CSIS 429 Operating Systems

Lecture 9: Smaller Tables

October 4th 2020

Textbook chapters

- Read “Advanced Page Tables” and “Swapping”

Intro	Virtualization		Concurrency	Persistence	Appendices
Preface	3 *Dialogue*	12 *Dialogue*	25 *Dialogue*	35 *Dialogue*	*Dialogue*
TOC	4 Processes	13 Address Spaces	26 Concurrency and Threads code	36 I/O Devices	Virtual Machines
1 *Dialogue*	5 Process API code	14 Memory API	27 Thread API	37 Hard Disk Drives	*Dialogue*
2 Introduction code	6 Direct Execution	15 Address Translation	28 Locks	38 Redundant Disk Arrays (RAID)	Monitors
	7 CPU Scheduling	16 Segmentation	29 Locked Data Structures	39 Files and Directories	*Dialogue*
	8 Multi-level Feedback	17 Free Space Management	30 Condition Variables	40 File System Implementation	Lab Tutorial
	9 Lottery Scheduling code	18 Introduction to Paging	31 Semaphores	41 Fast File System (FFS)	Systems Labs
	10 Multi-CPU Scheduling	19 Translation Lookaside Buffers	32 Concurrency Bugs	42 FSCK and Journaling	xv6 Labs
	11 *Summary*	20 Advanced Page Tables	33 Event-based Concurrency	43 Log-structured File System (LFS)	Flash-based SSDs
		21 Swapping: Mechanisms	34 *Summary*	44 Data Integrity and Protection	
		22 Swapping: Policies		45 *Summary*	
		23 Case Study: VAX/VMS		46 *Dialogue*	
		24 *Summary*		47 Distributed Systems	
				48 Network File System (NFS)	
				49 Andrew File System (AFS)	
				50 *Summary*	

Address Translation with Paging

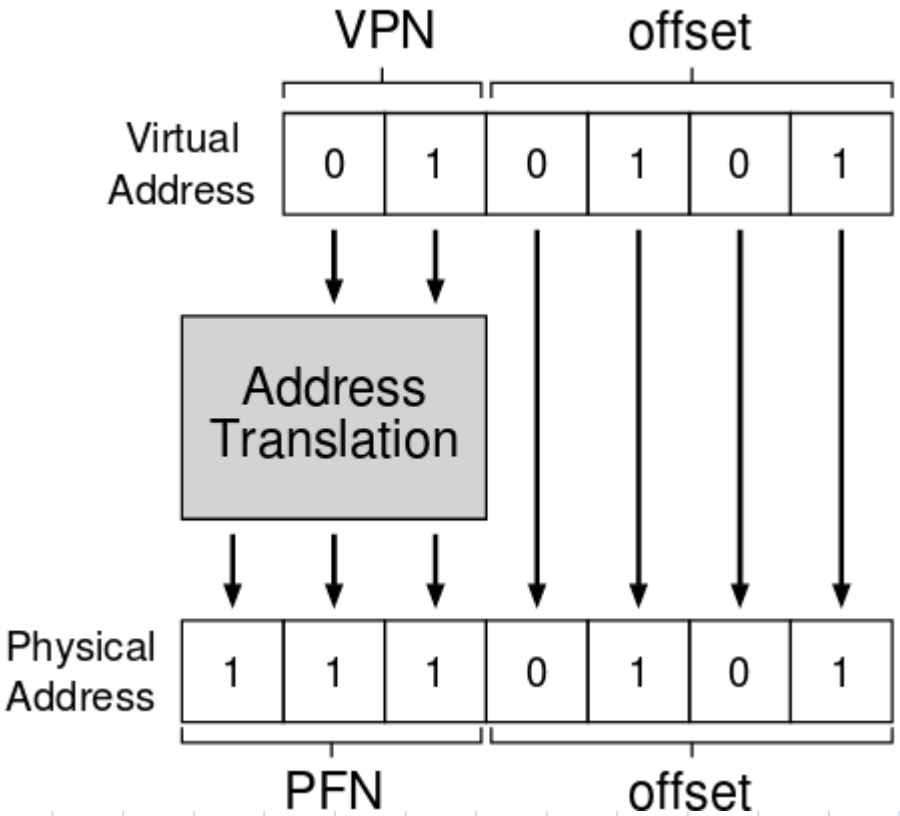
Example: Instruction to load data from memory to register

```
movl <virt-addr>, %eax
```

64 byte virtual address space

page size is 16 bytes

128 byte physical address space



Translation Lookaside Buffer

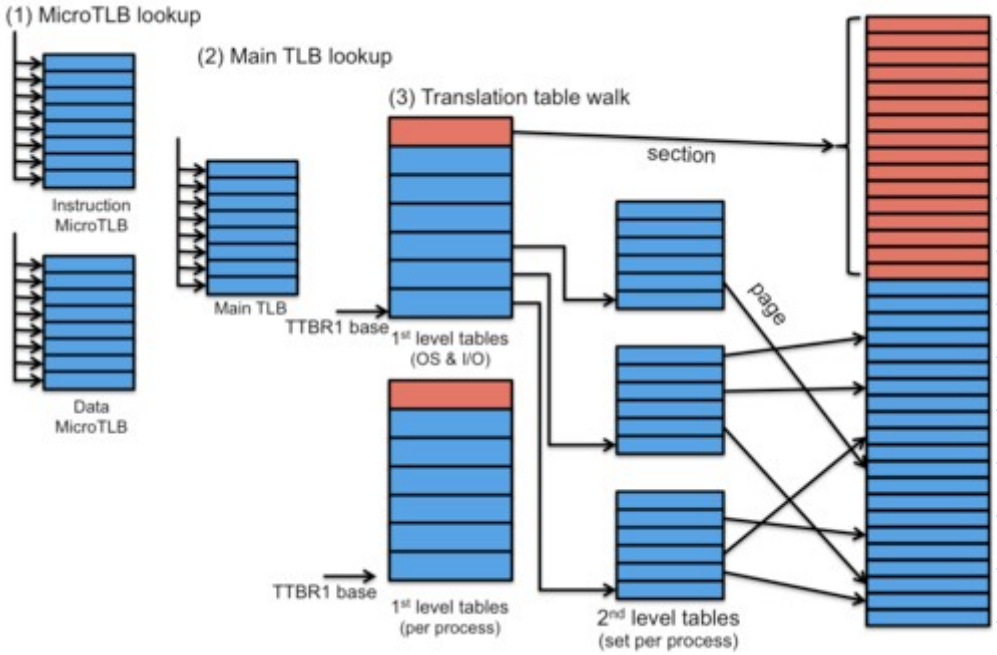
- Every virtual memory reference will involve the TLB to speed address translations. A TLB miss is expensive → shows up as slow execution time.

- We can see speed-ups due to TLB hit rate in our programs – mainly in executing loops.

- Why loops? Because repeatedly accessing the same or nearby data can increase the TLB hit rate if we do it right.

