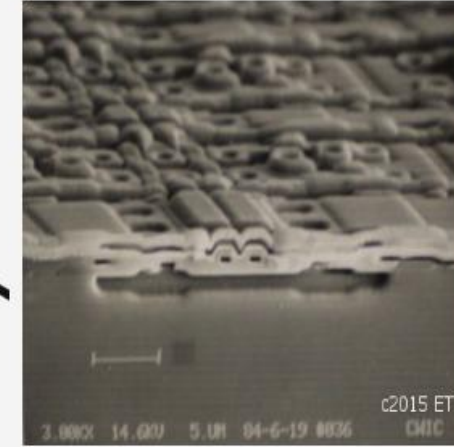
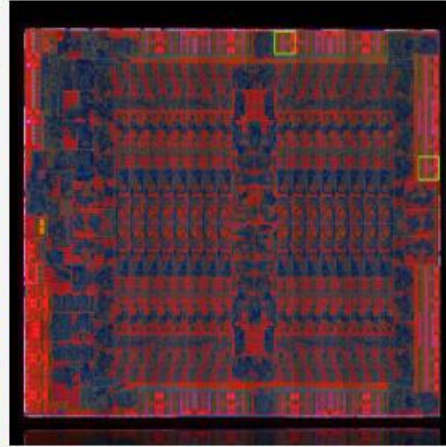
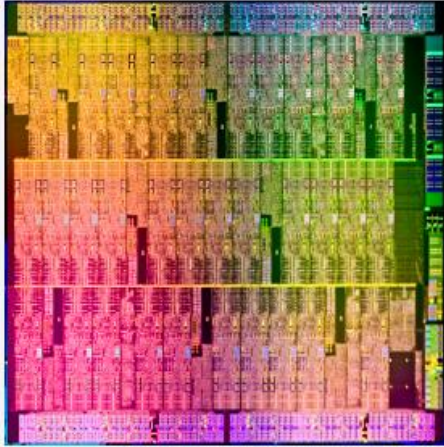




Course Description

Lecture 13: Virtual Memory

This course covers the design and implementation of computer systems from the perspective of the hardware software interface. The purpose of this course is for students to understand the relationship between the operating system, software, and computer architecture. Students that complete the course will have learned operating system fundamentals, computer architecture fundamentals, compilation to hardware abstractions, and how software actually executes from the perspective of the hardware software/boundary. The course will focus especially on understanding the relationships between software and hardware, and how those relationships influence the design of a computer system's software and hardware. The course will convey these topics through a series of practical, implementation-oriented lab assignments.

Credit: Brandon Lucia

Today (& Next Time): Virtual Memory

- Basic dimensions of a virtual memory system: paging, protections, process isolation, address mapping
- Working through operation of a virtual memory system example, including page fault handling and page table walking
- Start looking at hardware support for virtual memory (TLB)

What is virtualization?



Virtualization - Purpose

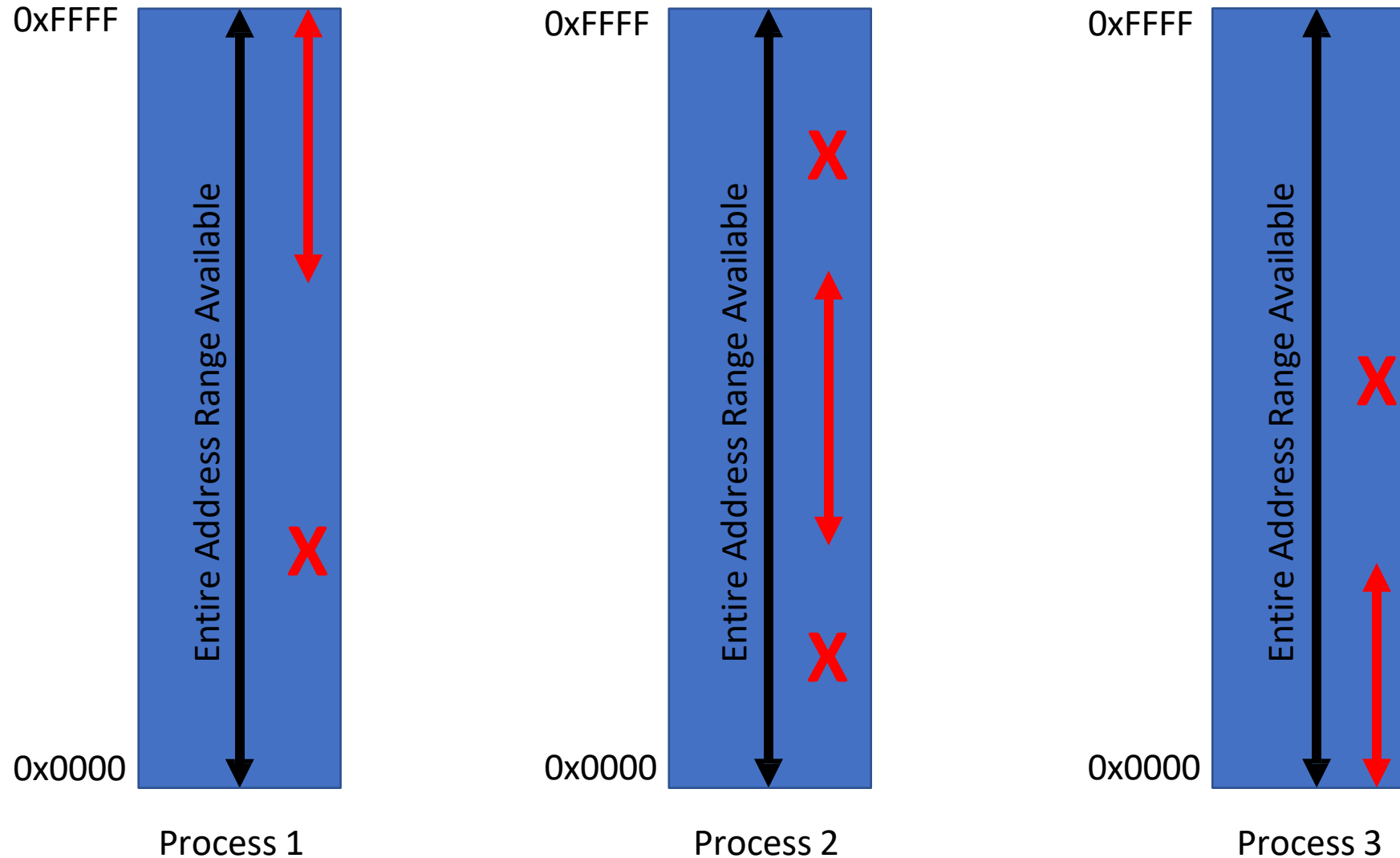
- Expose abstraction of abundant resource despite limited resource
- Expose abstraction of uniform resource despite heterogeneity of resource
- Expose abstraction of isolated resource despite sharing of resource

Virtualization – What resources?

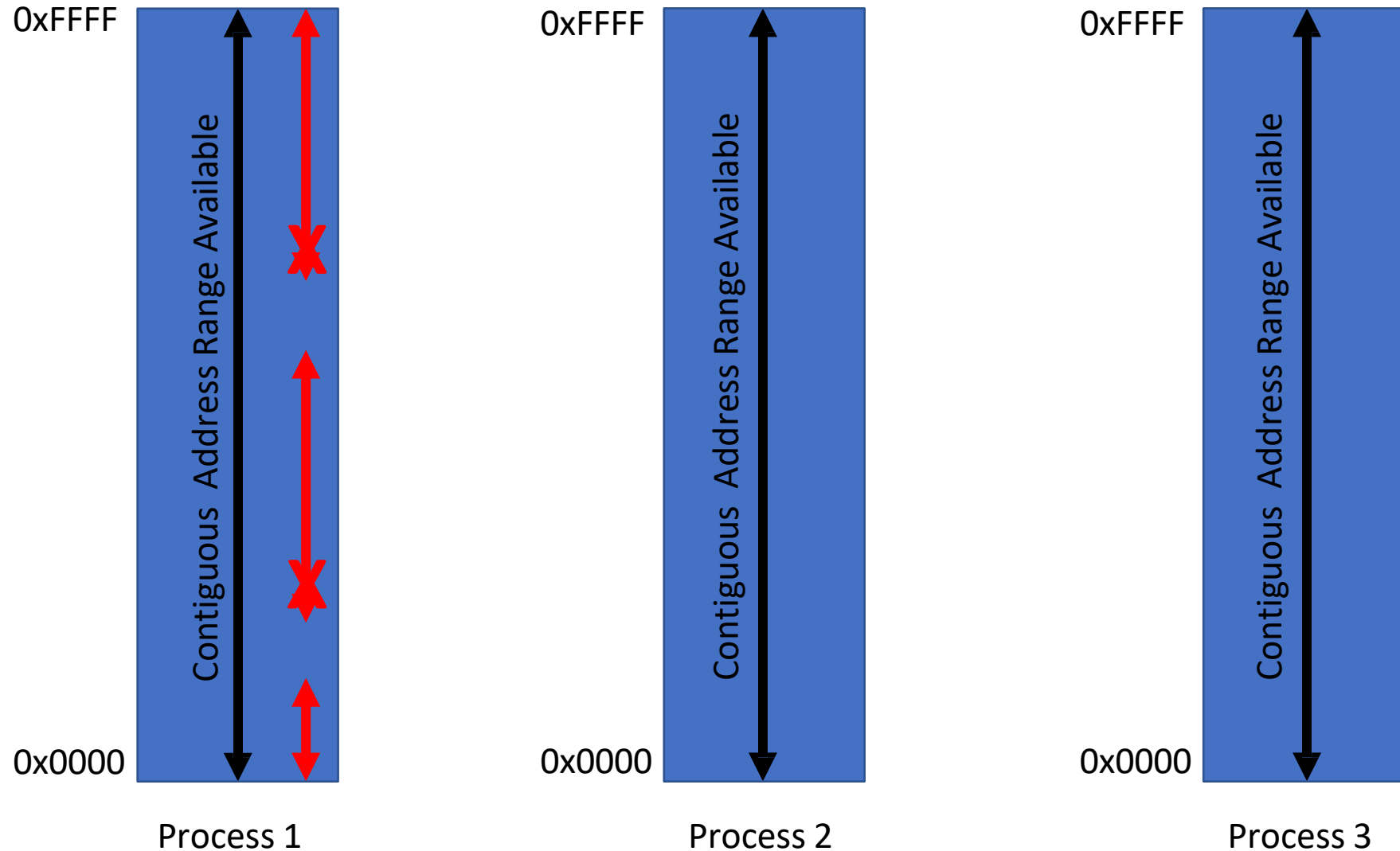
- Entire machines (VMMs)
- Storage (Disk controllers / Flash controllers)
- Memory (Virtual Memory)
- Network connectivity / bandwidth (Software-defined Networks)

Memory Virtualization

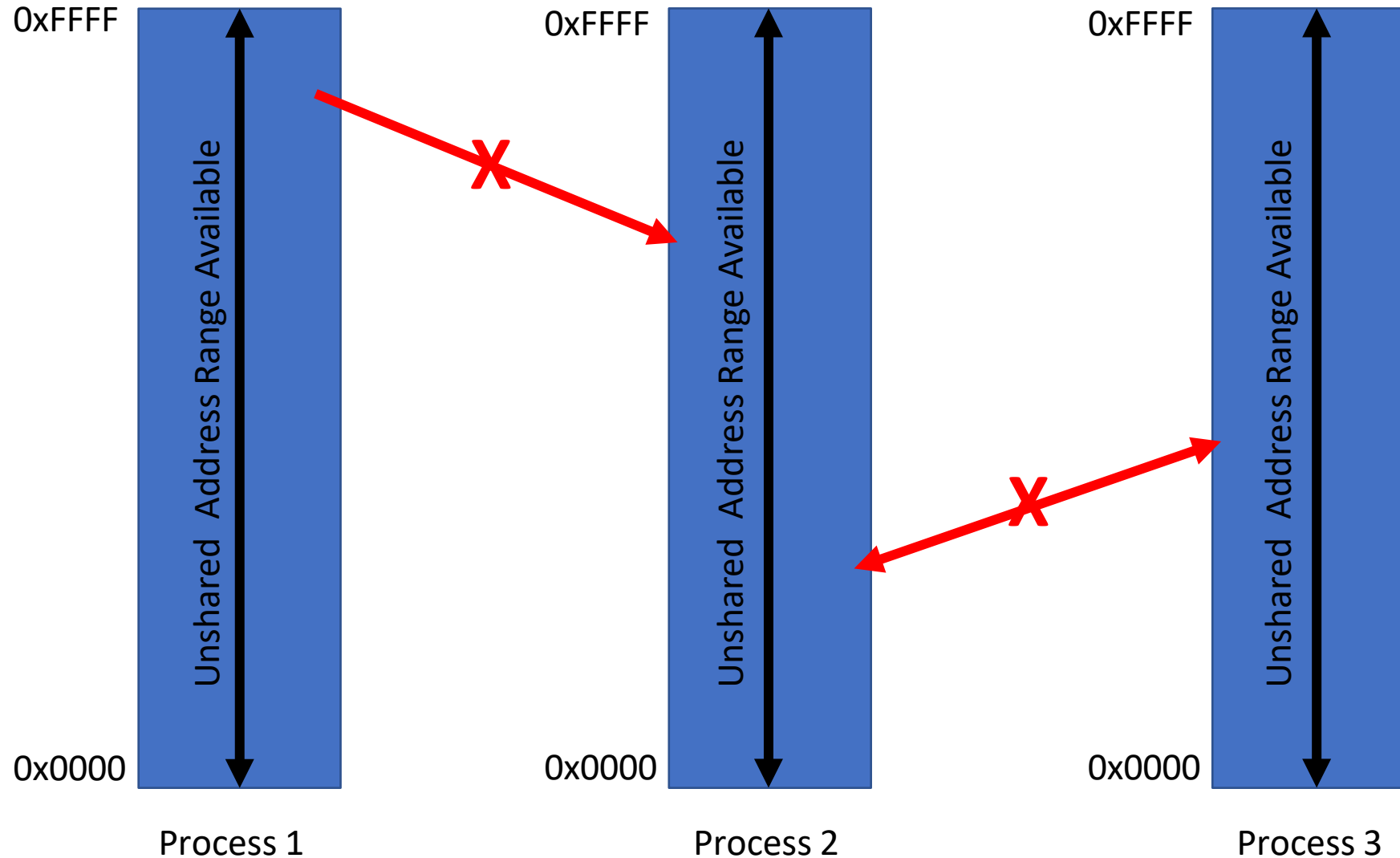
Virtual Memory – Abstraction of Abundance



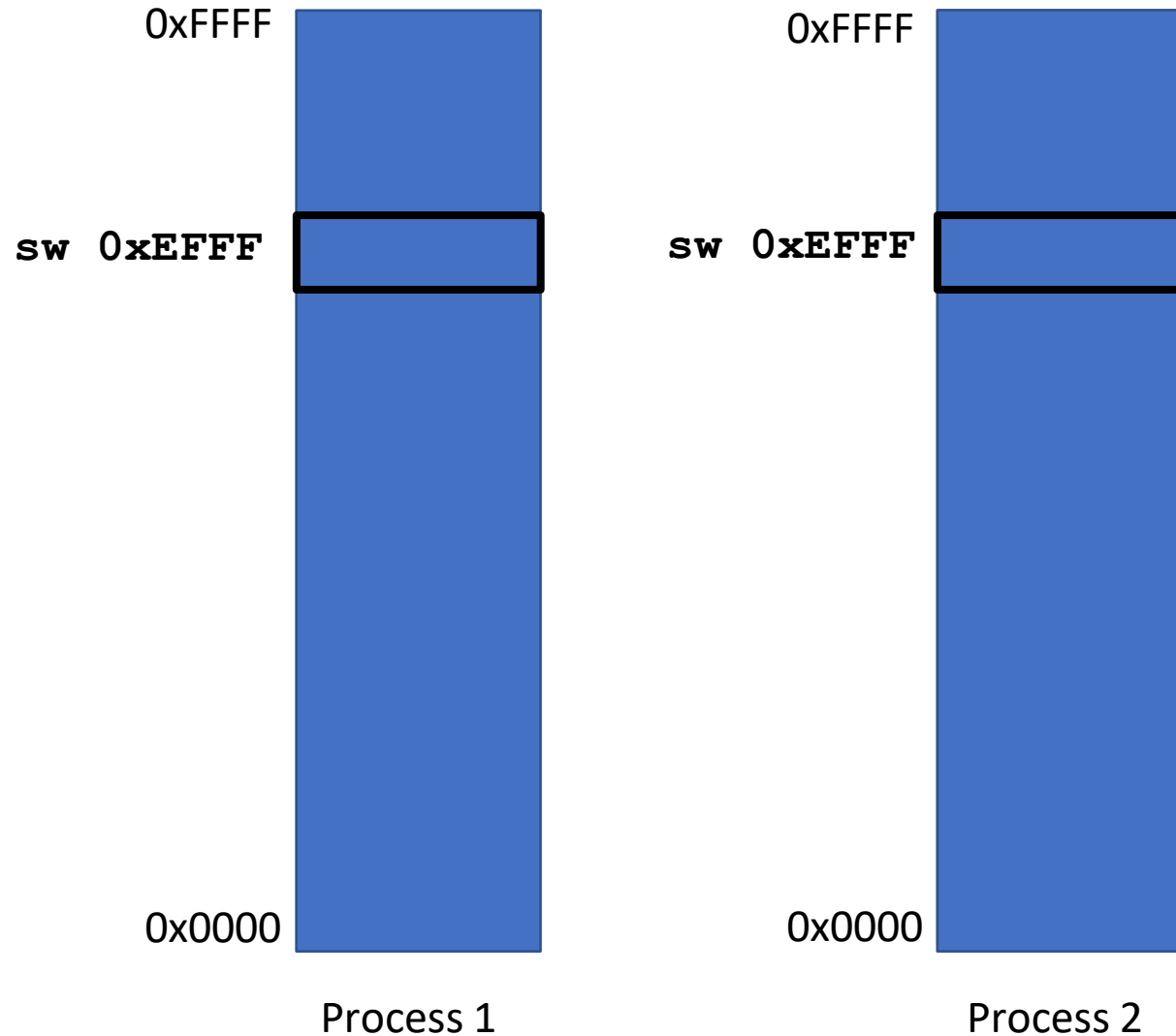
Virtual Memory – Abstraction of Uniformity



