

Virtual Memory - 1

(Address Translation)

Virtual Memory addresses all Memory Management issues simultaneously

- **Spatial issues in Allocation**
 - **Allows non-contiguous physical allocation, while making it appear contiguous virtually**
- **Physical memory (DRAM) is limited**
 - **Allowing address spaces larger than physical memory size (using disk space)**
- **Disallow 1 process from accessing another processes memory.**

与某物相邻的

Non-contiguous Allocation

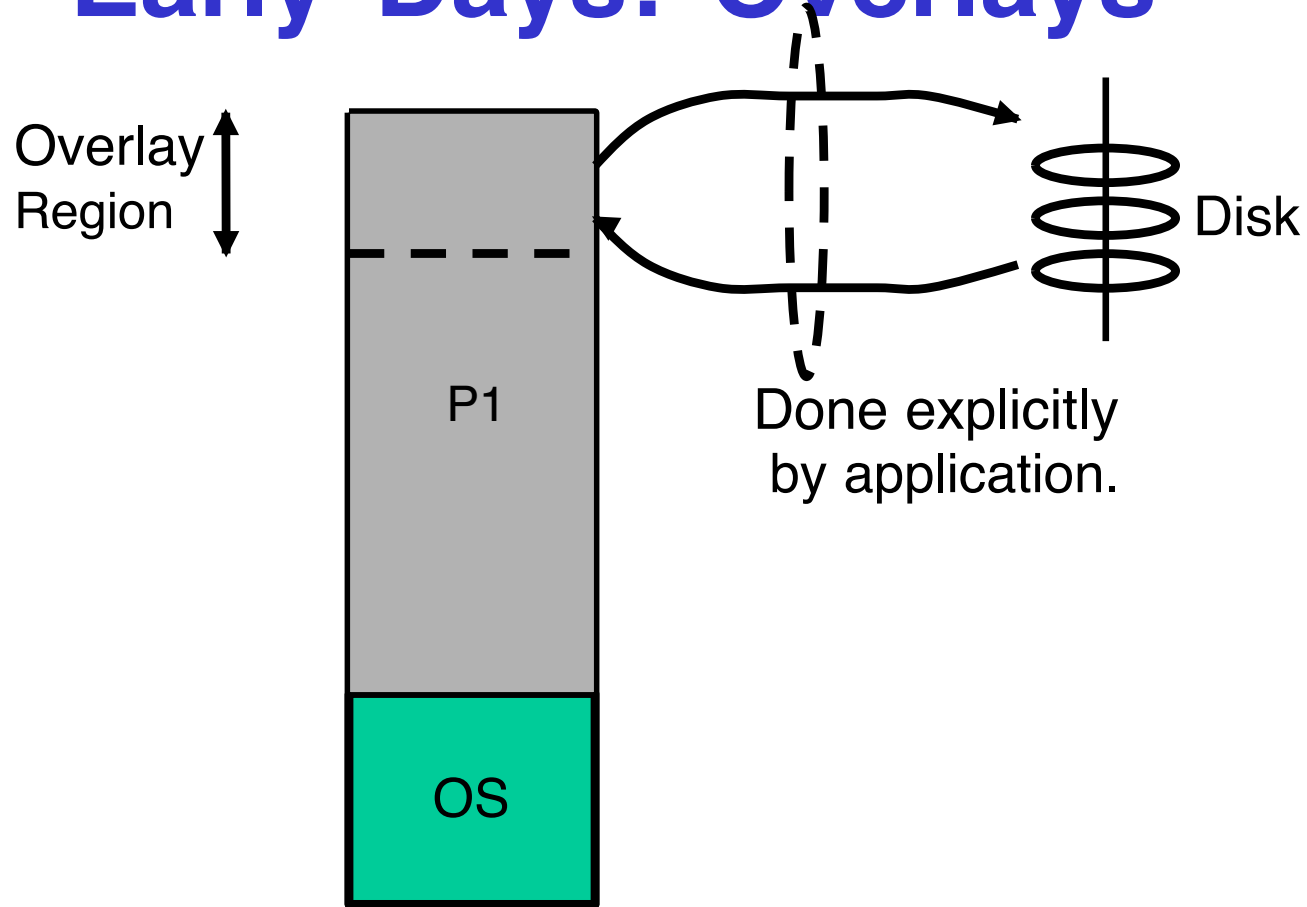
- **You may get several chunks of physically separate DRAM space.**
- **However, these get stitched together as 1 contiguous “virtual” space that the program sees.**
- **We will get back to this later!**

Dealing with Limited Memory

- If something (either part of a process, or multiple processes) does not fit in memory, then it has to be kept on disk.
- Whenever that needs to be used by the CPU (to fetch the next instruction, or to read/write a data word), it has to be brought into memory from disk.
- Consequently, something else needs to be evicted from memory.
- Such transfer between memory and disk is called **swapping**.

