CSC 112: Computer Operating Systems
Lecture 8

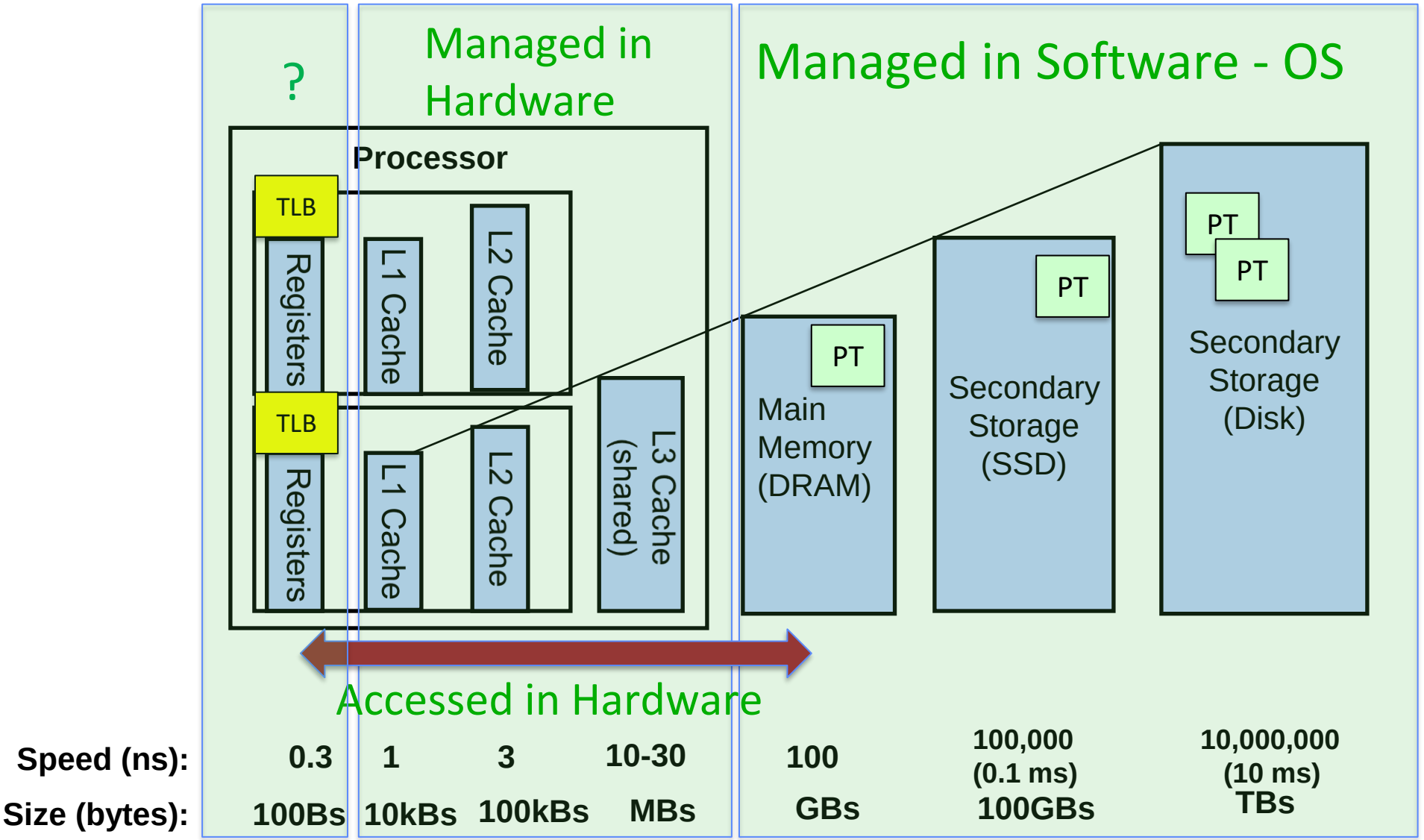
Memory System II: Paging

Department of Computer Science,
Hofstra University

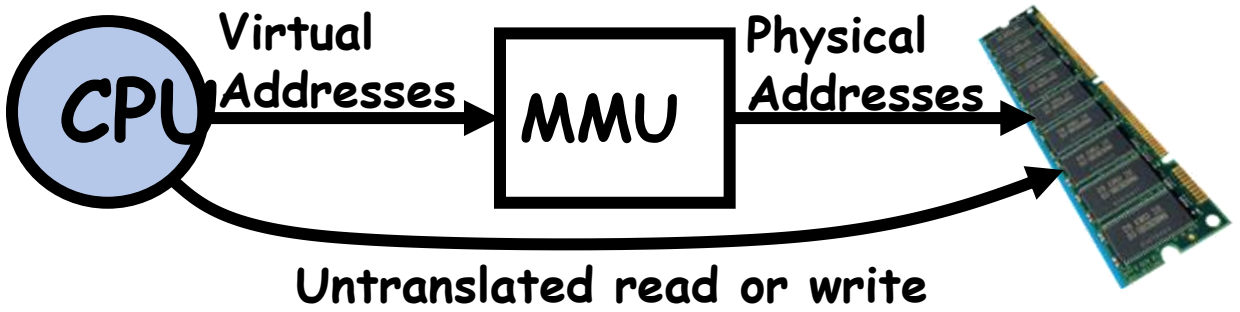
Outline

- Paging
- Page Translation
- Page Table
- Table Lookaside Buffer (TLB)
- Multi-level paging
- Page Swapping
- Page Replacement

Typical Memory Hierarchy



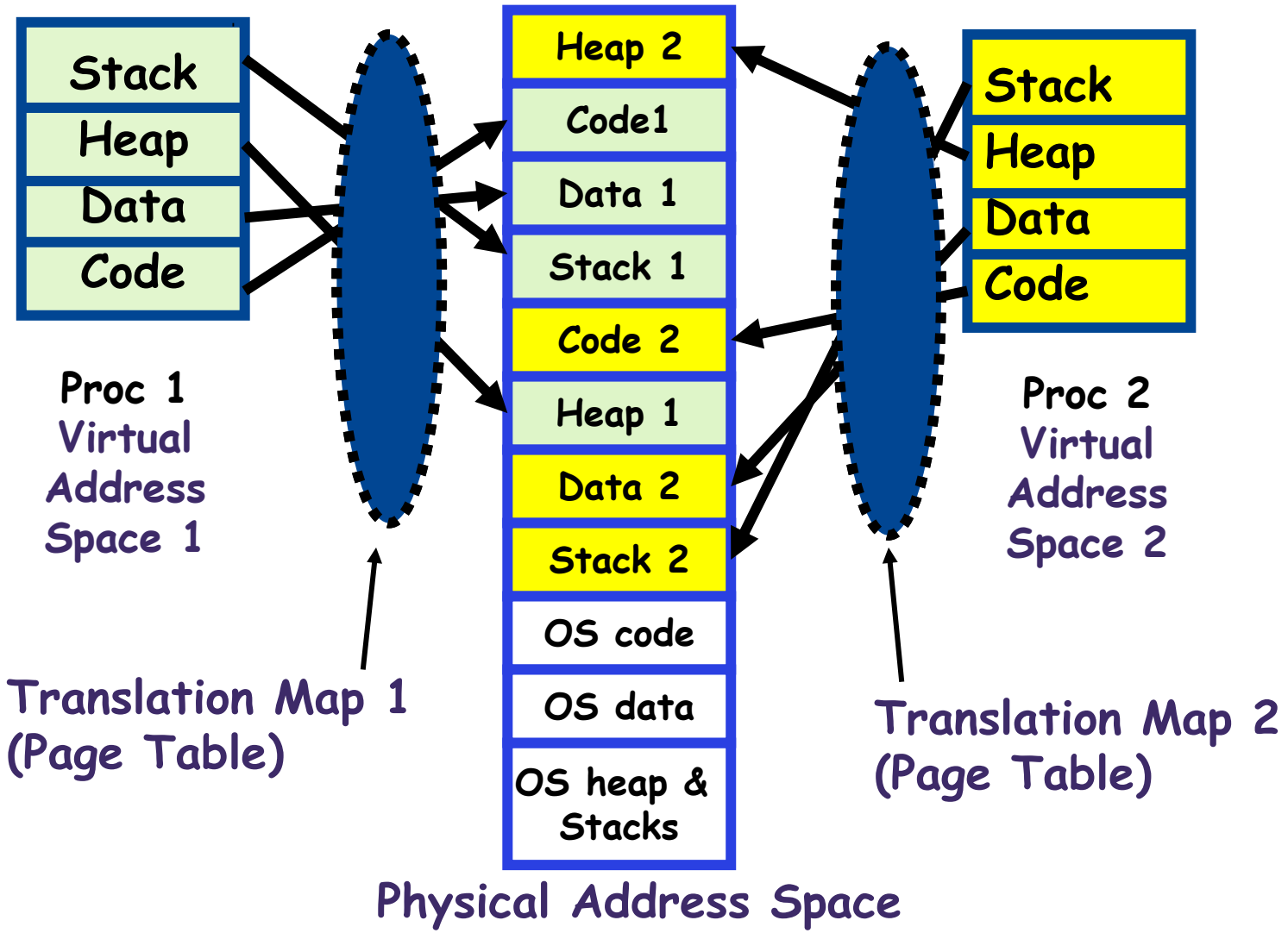
Two Views of Memory



- Two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Memory management unit (MMU) converts between the two views
  - Kernel accesses physical memory directly without translation
- Translation helps to implement protection
  - The same virtual address in different processes is mapped to different physical addresses, hence different processes cannot read/write each other's memory
  - Every program can be linked/loaded into same region of user address space
  - Isolation achieved through translation, not relocation

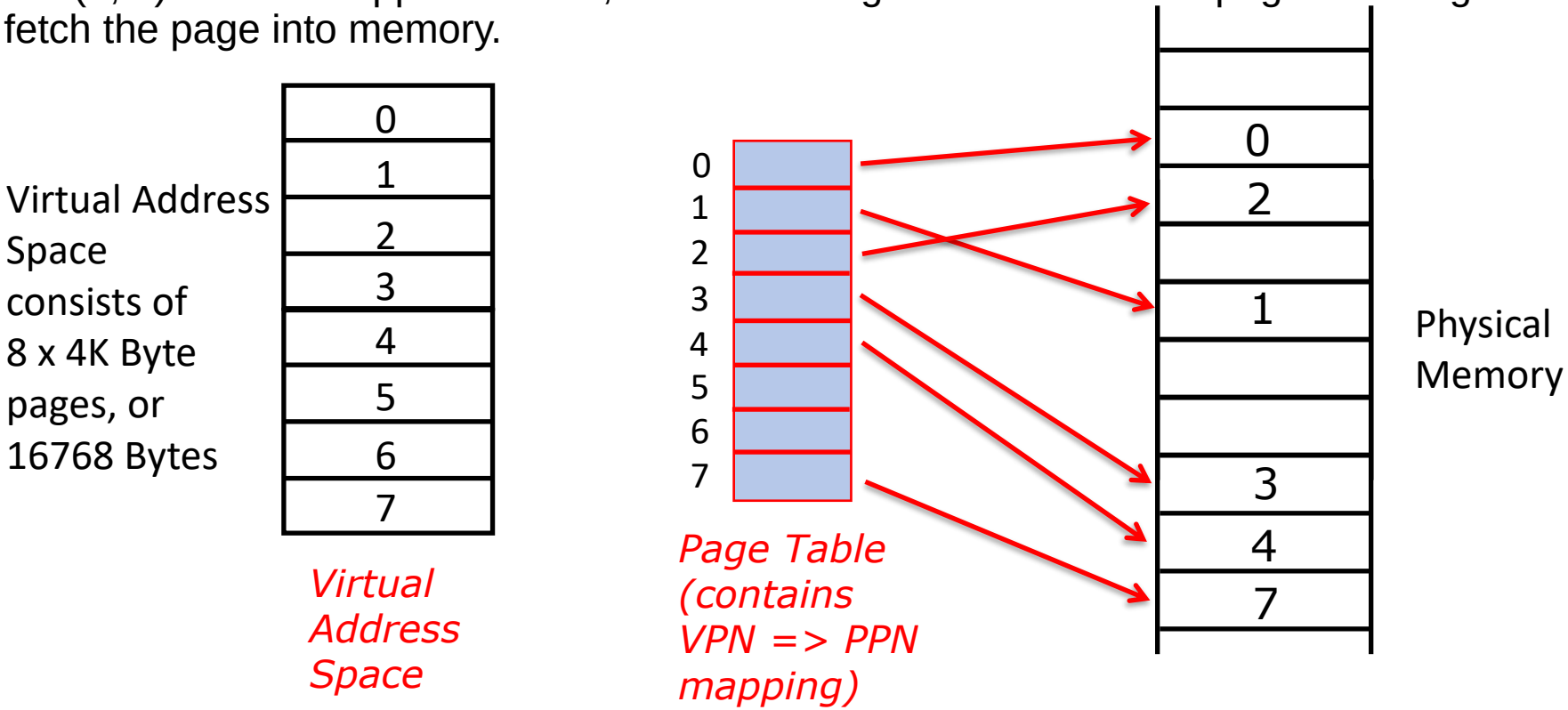
Paging

- Divide physical memory into **fixed-sized** blocks, called **physical page frames (**or simply **frames)**
- Divide virtual memory into blocks of the same size called **virtual pages (**or simply **pages)**
- Set up a **page table** to map from pages to frames
- Need both OS and hardware support (MMU)



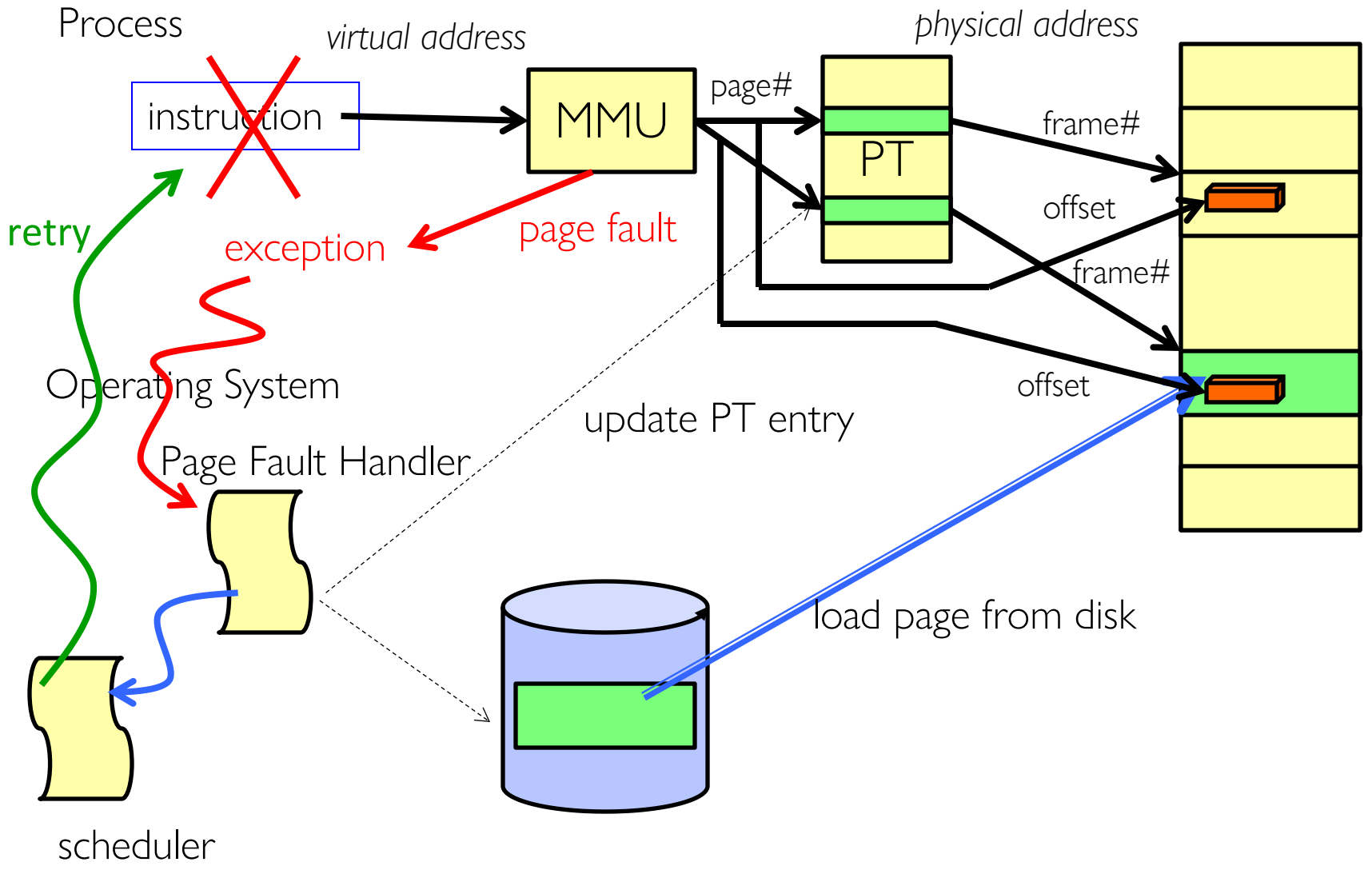
Paging Example

- Suppose page size is 4 KB, and the virtual address space has 8 pages
- Page Table maps from Virtual Page Number (VPN) to Physical Page Number (PPN)
  - PPN also called Page Frame Number (PFN).
  - Some VPNs (5, 6) are not mapped to PPN, and accessing them will cause a page fault to go to disk and fetch the page into memory.



Page Table makes it possible to store the pages of a process non-contiguously in physical memory.

Steps in Handling Page Faults



Paging Benefits

- **Flexibility:** Supporting the abstraction of address space effectively
  - Don’t need assumption how heap and stack grow and are used.