Modern TLBs

- ARM – RISC CPU with 64-bit address space
- Two levels of TLBs:
  - Fast 32-entry fully-associative Micro TLBs for Data and Instruction
  - Slower back-up Main TLB with 8 fully associative plus 64 set-associative (variable # of clock cycles for search)



Page Table Sizes

- A 32-bit (virt & phys) system with 4KB pages will have 12-bit offsets → 20-bit VPNs and PFNs

  If each PTEntry requires 20 bits for PFN and 12 bits for other information (Ref bit, Protection, etc.) → 4 bytes/PTE

  → 4MB for the process table of each process!!!

  How could this be smaller? Bigger pages? 8KB? 16 KB?

The Crux of the Problem: Page Tables Are Too Big

4 MB is too big – too much overhead

How can we make page tables smaller?

What are the trade-offs?

Linux allows 4 MB Pages

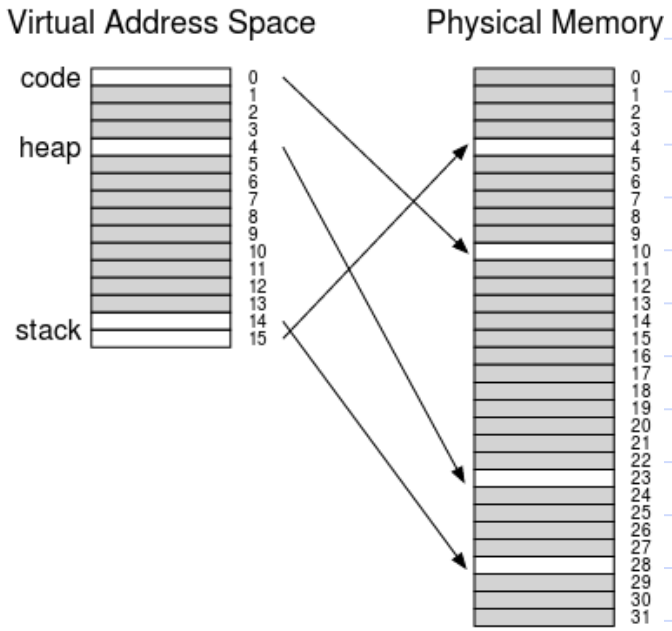
4MB is 1024 times larger than 4KB → PTEs are smaller, Page Tables would be 4 KB

Not good in general b/c of Internal Fragmentation

Mainly used in high performance DBMS because of the huge TLB performance gains

PT for Small Example System

- 32 KB Virtual, 64 KB Physical Space, 1 KB pages



PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
23	1	rw-	1	1
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

Page Table For 16KB Address Space

Most entries are invalid!

Large Page Tables are expensive/wasteful

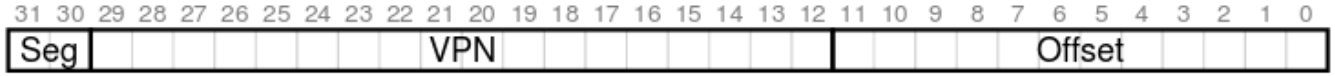
- Recall, we used segmentations to reduce waste – internal fragmentation

- How about combining Segmentation + Paging?

32-bit System with 4KB pages

- Let's use 3 segments: code, heap & stack

Virtual address:

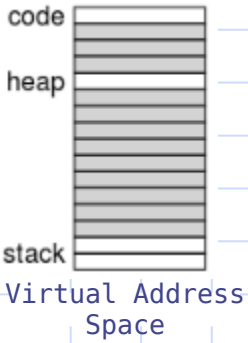


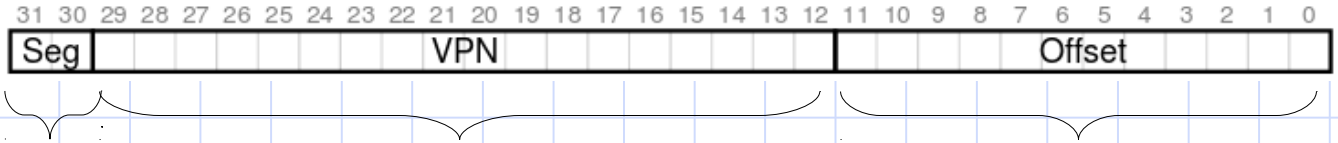
Plus 3 Base+Bounds register pairs, only the Base Register will contain the physical address of the Page Table for that Segment

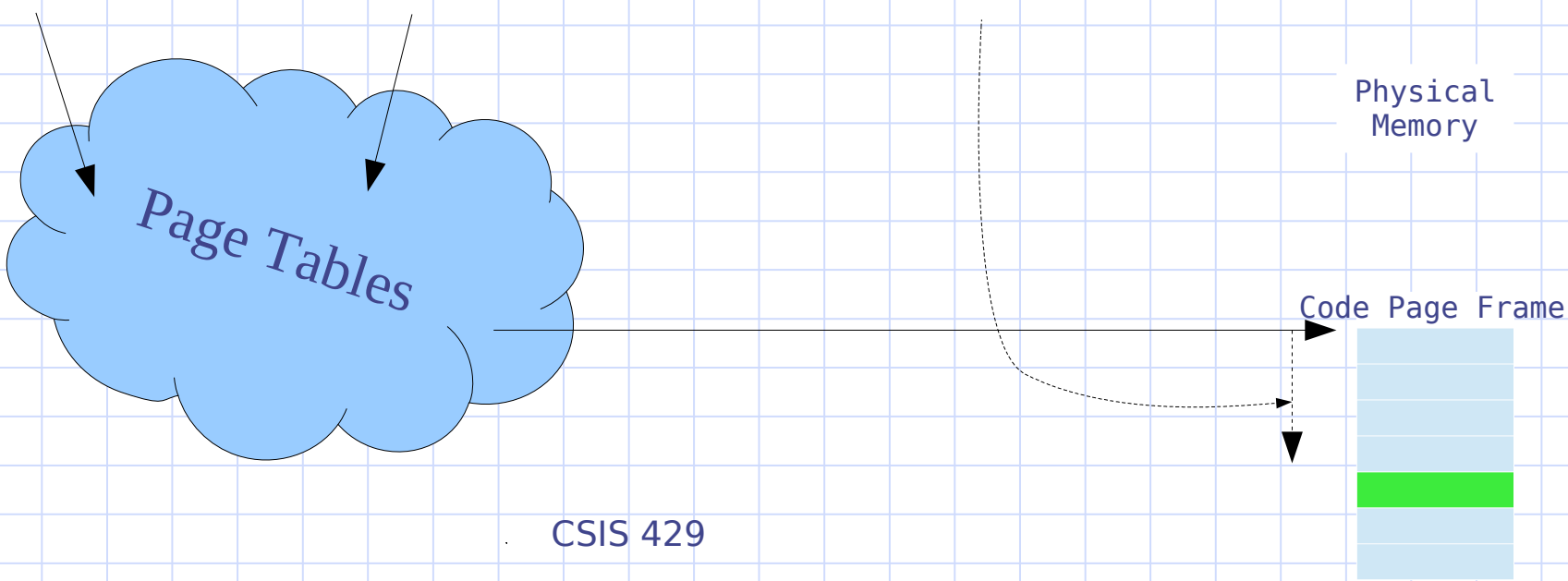
→ A process will have 3 page tables!

Segmentation + Paging

- 3 segments: code, heap & stack → 3 page tables!