Virtual Memory – Abstraction of Isolation*



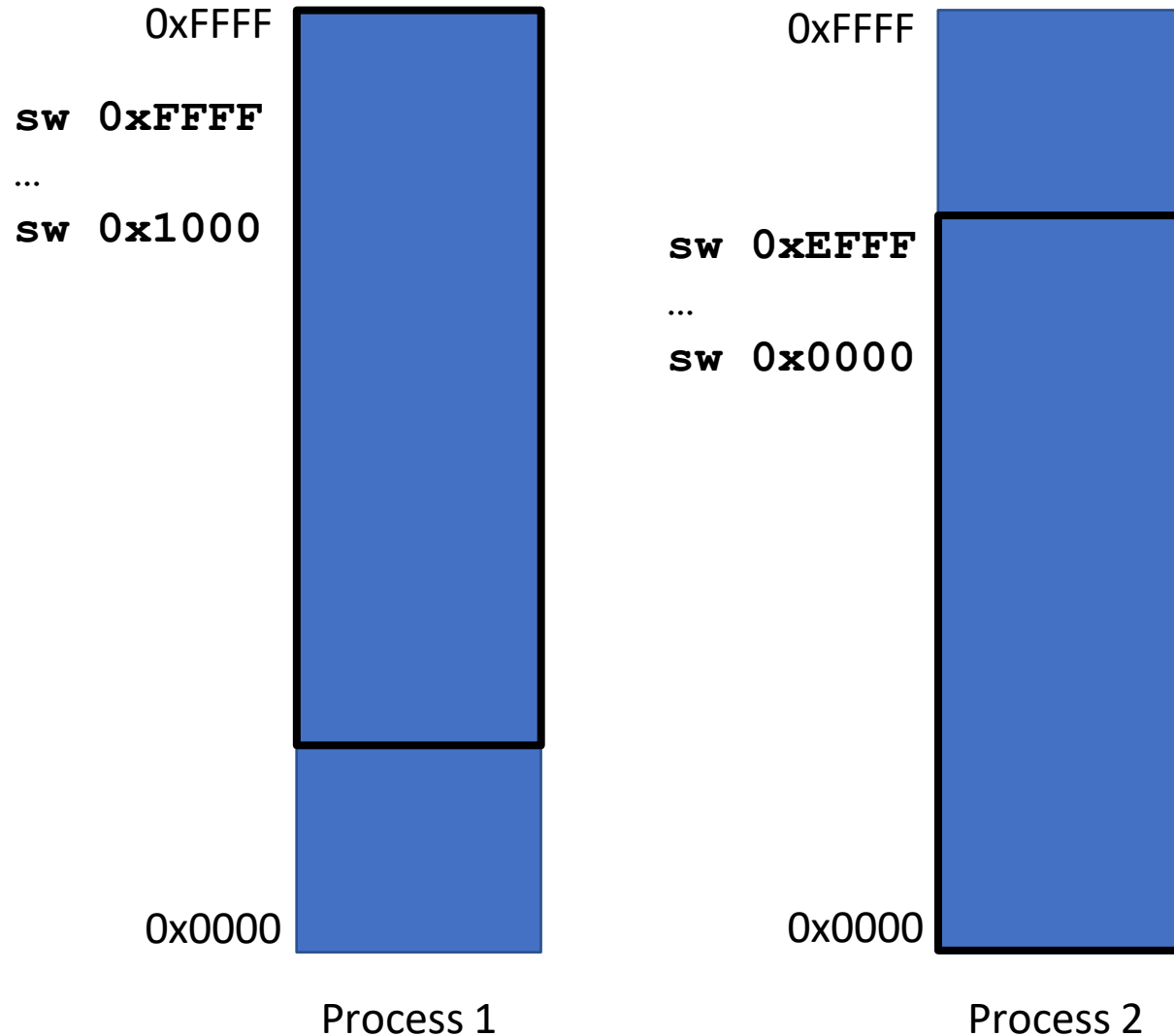
Virtual Memory – Thinking about mechanism



First obvious problem:

Two processes access same location
violates isolation abstraction

Virtual Memory – Thinking about Mechanism



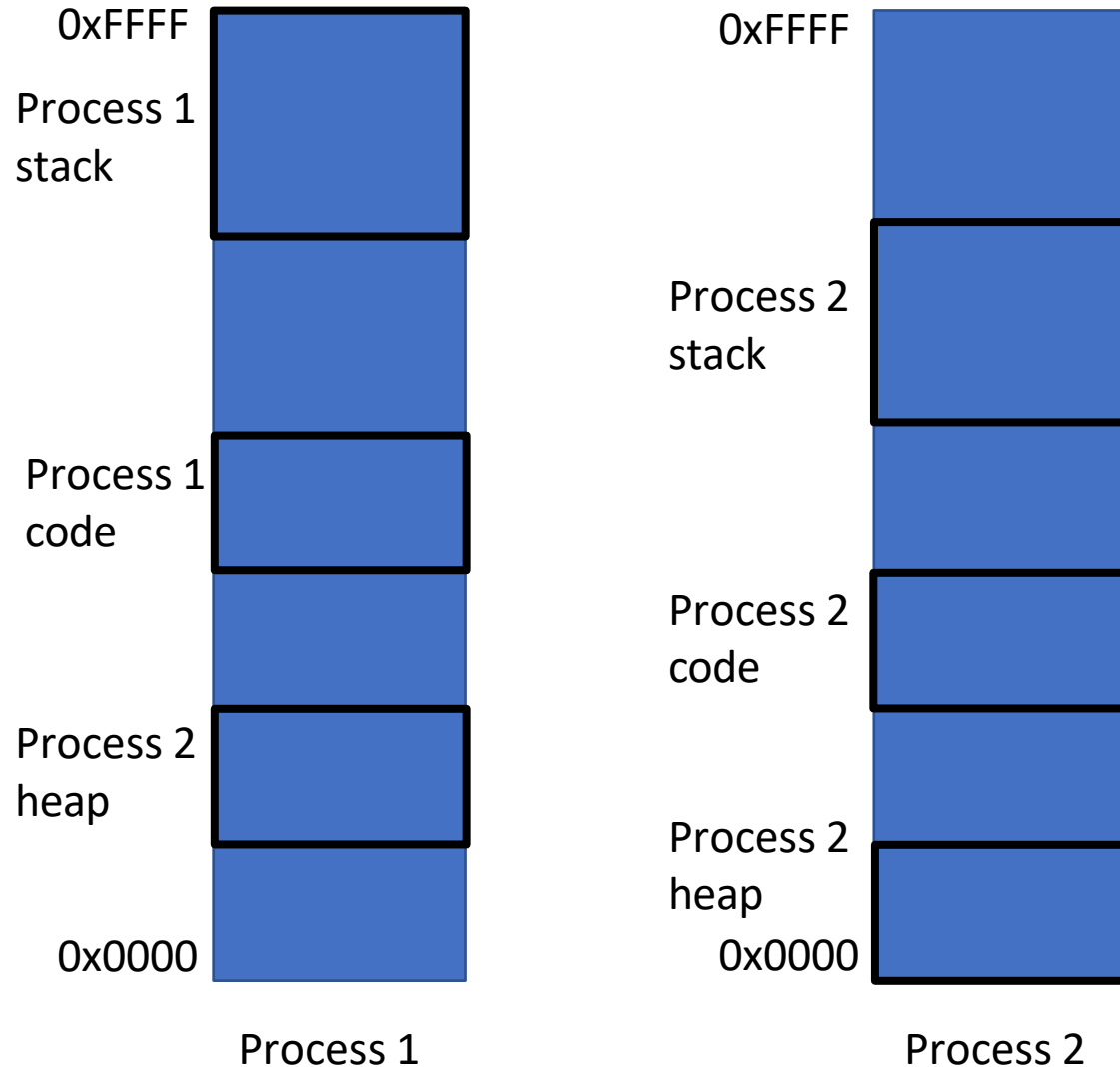
First obvious problem:

Two processes access same location
violates isolation abstraction

Second obvious problem:

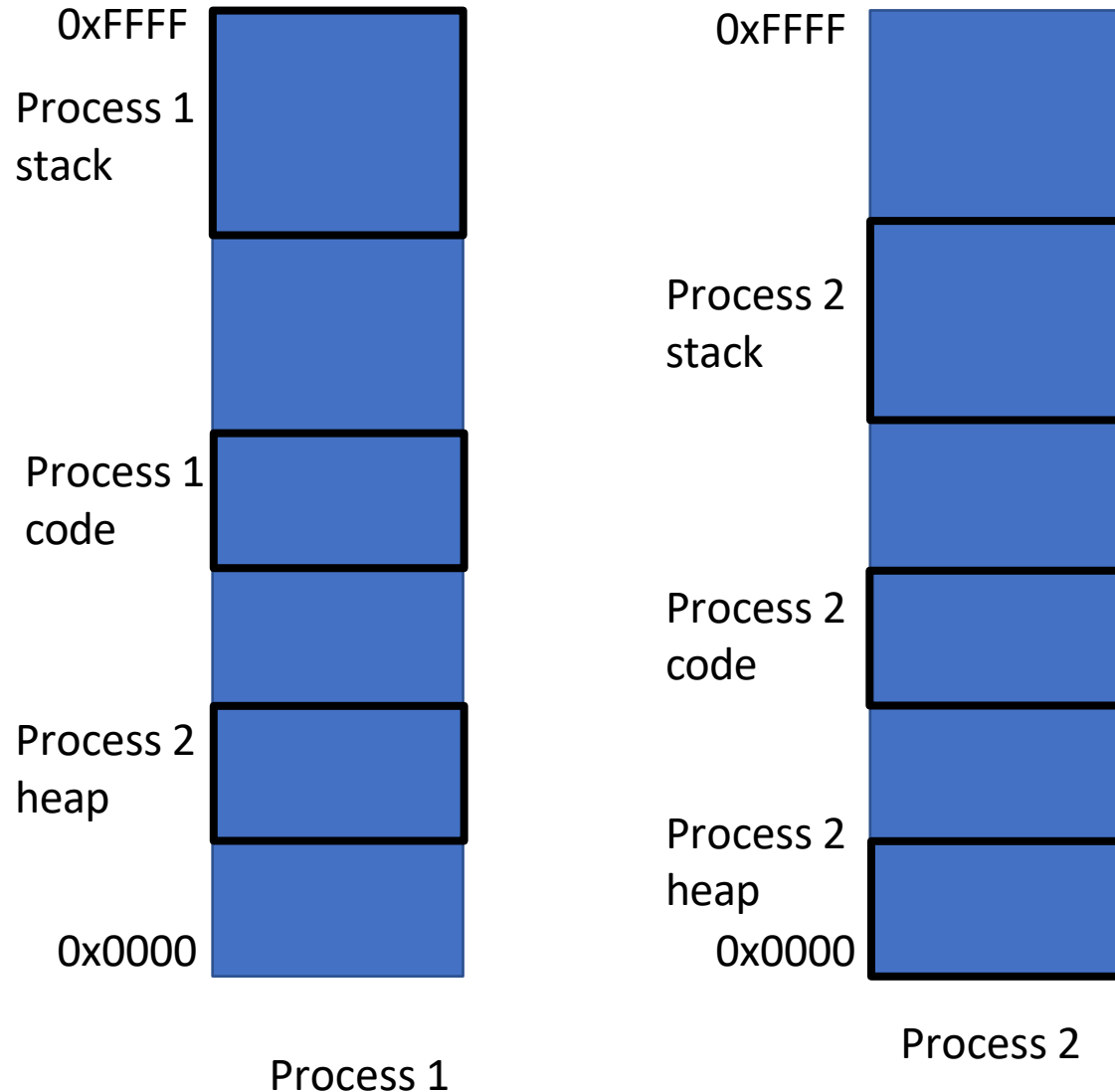
Two processes access #bytes > total memory size
violates abundance abstraction (and isolation)

First Attempt: Static Partitioning [Opal, SASOS, bare-metal micros]



Statically partitioning the address space
violates abundance and uniformity (but not isolation)

