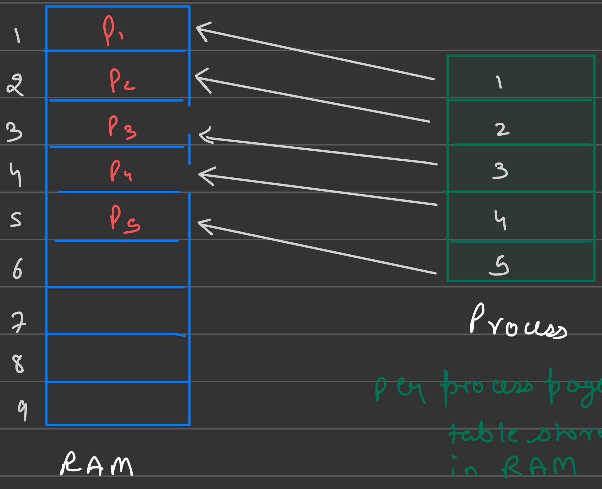
Virtual Memory And Segmentation

Virtual Memory

RAM is split into fixed size partitions called as Page frames.

Page frames in old processors are of typically 4KBs.



per process page table stored in RAM

Process also splits into blocks of equal size where block size == page frame

Block	page frame
1	1
2	2
3	3
4	4
5	5

Then the process blocks is allocated to frames

Because of the per process page table, blocks of process need not to be in contiguous frames. The page frame can be identified by page table.

So every memory access has an additional overhead of the lookup in page table. This canbe optimised using a TLB (Translational lookaside buffer) cache.

Every executing process will be having it's own process-page table.

★ **NOTE** Memory associated with the process is in the user region of memory whereas the process page table is in the kernel region.
Depending on the active process, the active page table will vary. So process can't access page frames of other process.

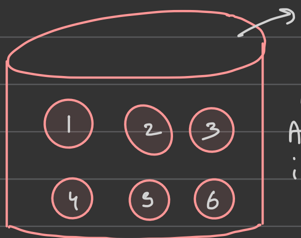
Q⇒ Do we really need to load all blocks into memory before the process starts executing ?

NO.

Not all parts of program are accessed simultaneously. Infact some code may never be executed.

So virtual memory takes advantage of this by using a concept called Demand Paging.

# Demand Paging



Swap space on disk

In our secondary storage a space is allocated (swap space). All blocks of executable is present. On demand blocks will be loaded from this space to RAM

present bit ↓

block	page frame	p
1	14	1
2		0
3		0
4		0
5	8	1

process page table in RAM

- ⭑ Pages are loaded from disk to RAM, only when needed.
- ⭑ A present bit in table represents if block is in RAM or not.
- ⭑ if (present ==1) { block in RAM} (else) { block not in RAM}

⇒ Scenario → Let's say block 3 of the executing process wants to access block 5 but block 3 is not loaded in the RAM, then when it will check the page table the present bit will be zero.
If a page/block is accessed that is not present in RAM the processor issues a page fault interrupt. This triggers the OS to load the page into the RAM and mark the present bit to 1.

# Page replacement policies

Now let's say OS wants to load a block into a page frame & no pages are free for a new block to be loaded then the OS makes a decision to remove another block from RAM.

This is based on the replacement policy implemented in the OS.

⭑ Note → Page tables for the processes under consideration are updated accordingly during page replacement.

Some replacement policies are -

- → first In first Out
- → Least Recently used
- → Least frequently used

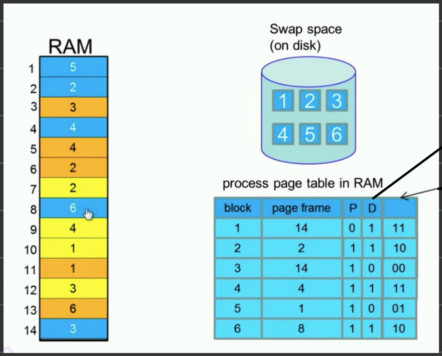
Based on the policy implemented the OS swaps out a block from memory with a block to be added in RAM. Present bits are changed accordingly.