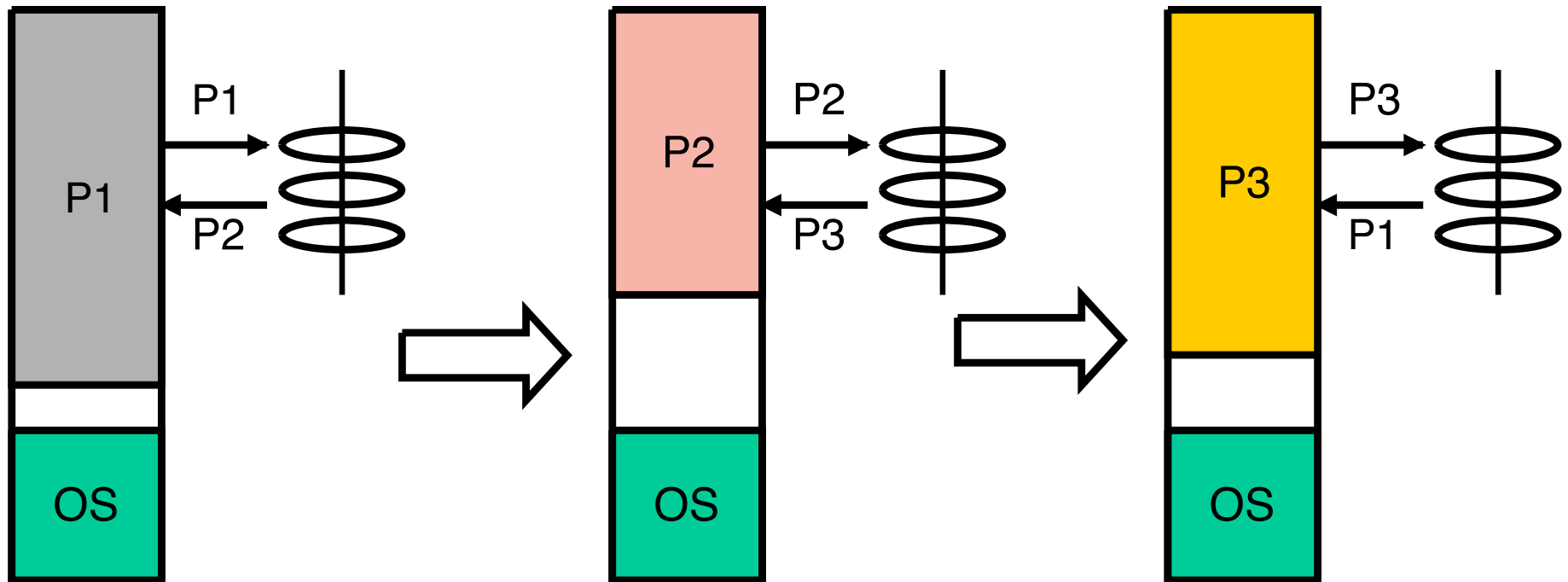
- **Usually, a separate portion of the disk is reserved for swapping, that is referred to as swap space.**
- **Note that swapping is different from any explicit file I/O (read/write) that your program may contain.**
- **Typically swapping is transparent to your program.**

Early Days: Overlays



In this case, even a single application process does not fit entirely in memory

**Even if a process fits entirely in memory,
we do not want to do the following ...**



Context switching will be highly inefficient, and it defeats the purpose of multiprogramming.

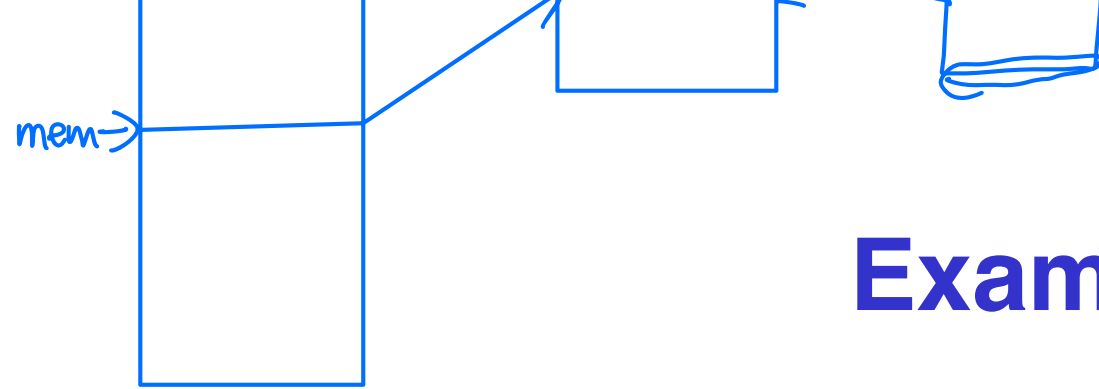
Need for multiprogramming

- Say your program does explicit I/O (read/write) for a fraction **f** of its execution time, then with **p** processes, CPU efficiency =
$$(1 - fp)$$
- To maintain high CPU efficiency, we need to increase **p**.
- But as we just saw, these processes cannot all be on disk. We need to keep as many of these processes in memory as possible.
- So even if we are not keeping all of the process, keep the essential parts of as many processes as possible in memory.
- *We will get back to this issue at a later point!*