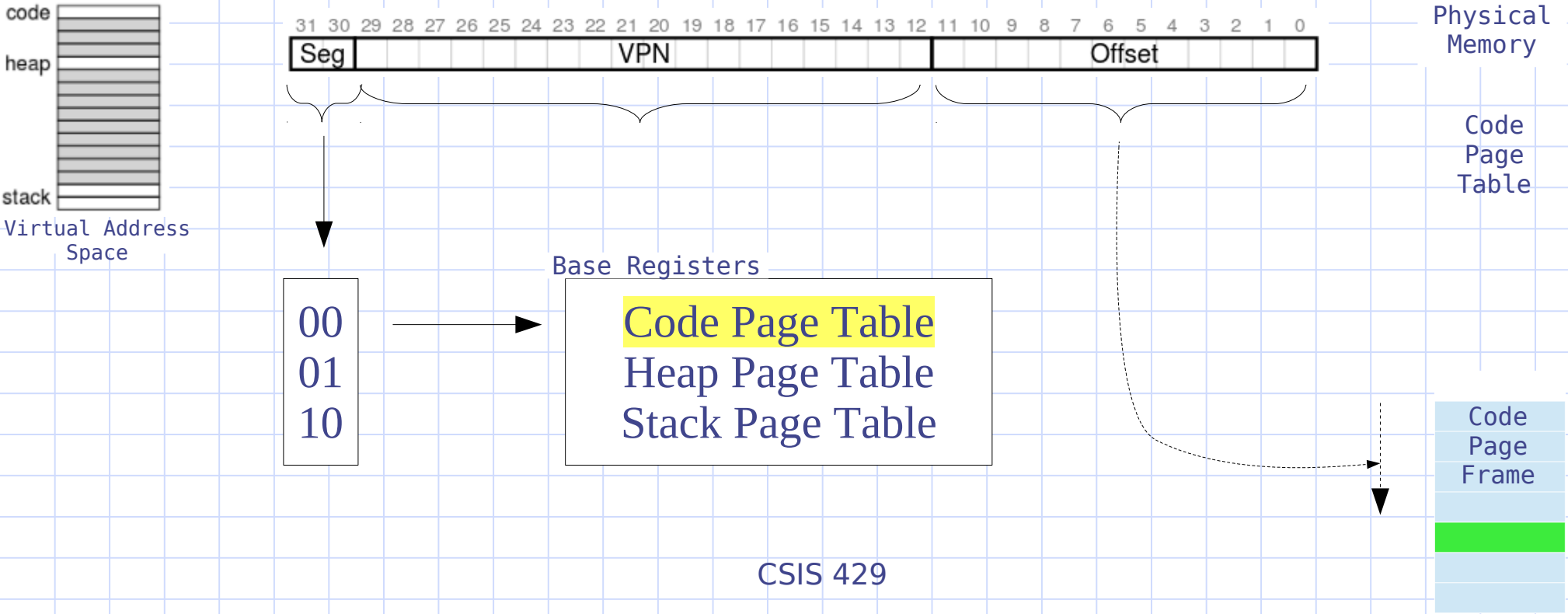






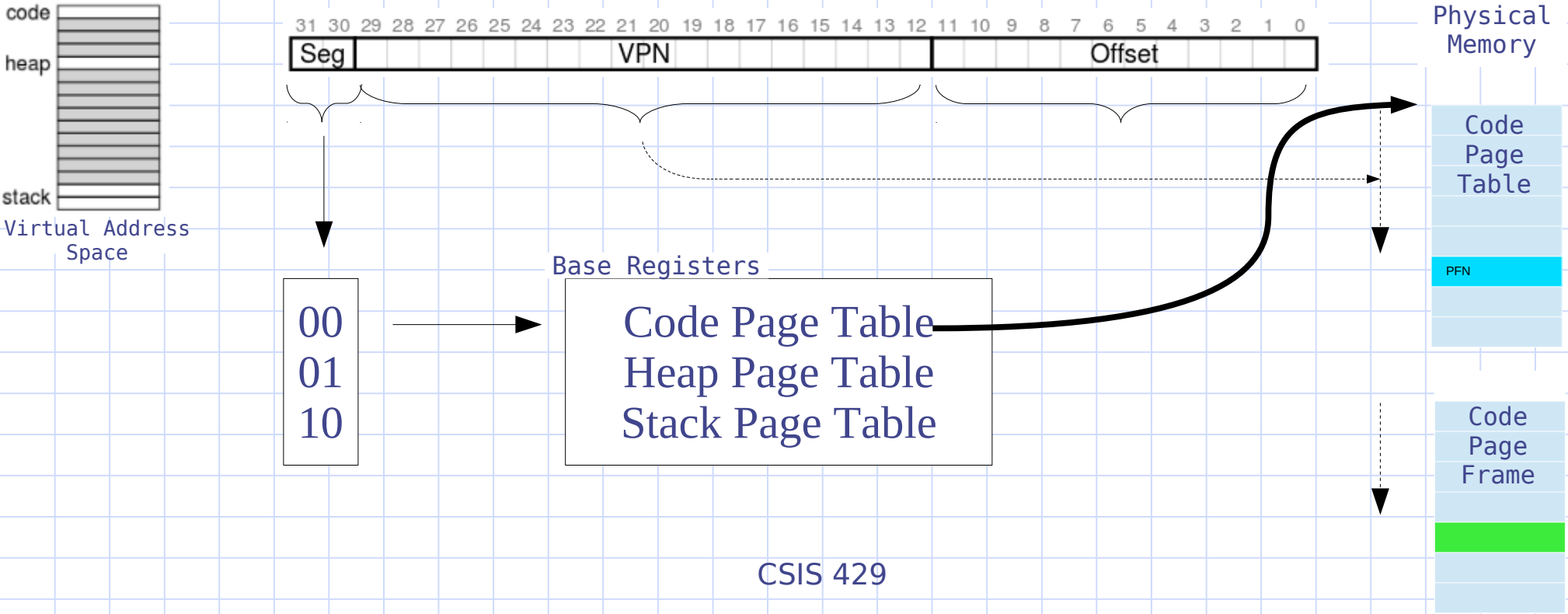
Segmentation + Paging

- 3 segments: code, heap & stack → 3 page tables!