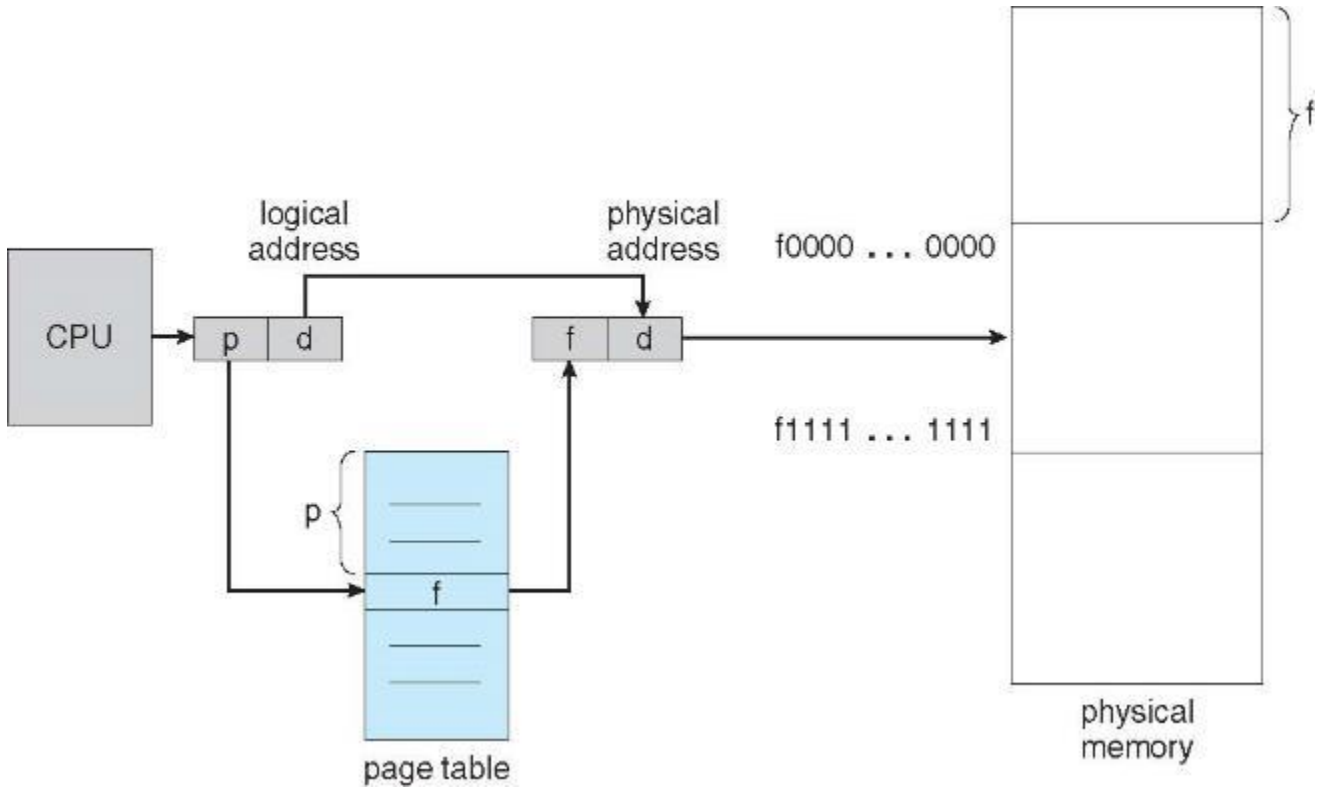
- **Simplicity**: ease of free-space management
  - The page in virtual address space and the physical page frame are the same size.
  - Easy to allocate and keep a free list

Page Translation

- A virtual address is split into **two parts**
  - **Virtual page number** (or **Page number**) – used as an index into a page table which contains base address of each page in physical memory
    - High bits to indicate page number
  - **Offset** – combined with base address to define the physical memory address within the page
    - Low bits to indicate offset
- Given virtual address space 2^m and page size 2^n Bytes, **n** bits for offset and **m-n** bits for page number
  - Only need to translate the page number to determine where the physical page is
  - Page offset determines page size, which is the same for both virtual and physical memory
    - Page offset refers to byte address within a memory page (e.g., 4KB); compare with byte offset in caching, which refers to byte address within a cache/memory block (e.g., 8 Bytes)

Page Table

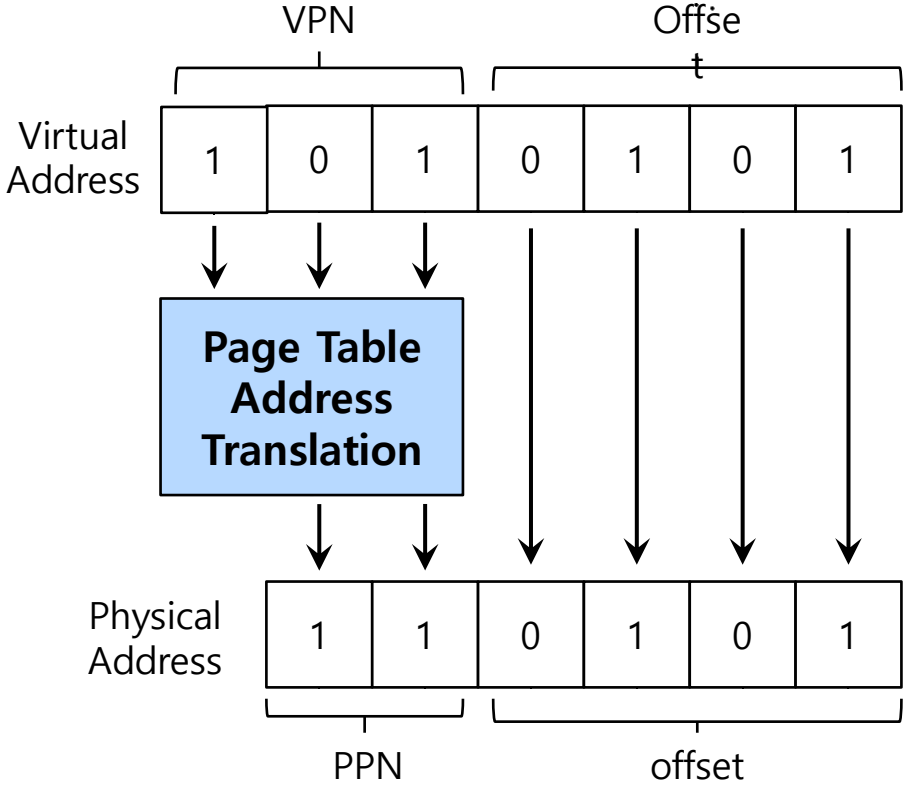
- **Page Table**
  - Keeps track mapping of virtual to physical addresses
  - Part of Process Control Block (PCB) for each process, kept in main memory