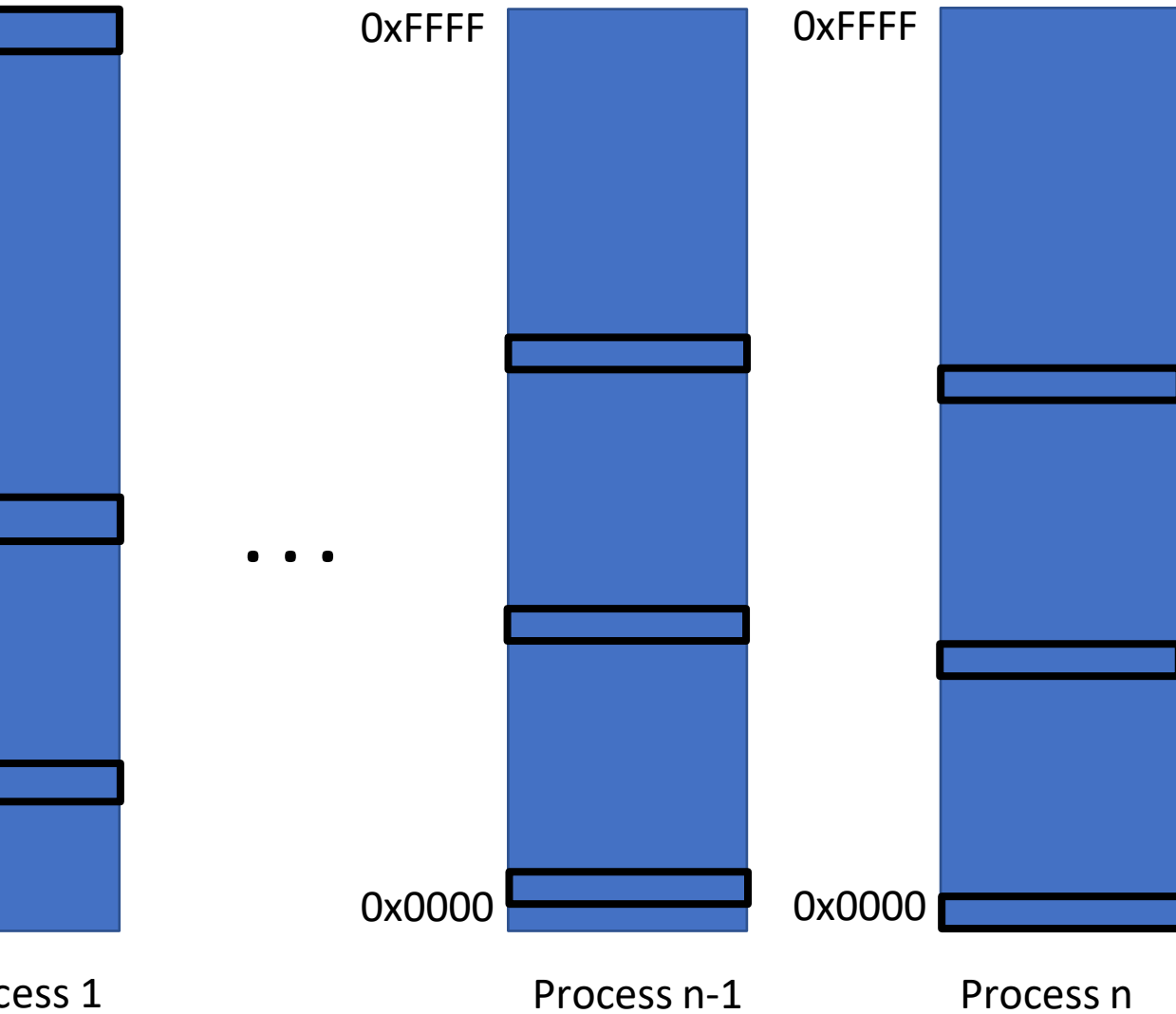
First Attempt: Static Partitioning



Statically partitioning the address space
violates abundance and uniformity

**Also need to be sure that neither process will go and
mess around with the other process' address ranges**

First Attempt: Static Partitioning



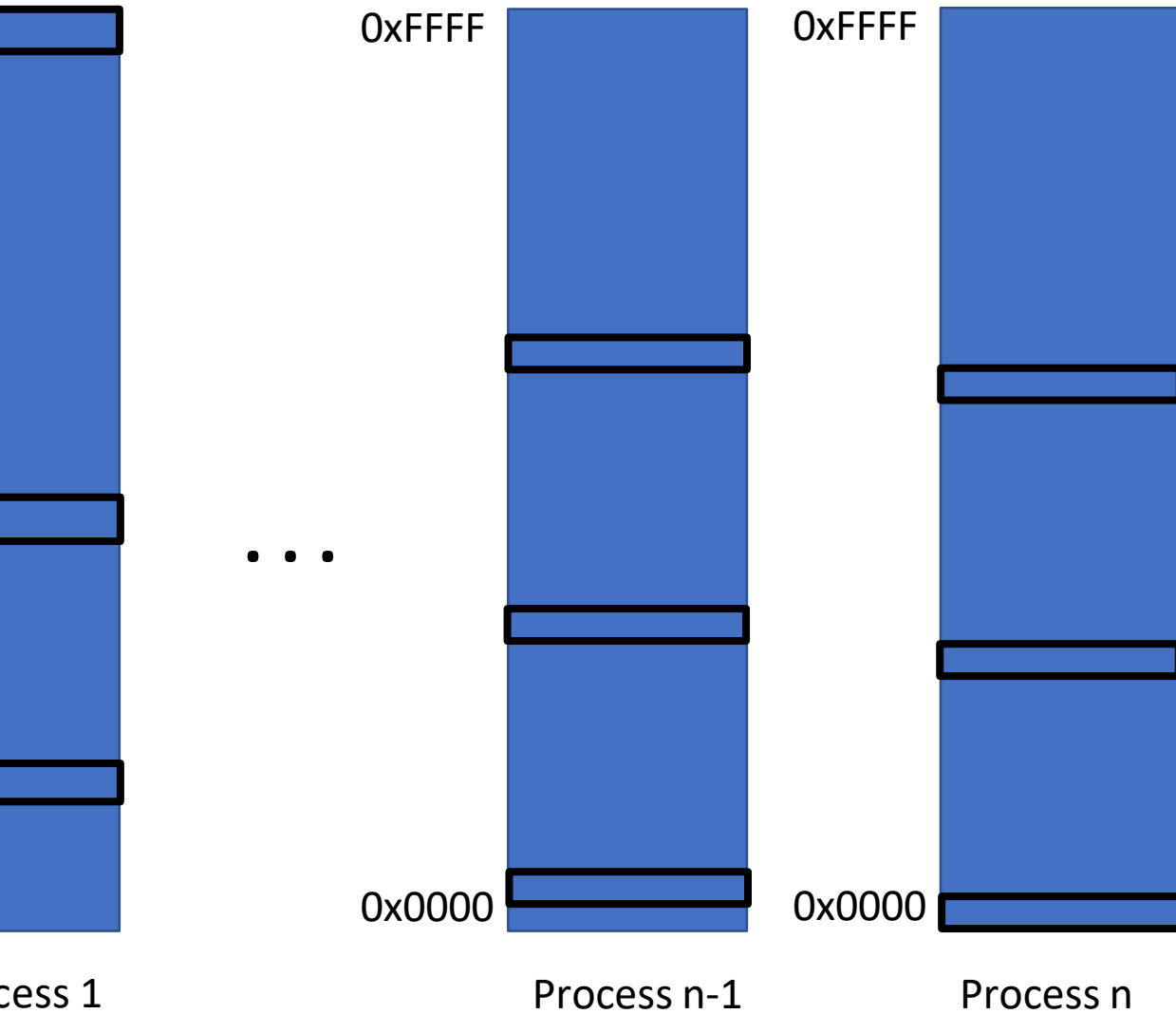
Statically partitioning the address space

Need to be sure that neither process will go and mess around with the other process' address ranges (isolation)

Need to use increasingly tiny partitions per process (abundance)

Need to know where your tiny partition starts so you can use it (uniformity)

First Attempt: Static Partitioning



Statically partitioning the address space

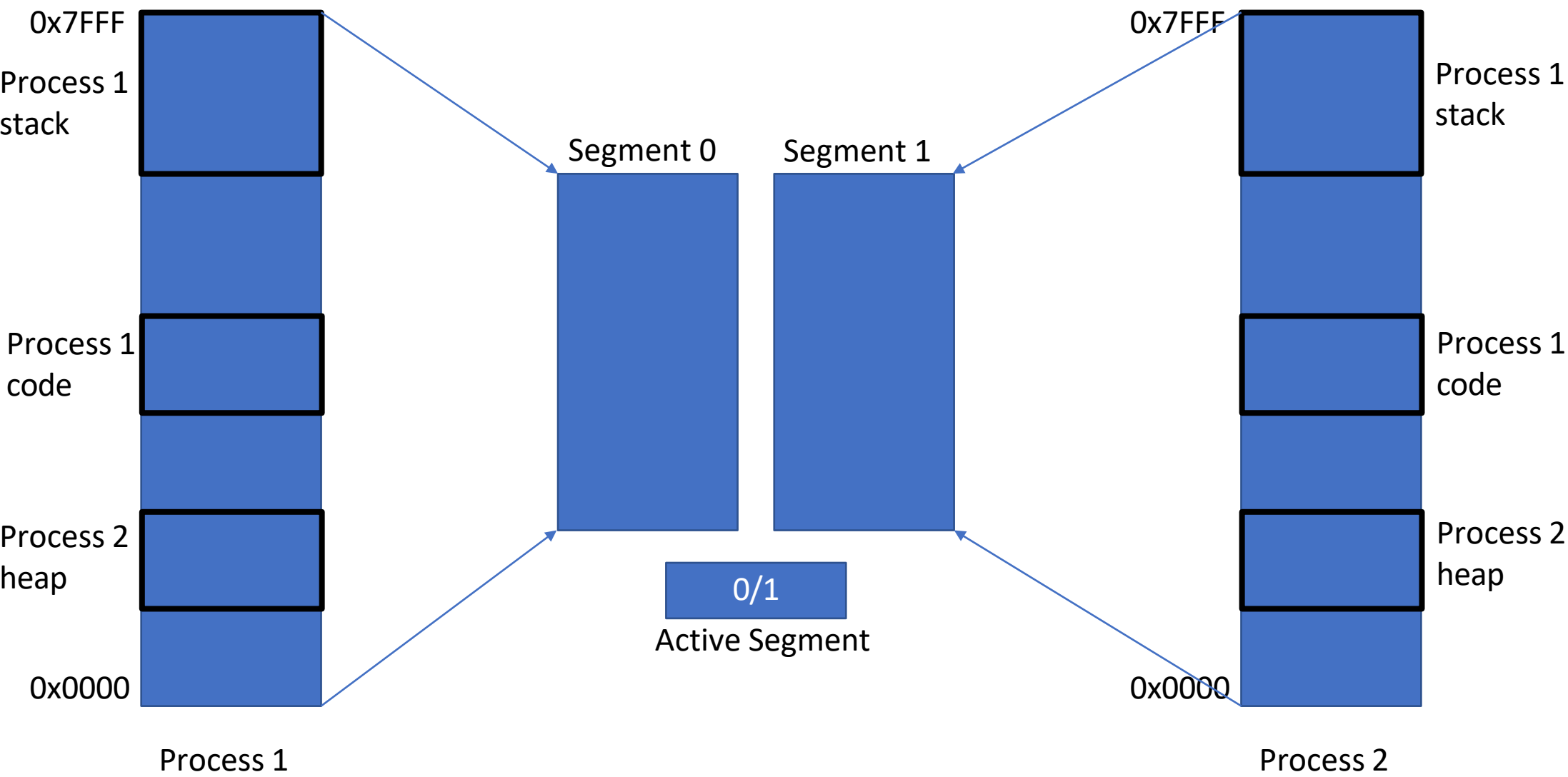
Need to be sure that neither process will go and mess around with the other process' address ranges (isolation)

Need to use increasingly tiny partitions per process (abundance)

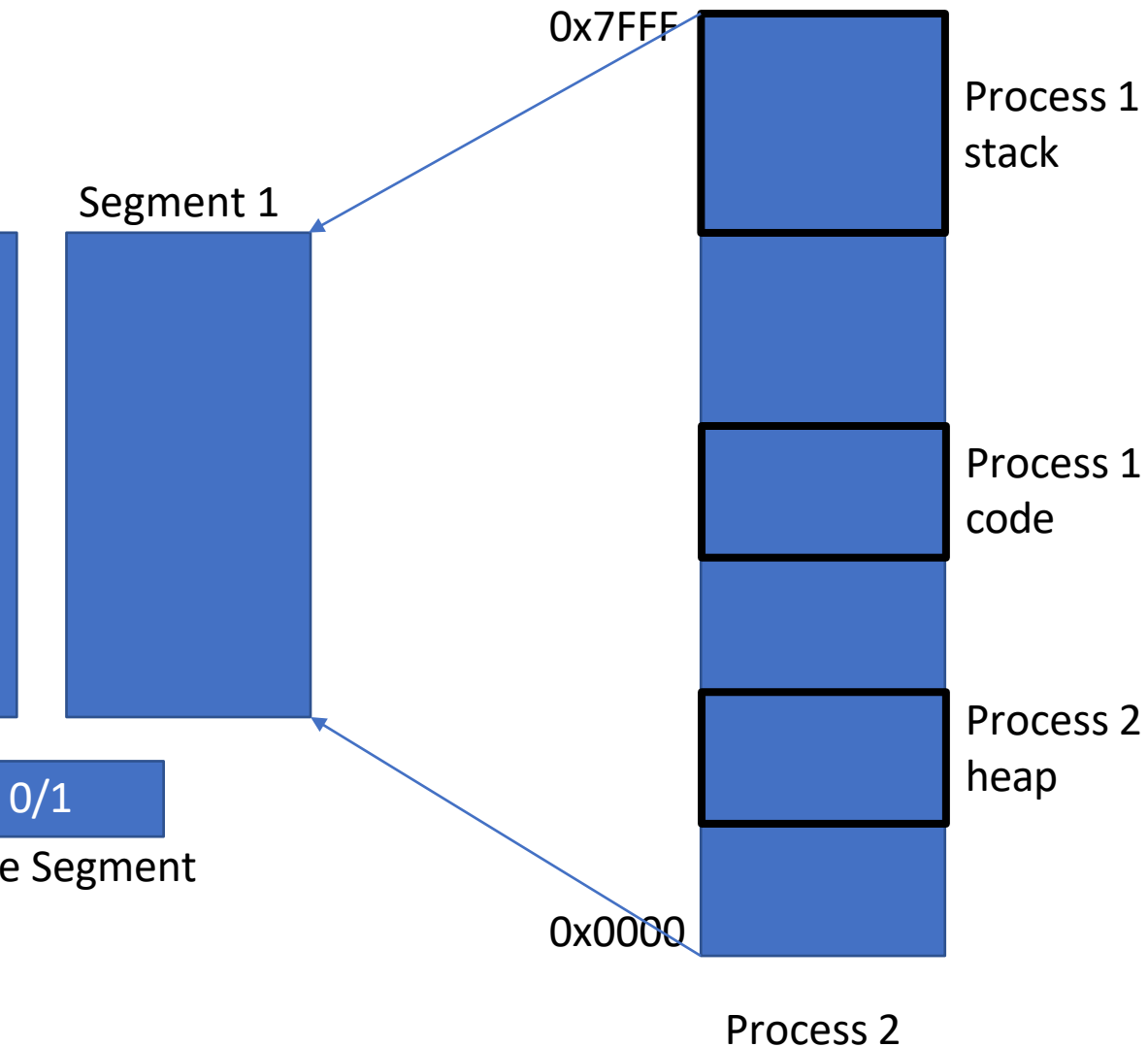
Need to know where your tiny partition starts so you can use it (uniformity).

Machine code can never refer to an address without knowing mix of other programs running on machine & where process loaded (uniformity, isolation)

Second Attempt: Segmented Memory [8086, IBM AS/400]



Second Attempt: Segmented Memory



Segment up the memory address space and switch segments

Benefit: Limited address size can address more memory (switch segment, another 16b space).

Abstraction of abundance.

Benefit: Processes can choose a segment and use predictable addresses off of that segment. **Abstraction of uniformity.**

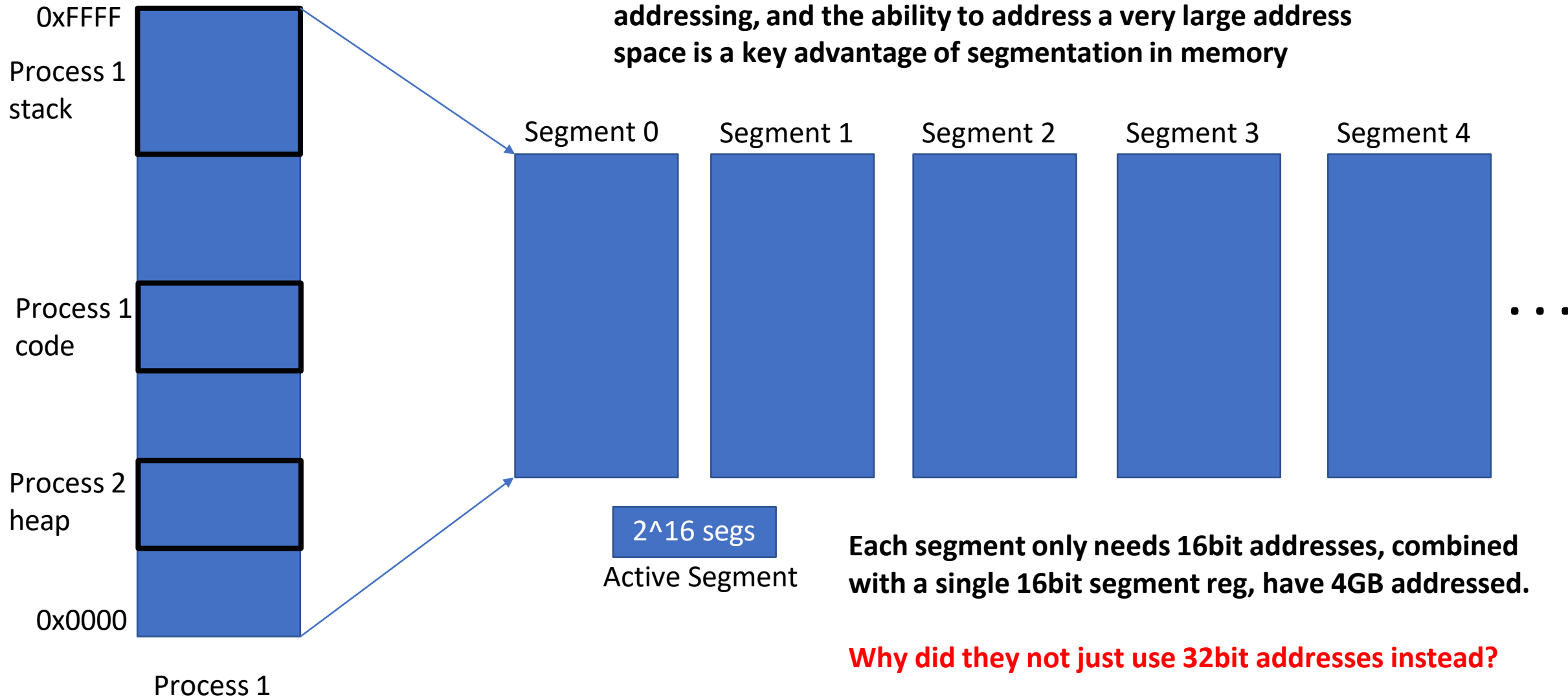
Benefit: If processes use independent segments, no interference.

Caveat: 8086 & others did not check permissions, segments could overlap. (isolation, abundance...)