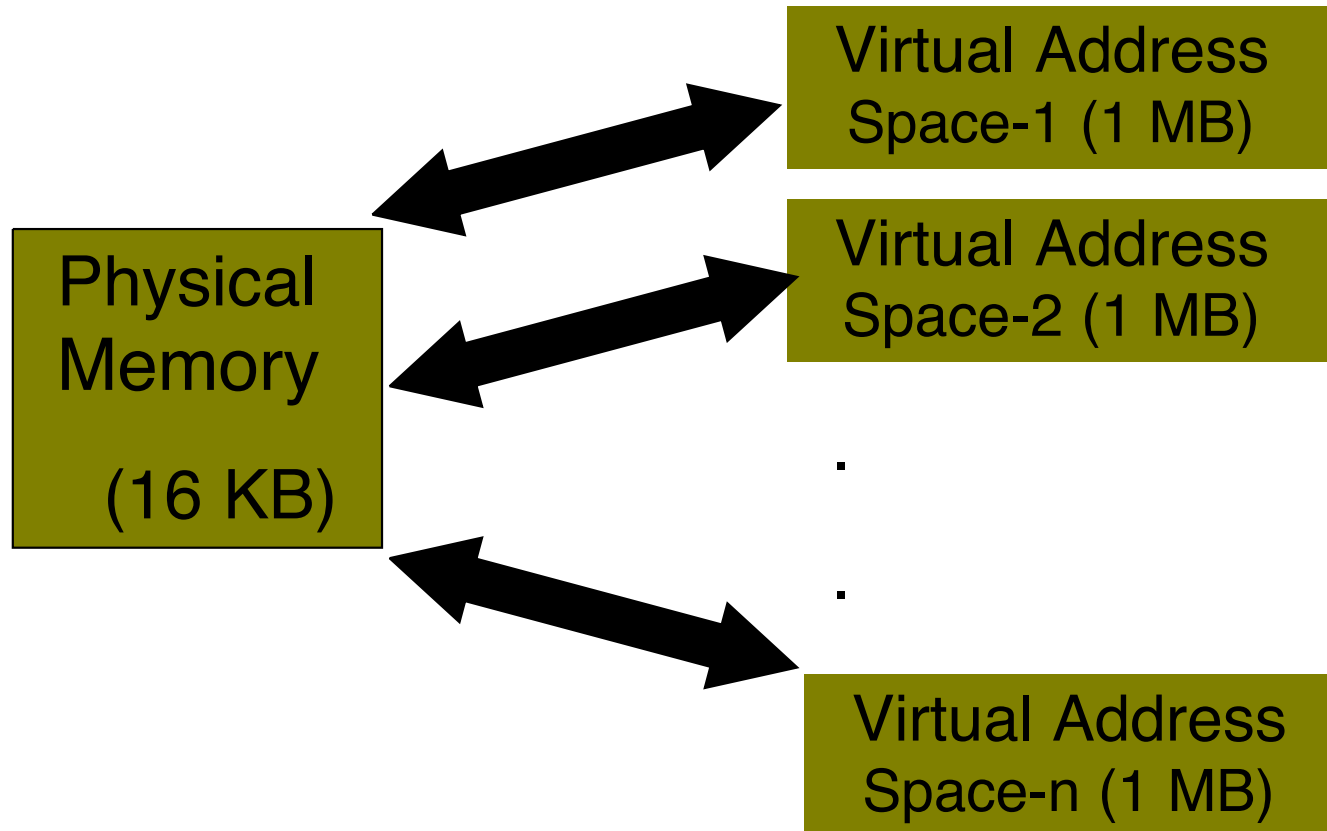
Virtual Memory

- Supports programs that are each larger than available physical memory.
- Allows several programs to reside in physical memory (or at-least the relevant portions of them).
- Allows non-contiguous allocation without making programming difficult.





Example



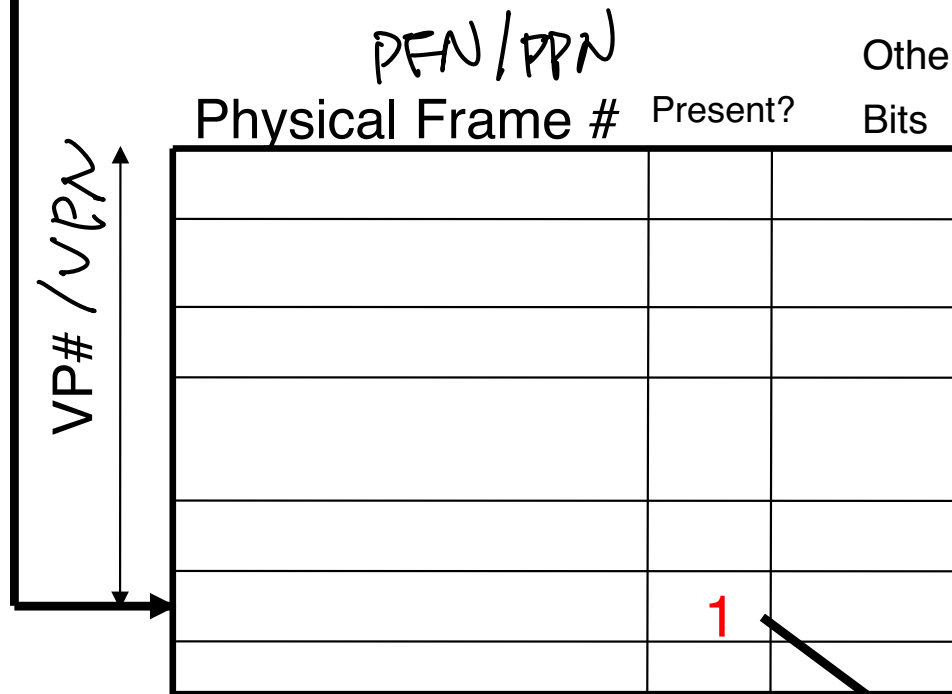
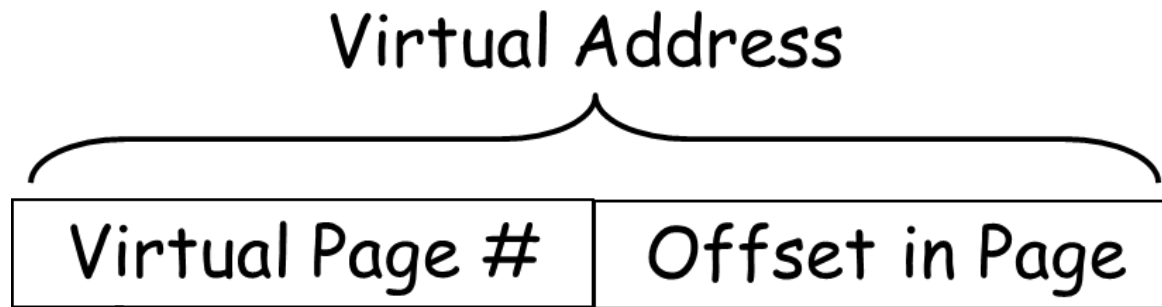
- **Programs are provided with a virtual address space (say 1 MB).**
- **Role of the OS to fetch data from either physical memory or disk.**
 - Done by a mechanism called **paging**.
- **Divide the virtual address space into units called “virtual pages” each of which is of a fixed size (usually 4K or 8K).**
 - Previous example, we have 256 4K-sized pages.
- **Divide the physical address space into “physical pages”/frames.**
 - Previous example, we would have 4 4K-sized pages.