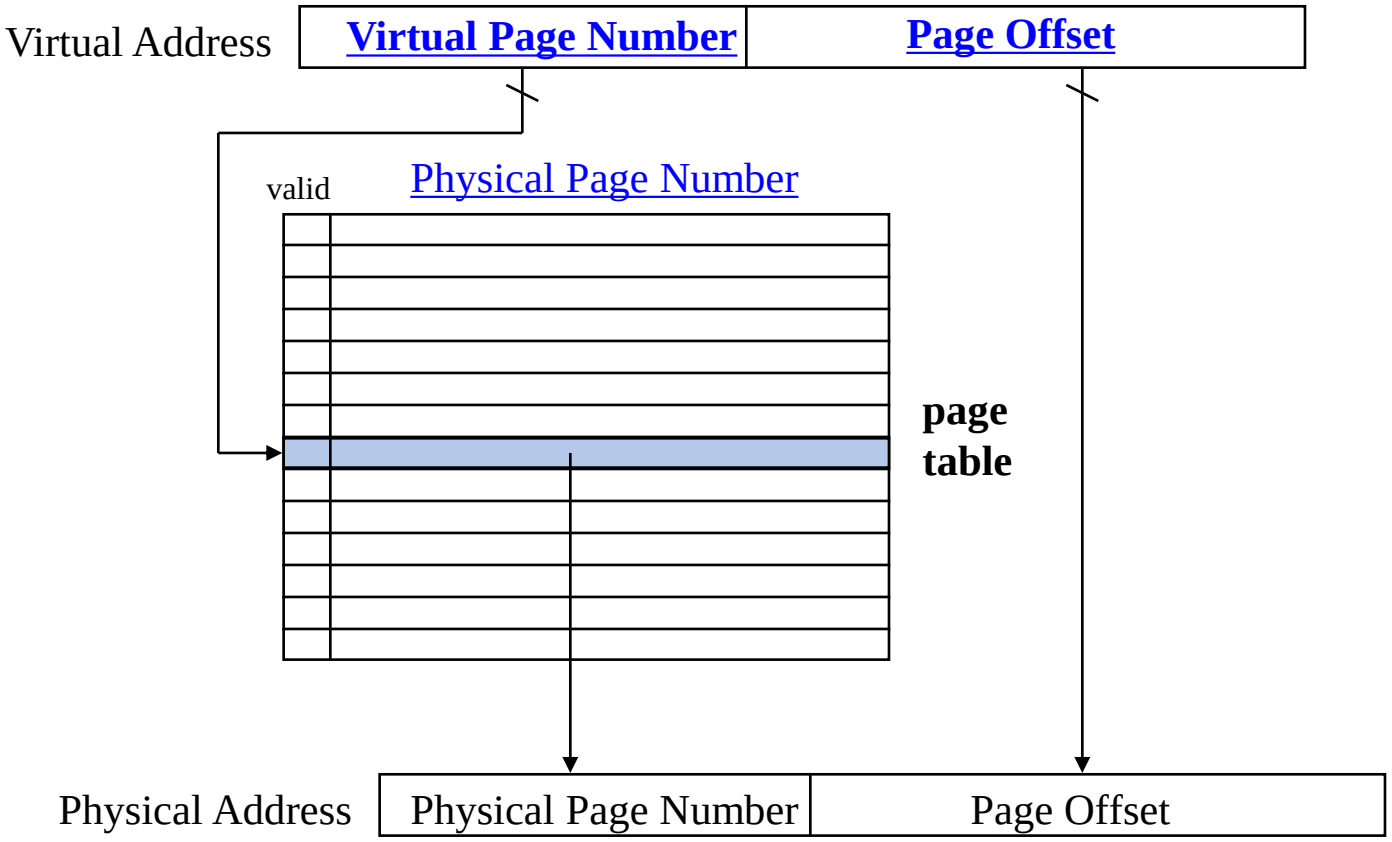



```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

Page Translation Mechanism

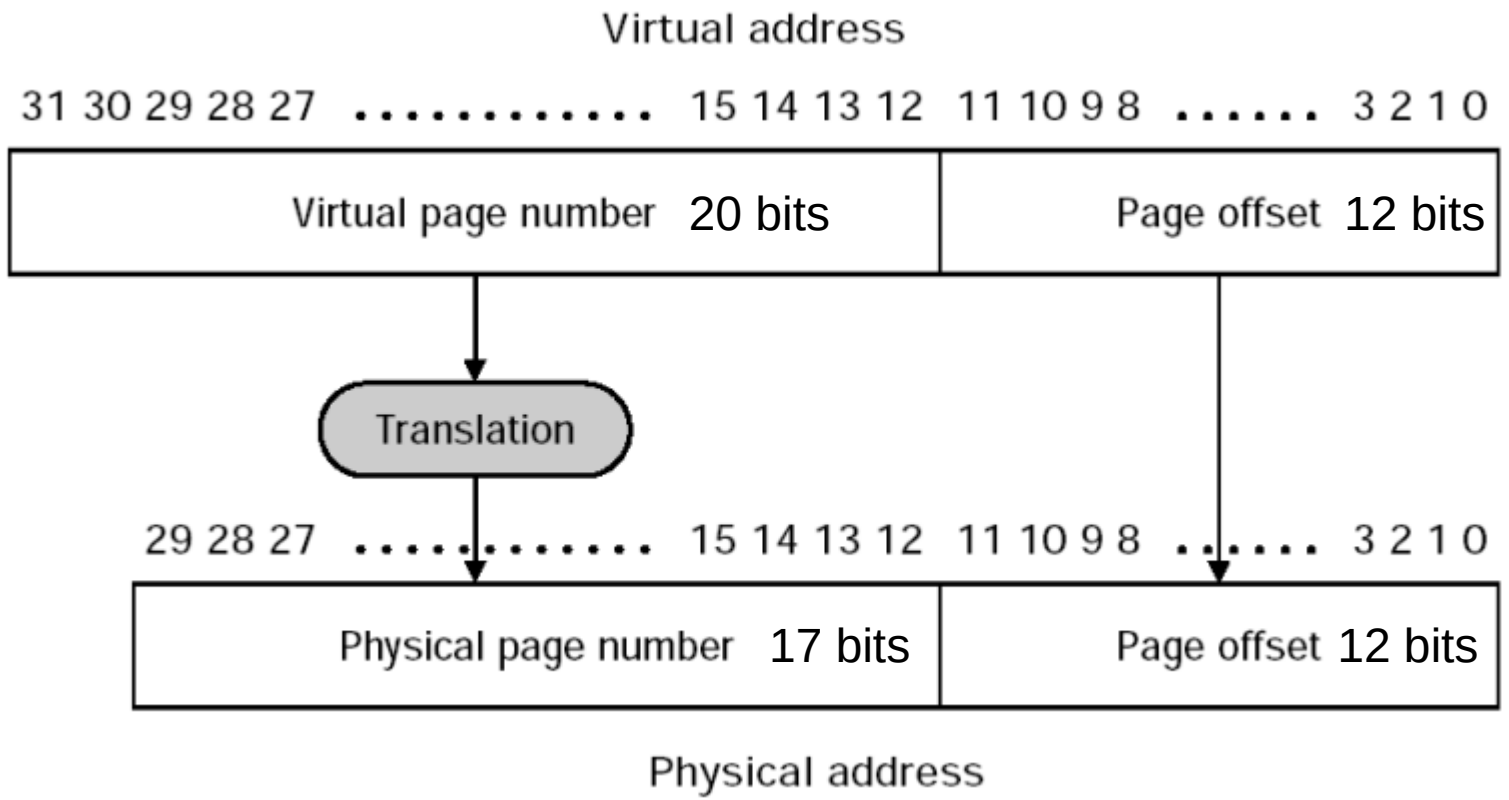
- Generally, VPN has more bits than PPN, since physical memory is smaller (# virtual pages ≥ # physical page)



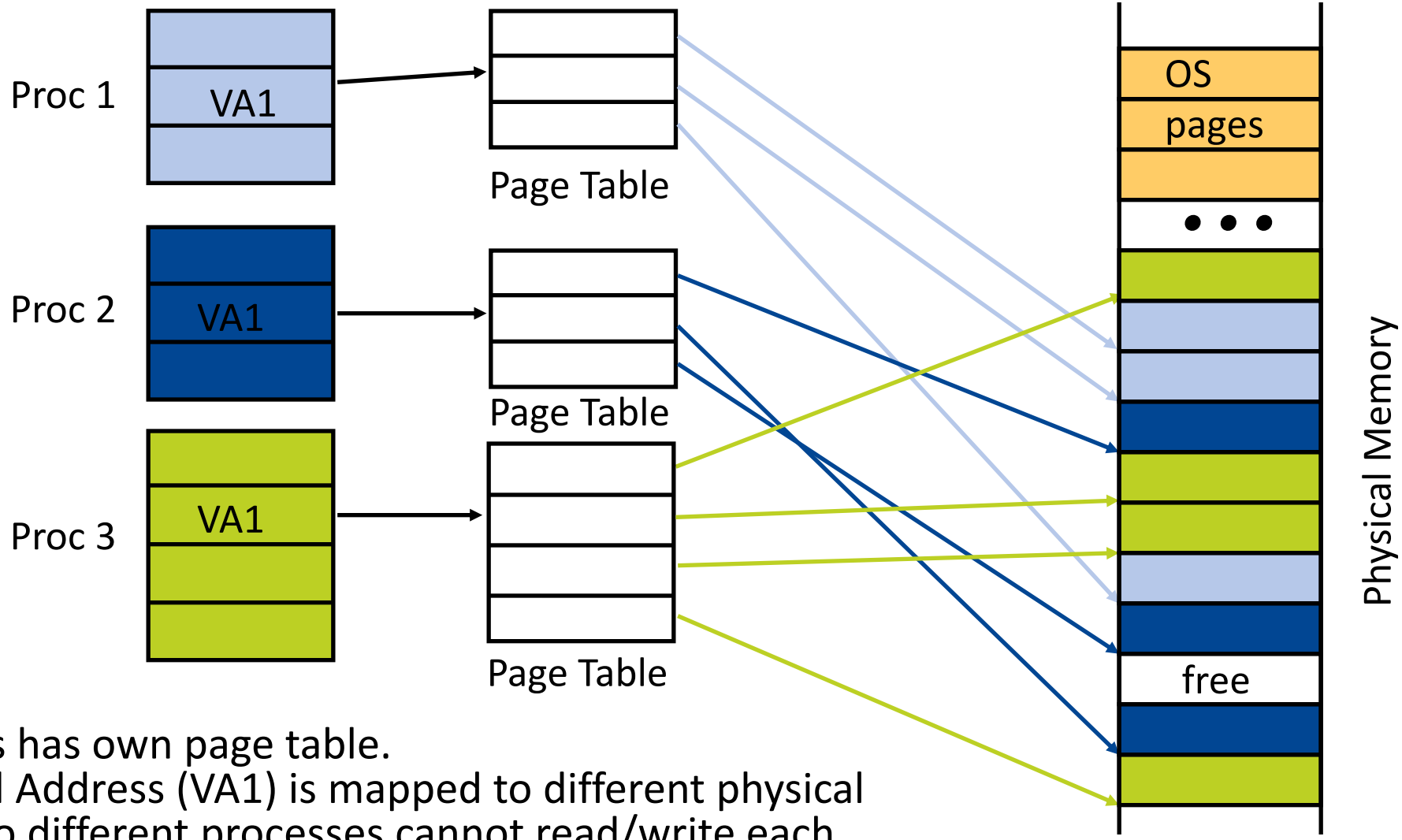


Page Translation Example

- Virtual Address: 32 bits, virtual address space 2^32=4GB
- Physical Address: 29 bits, physical address space 2^29=0.5GB
- Page size: 2^12=4KB, hence offset is 12 bits
  - VPN has 32-12=20 bits, PPN has 29-12=17 bits



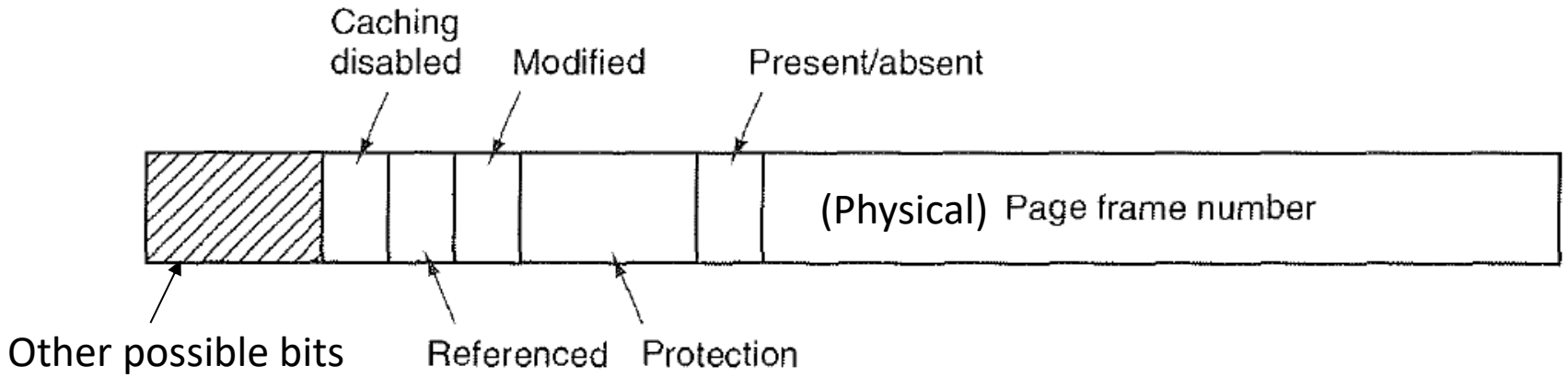
Separate Address Space per Process



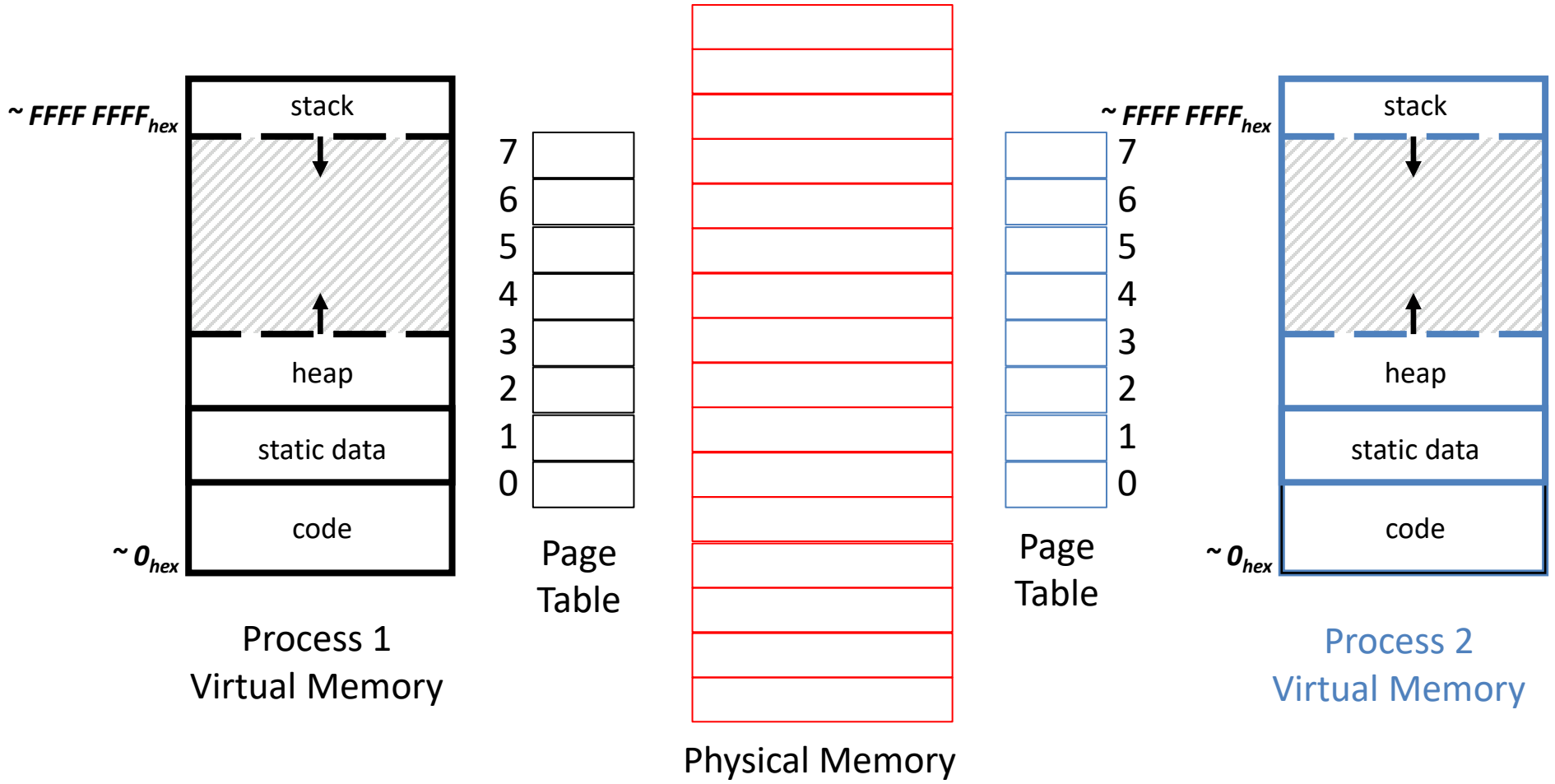
Each process has own page table.
Same Virtual Address (VA1) is mapped to different physical addresses, so different processes cannot read/write each other's memory

Page Table Entry (PTE)

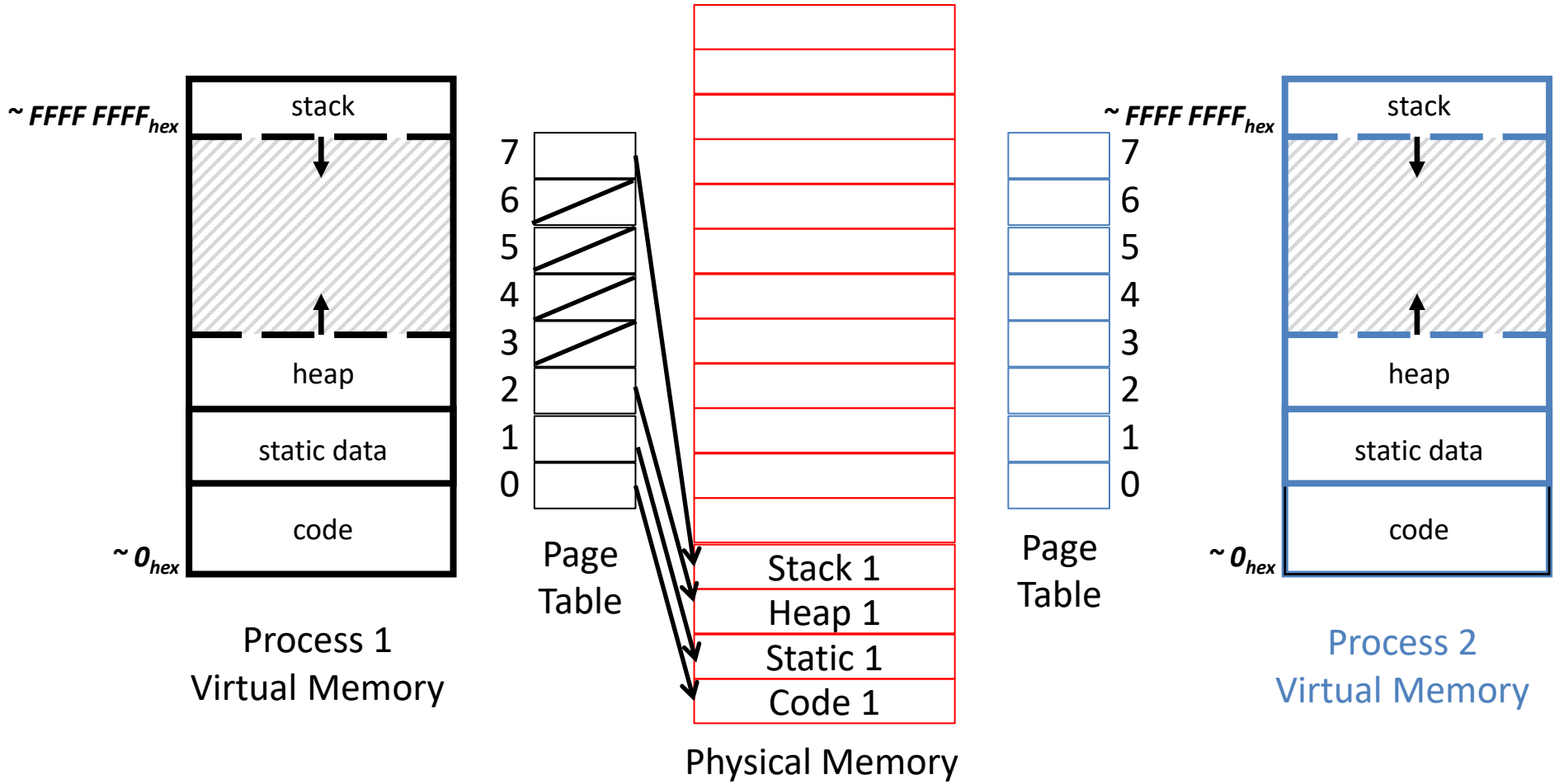
- A PTE contains:
  - *Physical Page Number (PPN),* also called *Page Frame Number*
  - *Present/absent bit,* also called *Valid bit.* If this bit is 1, the page is in memory and can be used. If it is 0, the page is not currently in memory. Accessing a page table entry with this bit set to 0 causes a page fault to get page from disk.
  - *Protection bits* tell what kinds of access are permitted on the page. 3 bits, one bit each for enabling read, write, and execute.
  - *Modified (M) bit,* also called *dirty bit,* is set to 1 when a page is written to
  - *Referenced (R) bit,* is set whenever a page is referenced, either for reading or writing.
    - M and R bits are useful to page replacement algorithms
  - *Caching disabled bit,* important for pages that map onto device registers rather than memory



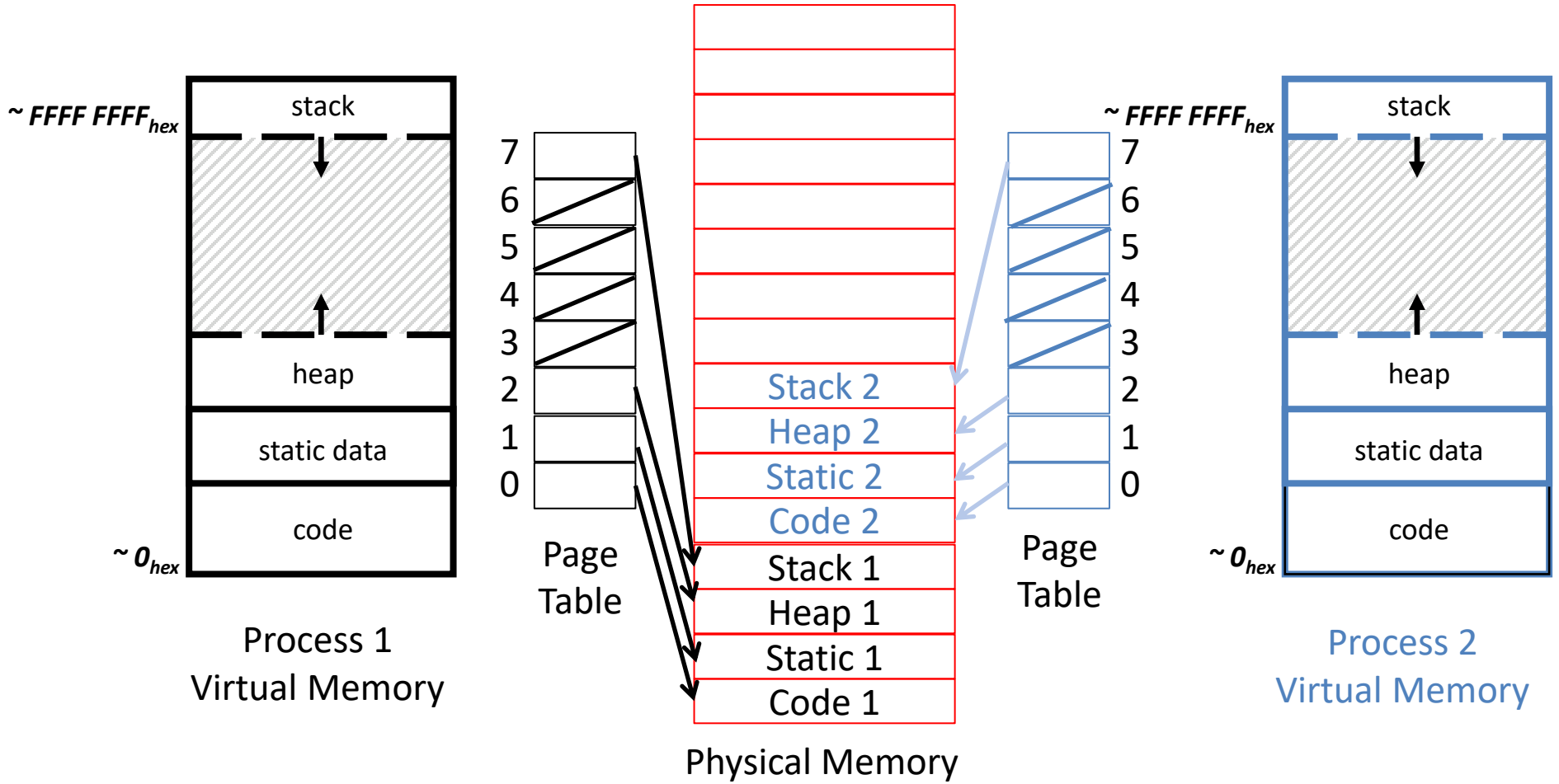
Paging example: initial state



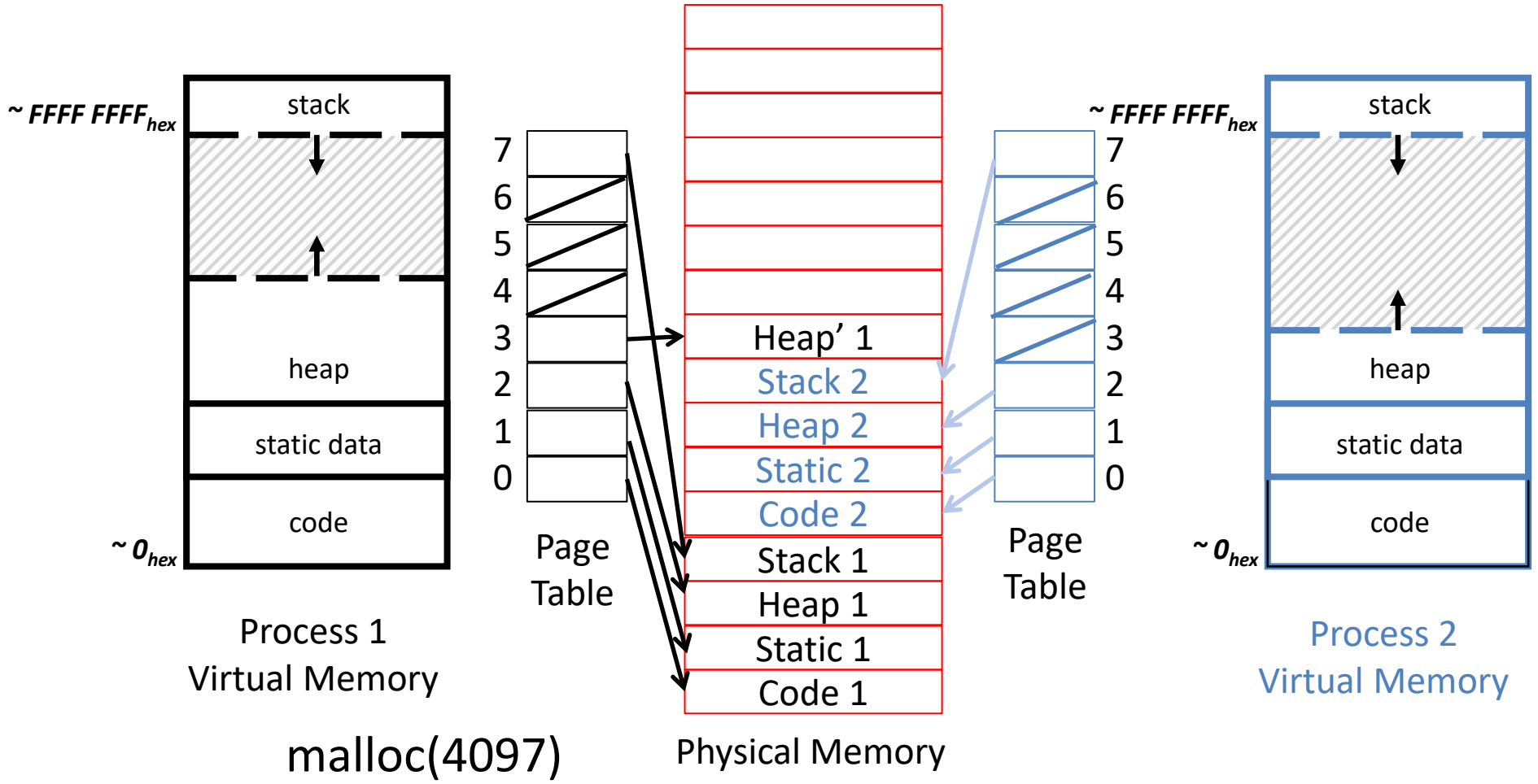
Paging example: Process 1 starts to run



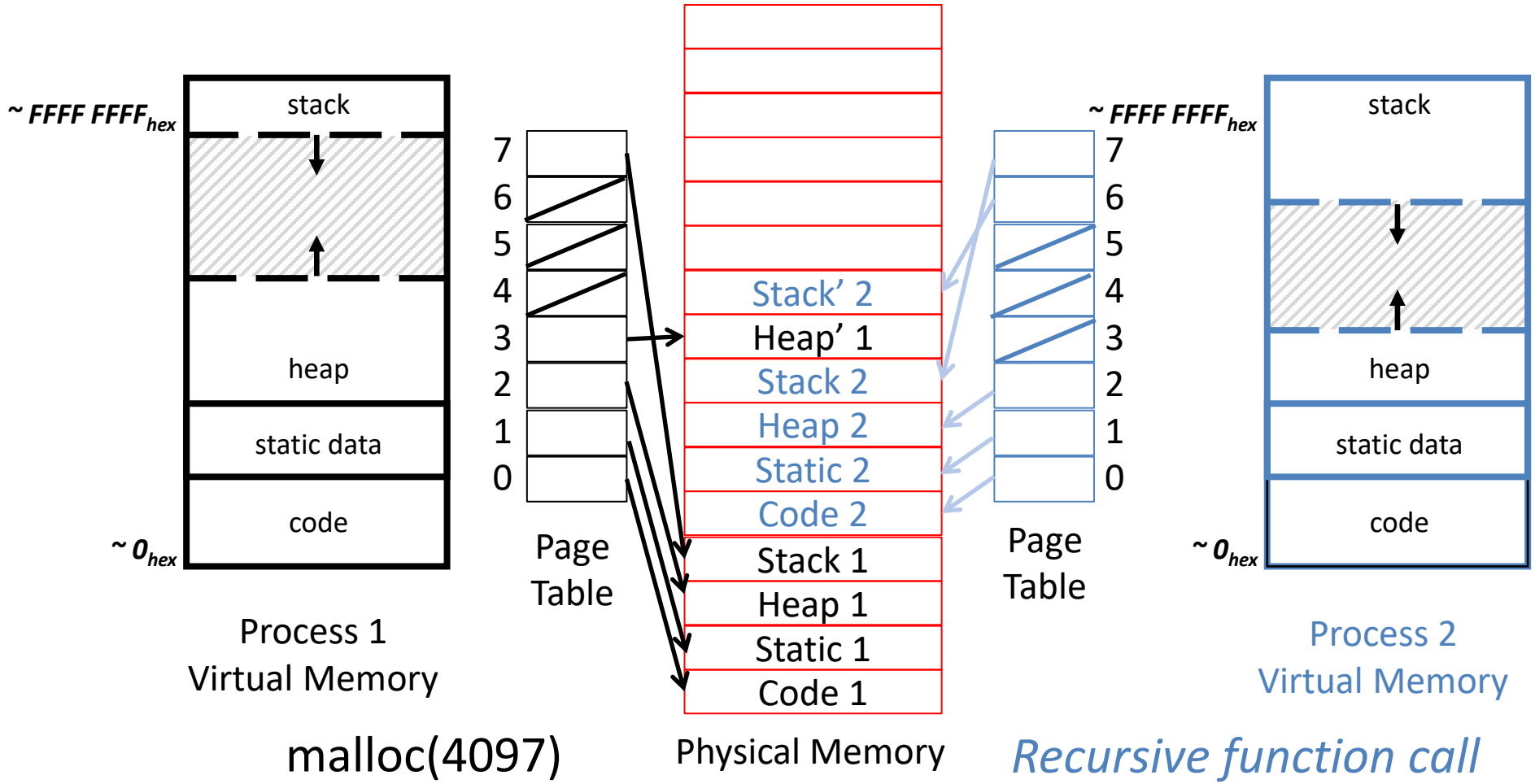
Paging example: Process 2 starts to run



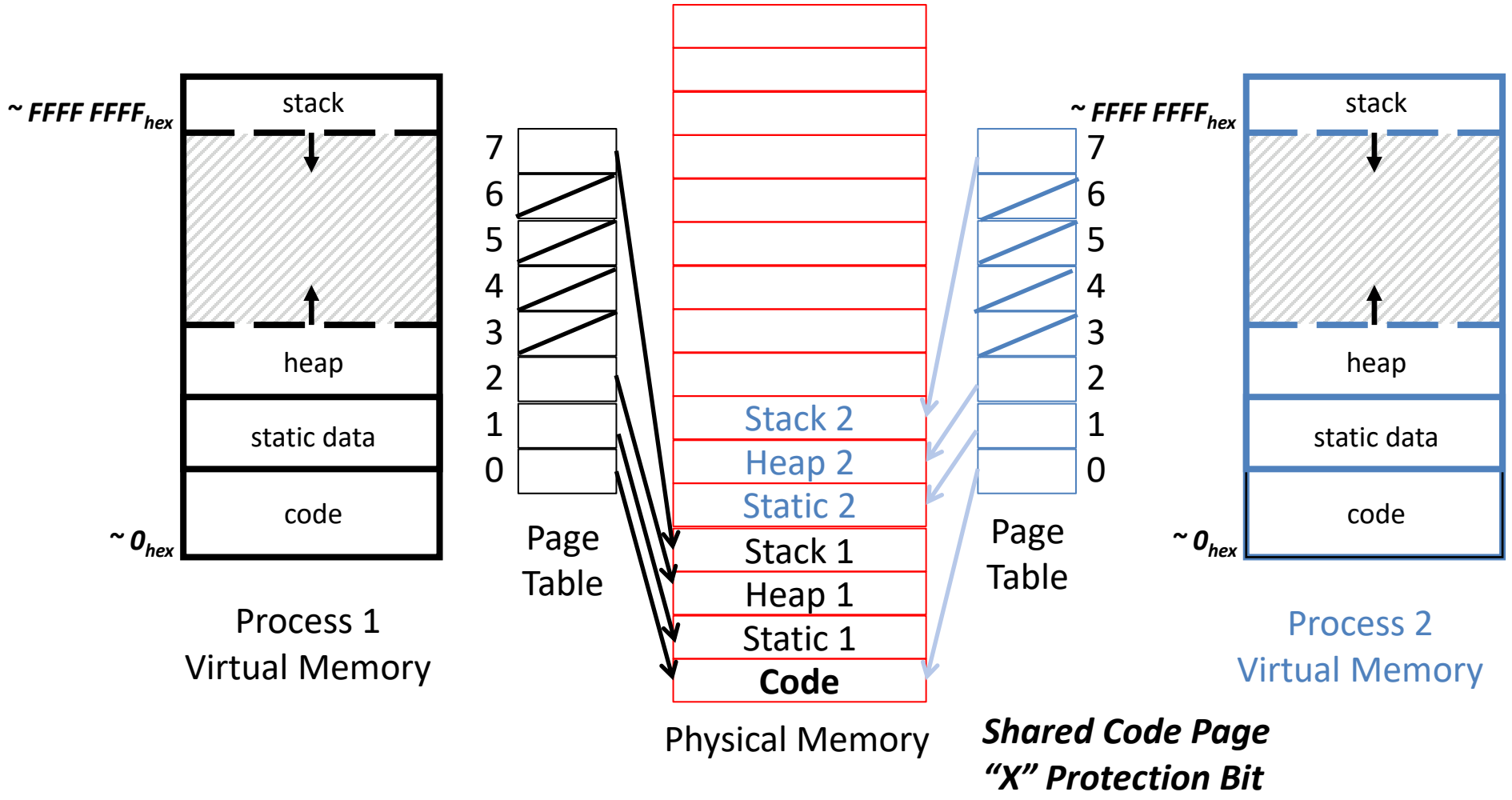
Paging example: Process 1 dynamic memory allocation on its heap with malloc()



Paging example: Process 2 dynamic memory allocation on its stack with Recursive function call

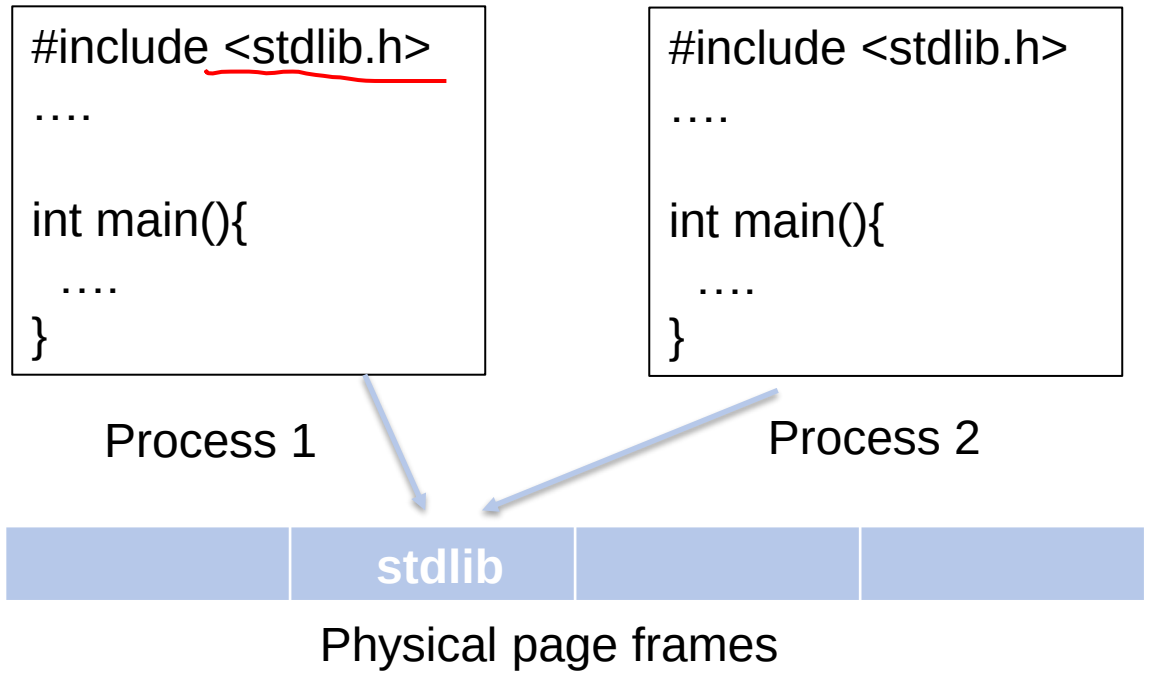


Paging example: controlled sharing of Code (Instruction) page

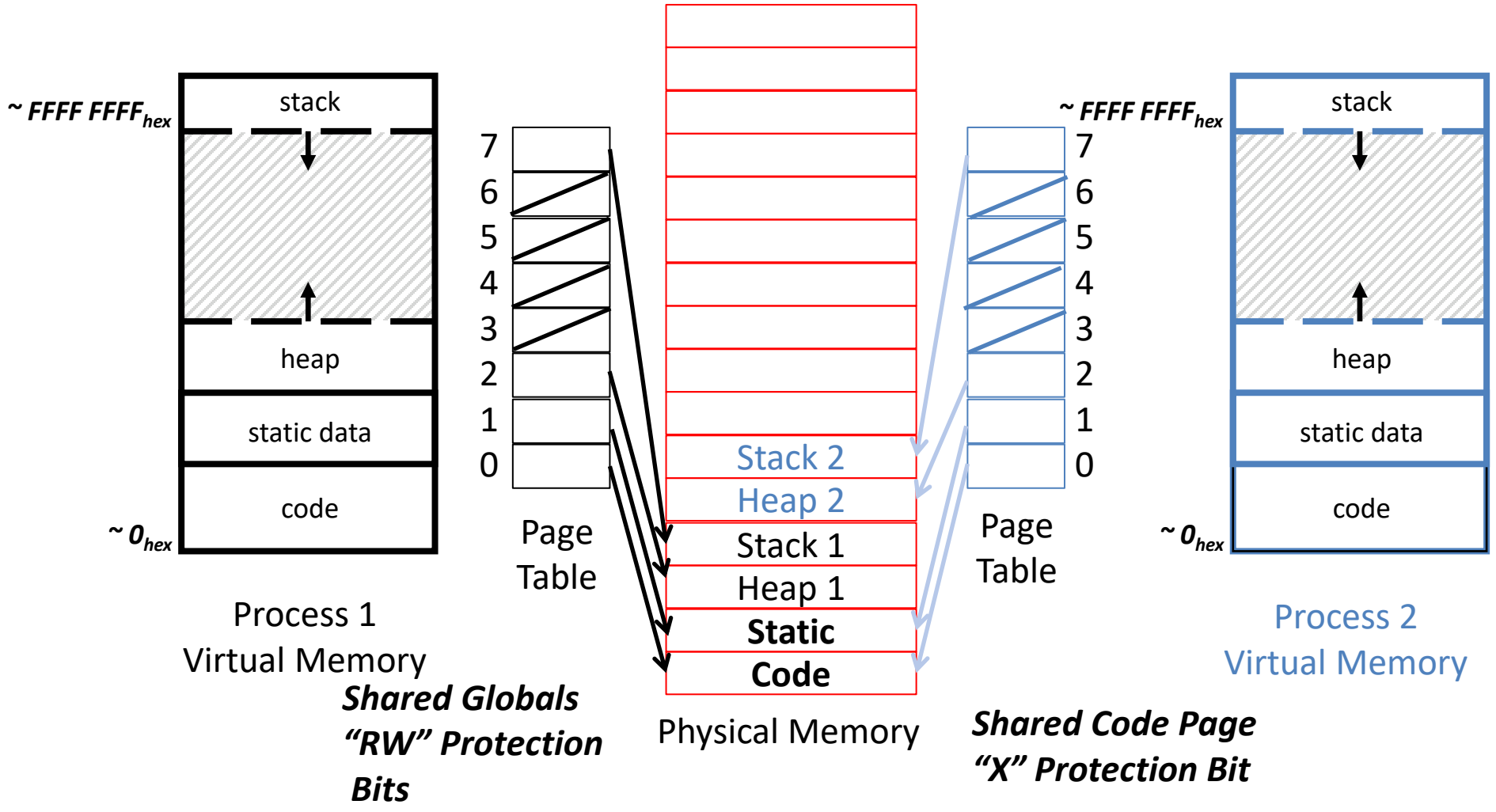


Code Page Sharing Use Case

- Shared code pages are memory pages that contain executable code and are shared among multiple processes. They are typically read-only to ensure that the code remains consistent across all processes accessing it. This approach is used to save memory and improve efficiency, as only one copy of the code resides in physical memory, while multiple processes map it into their virtual address spaces. Examples:
  - Shared libraries (e.g., dynamic link libraries or .so files).
  - Common executables like shells or system utilities.
  - Reentrant code, which can be safely executed by multiple processes simultaneously without modification.
- Implementation:
  - OS maps the same physical page containing the code into the virtual address spaces of multiple processes.
  - Protection mechanisms ensure that these pages are marked as non-writable to prevent accidental or malicious modification.
- The "X" bit in PTE stands for “Execute” permission.
  - It can be used to enforce policies like W^X (Write XOR Execute). This policy ensures that a memory page cannot be both writable and executable at the same time, which helps prevent certain attacks.



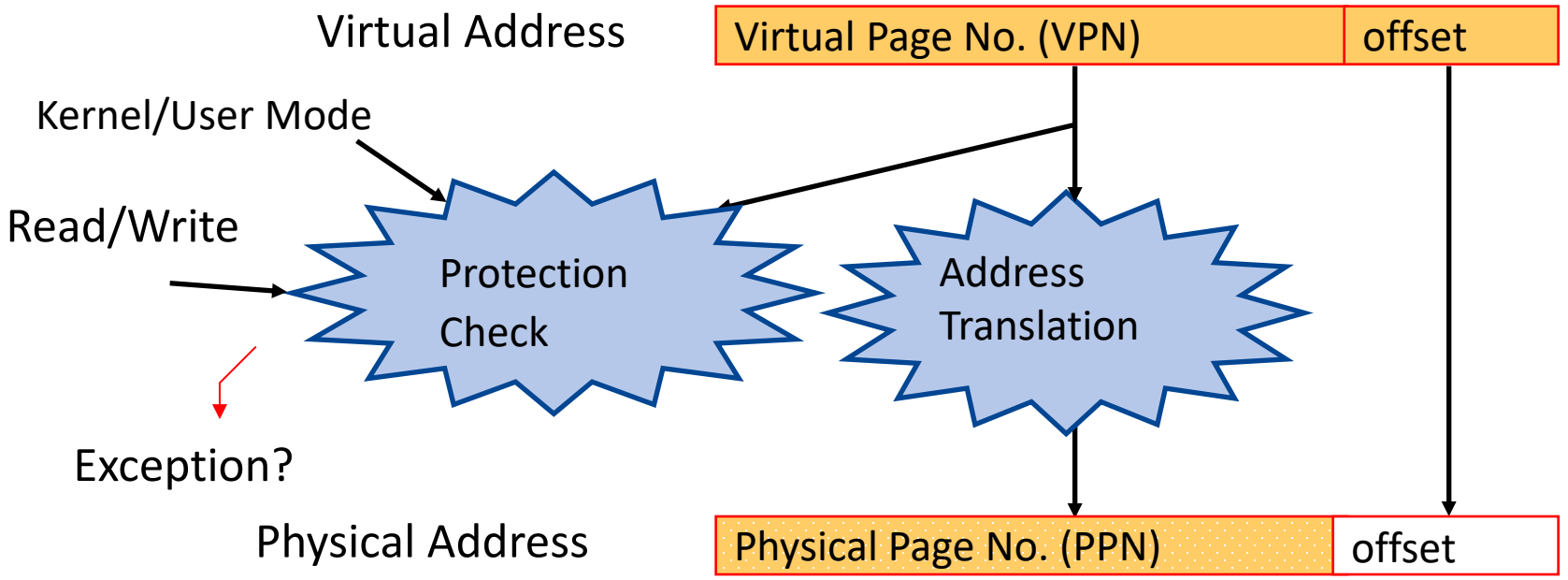
Paging example: controlled sharing of Global Data (Static Data) page



Data Page Sharing Use Case

- Shared global variables refer to data that can be accessed by multiple processes or threads. In most systems, global variables are private to each process by default, but they can be explicitly shared using mechanisms such as shared memory. Examples:
  - Inter-Process Communication: Shared globals in a writable memory region enable efficient communication between processes without copying data.
  - Shared Libraries: While code in shared libraries is typically read-only and executable, global data sections may require RW permissions if they store modifiable state.
  - Kernel Data Structures: The kernel may use RW-protected shared memory regions for managing system-wide states accessible by user-space applications under strict controls.