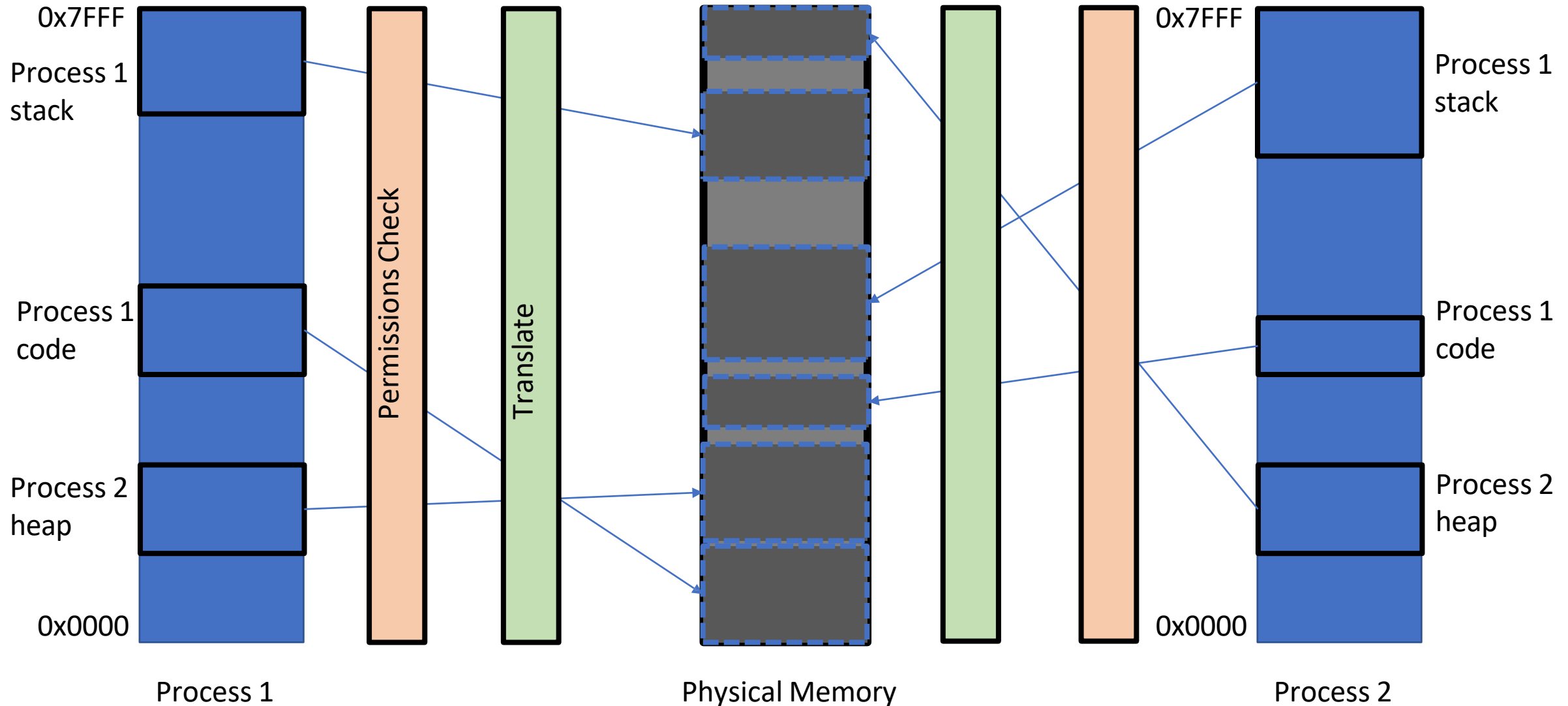
Caveat: need to select segment; how to choose which? (uniformity)

Second Attempt: Segmented Memory

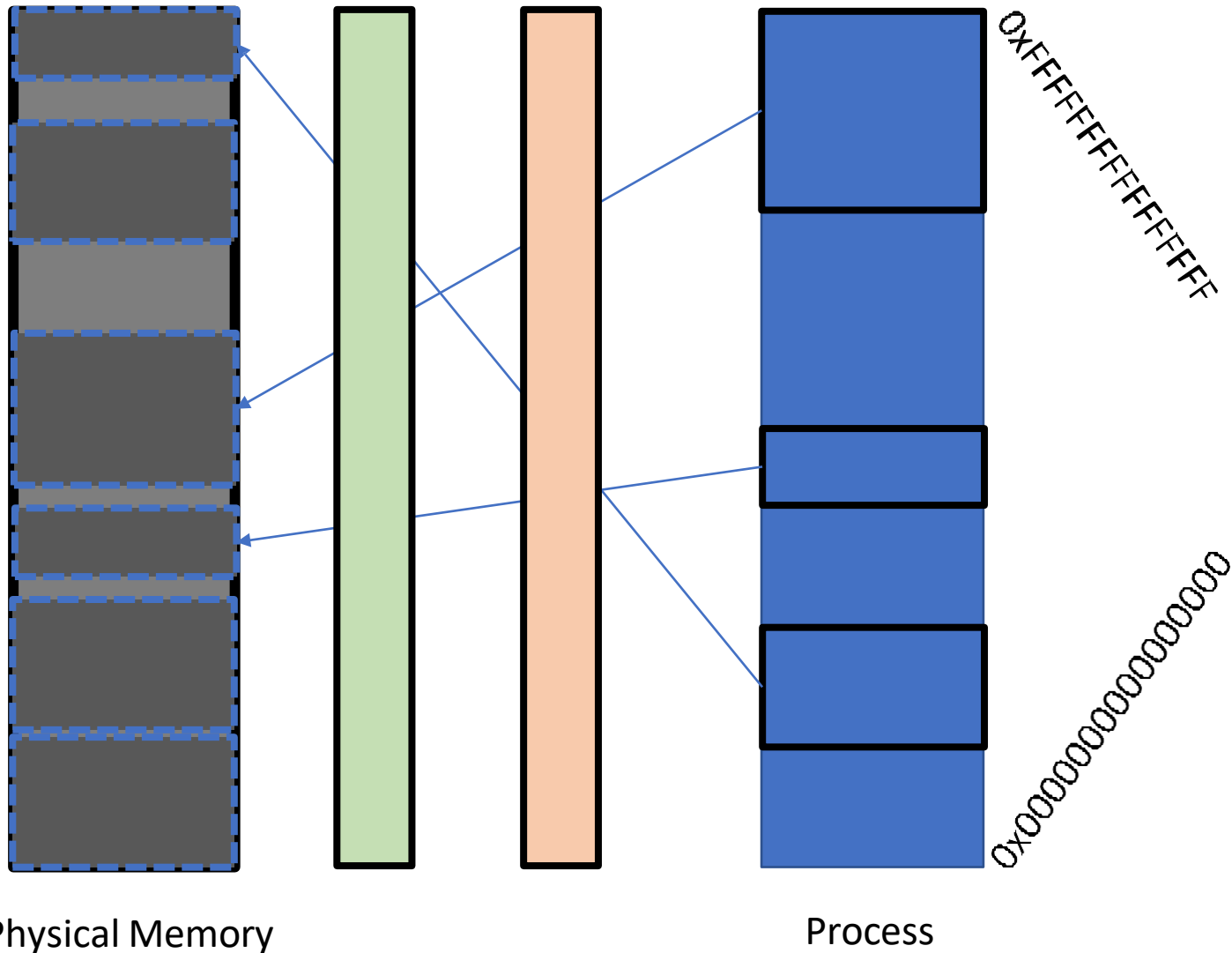
Abstraction of abundance, uniformity in segment relative addressing, and the ability to address a very large address space is a key advantage of segmentation in memory



Virtual Memory: Software Dynamic Address Translation (and Permission Checking)



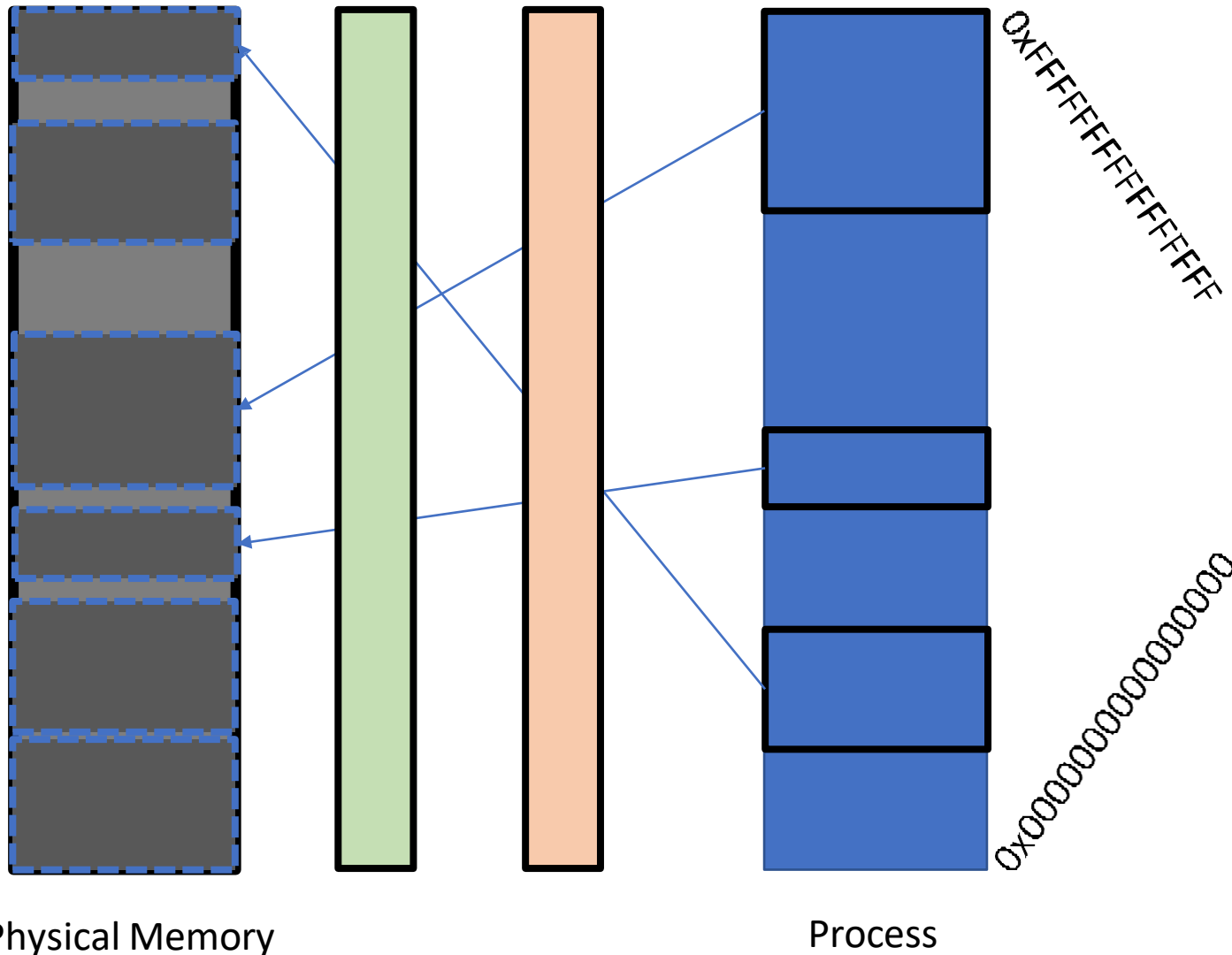
Virtual Memory: Software Dynamic Address Translation and Permission Checking



Key ideas behind virtual memory:

1. Physical memory acts like a cache of data that are mapped into process address space
2. Accesses always refer to VAs and VM translates them to usable physical addresses
3. Mapping makes a virtual address range accessible & unmapped regions are inaccessible
4. Virtual memory happens at granularity of pages (i.e., 4kB chunks of memory)
5. Page table entry per page contains: (1) is it in physical memory? (2) at what address? (3) with what access permissions?

Virtual Memory: Software Dynamic Address Translation and Permission Checking



On every memory access, translate memory address from *virtual* address to *physical* address

Benefit: Arbitrary hierarchy of memories / storage can back program data **Abstraction of abundance.**

Benefit: All processes have identical linear virtual address space that can use predictable addresses always. **Abstraction of uniformity.**

Benefit: Per-process address space are private by default. **Abstraction of isolation.**

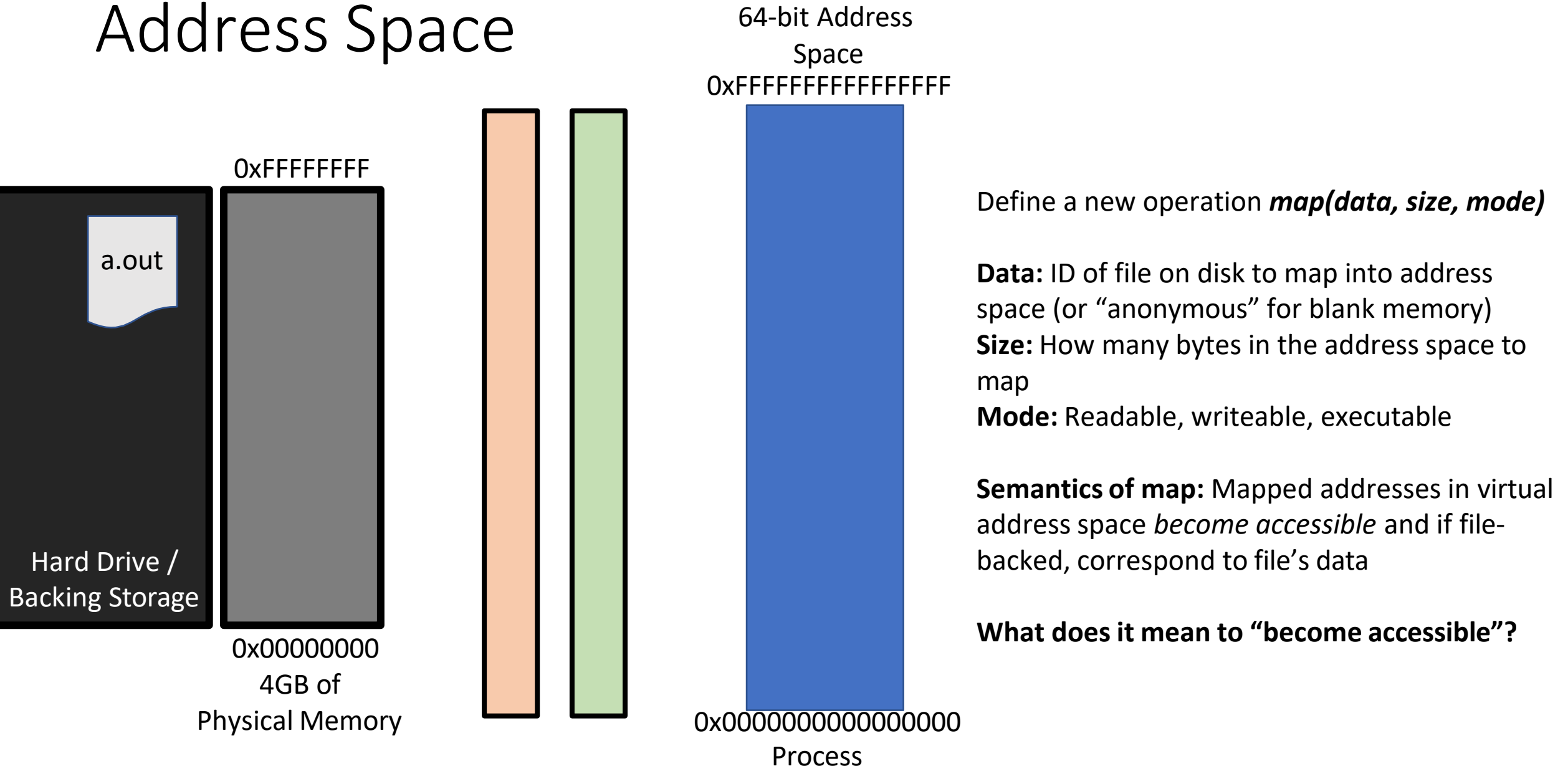
Caveat: need mechanism for mapping data in

Caveat: translation & permissions are *dynamic*

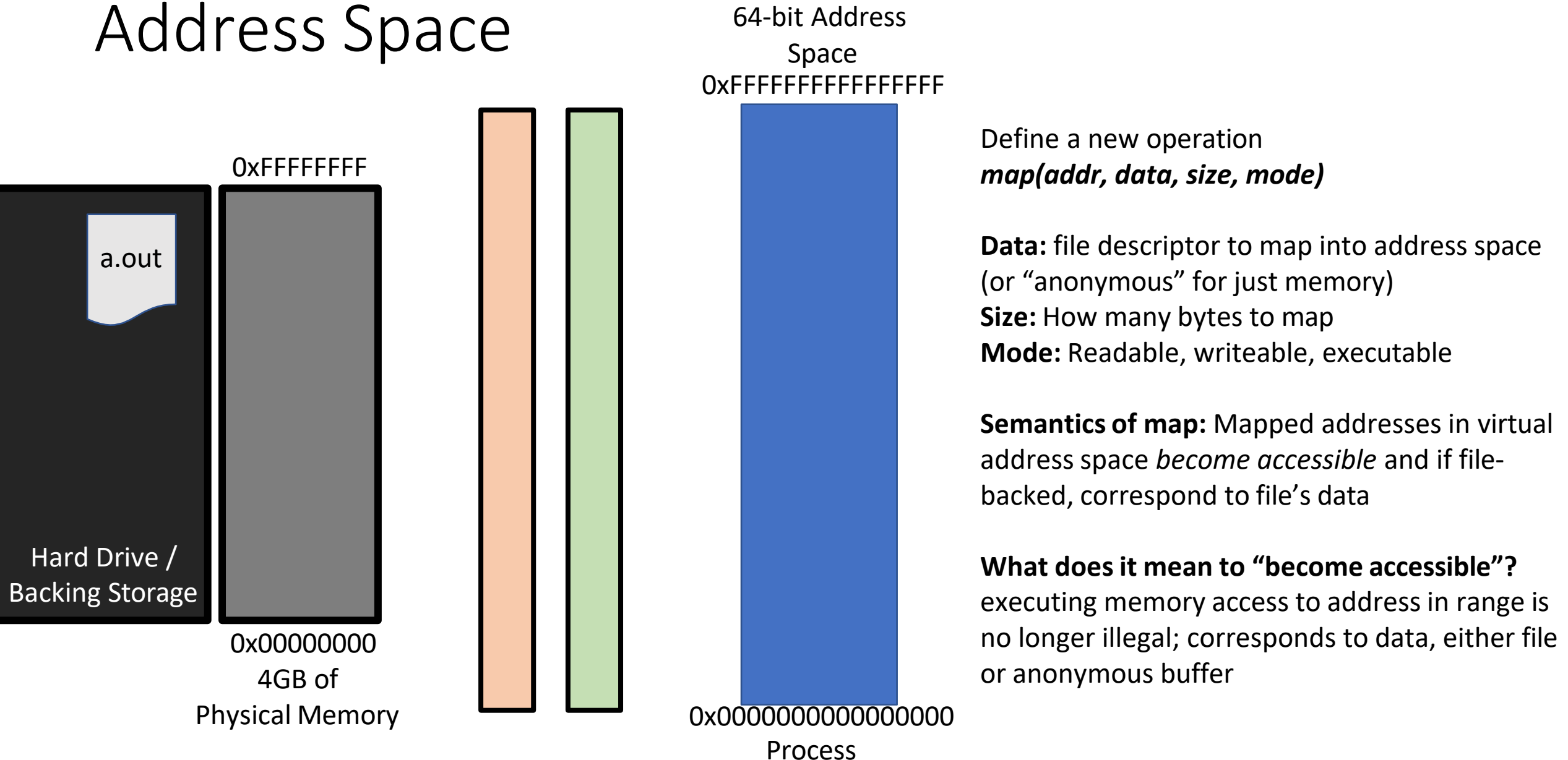
Caveat: translation granularity (i.e., page size) is a system-wide parameter