Process of Loading a block in RAM is called Swap IN
Process of pulling out a block from RAM is called Swap Out

* Swap Out Process → During the swap out process the changes in content of block to be swapped from RAM is copied to the disk so the disk has latest piece of changes. But if no changes has been done the no need to do a copy. To maintain this a Dirty bit is used

* Protection Bits → Shows if a block is executable or not. They also determine if it is a operating System code or a regular user code.

completely filled pages frames

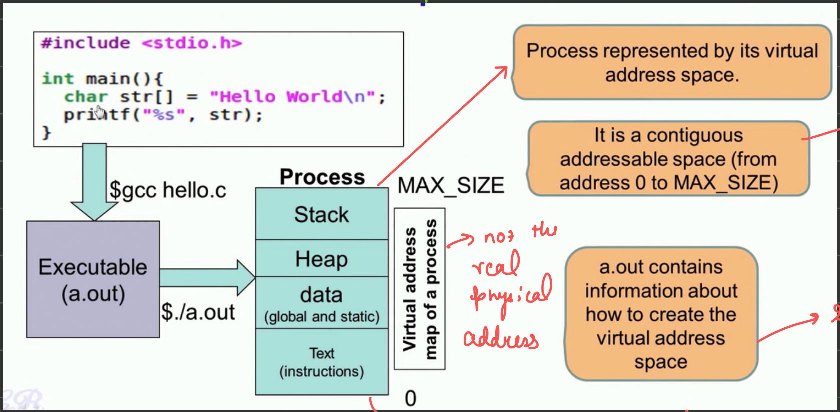


process page table in RAM

block	page frame	P	D	
1	14	0	1	11
2	2	1	1	10
3	14	1	0	00
4	4	1	1	11
5	1	1	0	01
6	8	1	1	10

dirty bit
protection bits

Virtual Address Space Of A Process



```
#include <stdio.h>

int main(){
   char str[] = "Hello World\n";
   printf("%s", str);
}
```

Process represented by its virtual address space.

It is a contiguous addressable space (from address 0 to MAX_SIZE)