- Implementation:
  - Shared global variables are typically placed in shared memory regions, which allow multiple processes to access the same physical memory.
- The "RW" (Read/Write) bit in PTE determines whether a page can be written to or only read.
  - If it is set, the page can be both read and written.
  - If it is not set, the page is read-only. Any attempt to write to it will trigger a protection fault (e.g., segmentation fault).

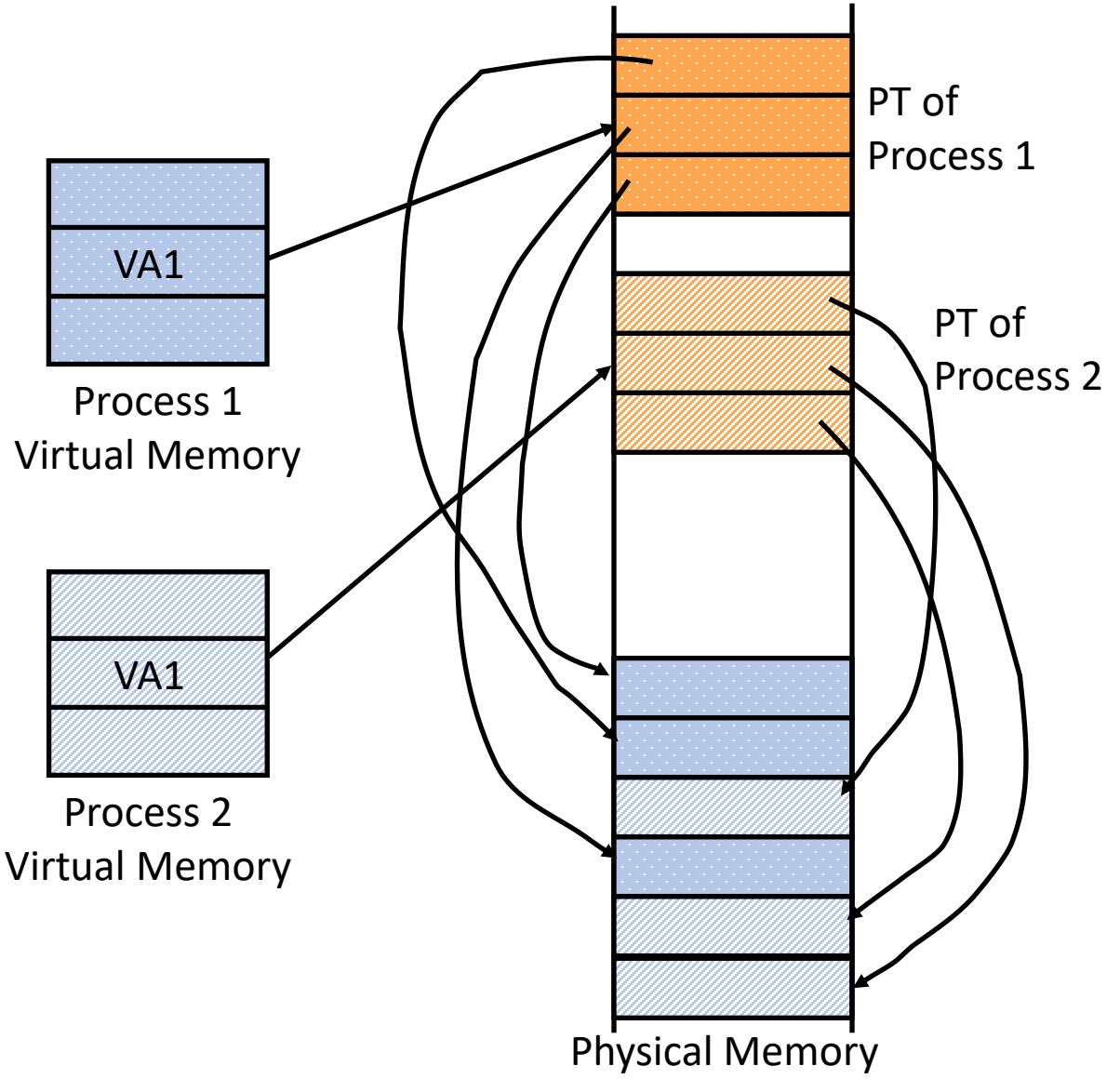
Address Translation & Protection



Every instruction and data access needs address translation and protection checks

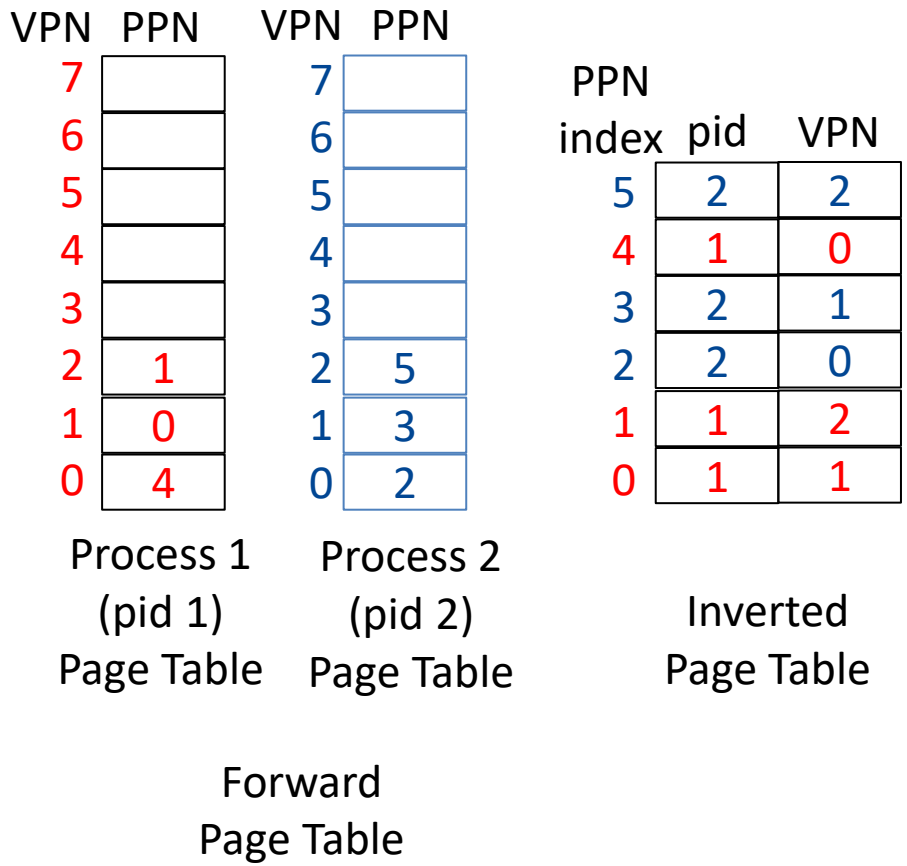
Where Should Page Tables Reside?

- Space required by the page tables is proportional to the address space, number of users, …
  - e.g., virtual address space 2^{32} Byte, page size 2^{12}=4KB
  - Number of pages: $2^{32}/2^{12}=2^{20}$, i.e., 2^{20} PTEs per process
  - If each PTE is 4 Bytes, then size of one page table is: 4 * 2^{20} = 4MB
- Each process has its own page table. Suppose 50 processes running on a system, then total size of all 50 page tables is 200MB!
- Too large to keep in cache. Keep in main memory
  - Keep physical address of page table in Page Table Base Register.
  - One access to retrieve the physical page address from table.
  - Second memory access to retrieve the data word
  - *Doubles* the number of memory references!
    - Use TLB to avoid the double memory access (later)
- What if Page Table doesn't fit in memory?
  - Multiple levels of page tables, or segmentation + paging (discussed later)



Inverted Page Table

- A single Inverted Page Table shared among all processes
- Indexed by PPN instead of VPN
  - One entry per PPN; # entries is equal to # PPNs, which is generally much smaller than #VPNs
- Each PTE contain the pair <process ID, VPN>.
  - It tells us which process is using this page, and which virtual page of that process maps to this physical page
- To translate a Virtual Address, current process ID and the the VPN are compared against each entry, scanning the table sequentially.
  - Reverse lookup from table entry to table index.
  - If a match is found, its index in the inverted page table is the PPN, e.g., pid=1, VPN=2 → PPN=1
  - If no match is found, a page fault occurs, e.g., pid=1, VPN=6 → page fault
- Pros:
  - Reduces memory overhead since there is only one global table for all processes.
  - Scales better with large physical memories compared to hierarchical page tables.
- Cons:
  - Table lookup is inefficient since finding a match may require searching the entire table. Poor cache locality because entries are scattered across the table.
  - Difficult to implement page sharing among processes.
- Processors that use IPT:
  - PowerPC, UltraSPARC, Itel IA-64 (Itanium)



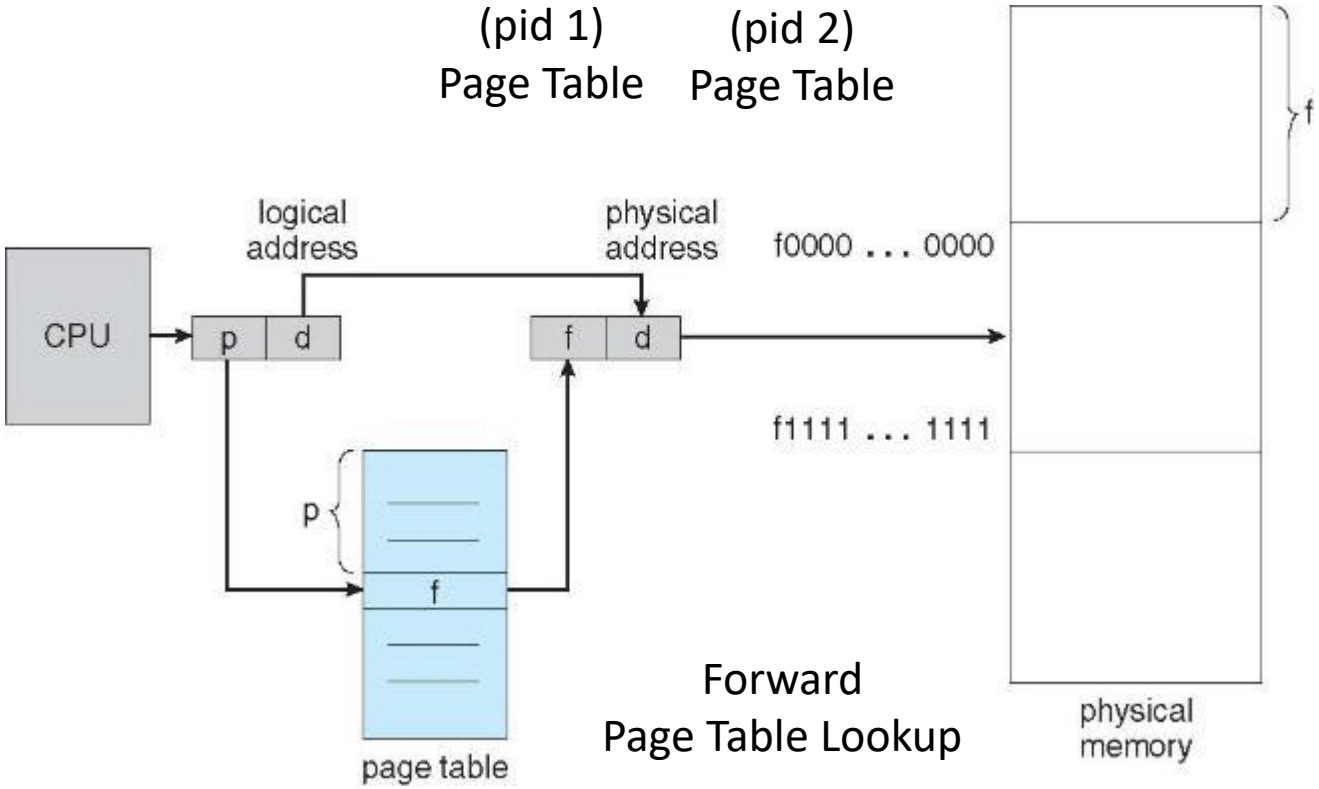
Forward Page Table

VPN	PPN
7	
6	
5	
4	
3	
2	1
1	0
0	4

Process 1 (pid 1) Page Table

VPN	PPN
7	
6	
5	
4	
3	
2	5
1	3
0	2

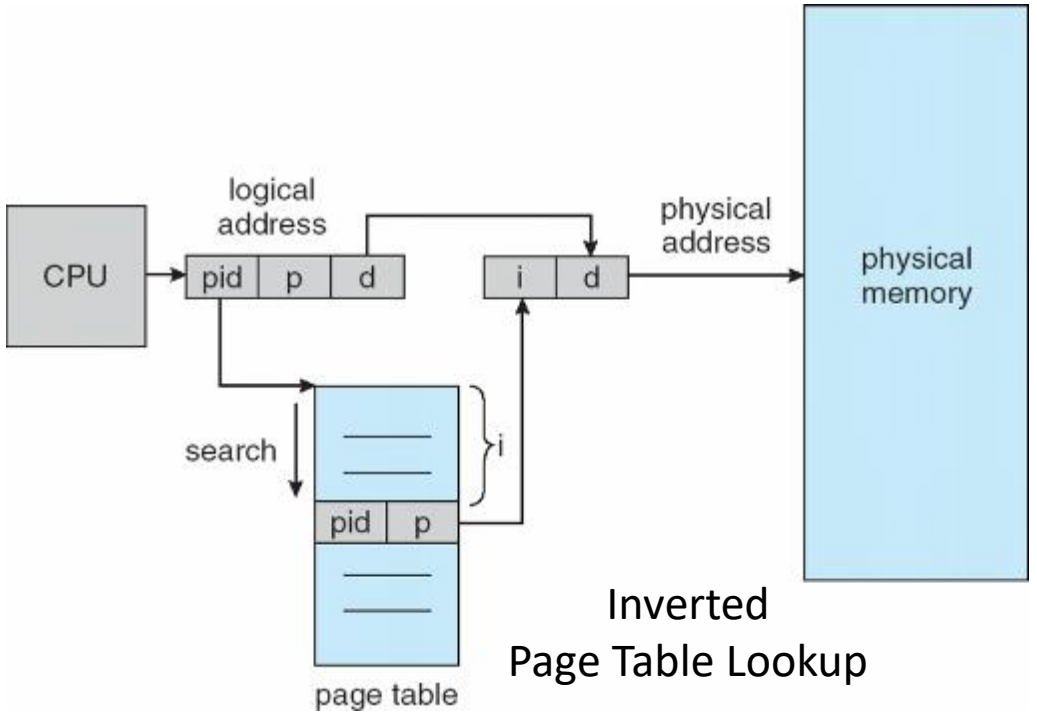
Process 2 (pid 2) Page Table



Forward Page Table Lookup

Inverted Page Table

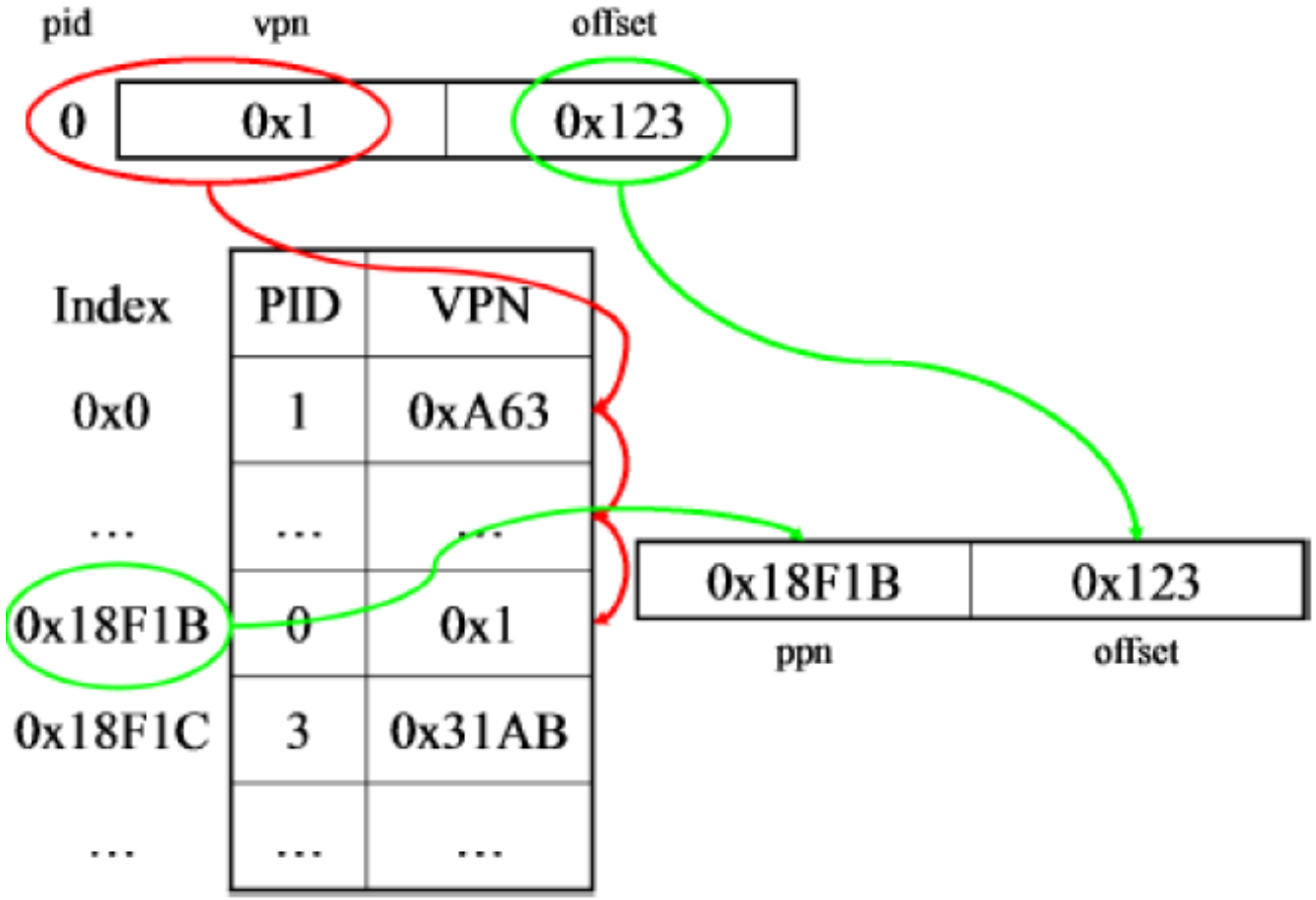
PPN index	pid	VPN
5	2	2
4	1	0
3	2	1
2	2	0
1	1	2
0	1	1



Inverted Page Table Lookup

Inverted Page Table Lookup Example

- pid=0, VPN=0x1 → PPN=0x18F1B



Inverted page table, EZCSE
https://www.youtube.com/watch?v=9pXnMfKq7Hw

Performance Implication of Paging

- The issue of paging:
  - Page Table stored in main memory
  - Fetch the translation from in-memory page table
  - Explicit load/store access on a memory address

- Every data/instruction access incurs **two** memory accesses
  - One for the page table
  - and one for the data/instruction

- Number of memory accesses is increased by **a factor of 2**!

Memory Accesses of Paging

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

VPN	PPN
0	3
1	2
2	8
3	5

Page Table stored at memory address 0x200C

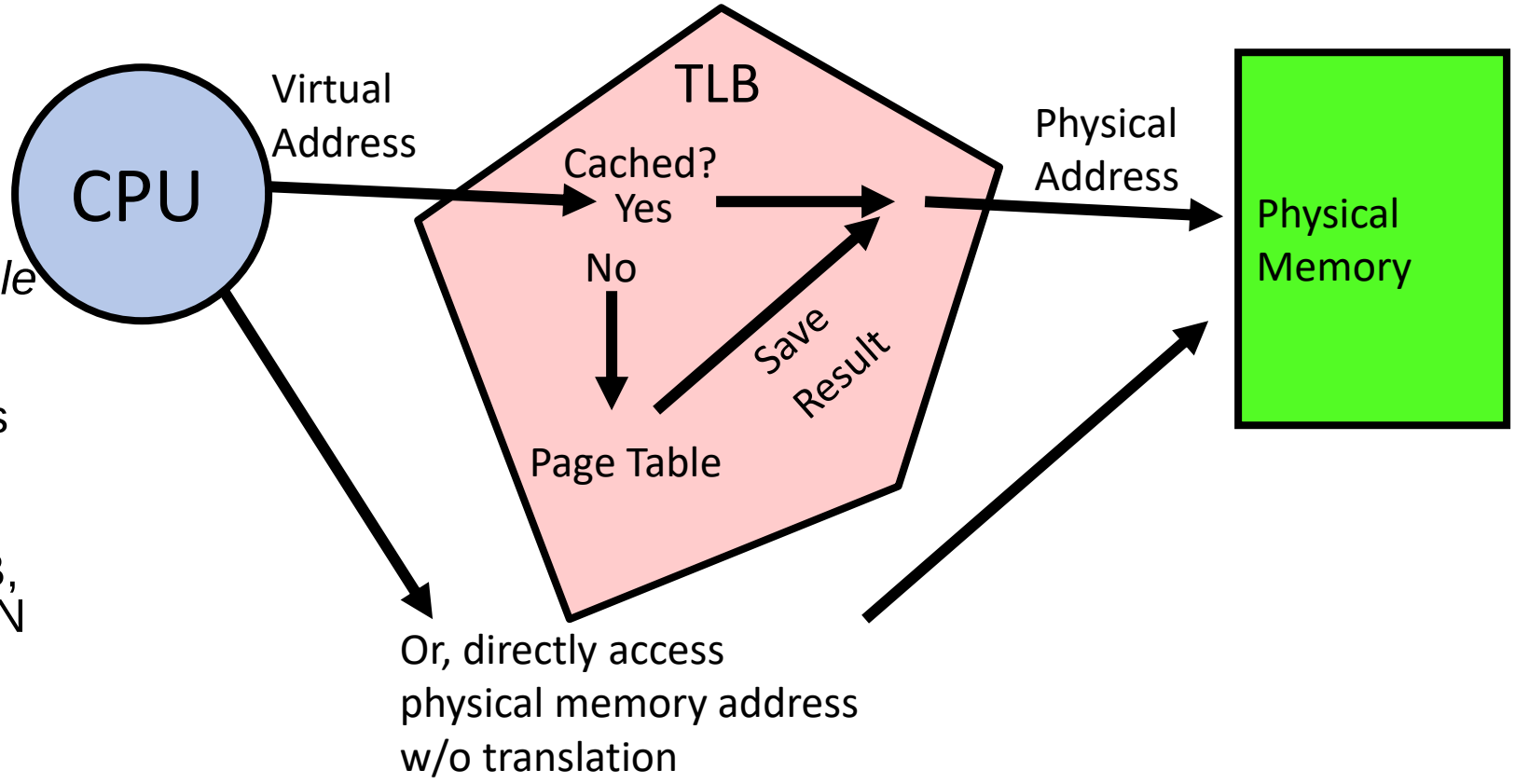
Virtual memory
load 0x3000
load 0x3004
load 0x3008
load 0x300C
…

Physical memory
load 0x200C
load 0x5000
load 0x200C
load 0x5004
load 0x200C
load 0x5008
load 0x200C
load 0x500C
…

- Page size: 2^{12}=4KB, hence offset is 12 bits. 16-bit virtual address space, hence VPN is 16-12=4 bits.