Mapping Data Into Virtual Address Space

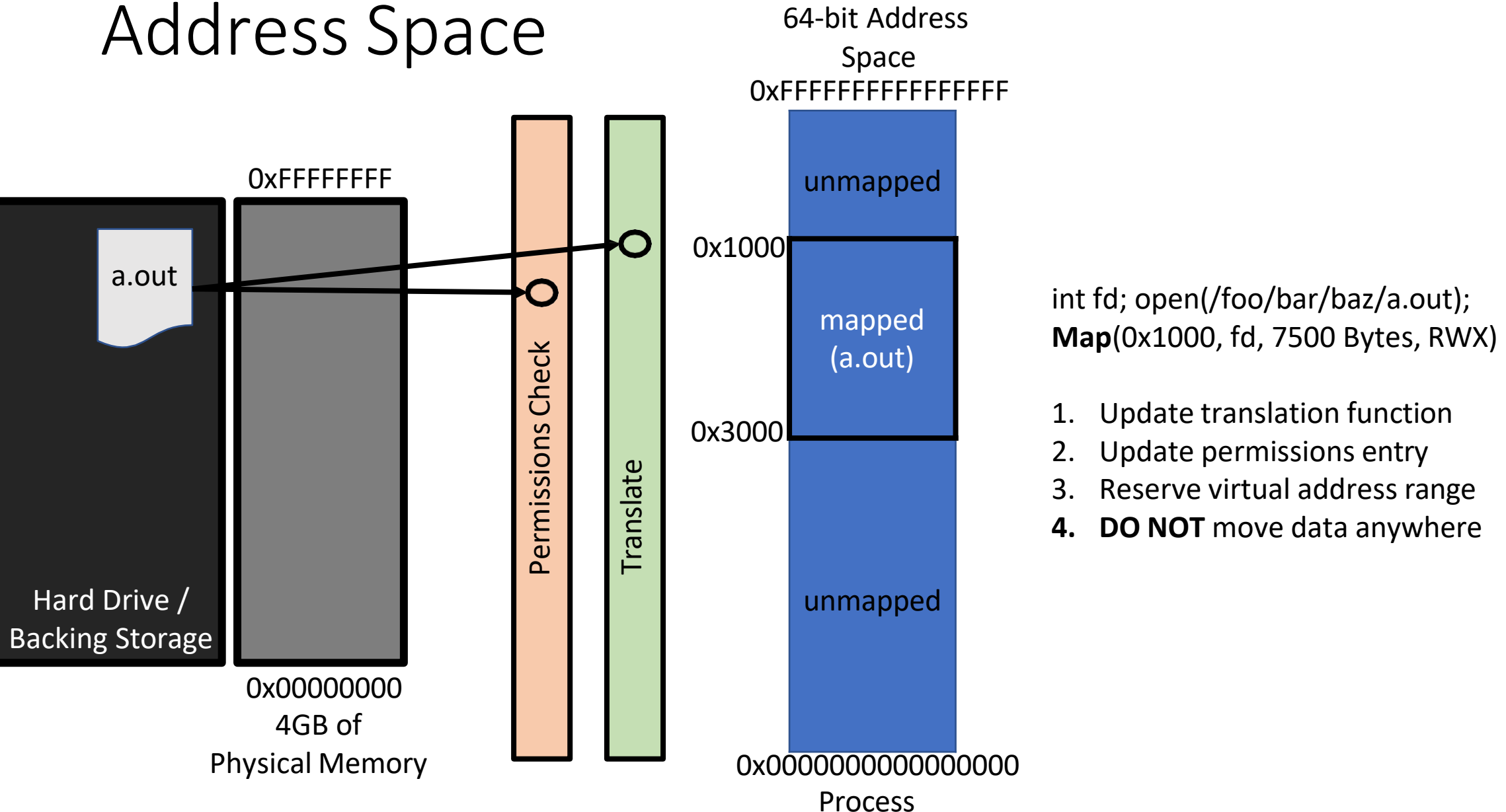
Virtual Memory: Mapping Data into Virtual Address Space



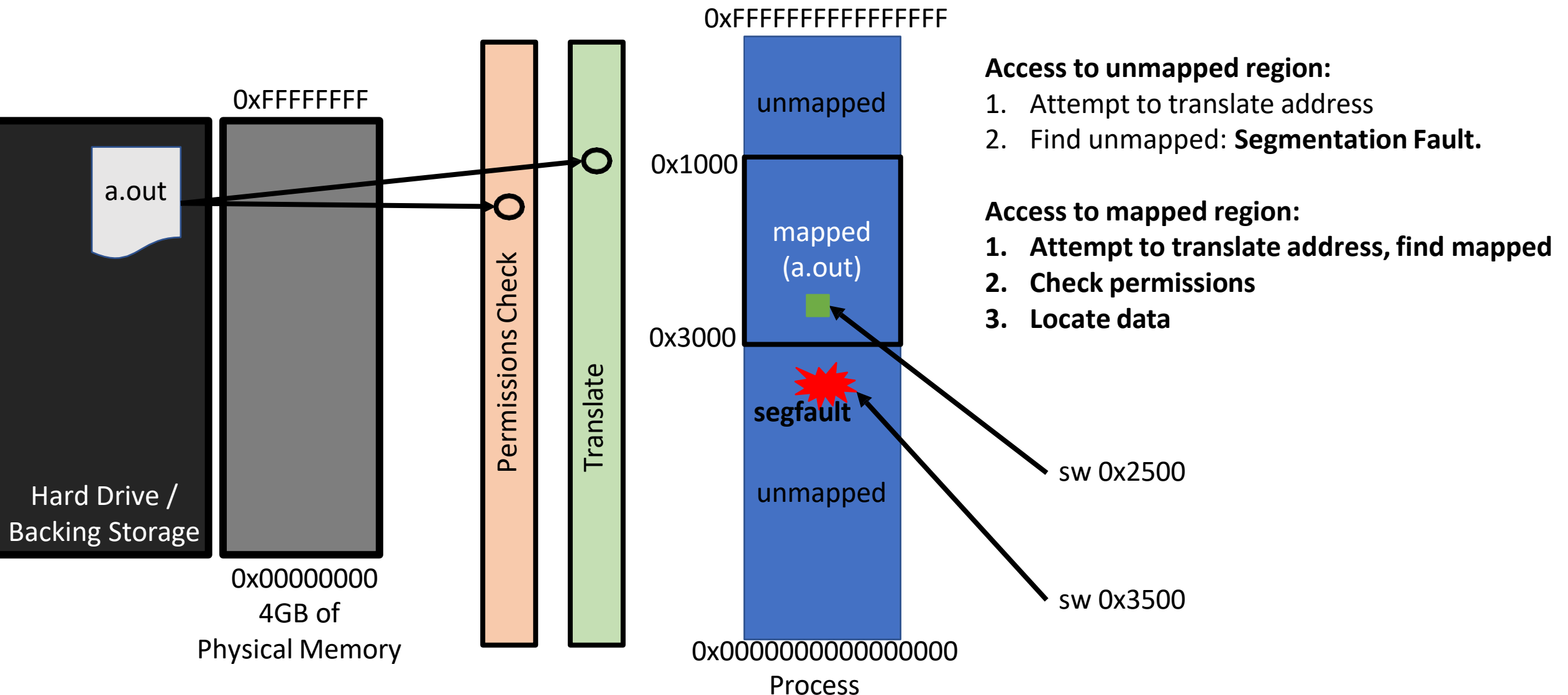
Virtual Memory: Mapping Data into Virtual Address Space



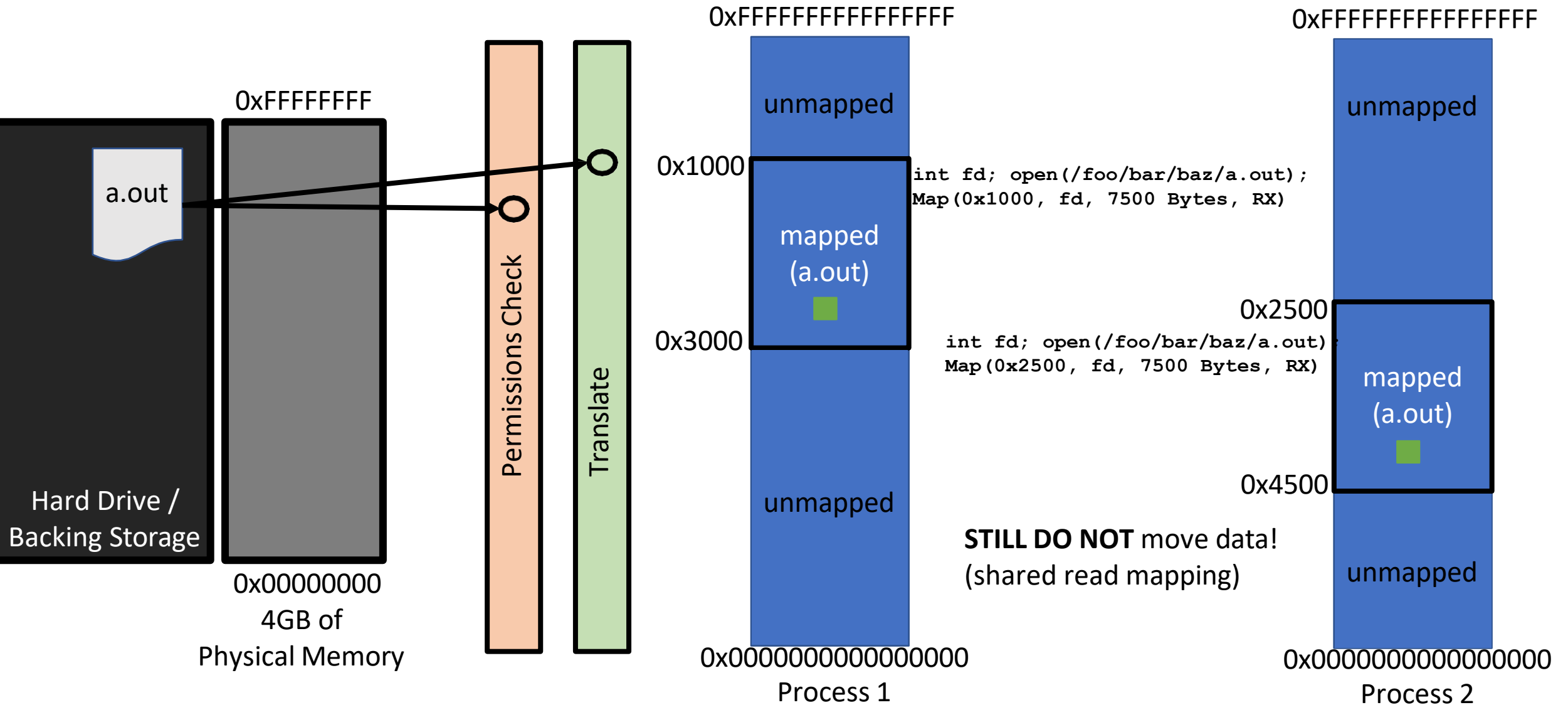
Virtual Memory: Mapping Data into Virtual Address Space



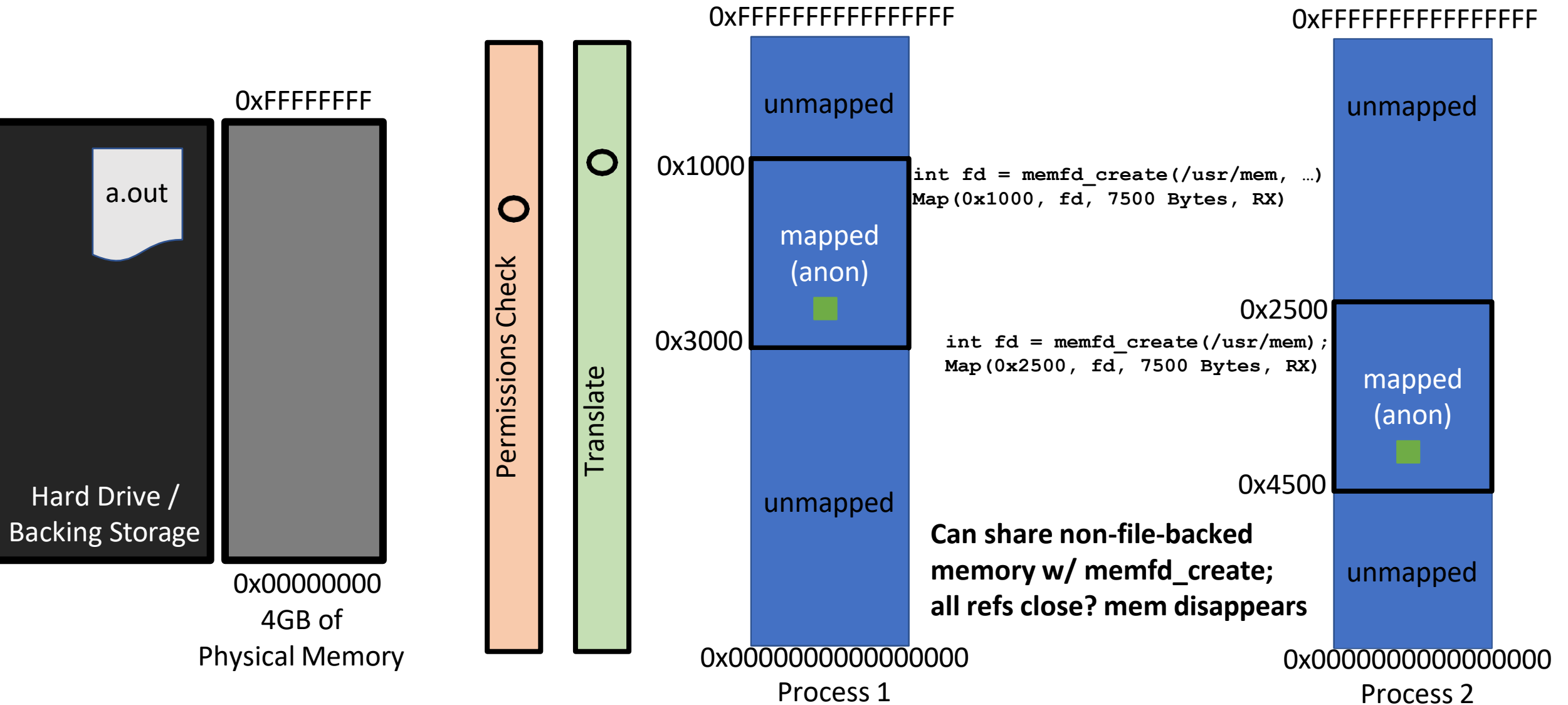
Virtual Memory: Accessing Data Mapped into the Virtual Address Space



Virtual Memory: Shared Mapping of File-backed Data into Address Space by Multiple Processes

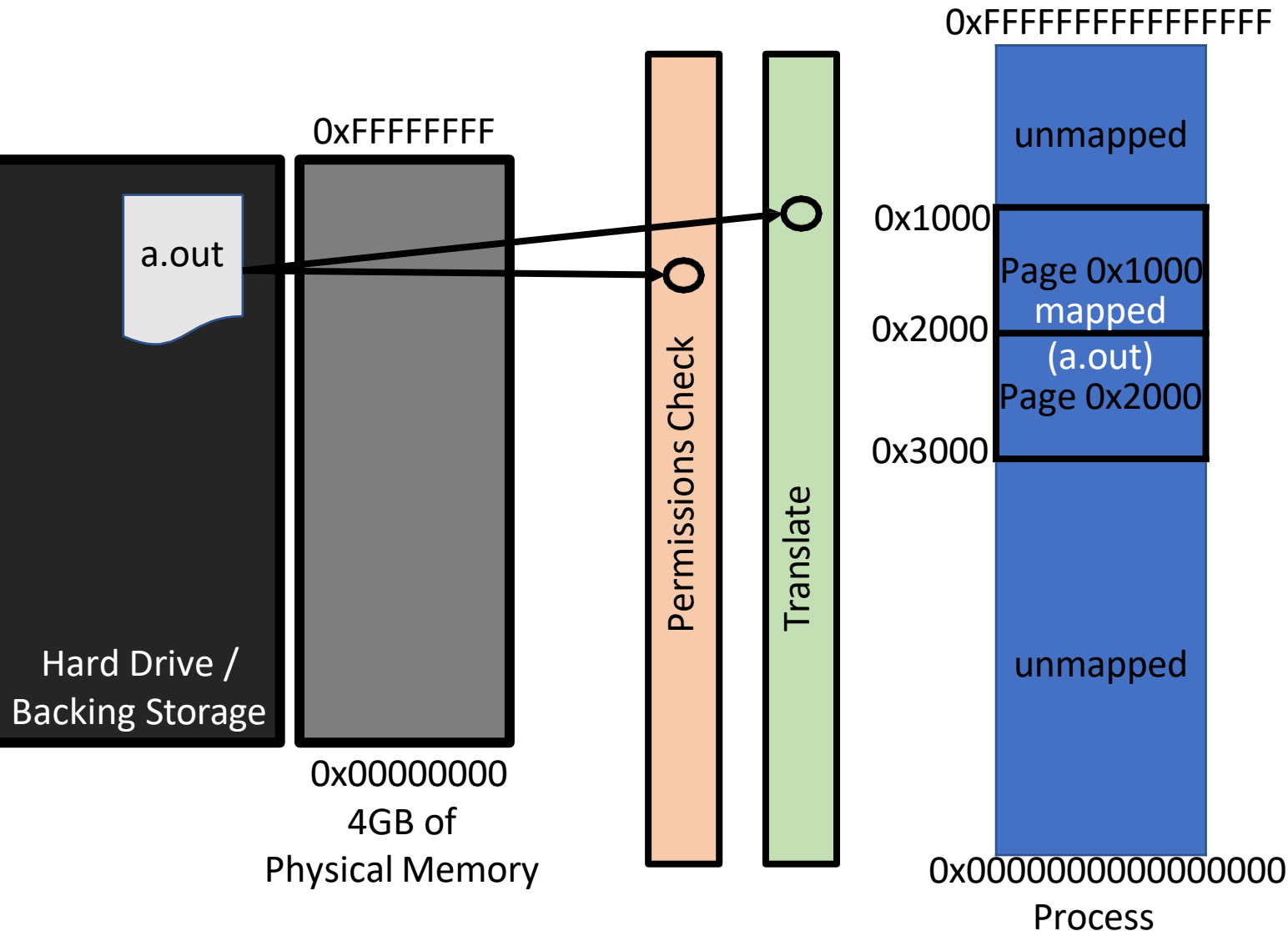


Virtual Memory: Shared Mapping of Anonymous Data into Address Space by Multiple Processes