a.out contains information about how to create the virtual address space

this is continuous but physical addressy is not

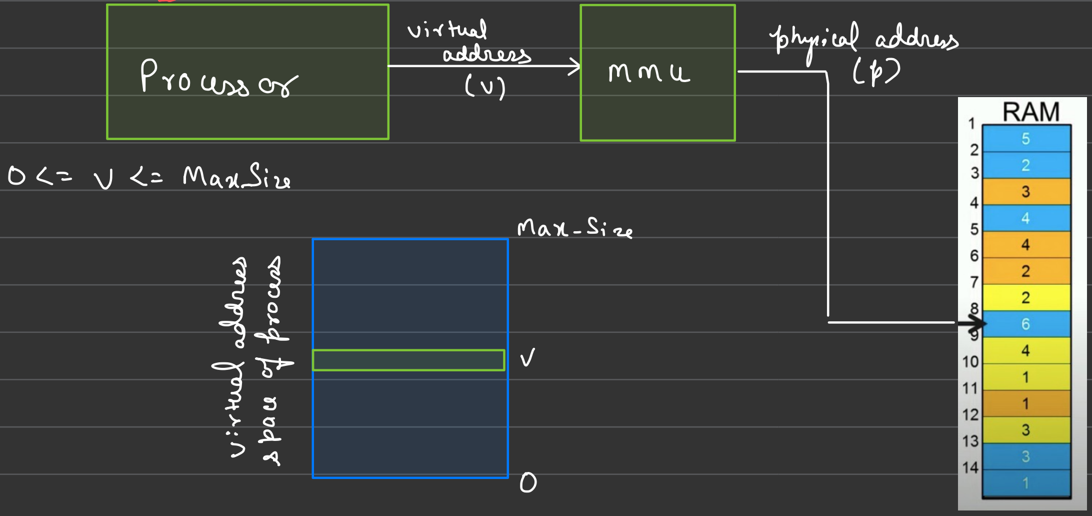
not the real physical address

so that we can extract the physical address

this is later divided into blocks

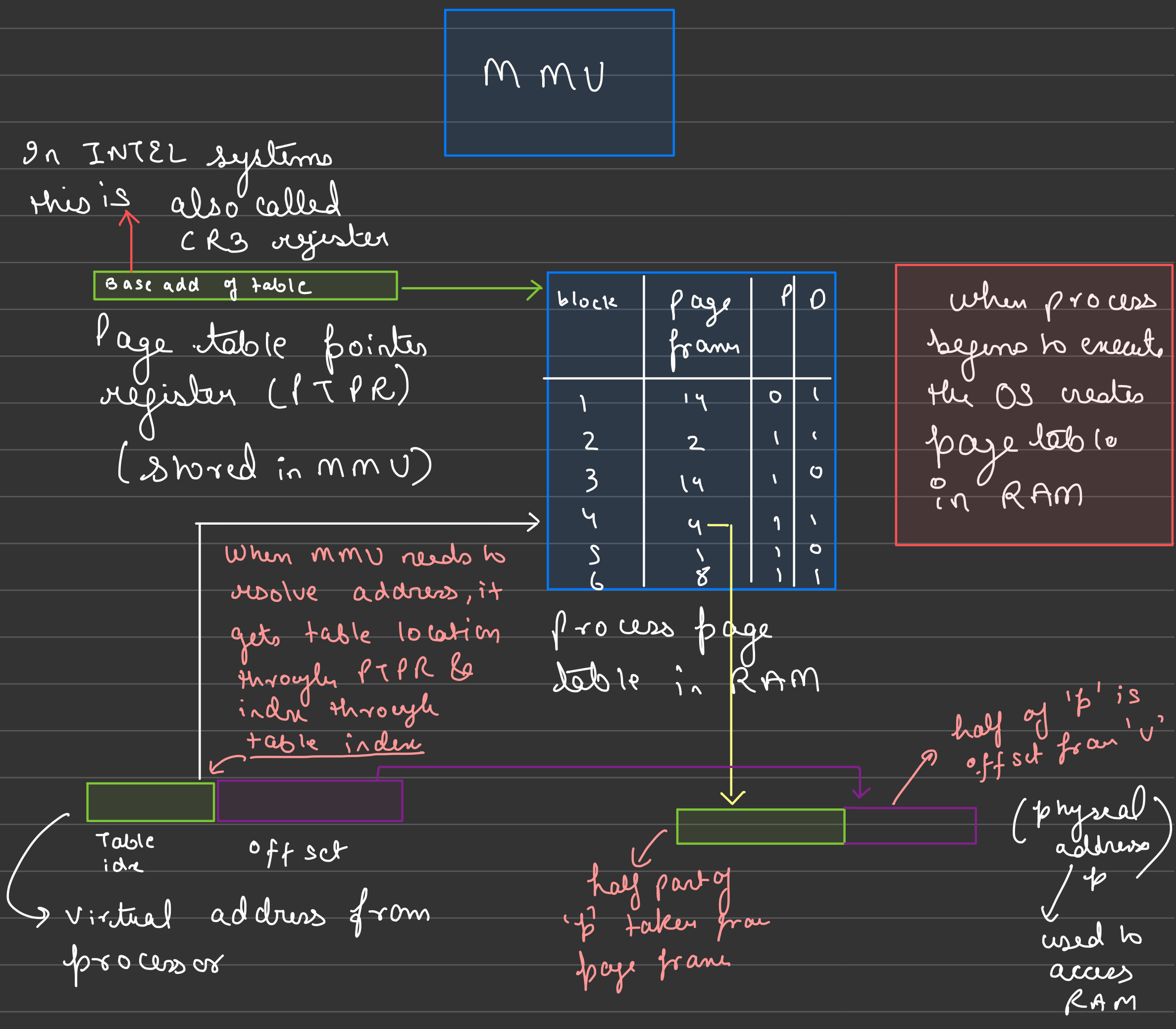
To access process memory the processor generates the corresponding virtual address