Basic Idea

- **At any time, only a small number of virtual pages can be in one of the physical pages.**
- **Rest would have to be stored on disk in a region called the “swap space”.**
 - **Previous example, we could only have 4 virtual pages residing in physical memory.**
- **A “page” is thus the unit of transfer between disk and physical memory.**

- **Role of the OS to keep track of which virtual page is in physical memory and if so where?**
 - **Maintained in a data structure called “page-table” that the OS builds.**
 - **“Page-tables” map Virtual-to-Physical addresses.**

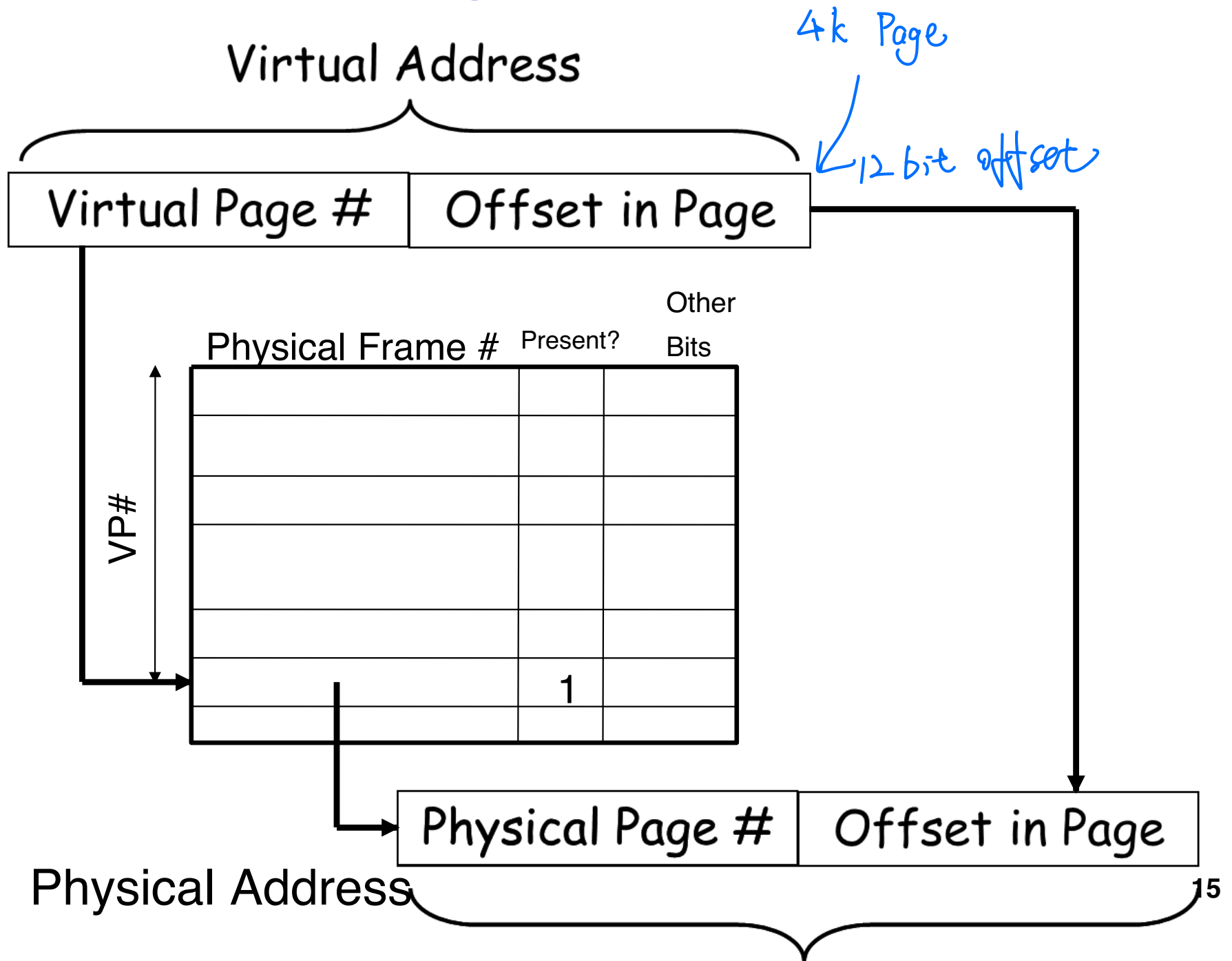
Page Tables



Compose physical address

Fetch data from the physical address

Page Tables



An example

Virtual address space
 $4\text{GB} = 2^{32}$ bytes

Page size
 $= 4096$ bytes
 $= 2^{12}$ bytes



No. of virtual pages ?

$$2^{32} / 2^{12} = 2^{20}$$

No. of bits in virtual address ?