- Suppose a[N] is an array of ints. Each int is 4 Bytes, hence Virtual Memory addresses in the for loop has stride of 4: 0x3000, 0x3004, 0x3008, etc. Suppose physical address space is also 16 bits (in practice it is smaller), hence PPN is also 4 bits, and page table maps VPN=3 to PPN=5. Suppose Page Table for all VPNs accessed by this program fit within one physical page at physical memory address 0x200C
- Suppose the virtual page with VPN=3 has virtual memory address 0x3000, and the physical page with PPN=5 has physical memory address 0x5000
- Each loop iteration incurs two memory accesses!

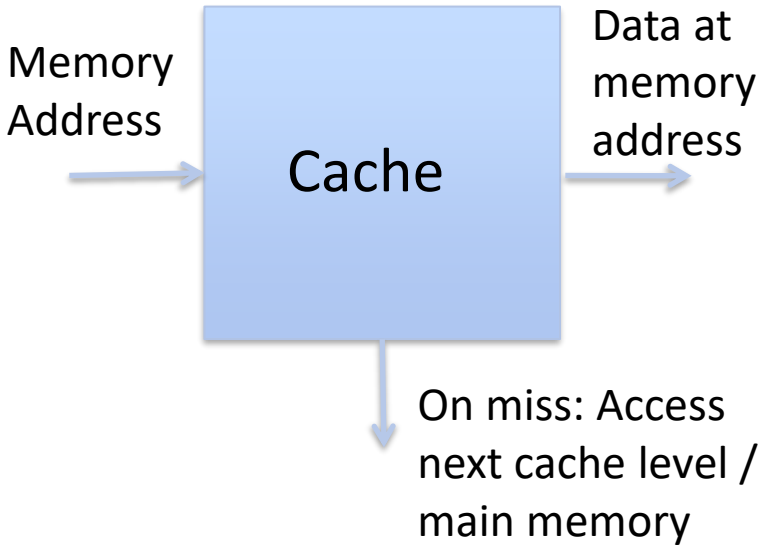
Translation lookaside buffer (TLB)

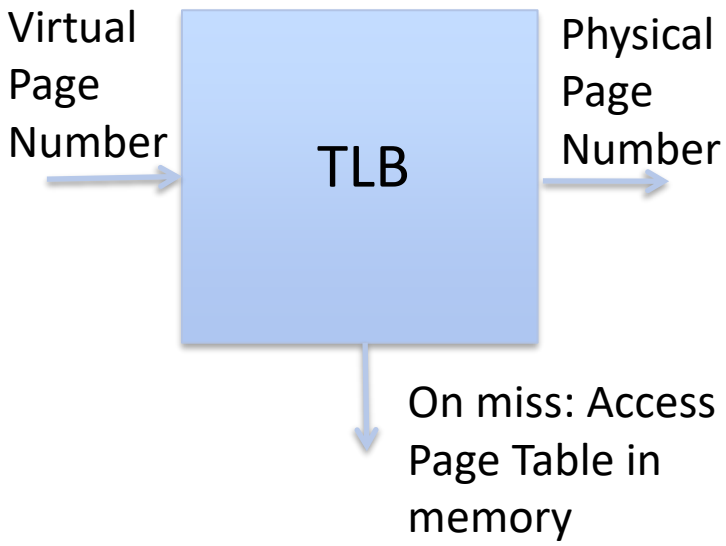
- TLB is a cache for entries in Page Table
  - Part of **Memory Management Unit** (MMU)
- For historical reasons, called *Translation Lookaside Buffer* (TLB)
  - More accurate name is *Page Table Address Cache;* should be small enough to fit in CPU cache.
  - Maps from VPN to PPN (same as Page Table)
- Memory reference with TLB
  - **TLB hit: VPN** is in cached in TLB, hence can get corresponding PPN w/o accessing Page Table
  - **TLB miss:** VPN is **not** in TLB**.** Access Page Table to get the translation to PPN, update the TLB entry with the translation



TLB is a Type of Cache

- TLB stores page table entries at Page Number granularity (not including the Byte Offset within a page), and each page is large (e.g., 4KB).
- Cache stores actual memory contents (instruction or data) at memory address granularity (including the Byte offset within a cache block).
- Hence TLB can be effective at a very small size (e.g.128-512 entries), and can fit within cache (L1 or L2) for fast access without touching memory.



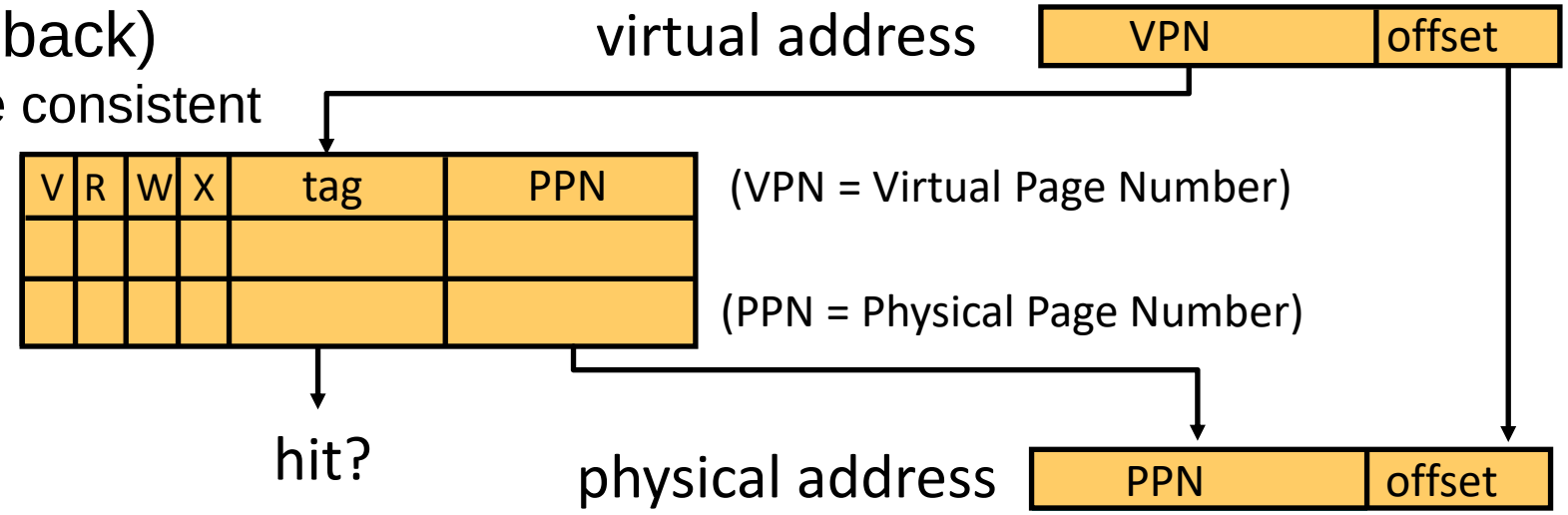


TLB Organization

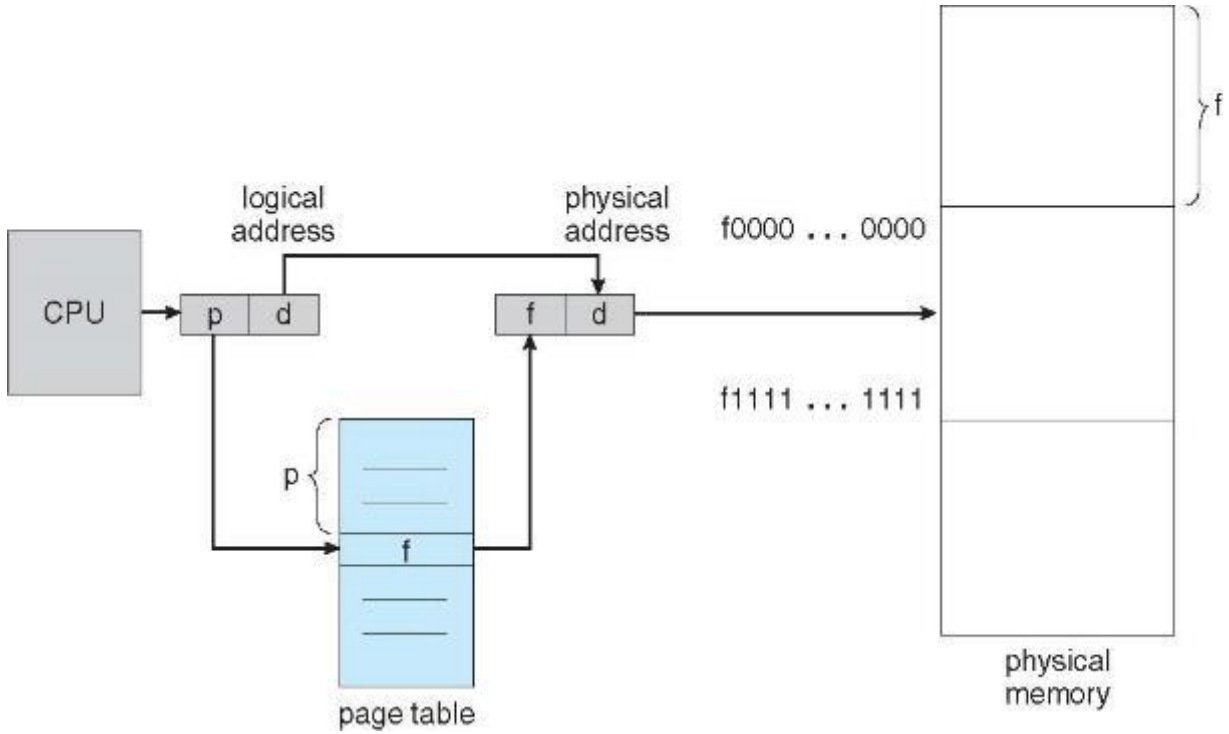
- TLB is usually fully-associative, but can also be set-associative
  - Since miss penalty is high (one memory access, similar to Last-Level Cache), FA or high associativity SA is adopted to minimize miss rate at the cost of slightly increased hit time. (Recall “Cache Design Considerations”.)
  - With FA, there is no set index bit, and the tag bits are the VPN. Any VPN entry can be anywhere in the TLB, and hardware searches entire TLB in parallel to find a tag match.
- TLB is write-through (not write-back)
  - Always keep TLB and Page Table consistent

Cache Design Considerations

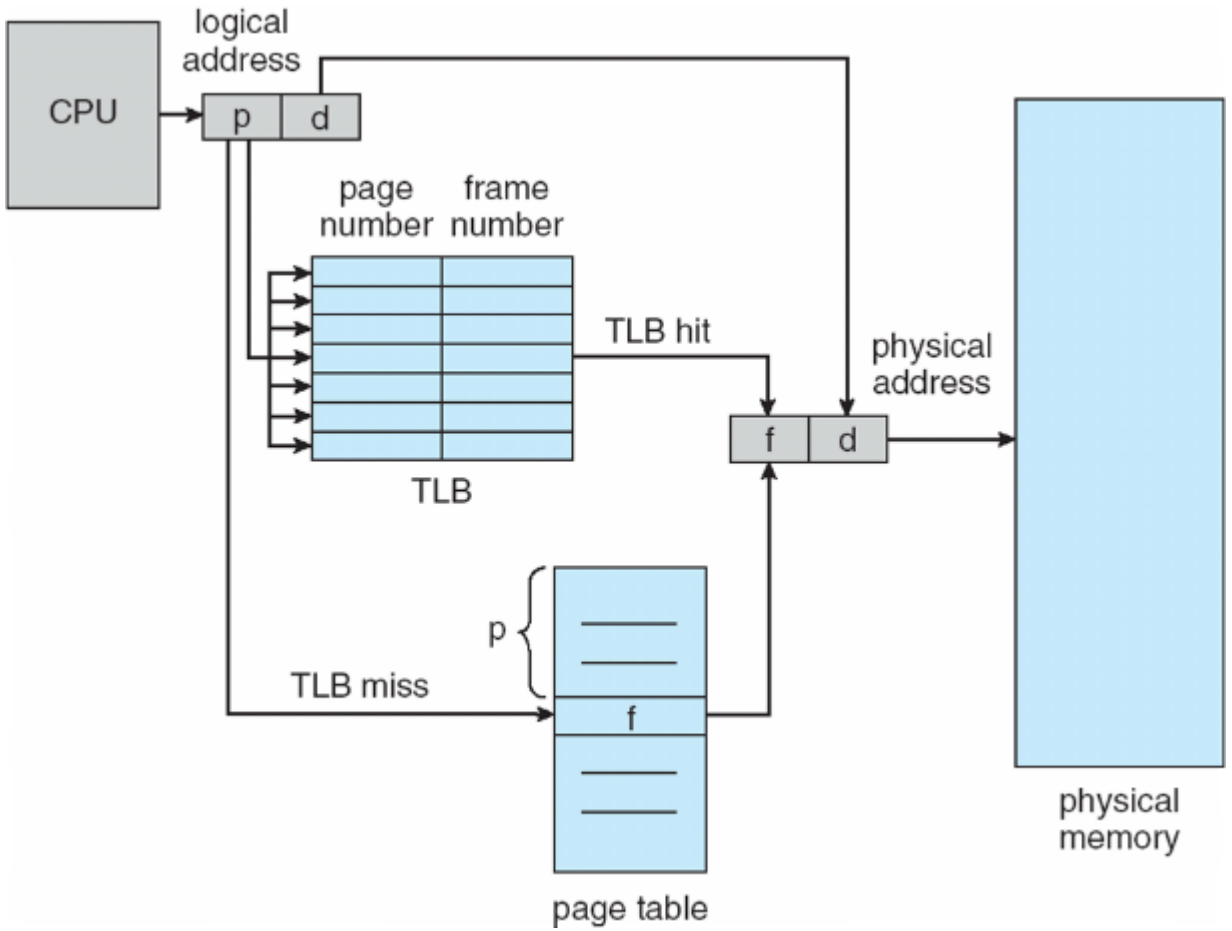
- Different design considerations for L1$ and L2$
  - L1$ focuses on fast access: minimize hit time to achieve shorter clock cycle, e.g., smaller $.
    - Since miss penalty of L1$ is significantly reduced by presence of L2$, so can be smaller/faster even with higher miss rate
  - L2$ (and L3$) focus on low miss rate: reduce penalty of long main memory access times: e.g., Larger $ with larger block sizes/higher levels of associativity
    - Since fast hit time is less important than low miss rate due to high miss penalty



Page Table Lookup w/ vs. w/o TLB



Page Table Lookup without TLB



Page Table Lookup with TLB

TLB Example

Suppose page table is stored within one physical memory page at address 0x200C (PFN=2, Offset=00C)

VPN	PFN
0	3
1	2
2	8
3	5

Page Table

Virtual memory
load 0x3000
load 0x3004
load 0x3008
load 0x300C
…

Physical memory

TLB

Valid	VPN	PFN
0		
0		
0		

TLB Example

Suppose page table is stored within one physical memory page at address 0x200C (PFN=2, Offset=00C)

VPN	PFN
0	3
1	2
2	8
3	5

Page Table

Virtual memory
load 0x3000
load 0x3004
load 0x3008
load 0x300C
…

Physical memory

TLB

Valid	VPN	PFN
0		
0		
0		

MISS!!

TLB Example

Suppose page table is stored within one physical memory page at address 0x200C (PFN=2, Offset=00C)

VPN	PFN
0	3
1	2
2	8
3	5

Page Table

Virtual memory
load 0x3000
load 0x3004
load 0x3008
load 0x300C
…

Physical memory
load 0x200C (PT)
load 0x5000

TLB

Valid	VPN	PFN
1	3	5
0		
0		

Update TLB

Table Lookaside Buffer (TLB)

Suppose page table is stored within one physical memory page at address 0x200C (PFN=2, Offset=00C)

VPN	PFN
0	3
1	2
2	8
3	5

Page Table

Virtual memory

load 0x3000

load 0x3004

load 0x3008

load 0x300C

…

Physical memory

load 0x200C (PT)

load 0x5000

load 0x5004

TLB

Valid	VPN	PFN
1	3	5
0		
0		

TLB hit

TLB Example

Suppose page table is stored within one physical memory page at address 0x200C (PFN=2, Offset=00C)