MMU maps the virtual address to the corresponding physical address



0 <= v <= Max_Size

MMU mapping



In INTEL systems this is also called CR3 register

Page table pointer register (PTPR)
(stored in MMU)

block	page frame	P	O
1	14	0	1
2	2	1	1
3	14	1	0
4	4	1	1
5	1	1	0
6	8	1	1

Process page table in RAM

when process begins to execute, the OS creates page table in RAM

When MMU needs to resolve address, it gets table location through PTPR & index through table index

virtual address from processor

half part of 'p' taken from page frame

half of 'p' is offset from 'v'

(physical address) p

used to access RAM

Virtual address is of 2 parts → table idx & offset

# MMU mapping for a 32 bit systems

Virtual address (v) is of 32 bits
The max process size is 2^{32} = 4GB

⟵ 32 bits ⟶

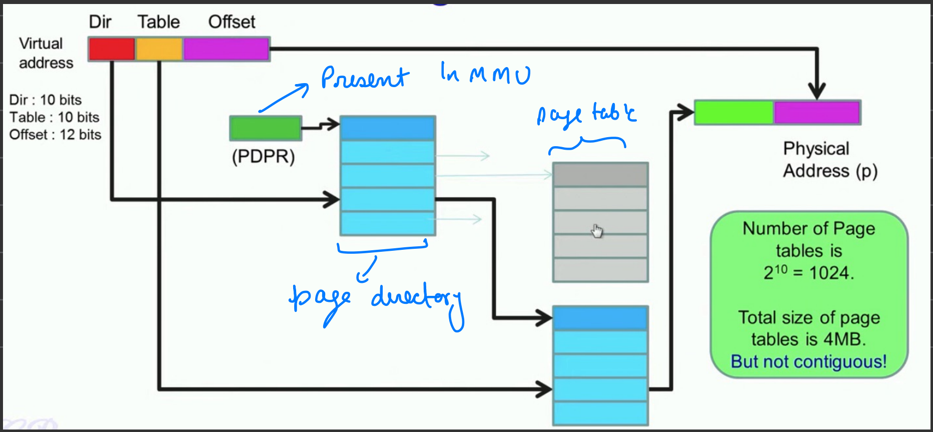
Table IDX	offset
⟵ 20 bits ⟶	⟵ 12 bits ⟶

if each page frame is of 4KB then 12 bits are reqd to address a page as $2^{12} = 4096$. Thus offset is of 12 bits

as table index is 20 bits then the total no. of entries in a page table is 2^{20} (around 4MB)
and this memory is reqd to be contiguous. Now a days 4MB is not that big but back then it was very large.

This is reqd to be contiguous bcoz table IDX is added directly to PTPR to get table index.

To avoid this some systems like Intel uses 2 level page translation.



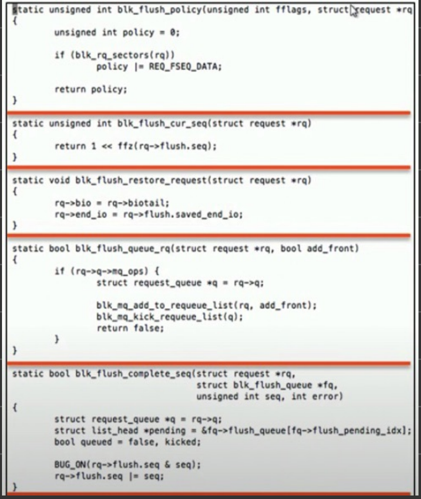
Number of Page tables is 2^{10} = 1024.