32 bits

An example

Virtual address space
4GB = 2^{32} bytes

Page size
= 4096 bytes
= 2^{12} bytes



No. of virtual pages ?
= $2^{32} / 2^{12}$
= 2^{20}

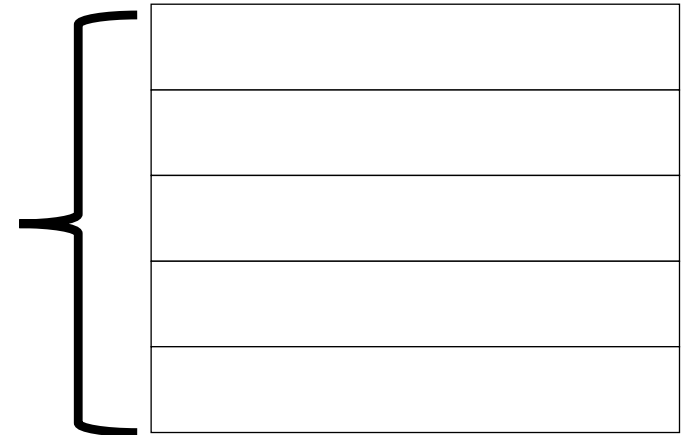
No. of bits in virtual address ?
= 32 bits
= {20 bits page id}{12 bits page offset}

An example

Virtual address space
4GB = 2^{32} bytes

Page size
= 4096 bytes
= 2^{12} bytes

Physical memory size
256KB = 2^{18} bytes



No. of physical frames ?

$$2^{18} / 2^{12} = 2^6$$

No. of bits in physical address ?

18 bits

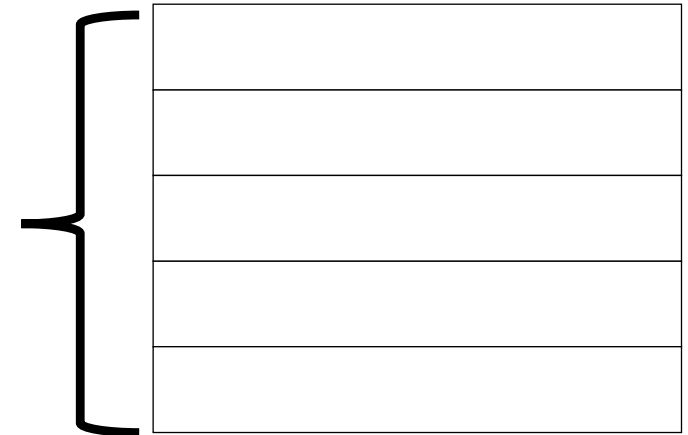
An example

Virtual address space
4GB = 2^{32} bytes



Page size
= 4096 bytes
= 2^{12} bytes

Physical memory size
256KB = 2^{18} bytes



No. of physical frames
= $2^{18} / 2^{12}$
= 2^6

Physical address
= 18 bits

= {6 bits frame id}{12 bits frame offset}₁₉

Page table lookup

Application issues: LOAD [0xBADC0FEE], R1

VA is provided by program

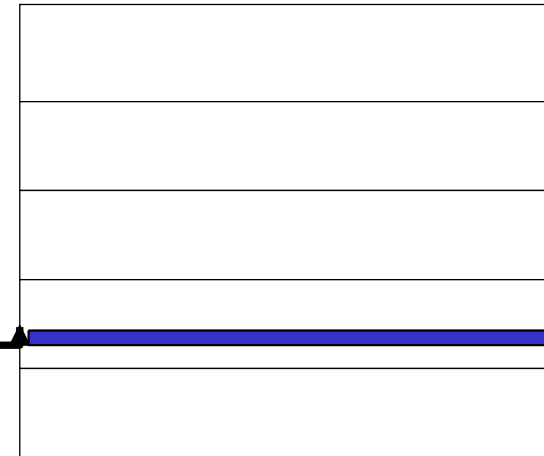
- Which instruction will be fetched according to Instruction Counter and some instructions need data.
- If Data is not in physical address,

Physical memory
Other to do swap. and VA in disk.

Virtual Page # 0xBADC0	Offset in Page 0xFEE
---------------------------	-------------------------

[illegible]

Physical frame # 0x11	Offset in Frame 0xFEE
--------------------------	--------------------------



- **Page-table lookup needs to be done on every memory-reference for both code & data!**
 - **Can be very expensive if this is done by software.**
- **Usually done by a hardware unit called the MMU (Memory-Management Unit).**
 - **Located between CPUs and caches.**

Role of the MMU

- Given a Virtual Address, index in the page-table to get the mapping.
- Check if the *present* bit in the mapping is set, and if so put out the physical address on the bus and let hardware do the rest.
- If it is not set, you need to fetch the data from the disk (swap-space).
 - We do not wish to do this in hardware!


↓ time consuming

Impact on Multi-programming

- Handled by either switching the address-mappings at context-switch time.
- Augment the mapping with a few more bits to encode the process/address space identifier.

$$\text{VA} \xrightarrow{\text{MMU}} \text{PT}_{\text{current}} \rightarrow \text{PA}$$

Page-table entry format

- Physical page Number.
- Present/Absent bit.
- Protection bits (Read / Write / Execute).