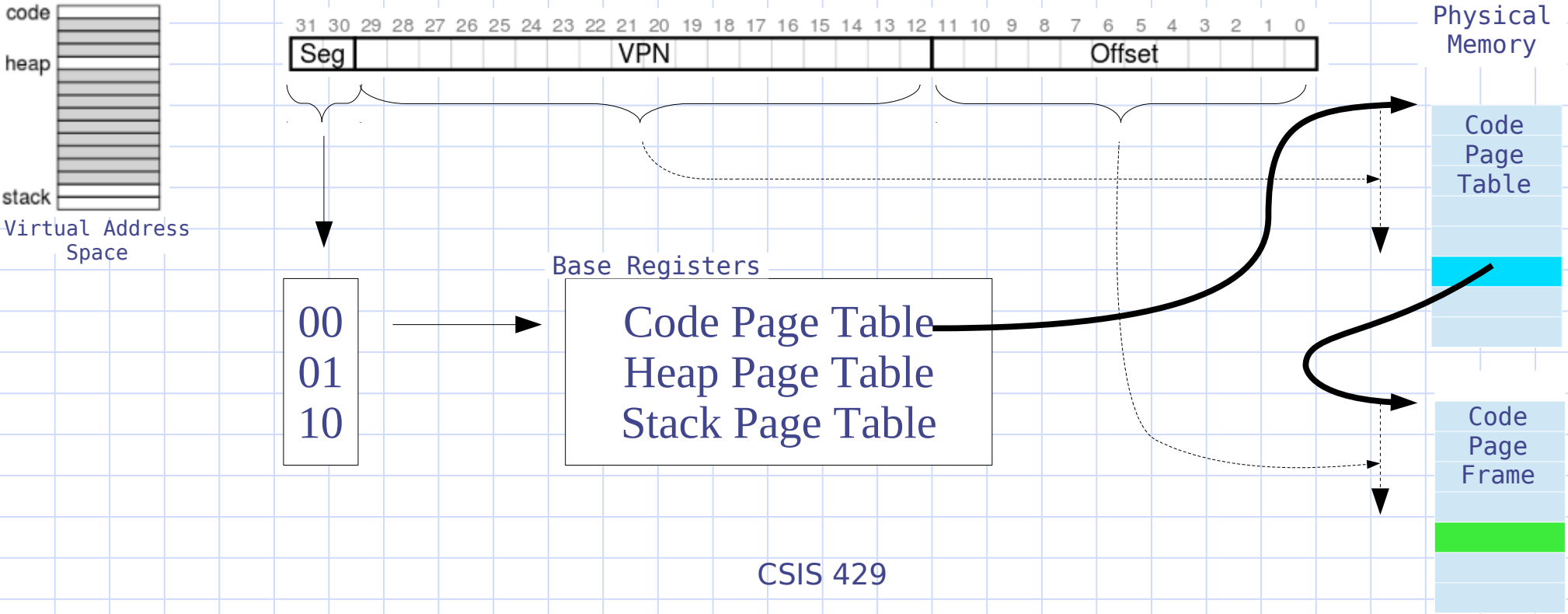
Segmentation + Paging

- 3 segments: code, heap & stack → 3 page tables!



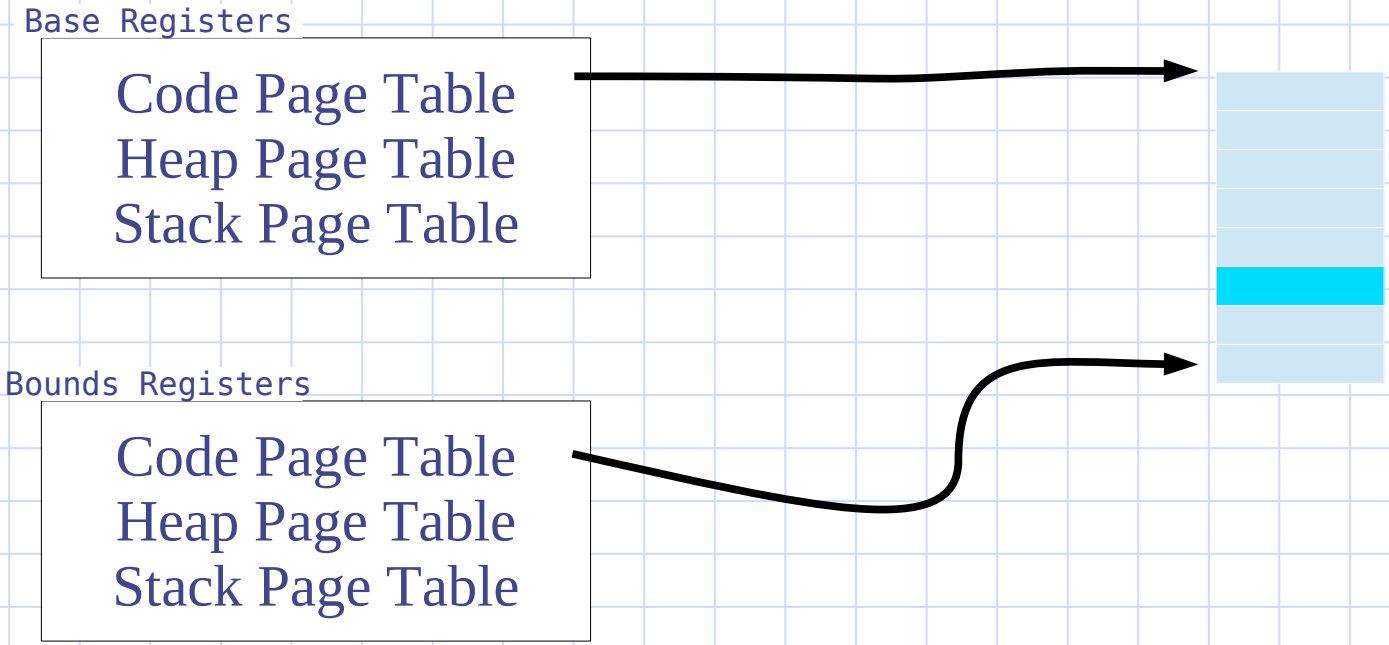
Segmentation + Paging

- 3 segments: code, heap & stack → 3 page tables!



Segmentation + Paging

- Bounds registers can point to end of (arbitrary sized) Page Tables.



Arbitrary size Page Tables → Fragmentation

- Combining Segmentation + Paging with arbitrary size page tables

  → external fragmentation again

Better solution: Break Page Tables into smaller pieces - use a page-directory to organize the page table.

Multi-level Page Table Example

Linear Page Table

PTBR 201

	valid	prot	PFN
PFN 201	1	rx	12
	1	rx	13
	0	-	-
	1	rw	100
PFN 202	0	-	-
	0	-	-
	0	-	-
	0	-	-
PFN 203	0	-	-
	0	-	-
	0	-	-
	0	-	-
PFN 204	0	-	-
	0	-	-
	1	rw	86
	1	rw	15

Multi-level Page Table

PDBR 200

	valid	PFN
PFN 200	1	201
	0	-
	0	-
	1	204

The Page Directory

	valid	prot	PFN
PFN 201	1	rx	12
	1	rx	13
	0	-	-
	1	rw	100

[Page 1 of PT: Not Allocated]

[Page 2 of PT: Not Allocated]

	valid	prot	PFN
PFN 204	0	-	-
	0	-	-
	1	rw	86
	1	rw	15

The parts of the page table that are not needed are not allocated.
Each piece would be the same size as a page

Multi-level Page Tables

Advantages:

- Save space - only allocate needed parts of page table.
- If each part is same size as a page → easy allocation.

No penalty for TLB hits → address translation in 1 clock cycle.