Total size of page tables is 4MB.
But not contiguous!

# Segmentation

Programs are collection of logical modules.
Logical modules : global data, stack, heap, functions, classes, namespaces etc.

Virtual memory does not split programs into logical modules, instead splits programs in fixed size blocks.



```
static unsigned int blk_flush_policy(unsigned int fflags, struct request *rq
{
        unsigned int policy = 0;

        if (blk_rq_sectors(rq))
                policy |= REQ_FSEQ_DATA;

        return policy;
}
```

```
static unsigned int blk_flush_cur_seq(struct request *rq)
{
        return 1 << ffz(rq->flush.seq);
}
```

```
static void blk_flush_restore_request(struct request *rq)
{
        rq->bio = rq->biotail;
        rq->end_io = rq->flush.saved_end_io;
}
```

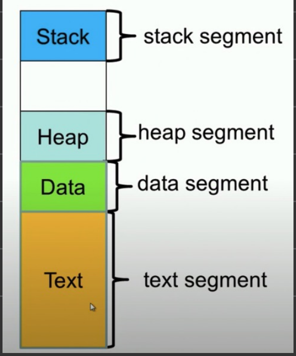
```
static bool blk_flush_queue_rq(struct request *rq, bool add_front)
{
        if (rq->q->mq_ops) {
                struct request_queue *q = rq->q;

                blk_mq_add_to_requeue_list(rq, add_front);
                blk_mq_kick_requeue_list(q);
                return false;
        }
}
```

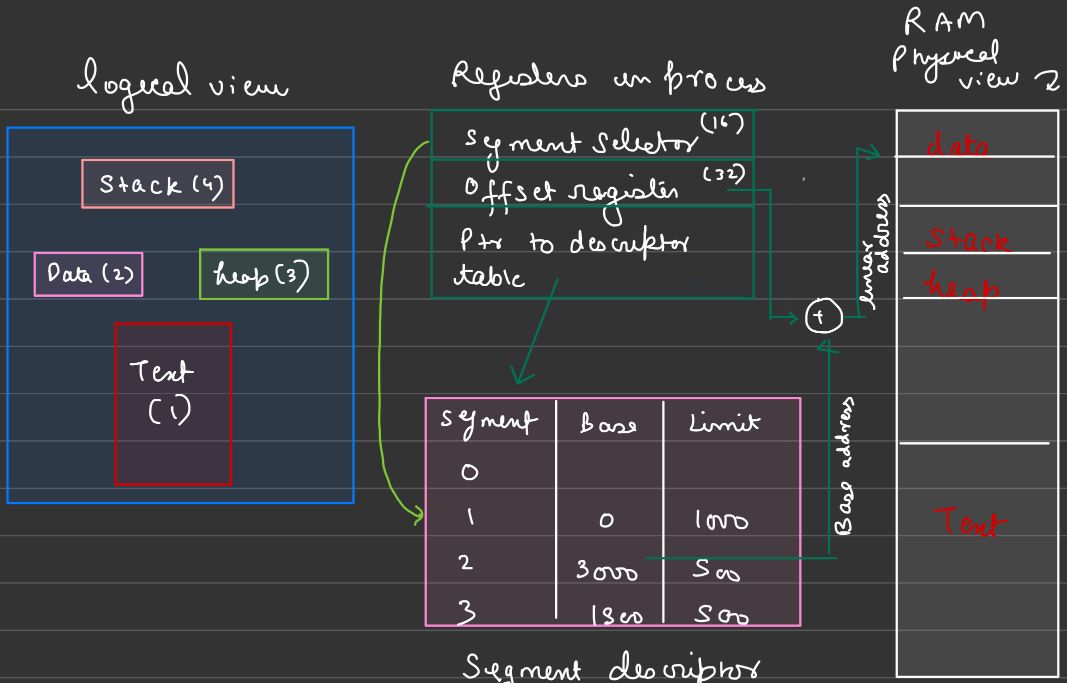
```
static bool blk_flush_complete_seq(struct request *rq,
                                   struct blk_flush_queue *fq,
                                   unsigned int seq, int error)
{
        struct request_queue *q = rq->q;
        struct list_head *pending = &fq->flush_queue[fq->flush_pending_idx];
        bool queued = false, kicked;

        BUG_ON(rq->flush.seq & seq);
        rq->flush.seq |= seq;
}
```

segmentation instead splits programs into segments that are more logical.