- Modified bit (set on a write/store to a page)
 - Useful for page write-backs on a page-replacement.
- Referenced bit (set on each read/write to a page).
 - Will look at how this is used a little later.
- Disable caching.
 - Useful for I/O devices that are memory-mapped.

Practical Issues to Address

Space taken by page tables

Time to look up page tables

Space Issues

- **Size of page-tables would be very large!**
 - For example, 32-bit virtual address spaces (4 GB) and a 4 KB page size would have ? 1 M pages/entries in page-tables.
 - What about 64-bit virtual address spaces?!
- **Process Address Space is Sparse – Leverage this**
 - Use multi-level page-tables. Equivalent to paging the page-tables.
 - Inverted page-tables (will not cover in this course)

– 稀疏的

Linear Page Table

Example:

Page Size = 4KB

Page Table Entry =
8bytes

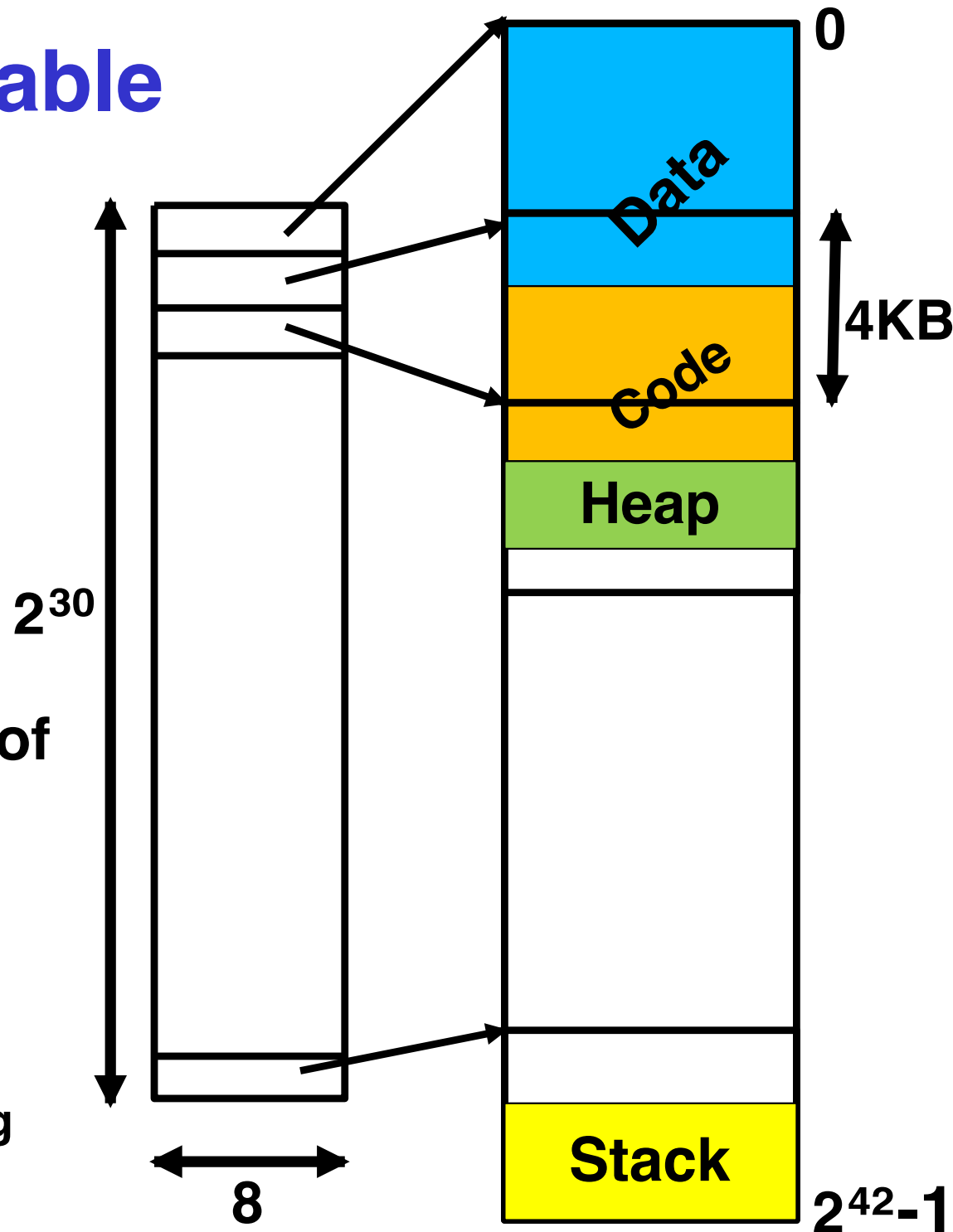
Process VAS = 2^{42}

What would be the size of
page table if we use a
linear page table?