VPN	PFN
0	3
1	2
2	8
3	5

Page Table

Virtual memory

load 0x3000

load 0x3004

load 0x3008

load 0x300C

…

Physical memory

load 0x200C (PT)

load 0x5000

load 0x5004

load 0x5008

TLB

Valid	VPN	PFN
1	3	5
0		
0		

TLB hit

TLB Example

Suppose page table is stored within one physical memory page at address 0x200C (PFN=2, Offset=00C)

VPN	PFN
0	3
1	2
2	8
3	5

Page Table

Virtual memory

load 0x3000
load 0x3004
load 0x3008
load 0x300C
…
load 0x2000

TLB

Valid	VPN	PFN
1	3	5
1	2	8
0		

Miss!!

Physical memory

load 0x200C (PT)
load 0x5000
load 0x5004
load 0x5008
load 0x500C
…
load 0x200C (PT)
load 0x8000

Effective Access Time with TLB

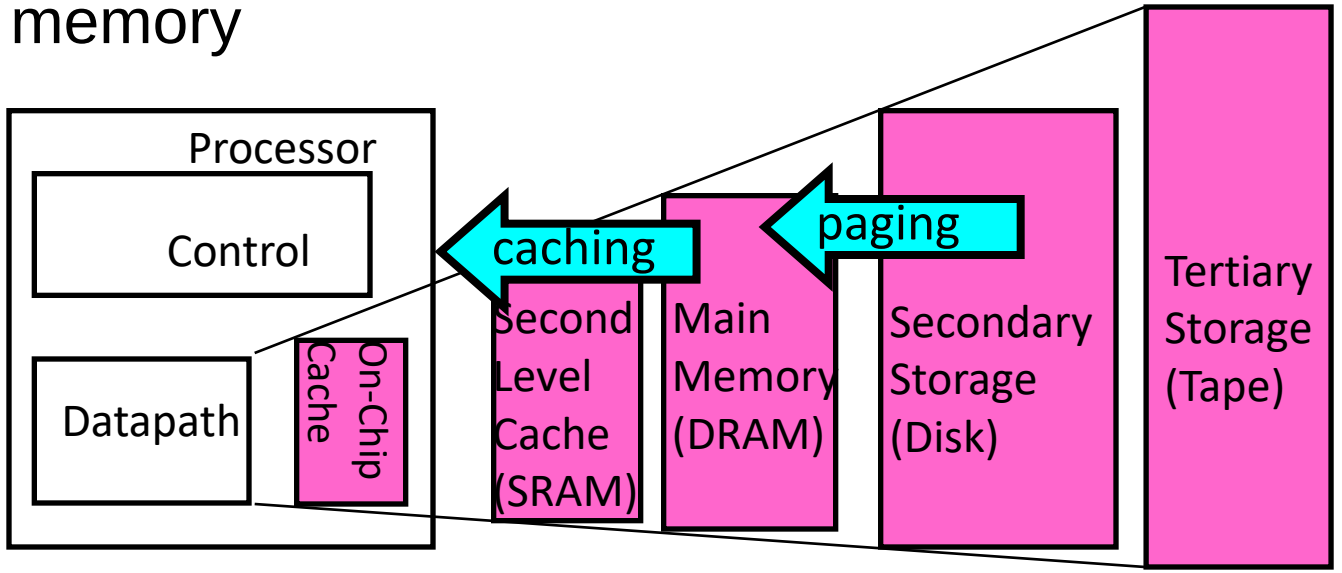
- TLB lookup time = σ time unit
- Memory access time = *m* time unit
  - Assume: Page Table needs single access (no multilevel page tables)
  - There is no cache
- TLB Hit Rate = η
- Effective access time:
  - *EAT* = Hit Time * Hit Rate + Miss Time * Miss Rate
  - = $(m + \sigma)\ \eta + (2m + \sigma)(1 - \eta) = 2m + \sigma - m\ \eta$
  - Assuming there is no cache. Upon TLB hit (w/ delay σ), access physical memory page directly (w/ delay *m*); upon TLB miss (w/ delay σ), first read Page Table (w/ delay *m*), then access physical memory page (w/ delay *m*)

Valid & Dirty Bits

- TLB entries have valid bits and dirty bits. Data cache blocks have them also.
  - The valid bit means the same in both: valid = 0 means either TLB miss or cache miss.
  - The dirty bit has different meanings. For cache, it means the cache block has been changed. For TLBs, it means that the page corresponding to this TLB entry has been changed.

Demand Paging

- Modern programs require a lot of physical memory
  - Memory per system growing faster than 25%-30%/year
- But they don't use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user's code to be in memory
- Solution: use main memory as "cache" for disk
  - Demand paging: If a requested page is not found in page table, then page fault occurs, OS brings the requested page in from secondary storage to memory



Summary

- Translation Lookaside Buffer (TLB)
  - Small number of PTEs and process IDs (< 512)
  - Often Fully Associative (Since conflict misses expensive)
  - On TLB miss, page table must be traversed and if located PTE is invalid, cause Page Fault
  - On change in page table, TLB entries must be invalidated
- Demand Paging: Treating the DRAM as a cache on disk
  - Page Table tracks which pages are in memory
  - Any attempt to access a page that is not in memory generates a page fault, which causes OS to bring missing page into memory
- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past

TLB Issue: Context Switch

- Each process has its own page table and virtual address space; But there is only a single TLB in the system
  - TLB entries no longer valid upon process context-switch
- Solution 1: **Flush**
  - Invalidate TLB by setting valid bits of all TLB entries to 0
  - Simple but expensive for frequent context switching between processes
- Solution 2: **Address space identifier (ASID)**
  - Hardware provides an address space identifier (ASID) field in the TLB (Think of ASID as process ID (pid)), to distinguish which process a TLB entry belongs to

VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

Problems of Paging

- Page Table is **too big**
- A linear page table array for 32-bit address space (2^{32} bytes) and 4KB page (2^{12} bytes)
  - How many pages: 2^{20} pages
  - How much memory: **4MB** assuming each page-table entry is of 4 bytes
    - 2 ^ (32-log(4KB)) * 4 = (2 ^ 20) * 4 = 4MB
  - One page table for one process:
    - 100 processes: **400MB**

Smaller Page Table

- Naïve solution:
  - **Bigger page size** -> **smaller page table**
  - 32-bit address space: 4KB page size -> 16KB
  - We can reduce the size by **4x** to 1MB per page table

- Page size: 2^x, **4KB – 1GB**
  - getconf PAGESIZE (MacOS and Linux)
  - 16KB for MacOS

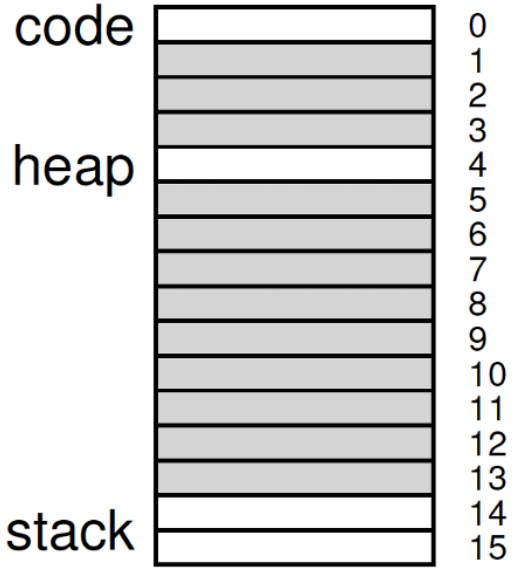
- **Problem: Internal fragment**
  - **Do not use up the whole page**

Variable Page Size

- TLB has **limited size**
  - 16-512
  - Multiple-level implementation, like cache
- Smaller page size → more TLB entries
  - A process of 64KB, **4KB page size**
    - **16 TLB entries**
  - **1MB page size**
    - 1 TLB entry