Page Granularity for Translation (& Permissions)

Virtual Memory: Translation & Permissions at Page Granularity



Still not talking about getting actual data yet;
first need to translate (we will talk about how
data moves in a few slides)

Translation happens at *page* granularity

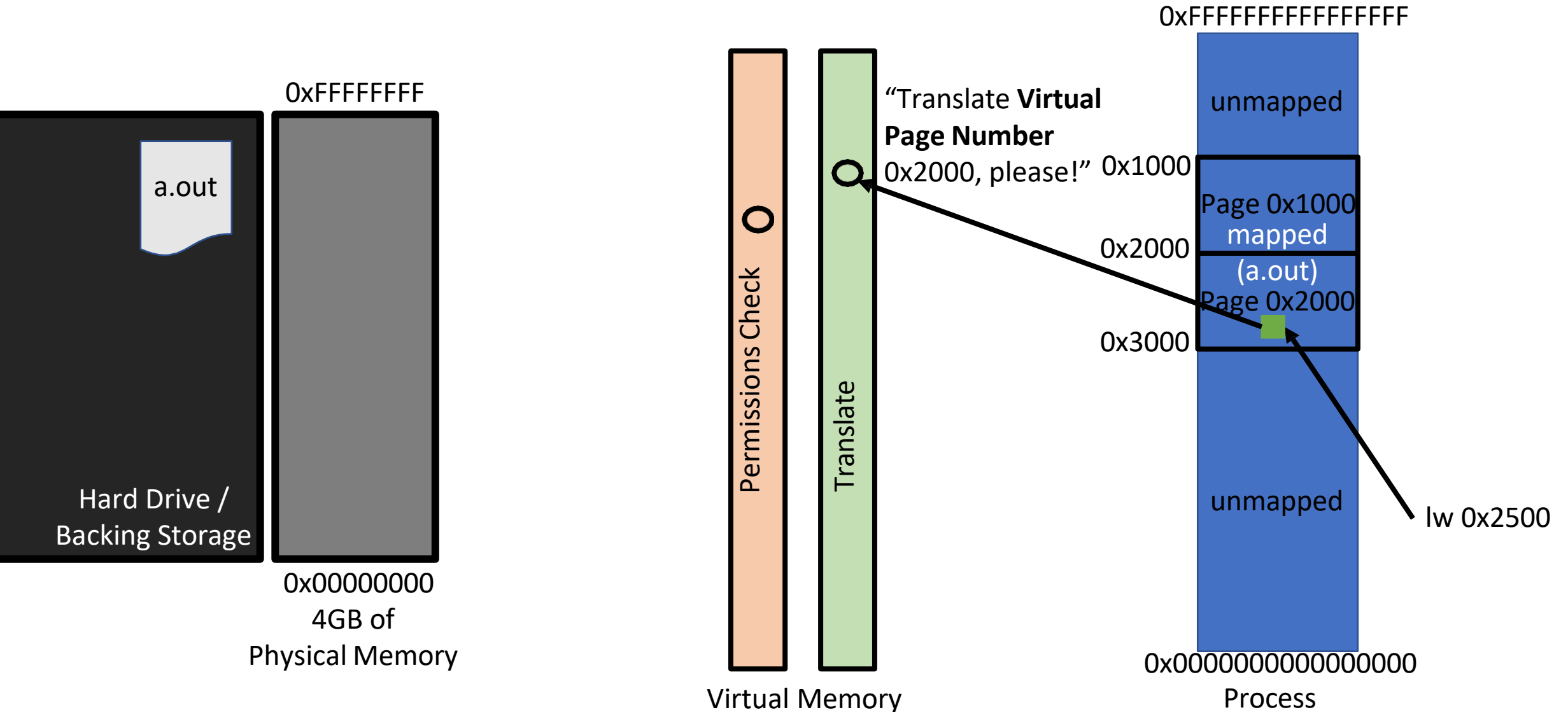
Pages are usually $2^{12} = 4096$ bytes

Memory access:

1. Look up translation for page virtual address
2. Find & fetch data after translation

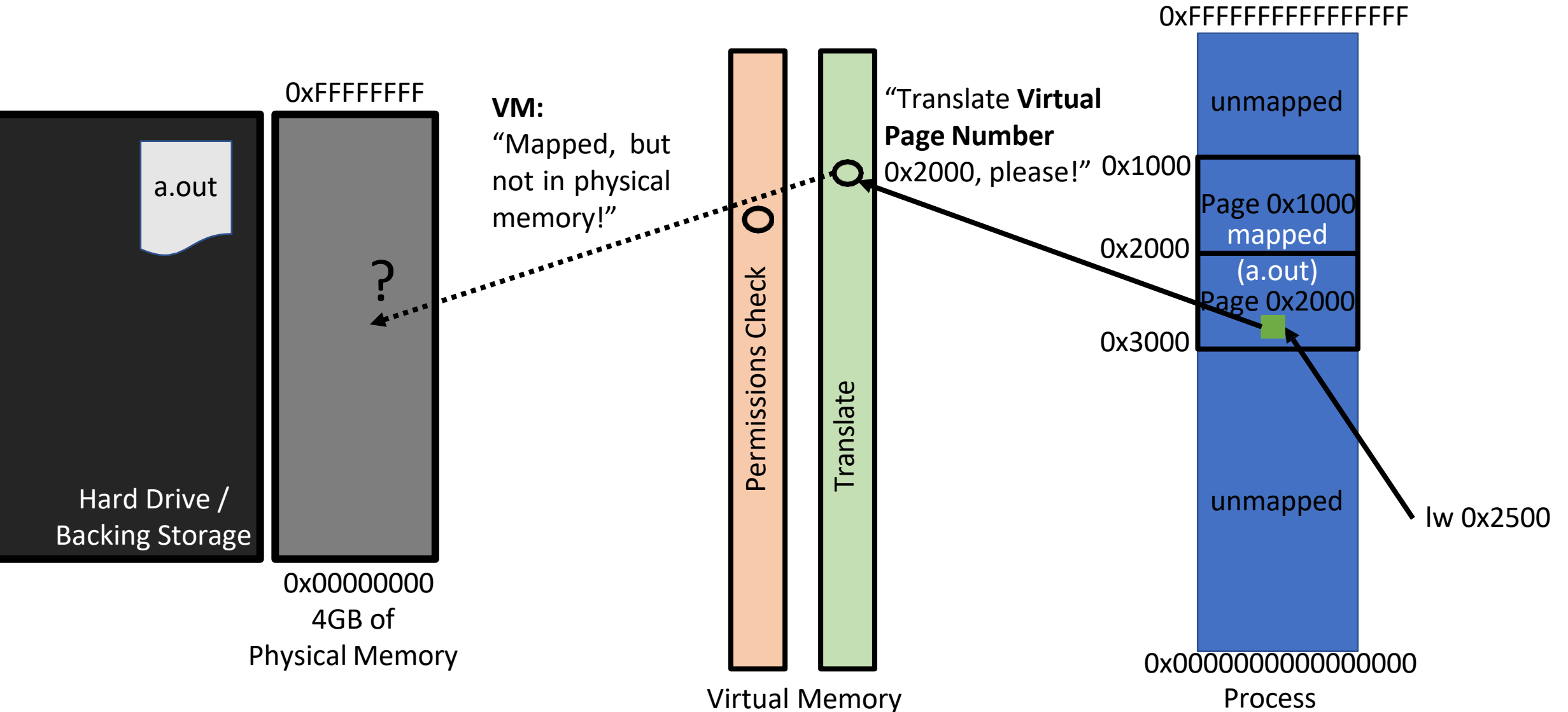
Virtual Memory: Translation and Finding Data

Outcome #1: Page Not in Physical Memory



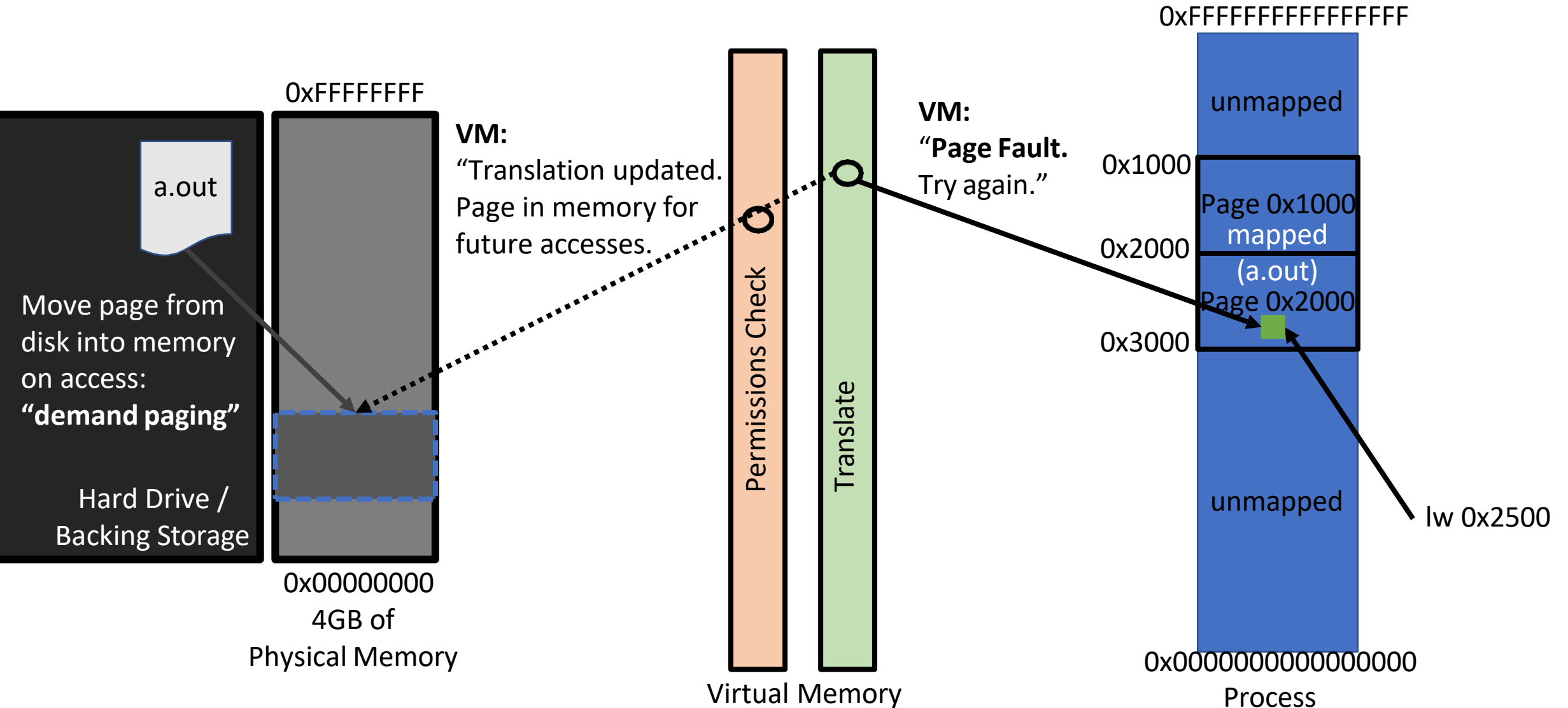
Virtual Memory: Translation and Finding Data

Outcome #1: Page Not in Physical Memory



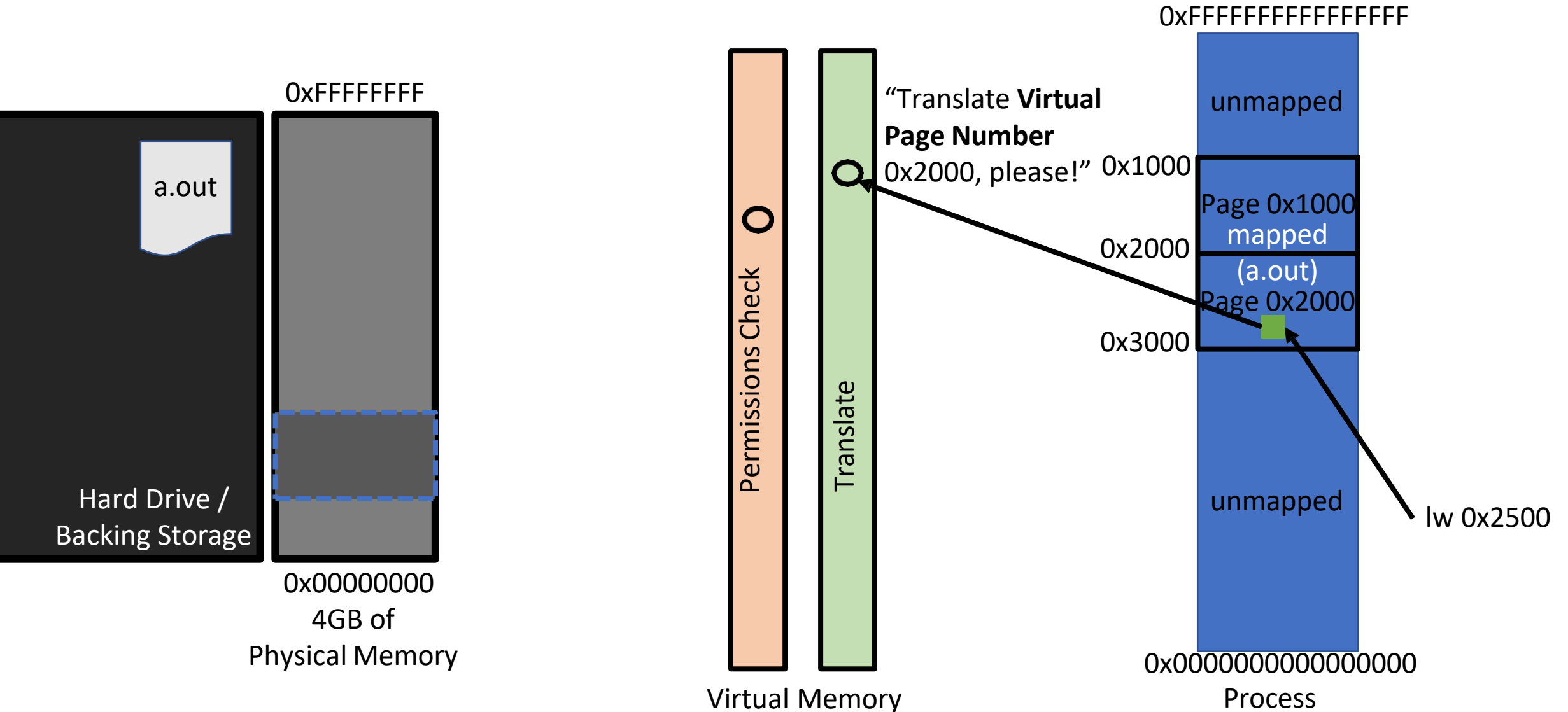
Virtual Memory: Translation and Finding Data