$$2^{30} * 8 = 8\text{GB}$$

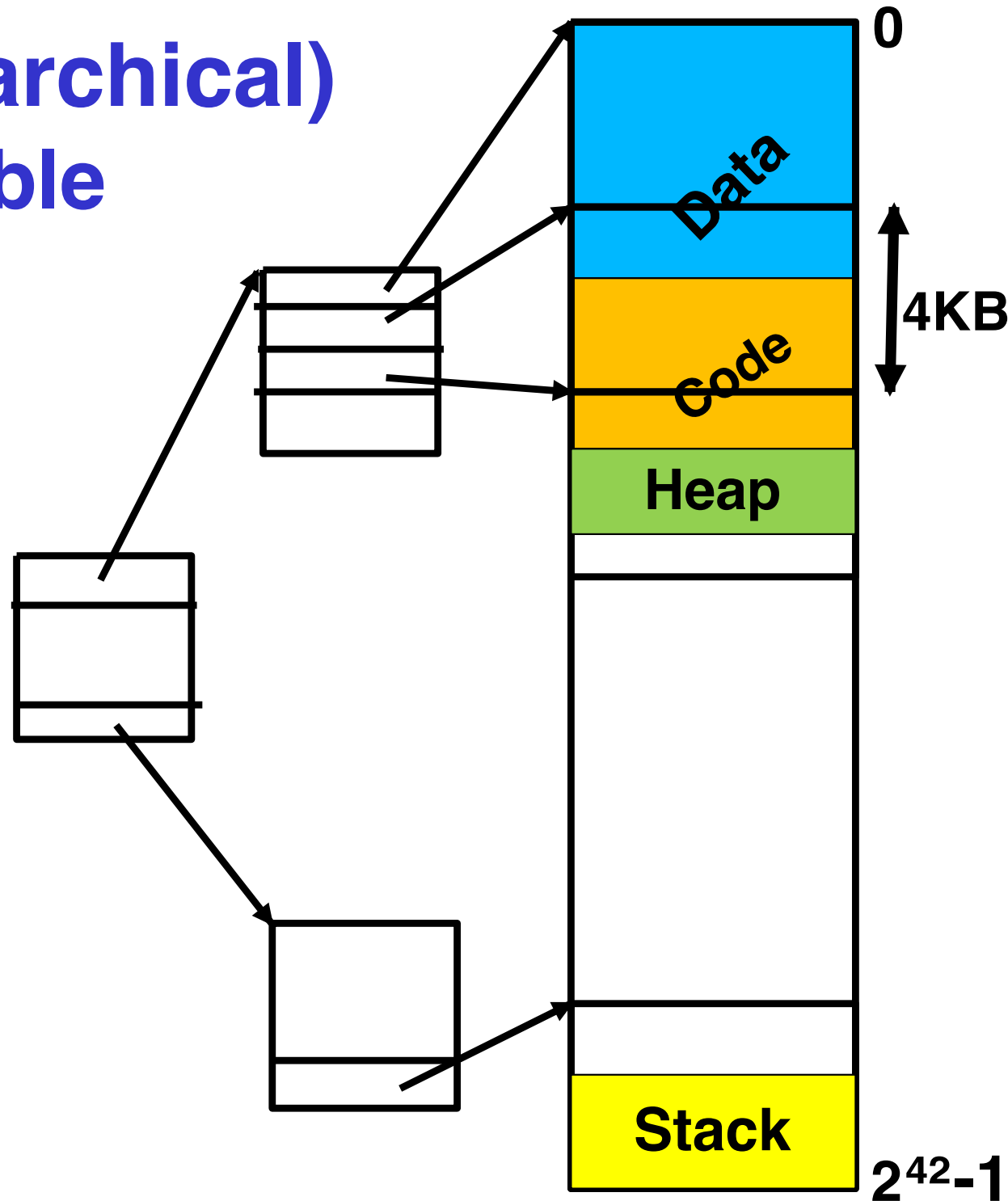
$$\downarrow \frac{2^{42}}{2^{12}}$$

Process is itself only occupying
 $4 * 4\text{KB} = 16\text{KB}$

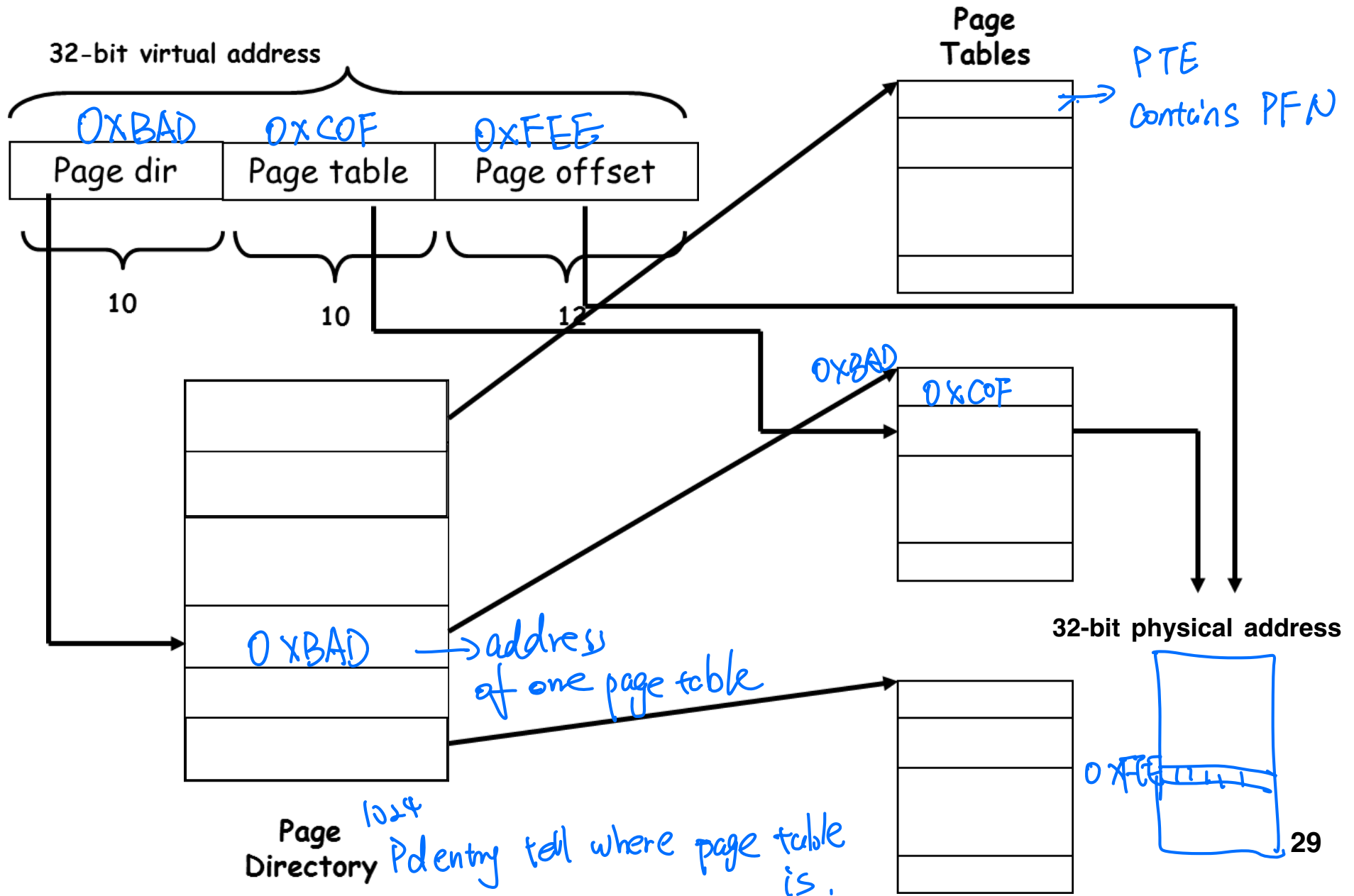


2-Level (Hierarchical) Page Table

Takes only 1 + 2
pages of PTEs
= $3 * 4\text{KB} =$
12KB



Example: A 2-level Page Table.



- **For example on SPARC, we have the following 3-level scheme, 8-bit index1, 6 bit index2, 6 bit index3, 12 bit offset.**
- **Note that only the starting location of the 1st-level indexing table needs to be fixed. Why?**
 - **The MMU hardware needs to lookup the mapping from the page-table.**
- **Exercise: Find out the paging configuration of x86**

Time for Address Translation

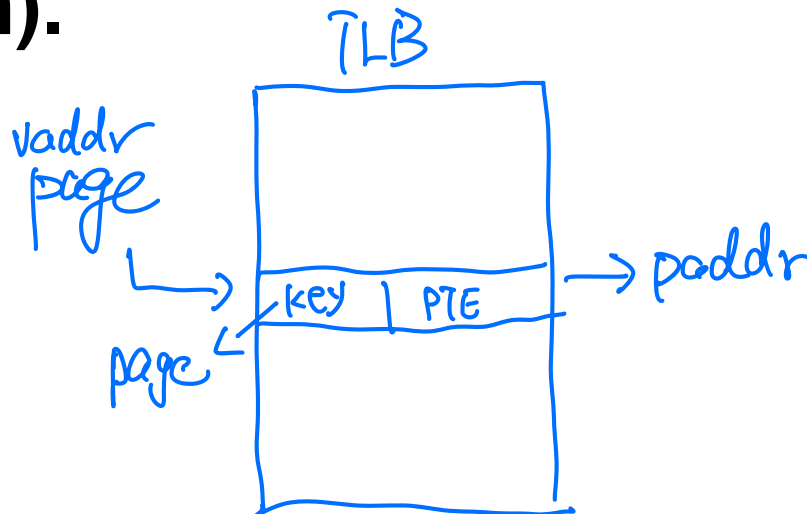
- **Address translation/mapping must be very fast! Why?**
 - Because it is done on every instruction fetch, memory reference instruction (loads/stores). Hence, it is in the critical path.
- **Previous mechanisms access memory to lookup the page-tables. Hence it is very slow!**
 - CPU-Memory gap is ever widening!
- **Solution: Exploit the locality of accesses.**

TLBs (Translation Look-Aside Buffers)

- **Typically programs access a small number of pages very frequently.**
- **Temporal and spatial locality are indicators of future program accesses.**