The segment size could range from a few bytes to max size (4GB in 32 bit Intel machines)

# commonly used segments

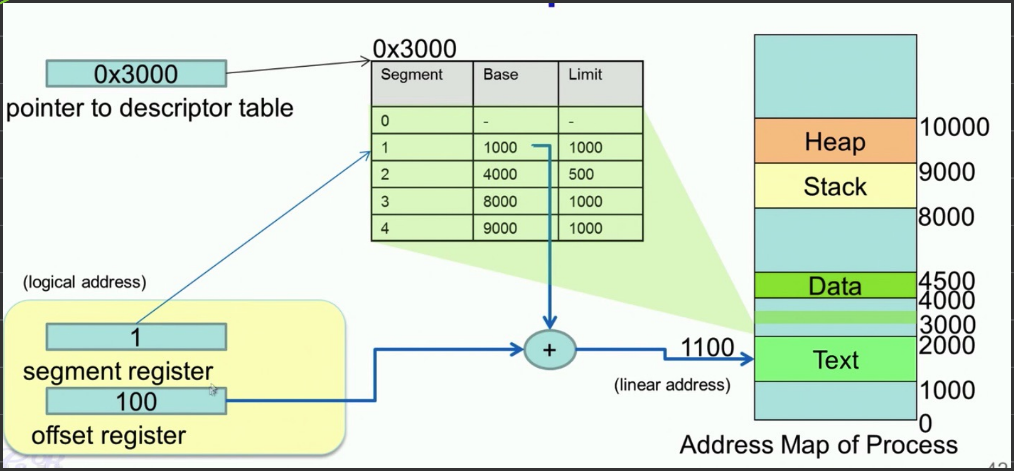




Segment	Base	Limit
0		
1	0	1000
2	3000	500
3	1500	500

Segment descriptor table (stored in memory)