Outcome #1: Page Not in Physical Memory



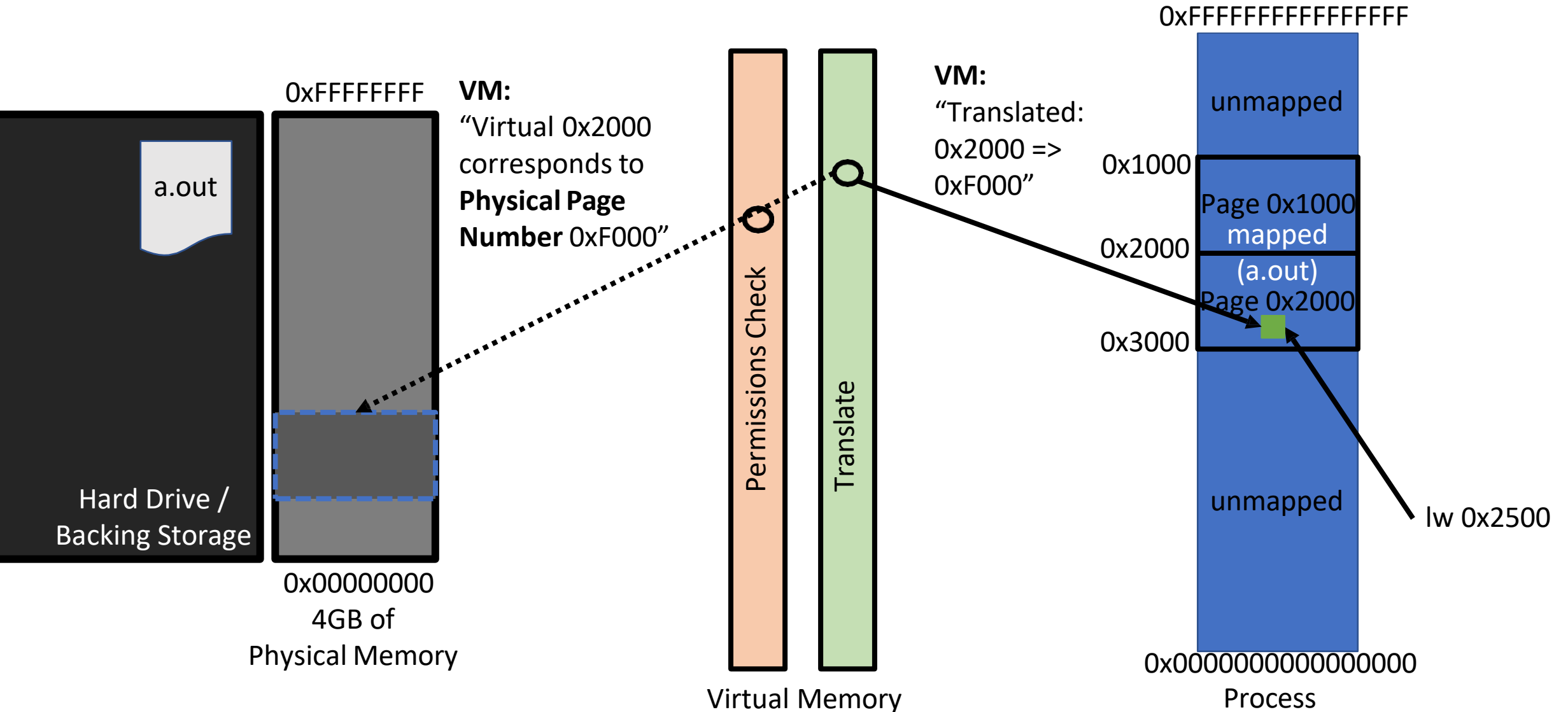
Virtual Memory: Translation and Finding Data

Outcome #2: Page in Physical Memory



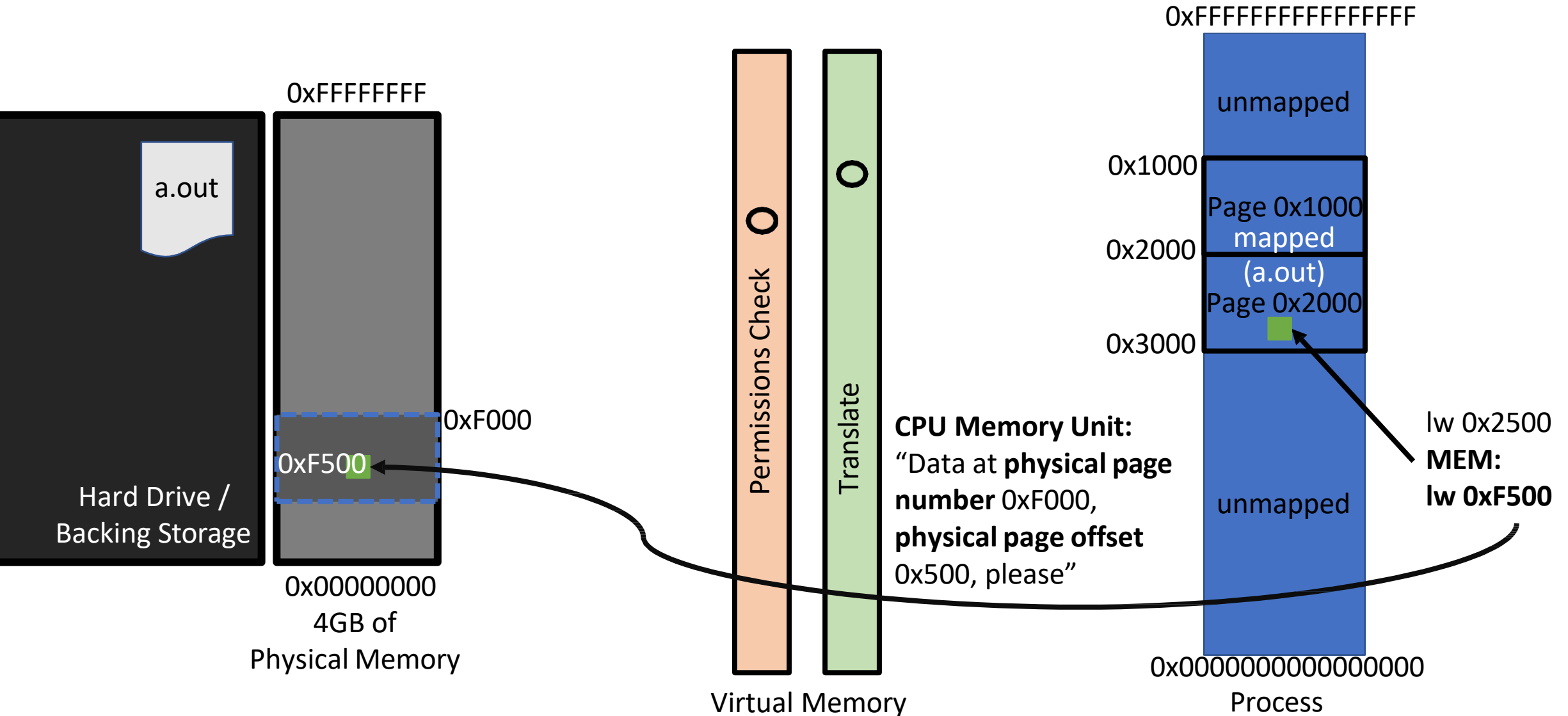
Virtual Memory: Translation and Finding Data

Outcome #2: Page in Physical Memory



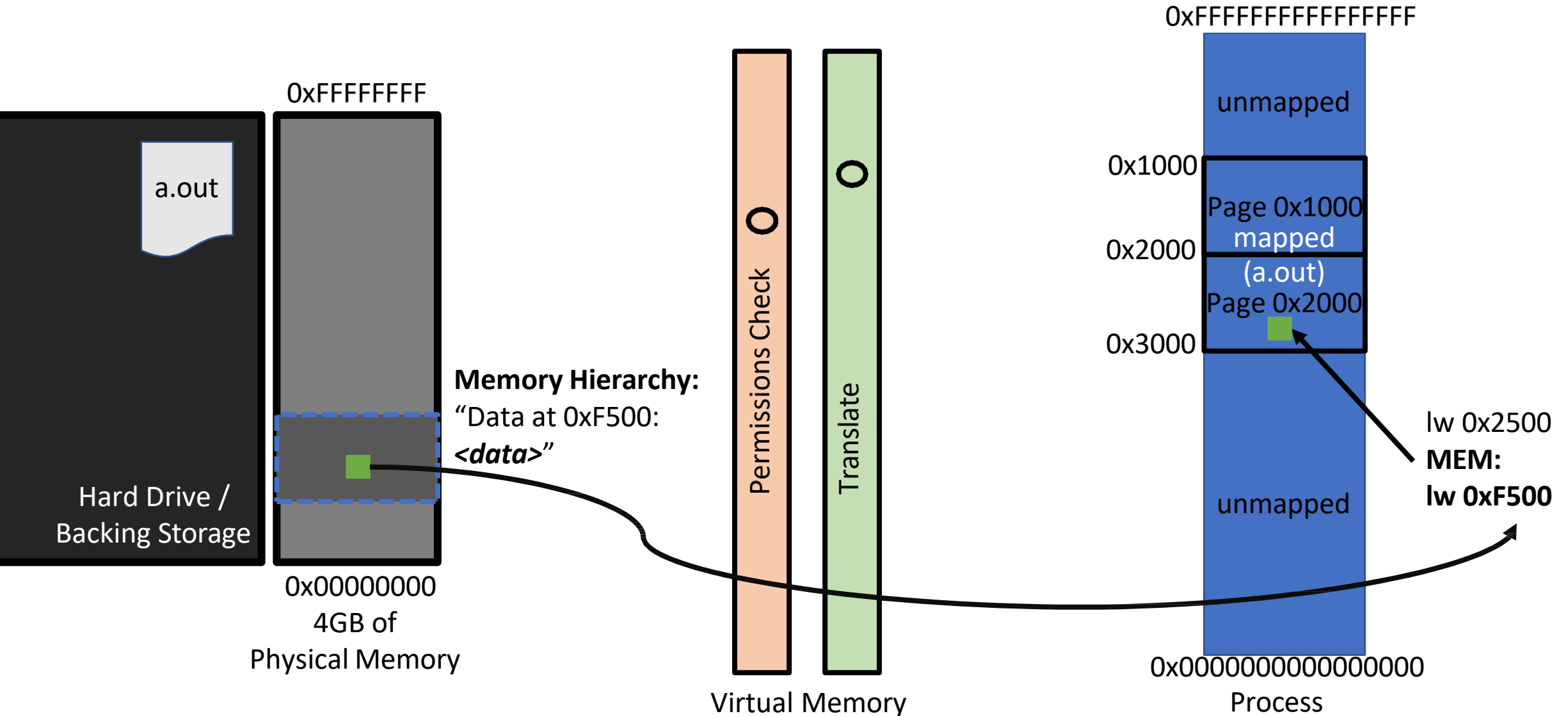
Virtual Memory: Translation and Finding Data

Outcome #2: Page in Physical Memory



Virtual Memory: Translation and Finding Data

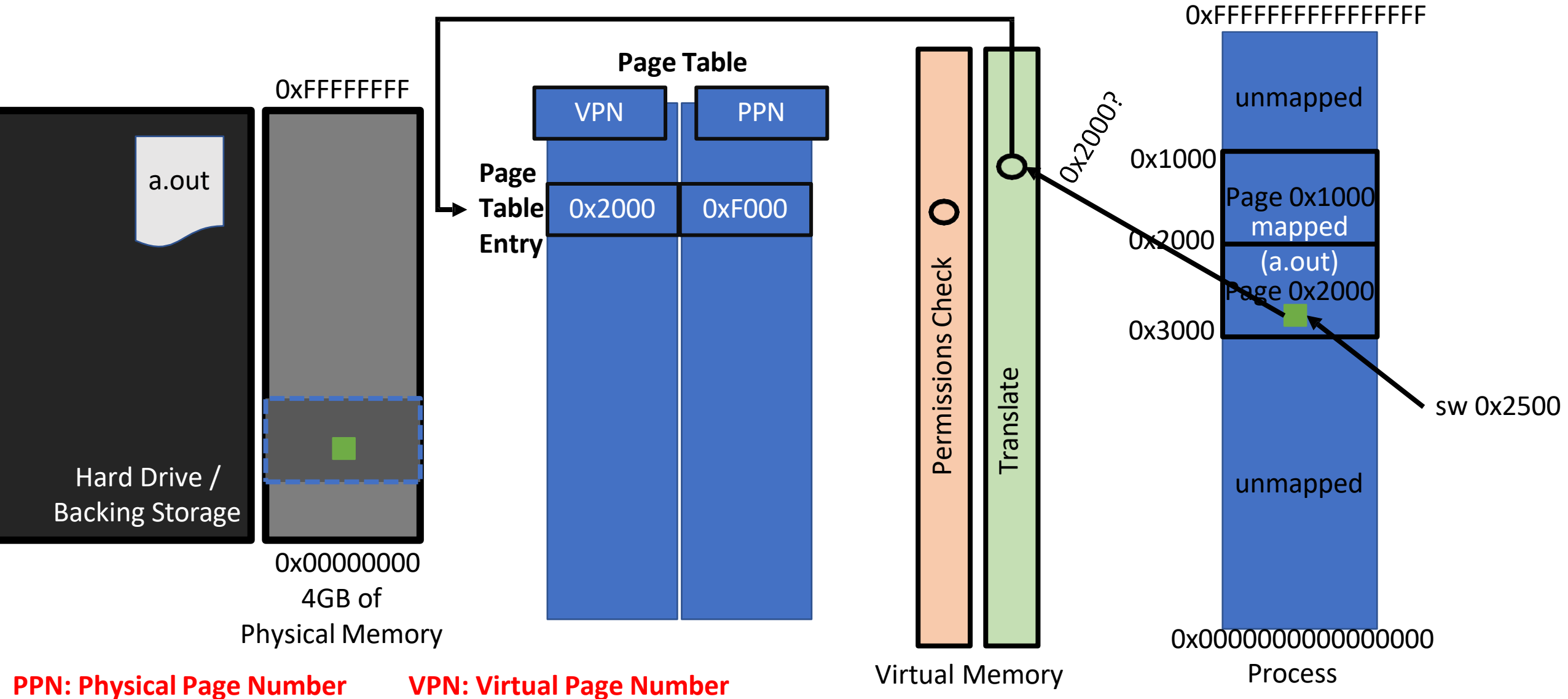
Outcome #2: Page in Physical Memory



The Translation Function & Its Use

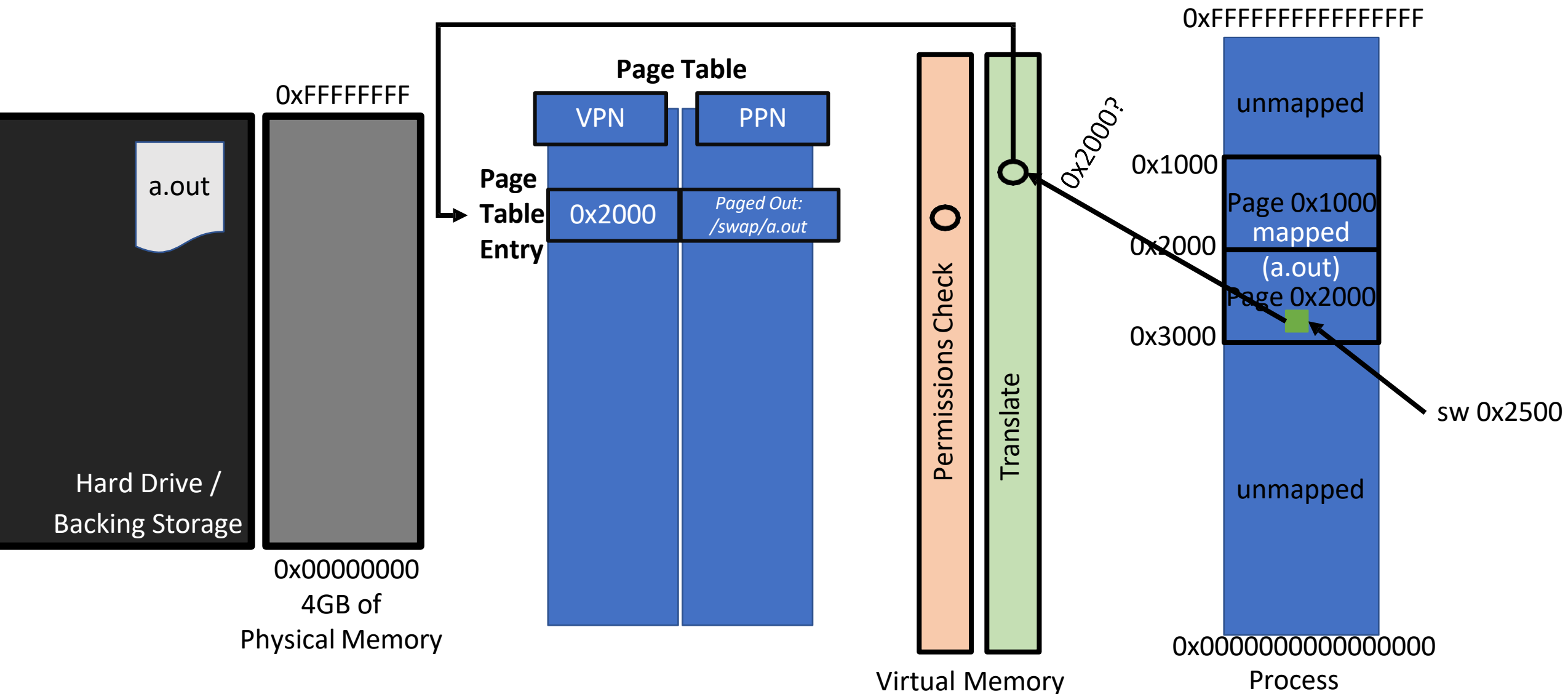
Virtual Memory: The Translation Function