Disadvantages:

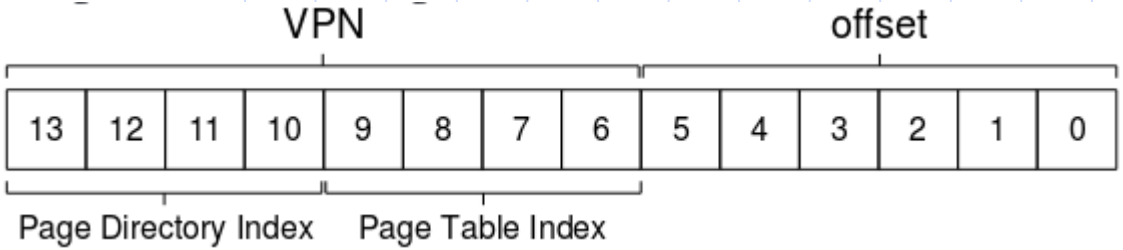
- More complex
- Adding a level of indirection: TLB miss → two memory accesses to do a single memory translation.

Detailed 2-level Page Table Example

- Ch. 20 pp 7-10

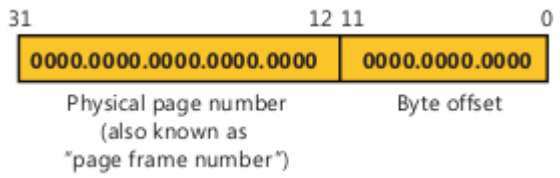
0000 0000	code
0000 0001	code
0000 0010	(free)
0000 0011	(free)
0000 0100	heap
0000 0101	heap
0000 0110	(free)
0000 0111	(free)
.....	... all free ...
1111 1100	(free)
1111 1101	(free)
1111 1110	stack
1111 1111	stack