- Variable page size:
  - This depends on hardware and OS
  - Windows 10 supports 4KB and 2MB
  - Linux has default page size (4KB) and huge page

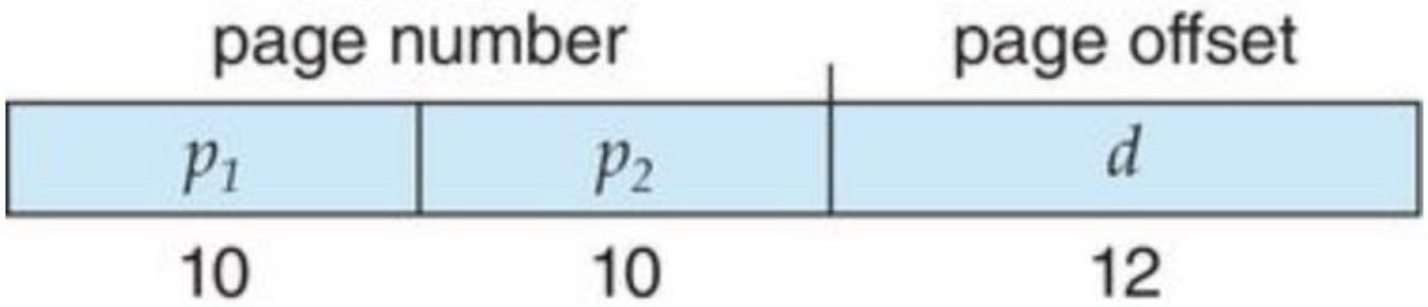
Multi-Level Paging

- Large page table is **contiguous** and may have some **unused pages**
- Allocate page table in a **non-contiguous manner**
- Break the page table into pages, i.e., page the page tables
- Create **multiple levels of page tables;** outer level **“page directory”**
  - **Page directory** to track whether a page of the page table is valid
- If an entire page of page table entries is **invalid**, **no allocation**



Multi-Level Paging

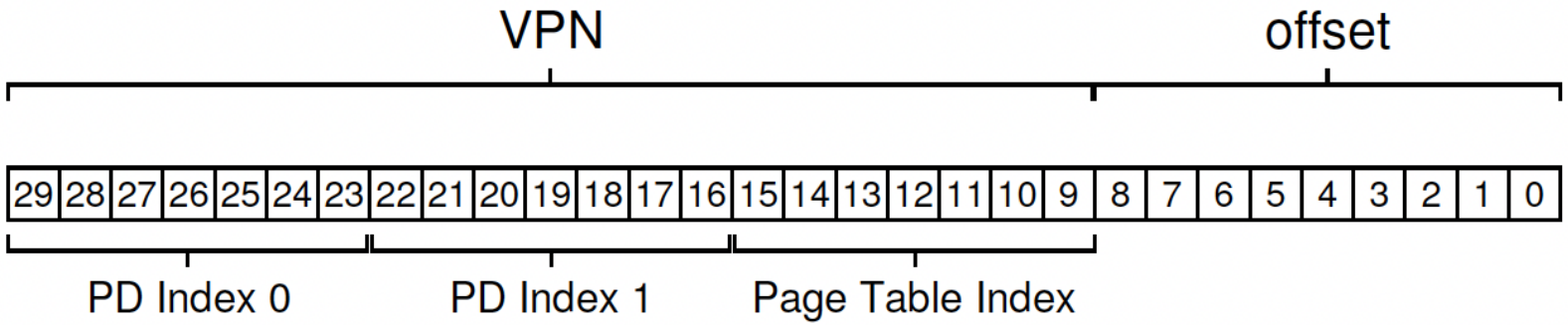
- A virtual address of 32-bit with 4KB page size is divided into
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- A page table entry is 4 bytes
- With two-level paging, the page number is further divided into two parts: p_1 is the page directory index, and p_2 is the page table index



Multi-Level Paging

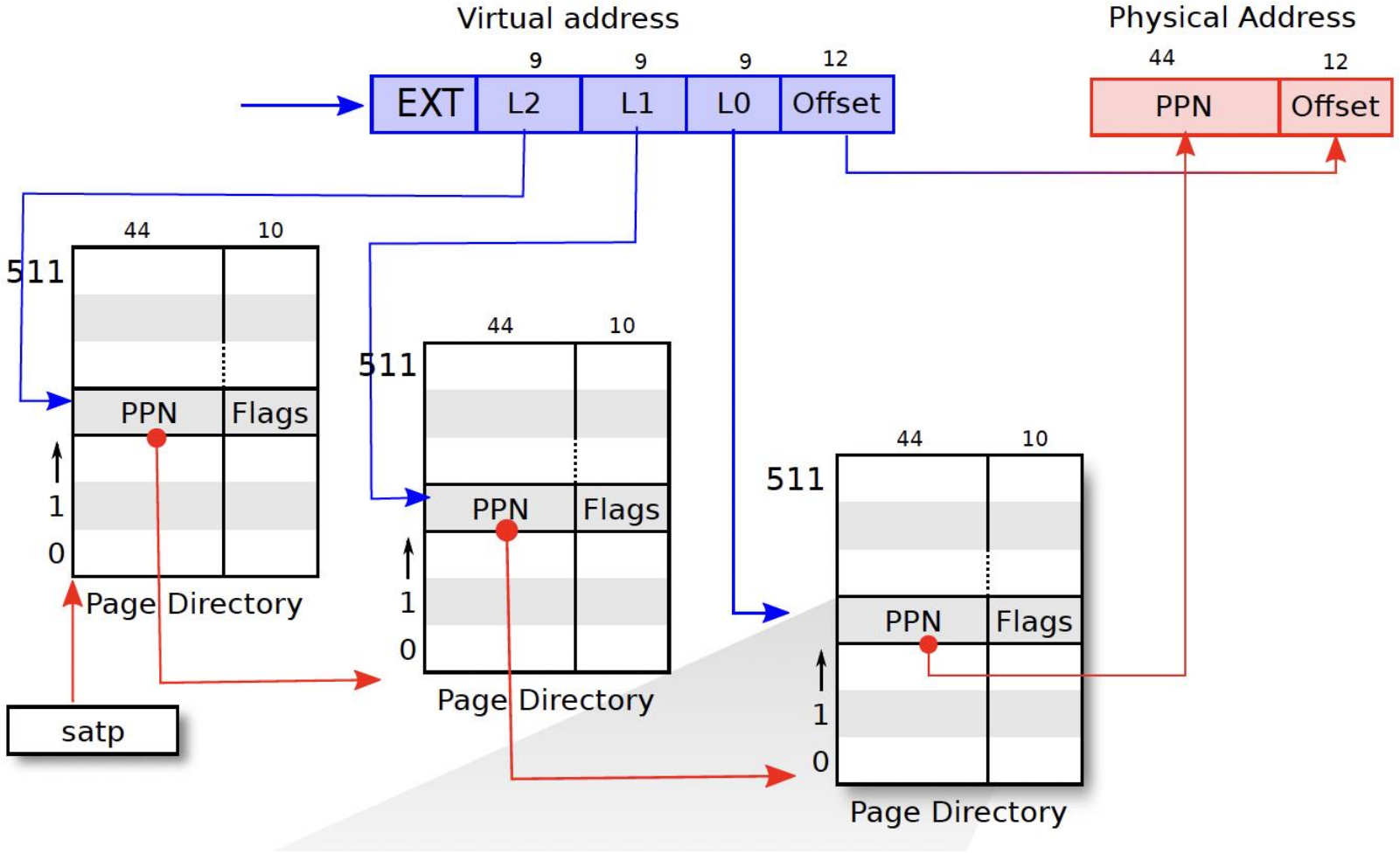
- Problem with 2 levels: **page directories may not fit in a page**
- Split page directories into pieces
- Multi-level page directories and each one can fit in a page

- 30-bit address space, 512-byte page size, 4byte PTE



Multi-Level Paging

- XV6-Sv39 - 3 level
- XV6-Sv48 – 4 level

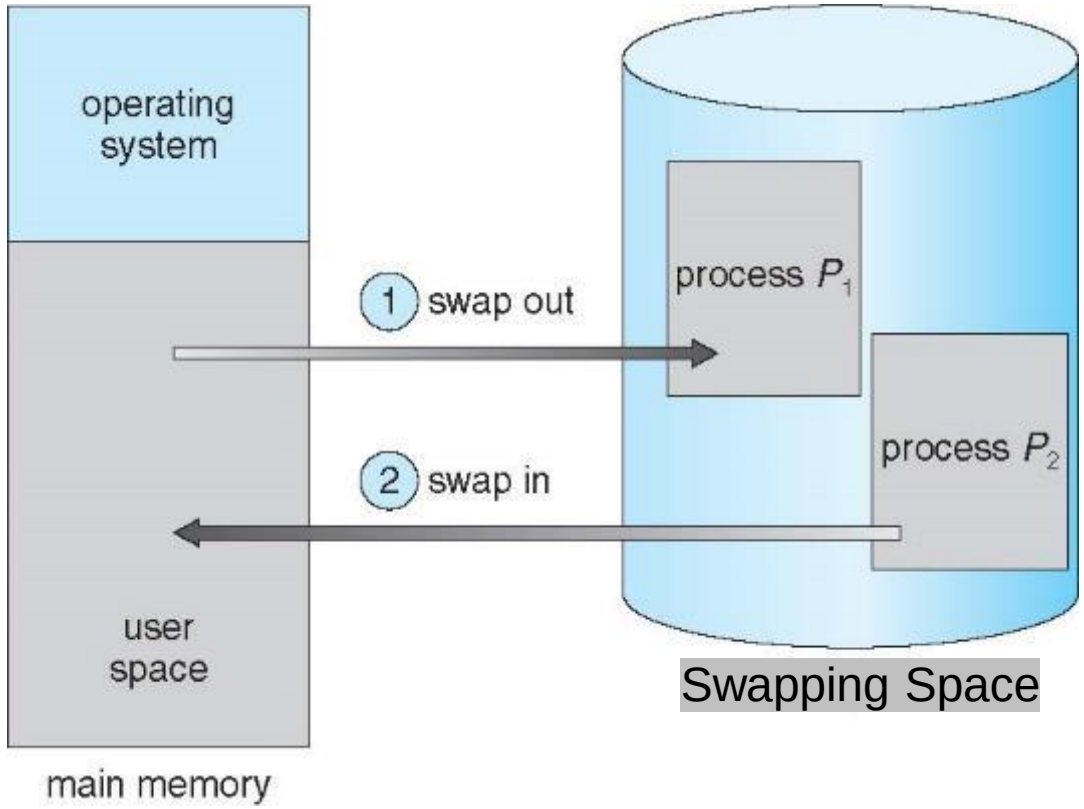


Page Swapping

- Motivation
  - Processes spend majority of time in small portion of code
    - The 90/10 rule: approximately **90%** of time in **10 %** of code
  - Process only uses small amount of address space (pages) at any moment
  - Only small amount of address space (pages) need to be resident in physical memory
- Hardware:
  - **Memory**: fast, but small, 2-100 GB/s
  - **Disk**: slow, but large, 80-160 MB/s (HDD) 500MB/s (SSD)
- Idea:
  - Process can run with **only some of its pages** in memory
  - Only keep **the actively used pages** in memory
  - Keep **unreferenced pages** on disk

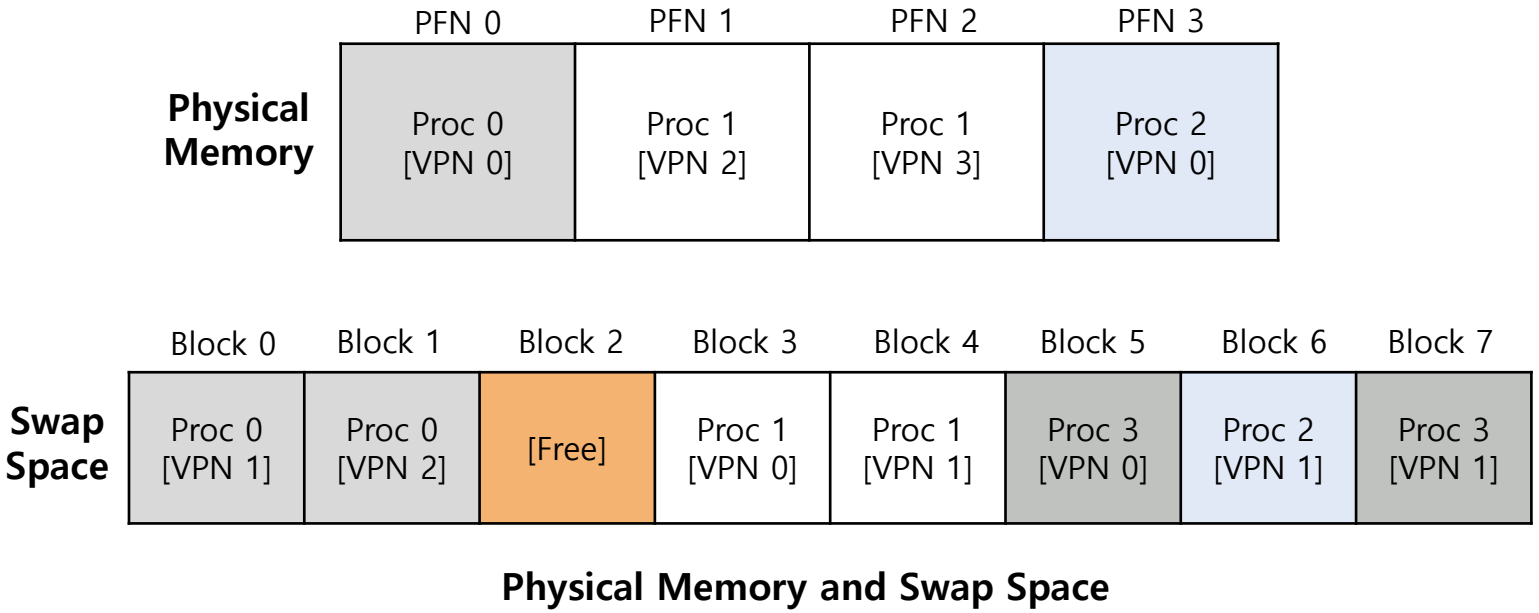
Page Swapping

- Swapping makes it possible for **the total physical address space of all processes to exceed** the real physical memory of the system



Page Swapping

- Reserve some space on the disk for moving pages back and forth —— **Swap space**
- OS keeps track of the swap space, in page-sized unit.

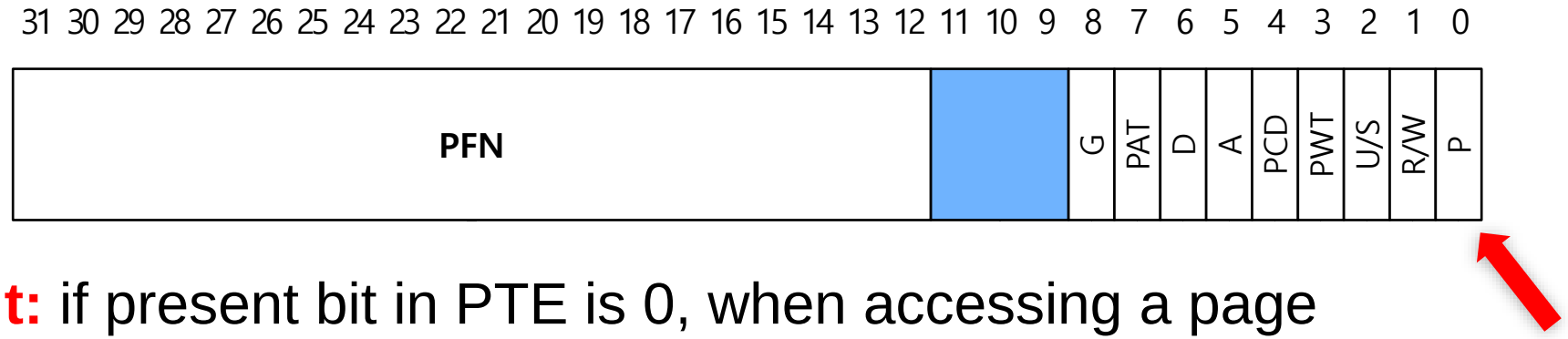


Physical Memory and Swap Space

Page Swapping

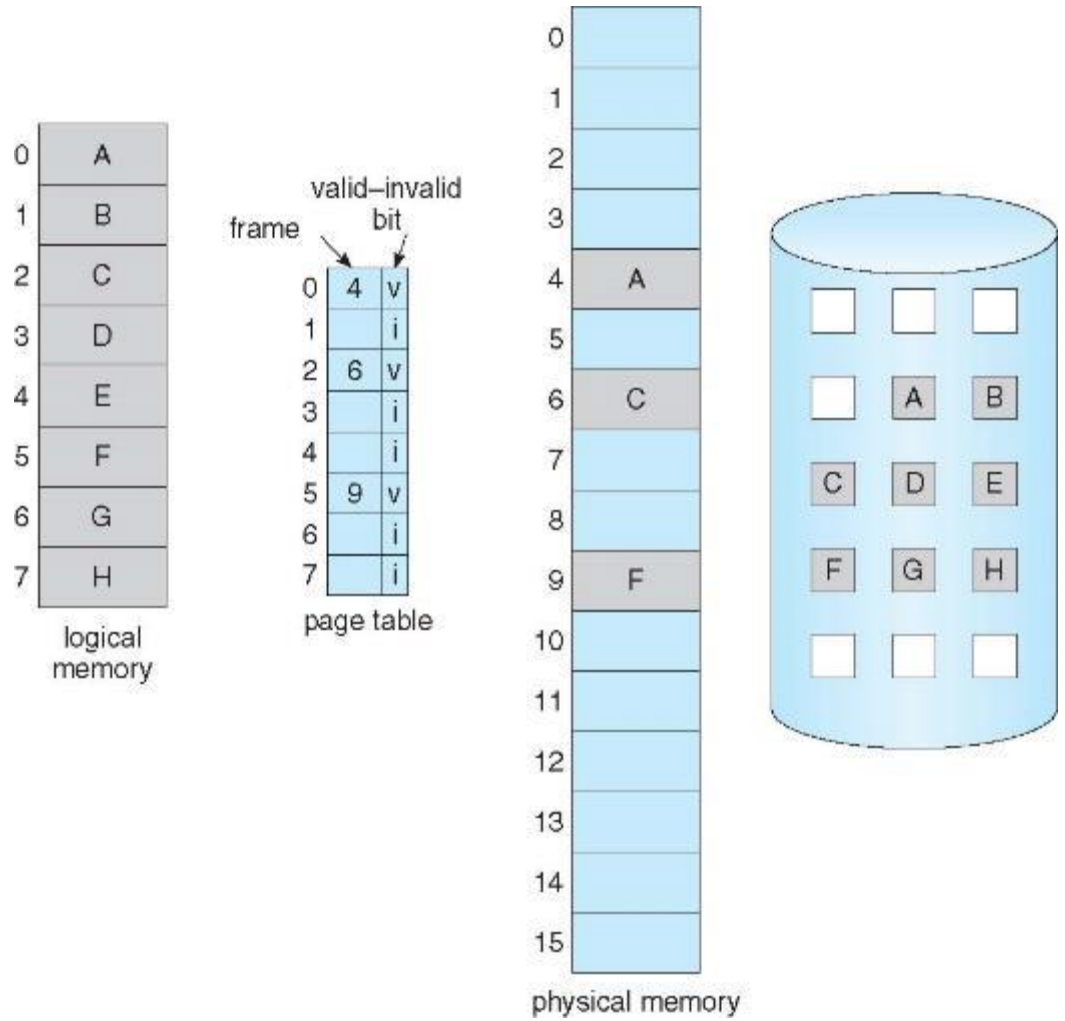
- How to know where a page lives?
  - **Present bit/Valid bit**
  - **1 indicates in-memory**
  - **0 indicates in-disk**

X86 page table entry (PTE)



- **Page fault:** if present bit in PTE is 0, when accessing a page

Page Swapping



Page Swapping Policies

- The objective of page swapping policies: to minimize the number of **page faults** (**cache misses**)

- Two decisions:
  - **Page selection**
    - **When** should a page on disk be brought into memory?
  - **Page replacement**
    - **Which** in-memory page should be evicted to disk?

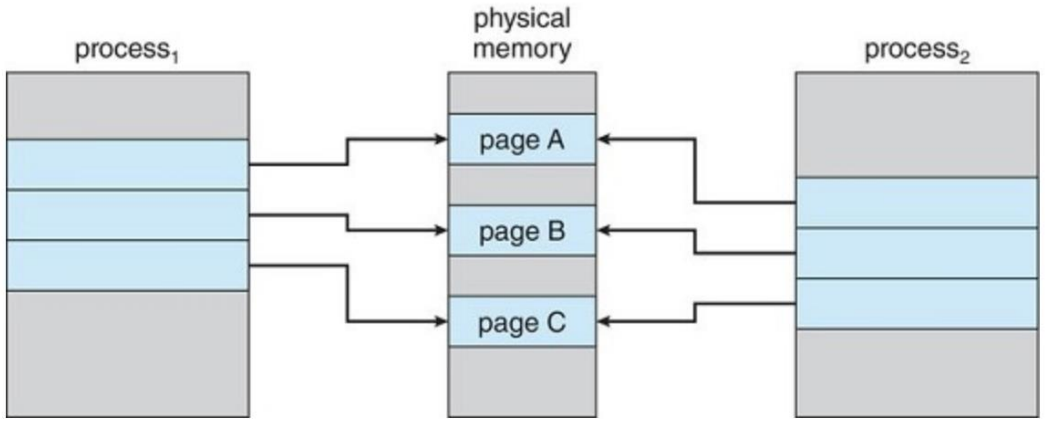
Page Selection

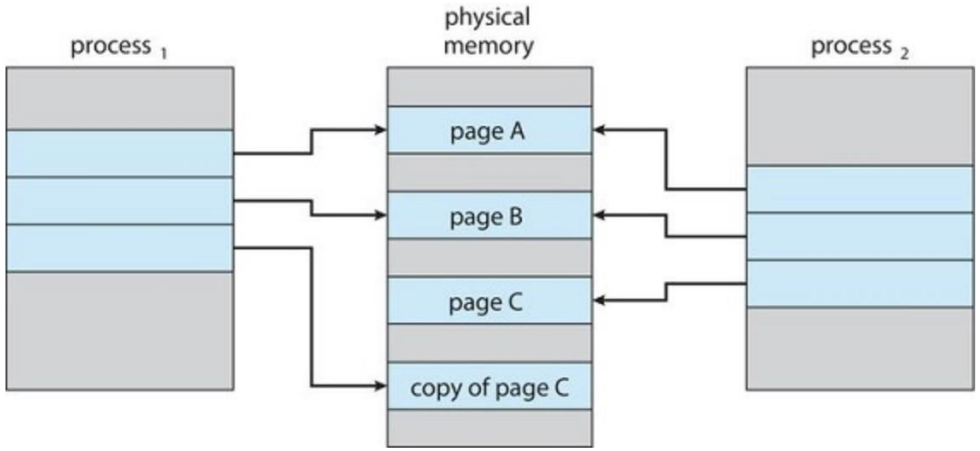
- **Demand paging**:
  - Load page only when it is needed (**demand**)
  - Less I/O, less memory
  - Problems: **High page fault cost**

- **Prefetch:**
  - Load page before referenced
  - OS predicts future accessed pages (oracle) and brings them into memory early
  - Works well for some access patterns, like sequential pages

Copy-On-Write Paging

- **Copy the page** only if a process writes to it (**demand**)
  - Process creation **fork() + exec()**





Page Replacement

- When does page replacement happen?
- **Lazy approach**
  - If memory is **entirely full**, OS then replaces a page to make room for some other page.
  - This is **unrealistic**.
  - The OS usually needs to **reserve some room** for the new pages

- **Swap Daemon, Page Daemon**
  - There are fewer than **LW (low watermark)** pages available, a background thread that is responsible for freeing memory is activated.
  - The thread evicts pages until there are **HW (high watermark)** pages available.

Page Replacement Policies

- OPT (Optimal) :
  - Replace the page that will not be used for **the longest time in future**
  - **Pros:** Minimal number of page faults
  - **Cons:** impractical, need to predict the future.
  - **Can be used as a comparison baseline**