 - **Temporal locality [?] Likelihood of same data being re-accessed in the near future.**
 - **Spatial locality [?] Likelihood of neighboring locations being accessed in the near future.**
- **TLBs act like a cache for page-table.**

- **Typically, TLB is a cache for a few (64/128/256) Page-table entries.**
- **Given a virtual address, check this cache to see if the mapping is present, and if so we return the physical address.**
- **If not present, the MMU attempts the usual address translation.**
- **TLB is usually designed as a fully-associative cache.**
- **TLB entry has**
 - **Used/unused bits, virtual page number, Modified bit, Protection bits, physical page number.**

- Fraction of references that can be satisfied by TLB is called “*hit-ratio(h)*”.
- For example, if it takes 100 nsec to access page-table entry and 20 nsec to access TLB,
 - average lookup time = $20 * h + 100 * (1 - h)$.



Address Translation Steps

- Virtual address is passed from the CPU to the MMU (on instruction fetch or load/store instruction).
- Parallel search of the TLB in hardware to determine if mapping is available.
- If present, return the physical address.
- Else MMU detects miss, and looks up the page-table as usual. (NOTE: It is not yet a page-fault!)
- If page-table lookup succeeds, return physical address and insert mapping into TLB evicting another entry.
- Else it is a page-fault.