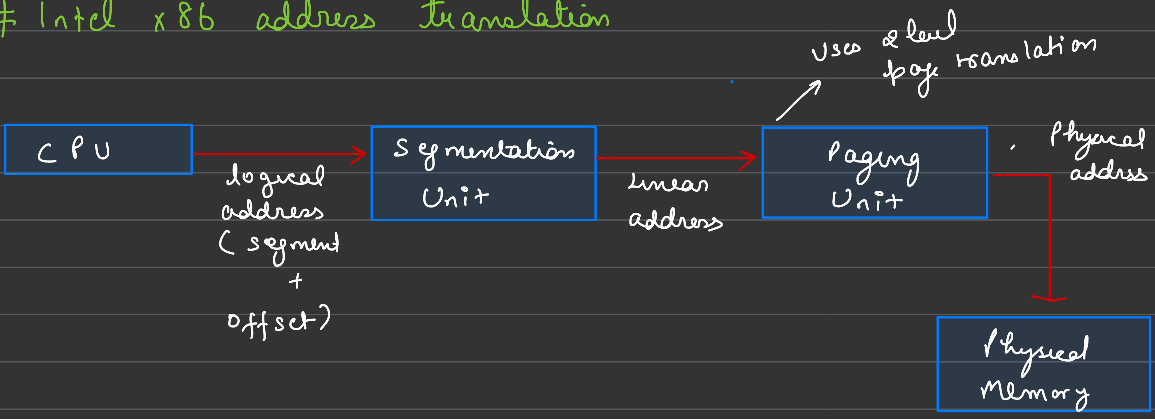
Example



Segment	Base	Limit
0	-	-
1	1000	1000
2	4000	500
3	8000	1000
4	9000	1000

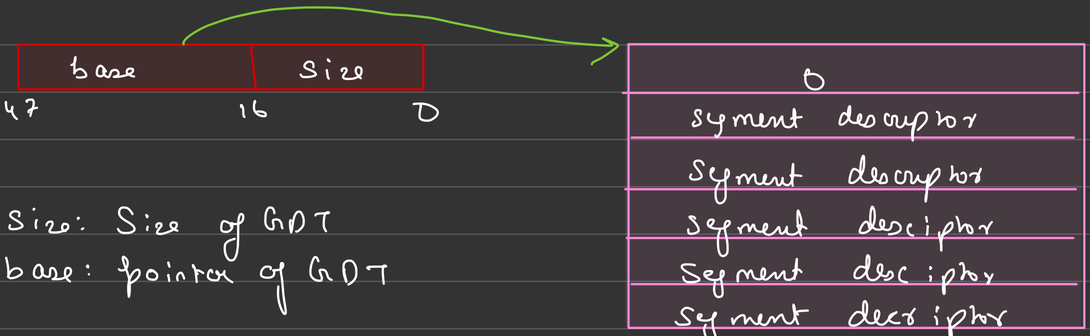
fragmentation is an issue with this approach because heap, stack, text etc are not stored in continuous fashion.

Intel x86 address translation



★ The segmentation Unit of x86 Systems

It contains a
GDT (global descriptor table)
Stored In Memory
Pointed to by GDTR(GDT register)



Size: Size of GDT
base: pointer of GDT

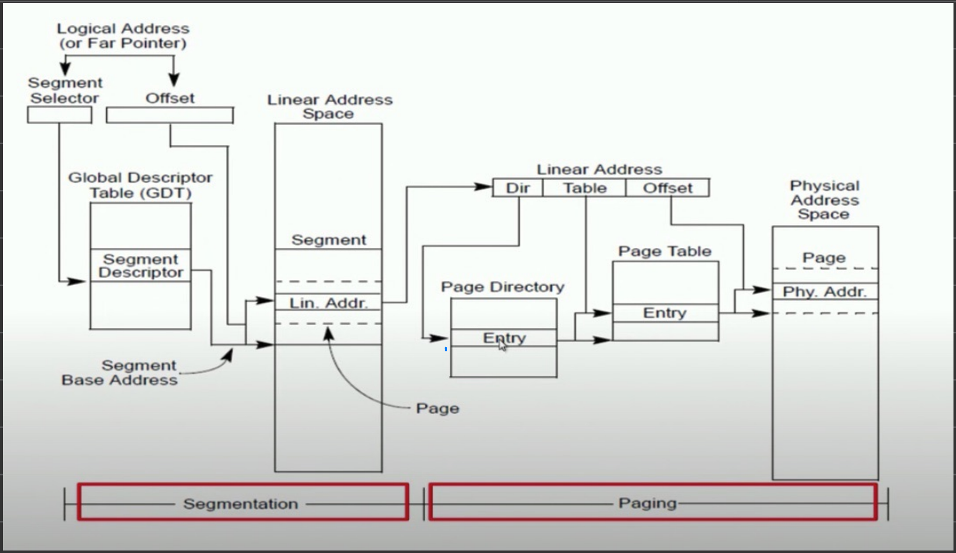
★ Segment descriptor inside GDT →

Access	limit
Base address	

Base add → 0 – 4GB
Limit → 0 – 4GB
Access Right
↦ execute , Read, write
→ Privilege level (0 – 3)

★ Segment and offset registers

- → Holds a 16 bit segment selector
  - ↳ Points to offsets in GDT
- → Offset registers are 32 bit registers
- → Segments associated with one of the 3 types of storage → code, stack, data



full picture