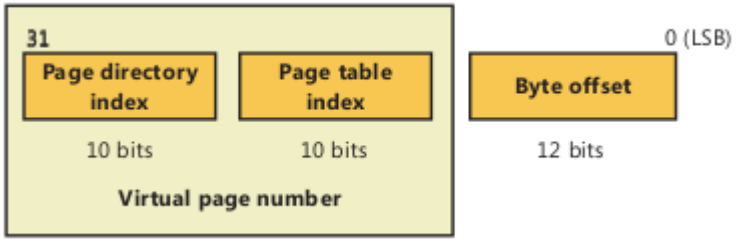
A 16KB Address Space With 64-byte Pages

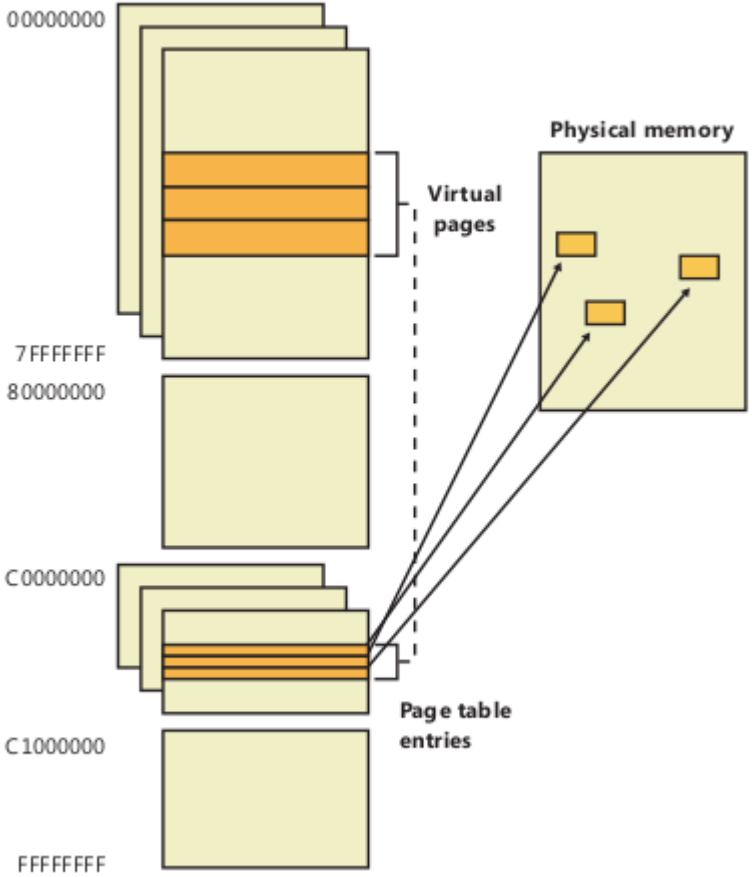


Win x86 Virtual Address Translation

- 32-bit (x86) Windows has a 4GB process address space

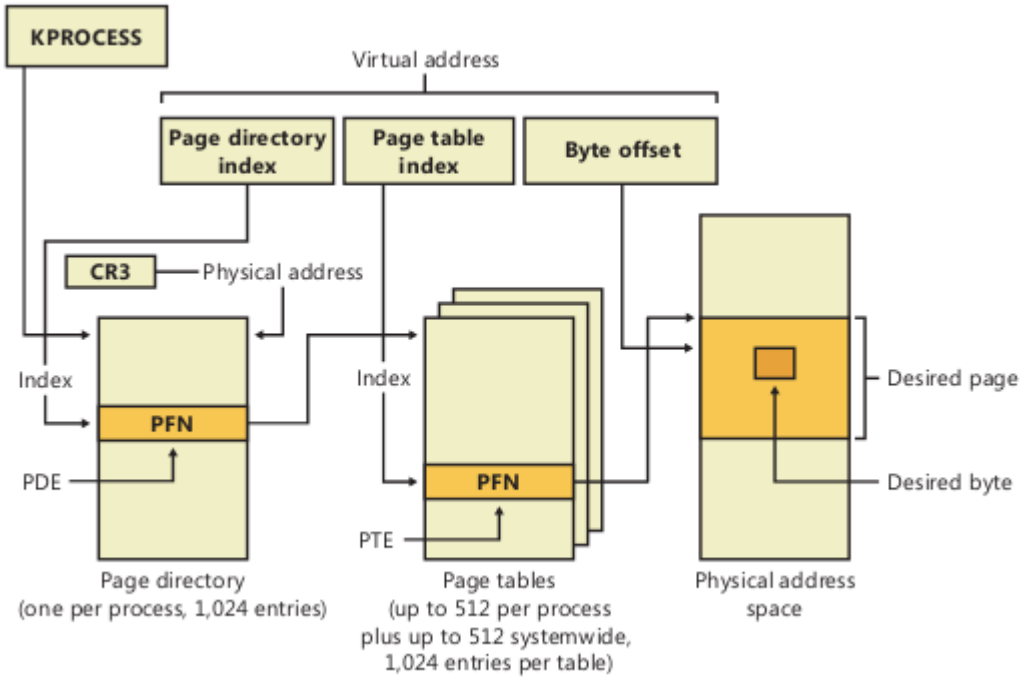


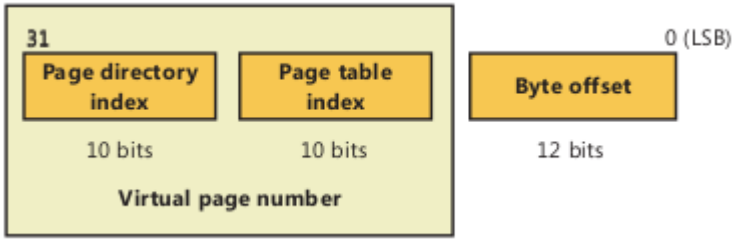




Win x86 2-level paging

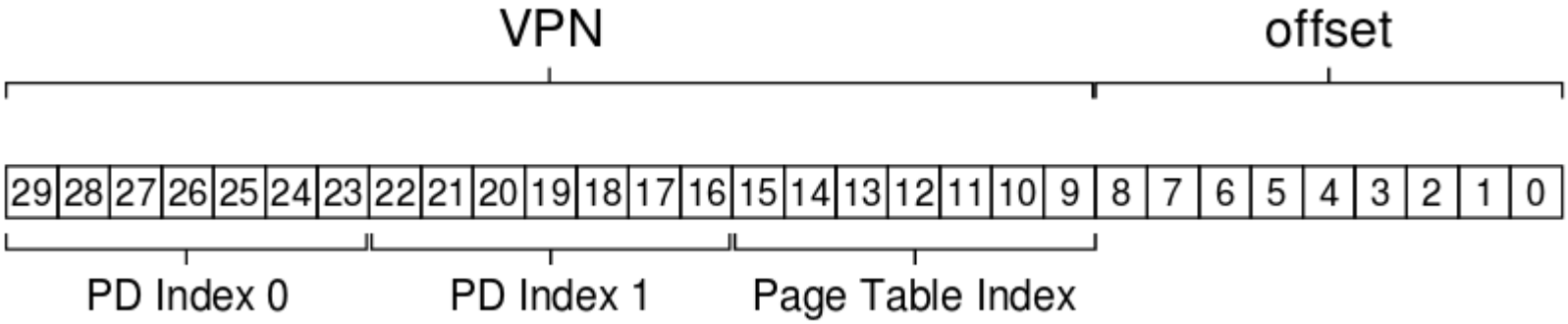
- Windows uses x86 privileged CPU/MMU register CR3 for physical address of page directory





3-level Page Table Example

- 30-bit virtual address space with 512-byte pages
  → 9-bit offset, 21-bit VPN



Goal: make each page table fit inside 1 page.

Segmentation with Paging – Intel 386

The Intel 386 uses segmentation with paging for memory management with a two-level paging scheme.

Both require hardware support in MMU but an OS does not have to use it.

Segmentation came with 8086

Paging came with 80386 (optional)

Segmentation with Paging – Intel 386

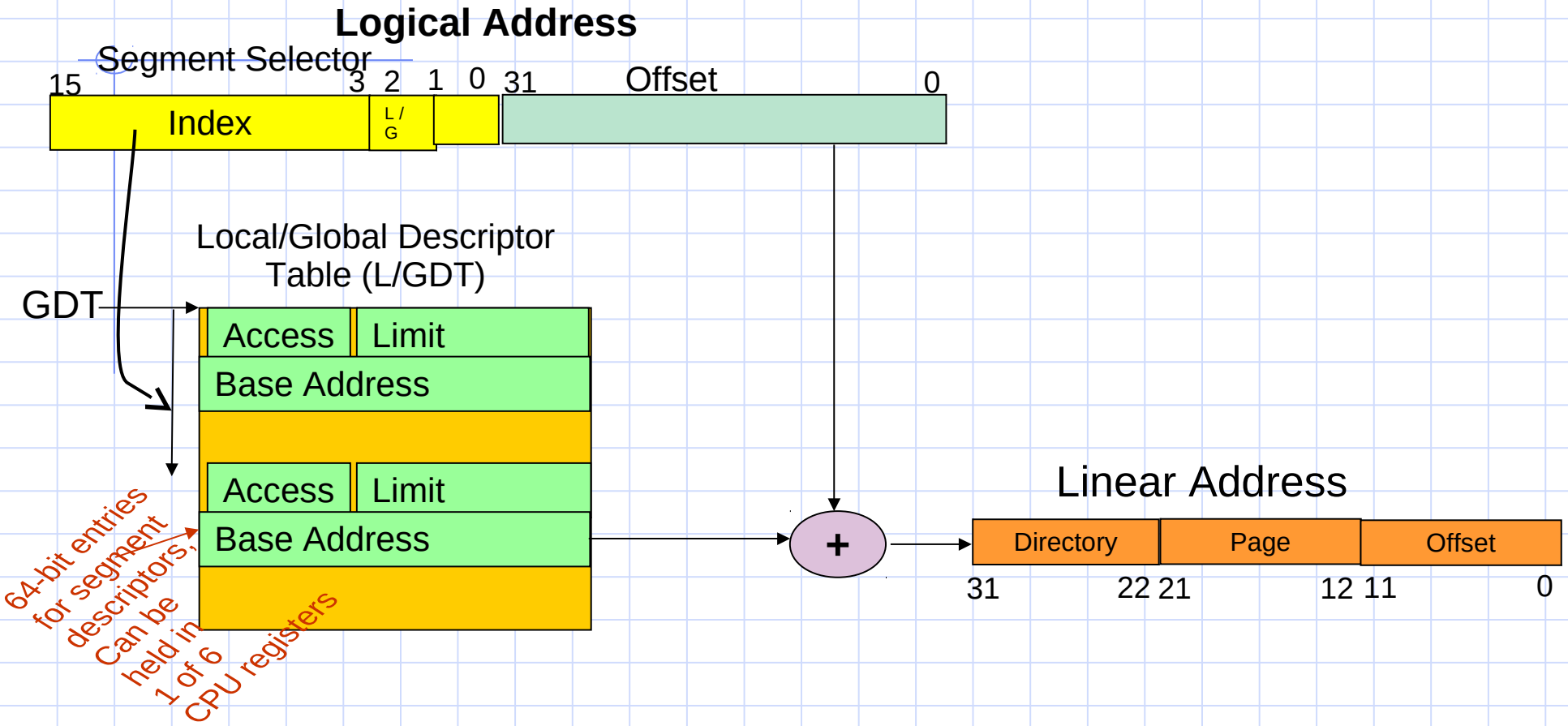
The Intel 386 uses segmentation with paging for memory management with a two-level paging scheme.