- FIFO:
  - Replace the oldest page, that was loaded into memory first
  - **Pros**: Fair, easy to implement
  - **Cons**: May evict useful pages
- Least-recently-used (LRU): (Predict using history)
  - Replace the page which has not been used for longest time
  - **Pros:** Approximate optimal replacement
  - **Cons**: Difficult to implement

Example 1

- Consider memory size of 3 frames, and following reference stream of virtual pages:
  - A B C A B D A D B C B
- FIFO incurs 7 page faults:
  - When referencing D, replacing A is a bad choice, since A is references again right away
- OPT incurs 5 page faults:
  - When D is first referenced, C is replaced, since it is the page not referenced farthest in the future
- LRU page replacement policy makes the same decisions as OPT in this case
  - but not always true!

Ref	A	B	C	A	B	D	A	D	B	C	B
F1	A	A	A	A	A	D	D	D	D	C	C
F2		B	B	B	B	B	A	A	A	A	A
F3			C	C	C	C	C	C	B	B	B

FIFO: 7 page faults

Ref	A	B	C	A	B	D	A	D	B	C	B
F1	A	A	A	A	A	A	A	A	A	C	C
F2		B	B	B	B	B	B	B	B	B	B
F3			C	C	C	D	D	D	D	D	D

LRU & OPT: 5 page faults

Example 2

- Consider memory size of 3 frames, and following reference stream of virtual pages:
  - A B C D A B C D A B C D
  - Cyclically referencing 4 pages A B C D
- FIFO & LRU both incurs 12 page faults, one for every page reference!
- OPT incurs 6 page faults, but it is not implementable

Ref	A	B	C	D	A	B	C	D	A	B	C	D
F1	A	A	A	D	D	D	C	C	C	B	B	B
F2		B	B	B	A	A	A	D	D	D	C	C
F3			C	C	C	B	B	B	A	A	A	D

FIFO & LRU: 12 page faults

Ref	A	B	C	D	A	B	C	D	A	B	C	D
F1	A	A	A	A	A	A	A	A	A	B	B	B
F2		B	B	B	B	B	C	C	C	C	C	C
F3			C	D	D	D	D	D	D	D	D	D

OPT: 6 page faults

Example 3

- Consider memory size of 3 frames, and following reference stream of virtual pages:
  - A A B B C D B A B A
- FIFO incurs 6 page faults
- LRU incurs 5 page faults
- OPT incurs 4 page faults

Ref	A	A	B	B	C	D	B	A	B	A
F1	A	A	A	A	A	D	D	D	D	D
F2			B	B	B	B	B	A	A	A
F3					C	C	C	C	B	B

FIFO: 6 page faults

Ref	A	A	B	B	C	D	B	A	B	A
F1	A	A	A	A	A	D	D	D	D	D
F2			B	B	B	B	B	B	B	B
F3					C	C	C	A	A	A

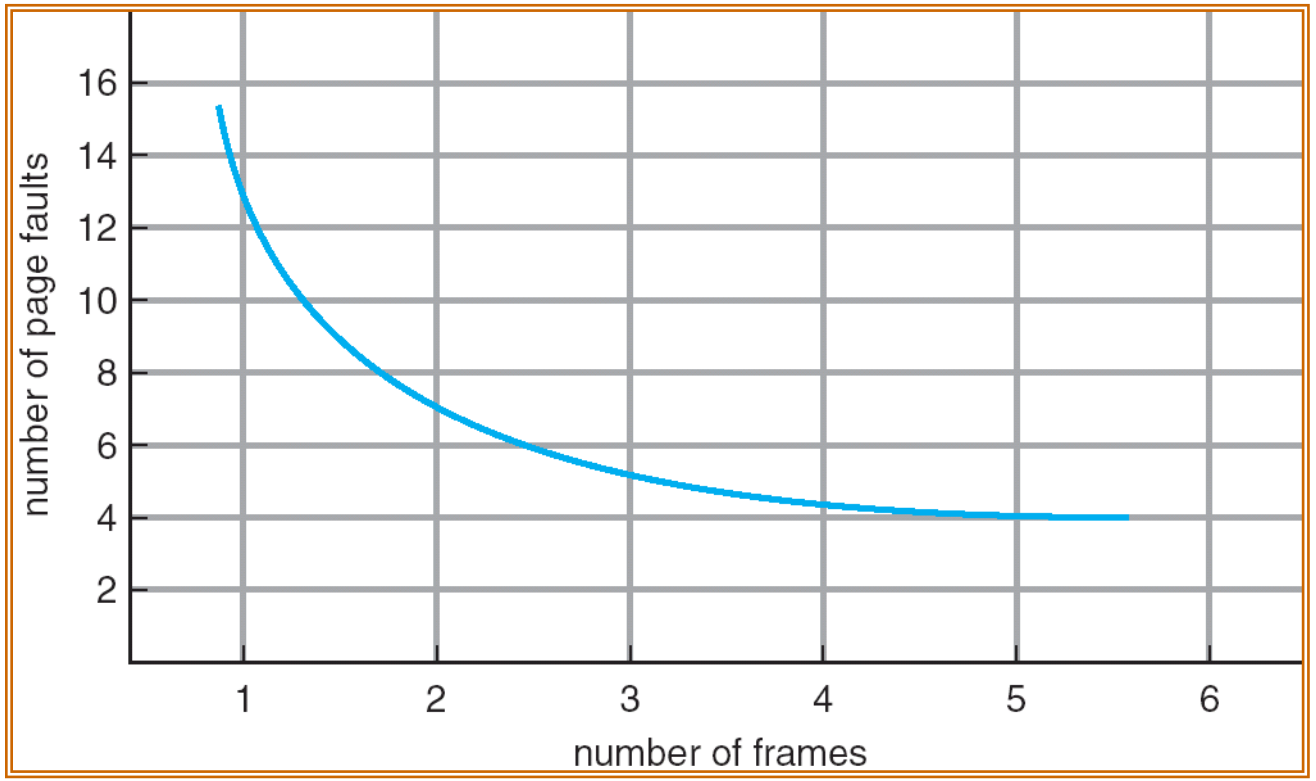
LRU: 5 page faults

Ref	A	A	B	B	C	D	B	A	B	A
F1	A	A	A	A	A	A	A	A	A	A
F2			B	B	B	B	B	B	B	B
F3					C	D	D	D	D	D

OPT: 4 page faults

Page faults vs. cache size

- When you increase memory size (number of frames), number of page faults should go down
  - Yes for LRU and OPT
  - Not always for FIFO (Called Belady's anomaly)



BeLady's anomaly

- After increasing memory size from 3 pages to 4 pages, for the given reference stream:
  - With FIFO, number of page faults increases from 9 to 10!
  - With LRU or OPT, memory contents with size of X pages are a subset of contents with size of X+1 Pages. Whereas for FIFO, memory contents can be completely different

Ref	A	B	C	D	A	B	E	A	B	C	D	E
F1	A	A	A	D	D	D	E	E	E	E	E	E
F2		B	B	B	A	A	A	A	A	C	C	C
F3			C	C	C	B	B	B	B	B	D	D

FIFO w/ 3 frames: 9 page faults

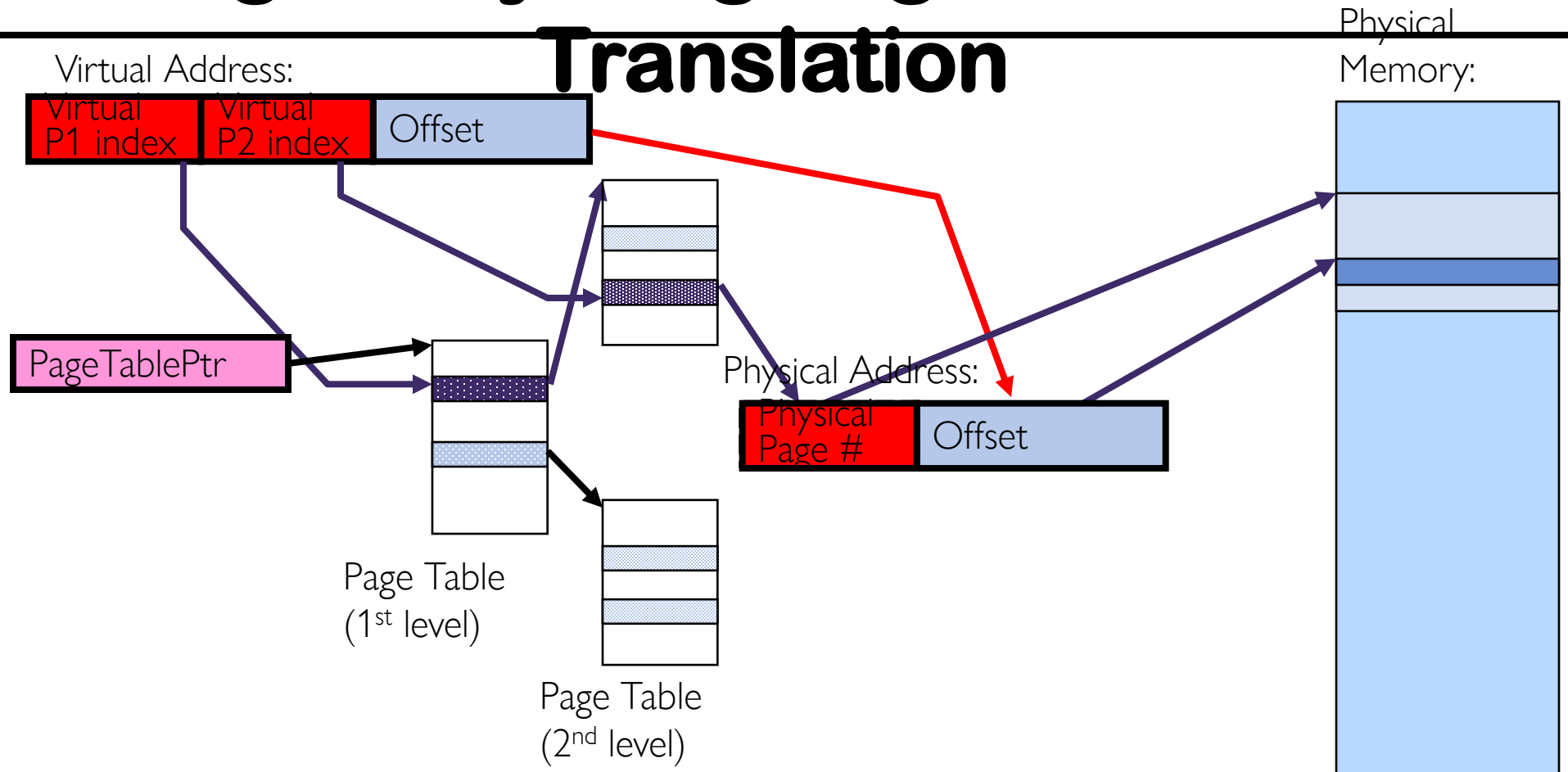
Ref	A	B	C	D	A	B	E	A	B	C	D	E
F1	A	A	A	A	A	A	E	E	E	E	D	D
F2		B	B	B	B	B	B	A	A	A	A	E
F3			C	C	C	C	C	C	B	B	B	B
F4				D	D	D	D	D	D	C	C	C

FIFO w/ 4 frames: 10 page faults

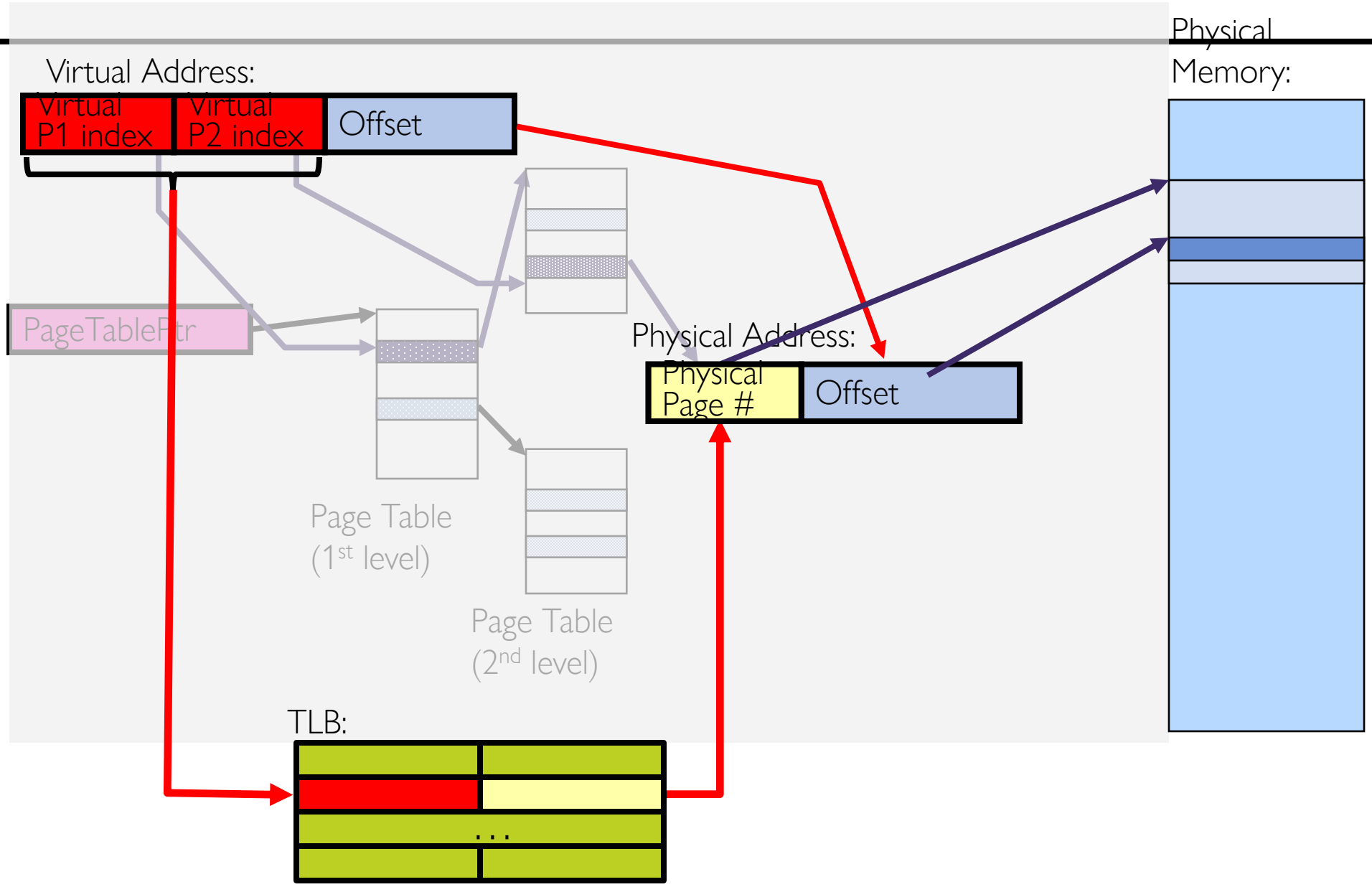
Page Replacement Policies

- Other policies:
  - **Random (RAND), Least-frequently used (LFU)**
- The performance of replacement policies also depends on workloads.
  - Random workload: LRU, RAND, and FIFO **no difference**
  - 80-20 workload: **LRU** is better than RAND and FIFO
  - Looping sequential workload: **RAND** is better than LRU and FIFO

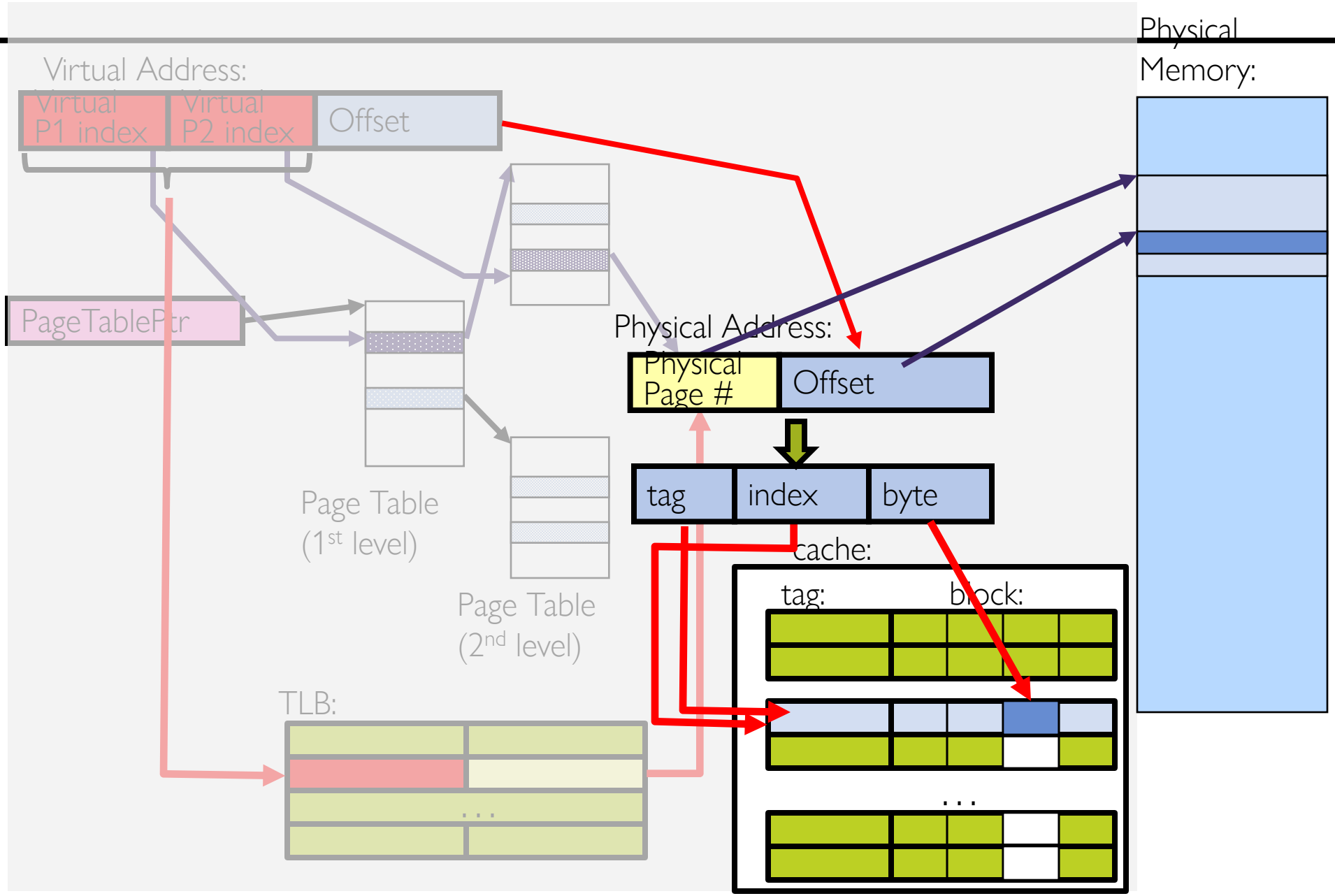
Putting Everything Together: Address Translation



Putting Everything Together: TLB



Putting Everything Together: TLB+Cache



Summary

- Paging: flexible virtual memory management
- Challenges with paging
  - Slow access
  - Big page table and high memory consumption
- Table Lookaside Buffer (TLB) for slow access
- Multi-level paging and inverted page tables for big page table
- Larger address space: swapping and replacement

References

- Memory Management (Virtual Memory / Paging / DMA), BitLemon
  - https://www.youtube.com/playlist?list=PL38NNHQLqJqZoDp4CrAueD1aBin7OebEL
- Lectures on Virtual Memory, by David Black-Schaffer
  - https://www.youtube.com/playlist?list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX
- Inverted page table, EZCSE
  - https://www.youtube.com/watch?v=9pXnMfKq7Hw