → victim block.

Q. Arrays Vs Linked Lists.

suppose → linear search.

Good spatial locality ?