Intel logical address is a SEGMENT:OFFSET pair

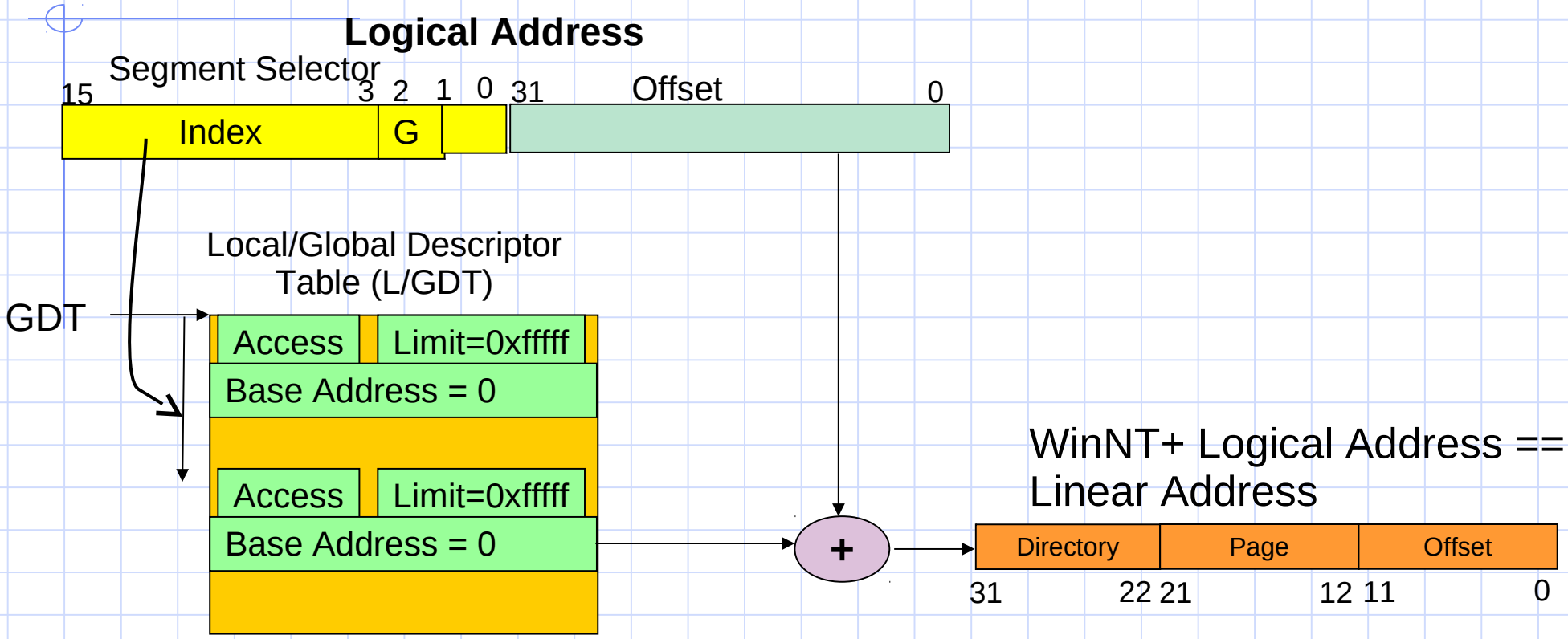
- Linux: Segmentation + Paging
- Win3: Segmentation + Paging
- WinNT+: No segmentation, paging only
  - WinNT+ virtual address is Intel linear address

Cf. Motorola 68000: No segmentation: flat address space; logical address is the same as the linear address

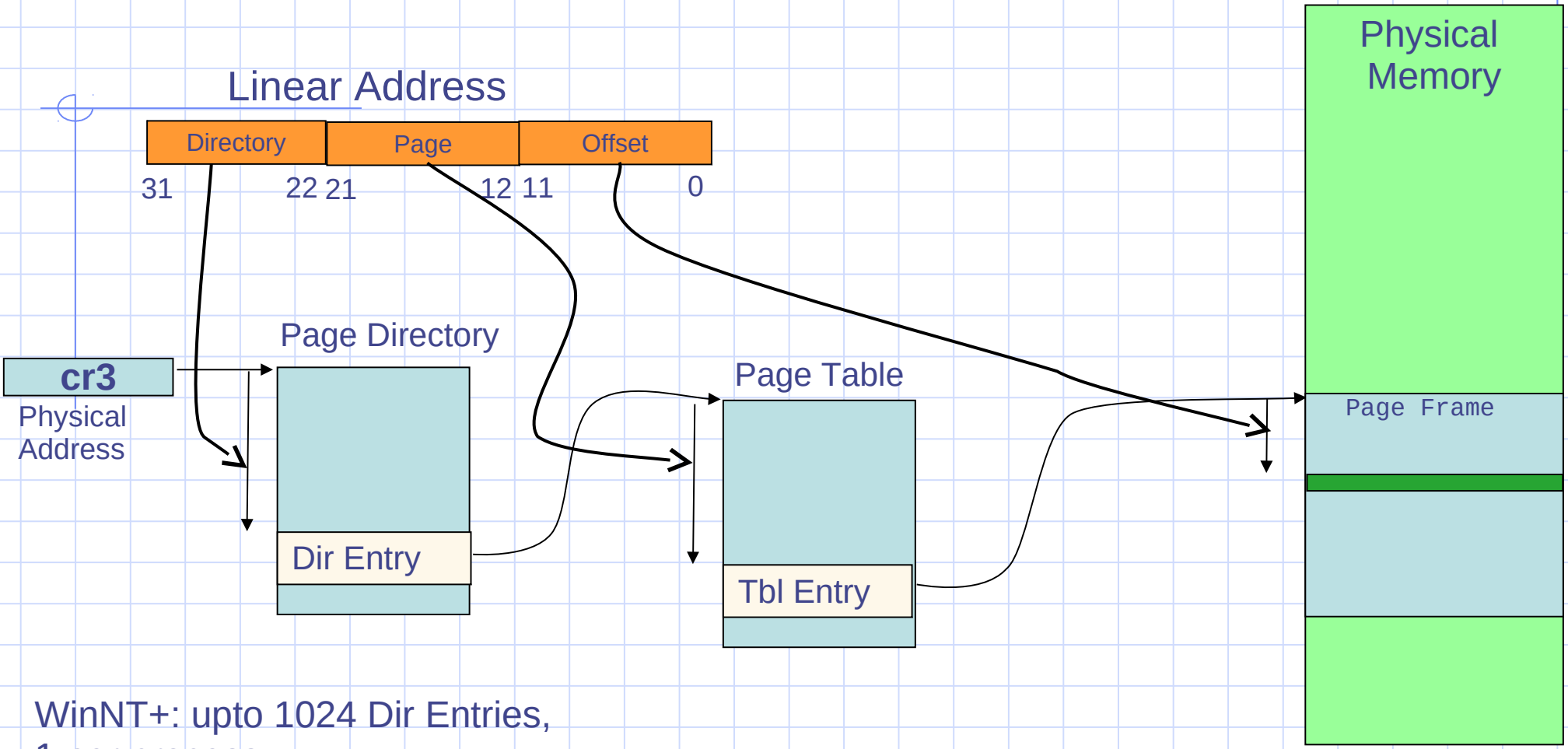
Intel 80386 Address Translation



Intel 80386 Address Translation in WinNT+

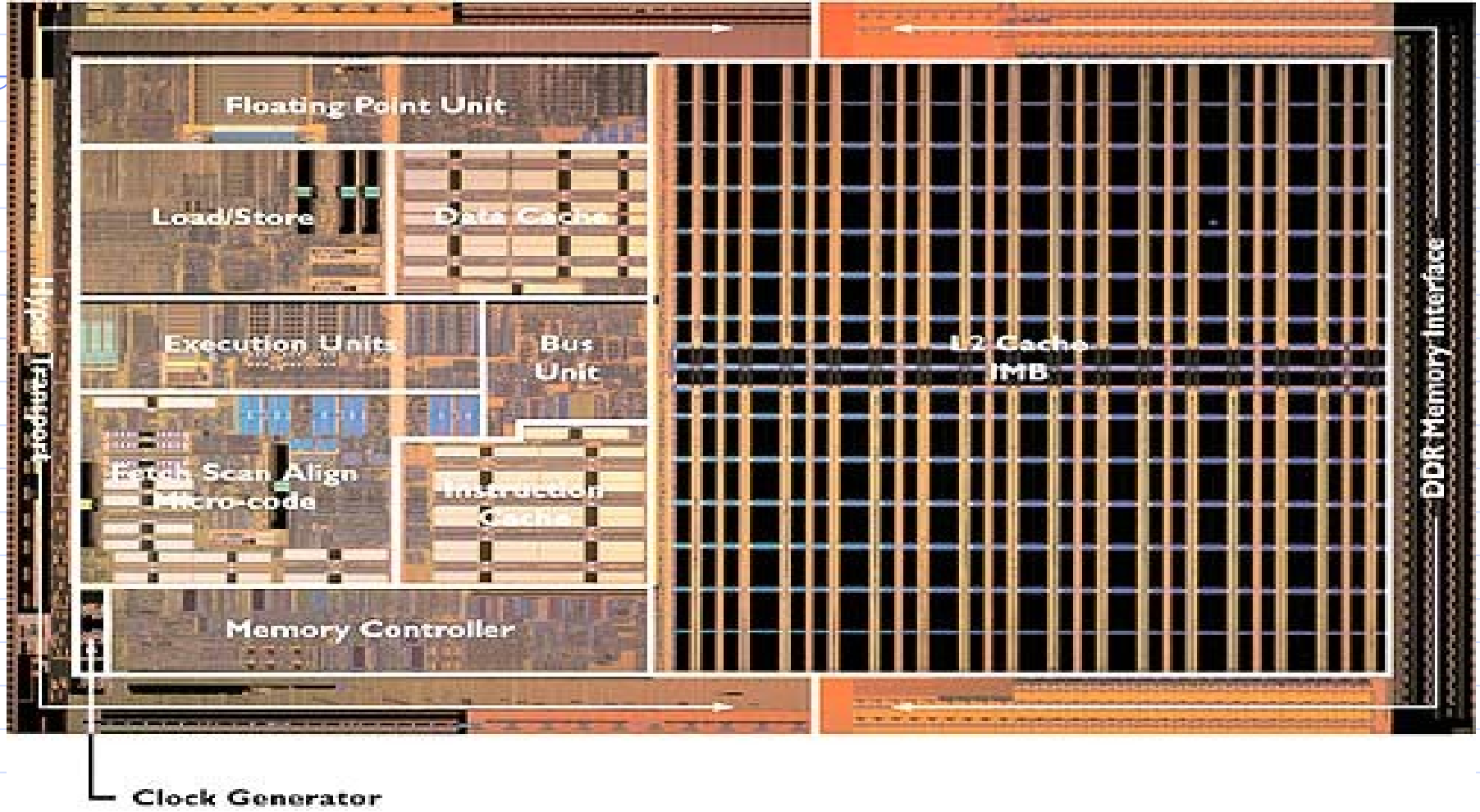


Intel 80386 Address Translation

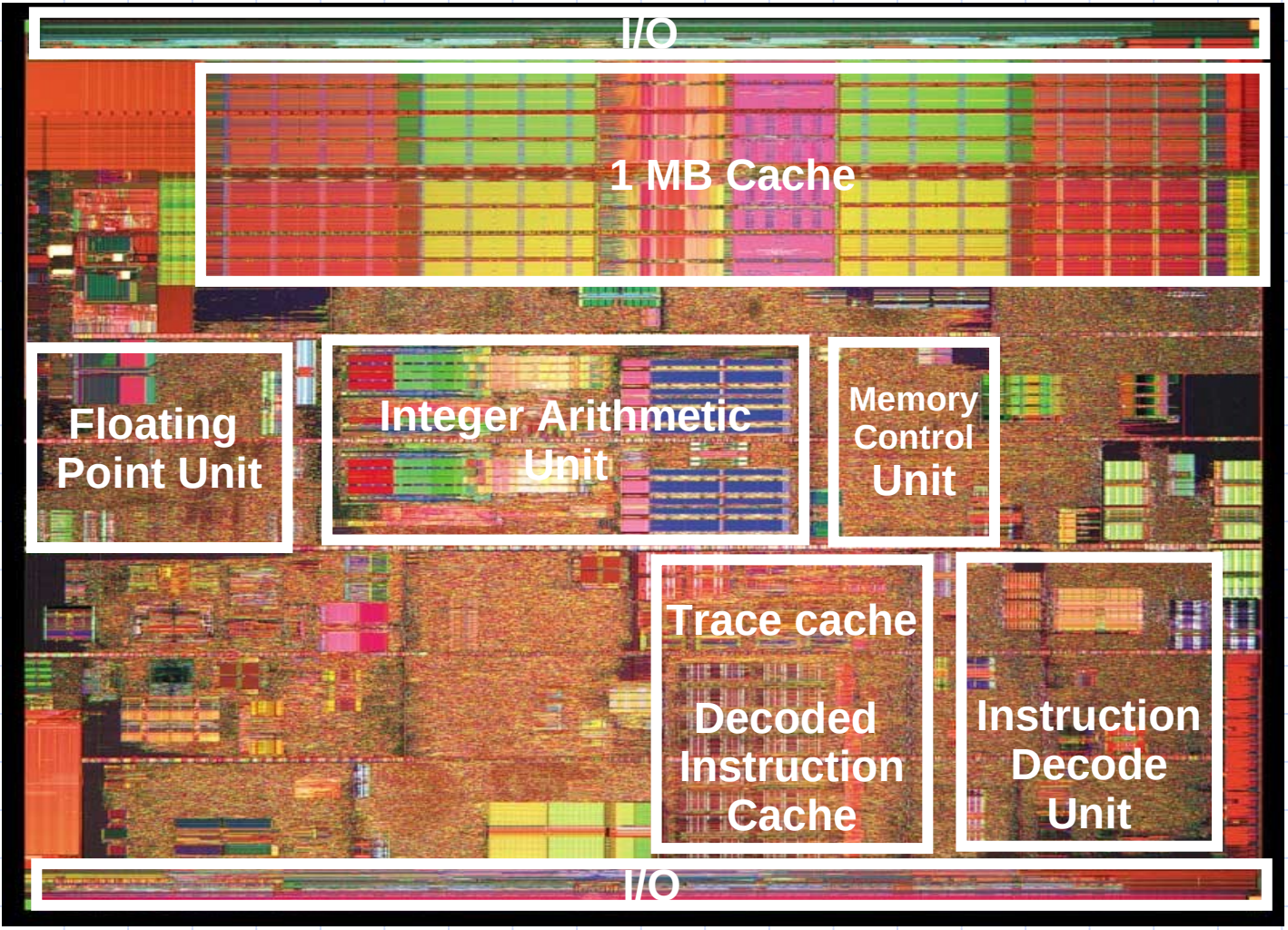


WinNT+: upto 1024 Dir Entries,
1 per process

AMD64 = Memory + some processing



Intel P4 Prescott



90 nm lithography
112mm² die
125 million transistors
L1 Cache: **16KB**
L2 Cache: **1MB**
3.4GHz Clock
Instruction Set:
- MMX
- SSE
- SSE2
- **SSE3**