Page Table Stores Translation for Paged-In Data



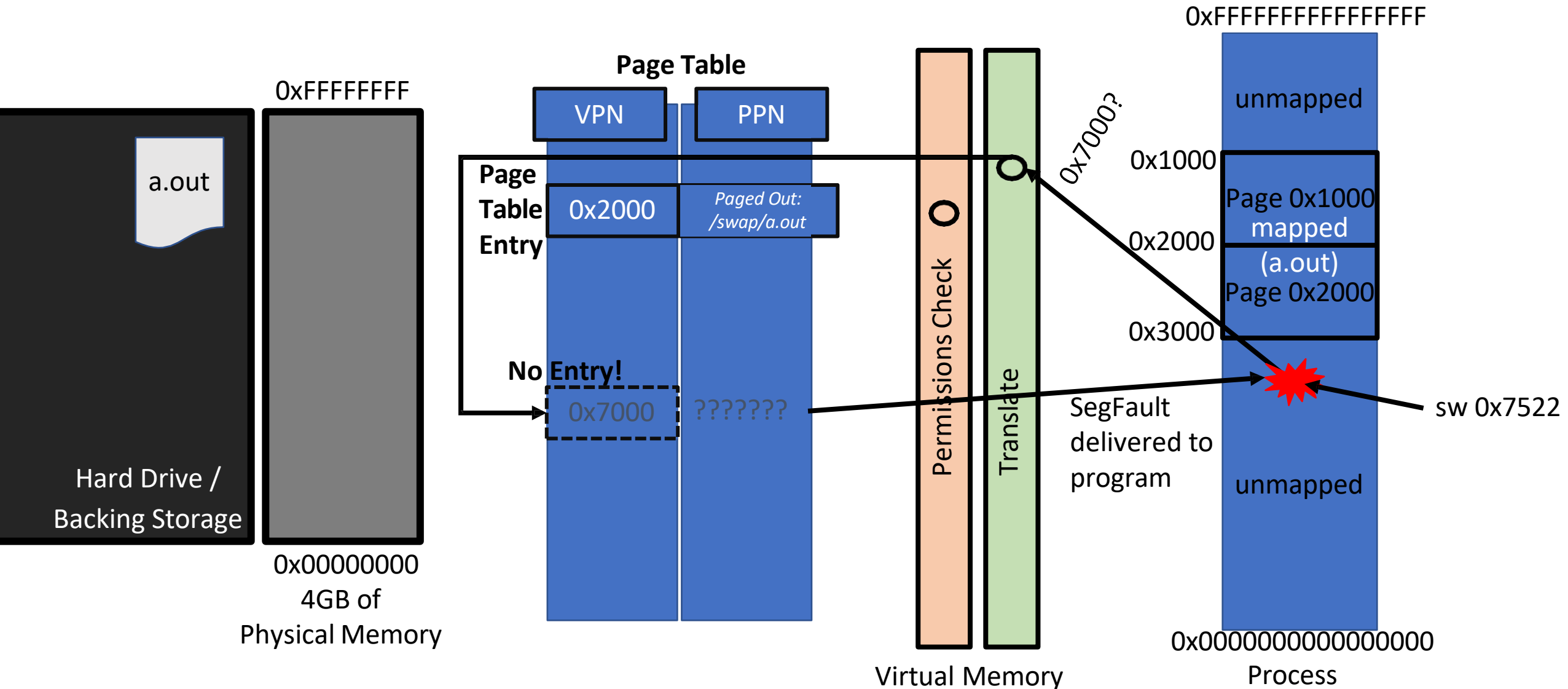
Virtual Memory: The Translation Function

Page Table Holds Disk Location for Paged-Out Data

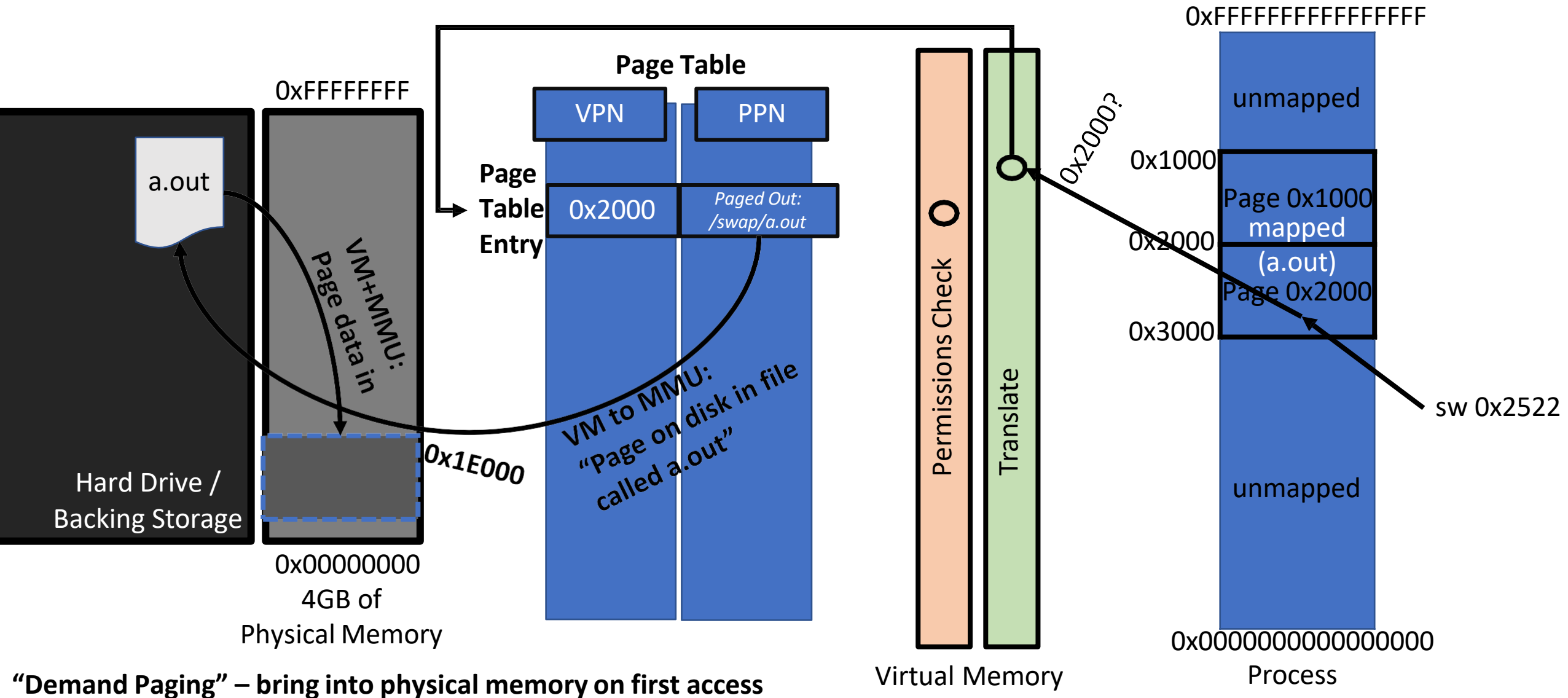


Virtual Memory: The Translation Function

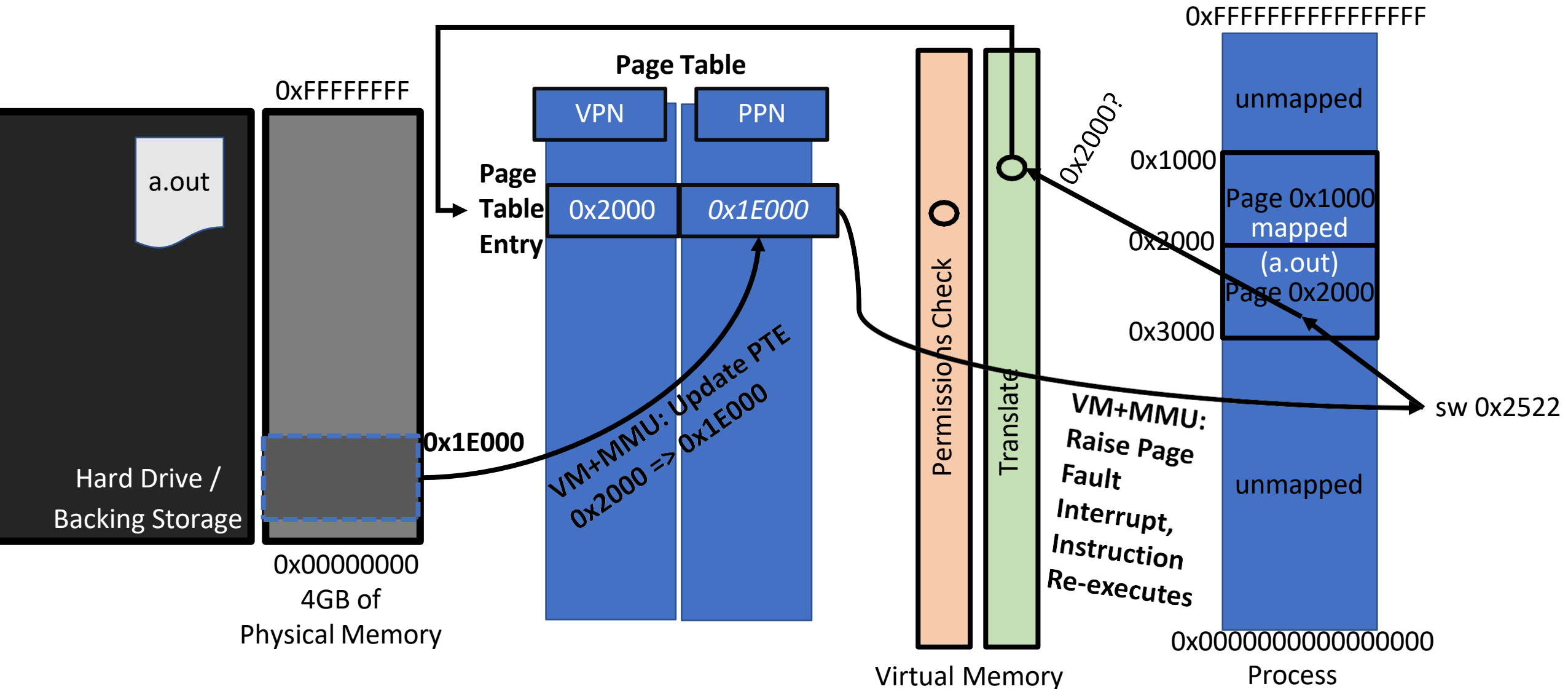
Page Table Holds No Entry for Unmapped Data



Physical Memory as a Cache of Data on Disk: Cache Miss Means Page Fault (1/2)



Physical Memory as a Cache of Data on Disk: Cache Miss Means Page Fault (2/2)

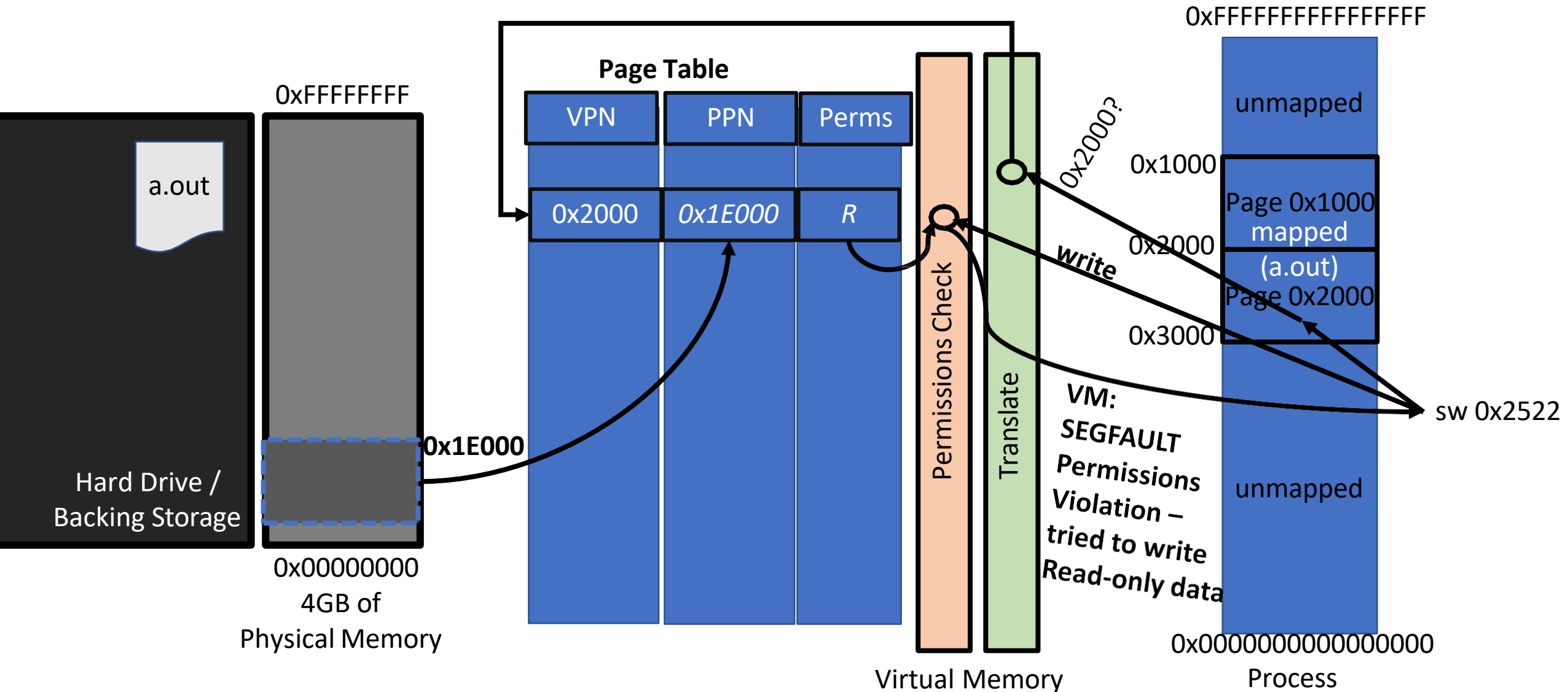


Virtual Memory Translation Algorithmically

```
vmTranslate(vaddr) {  
    //Compute Virtual Page Number & Virtual Page Offset  
    //from vaddr assuming  $2^{12}$  page size  
    (VPN,VPO) = (vaddr[63:12],vaddr[11:0])  
    PPN = PT.lookup(VPN)  
  
    if PPN == UNMAPPED:  
        kill(SIGSEGV)  
    else if PPN == PAGEDOUT@<diskloc>:  
        MMU.pageIn(VPN,PPN,<diskloc>) // move diskloc data to  
                                     // phys @ PPN, update PTE for VPN  
        MMU.raiseInterrupt(PAGE_FAULT, ...)  
        //Semantics of interrupt: replay instruction that caused interrupt  
        //In Lab 3 emulation: page data in, record page fault  
  
    else  
        return PPN //PTE contained usable VPN; hooray! MMU tells CPU the PPN  
}
```

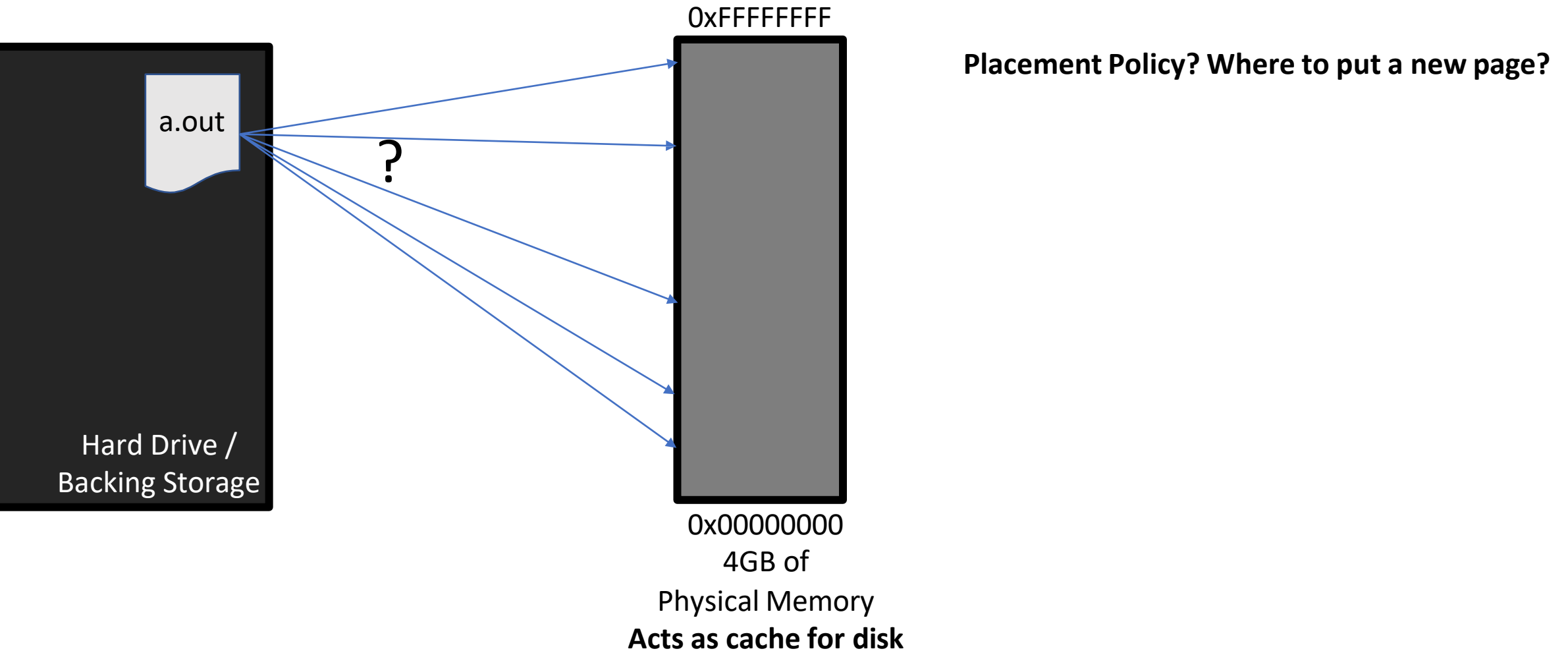
Permissions Checking

Permissions Checking Happens with Page Translation (Compare access type to permissions)

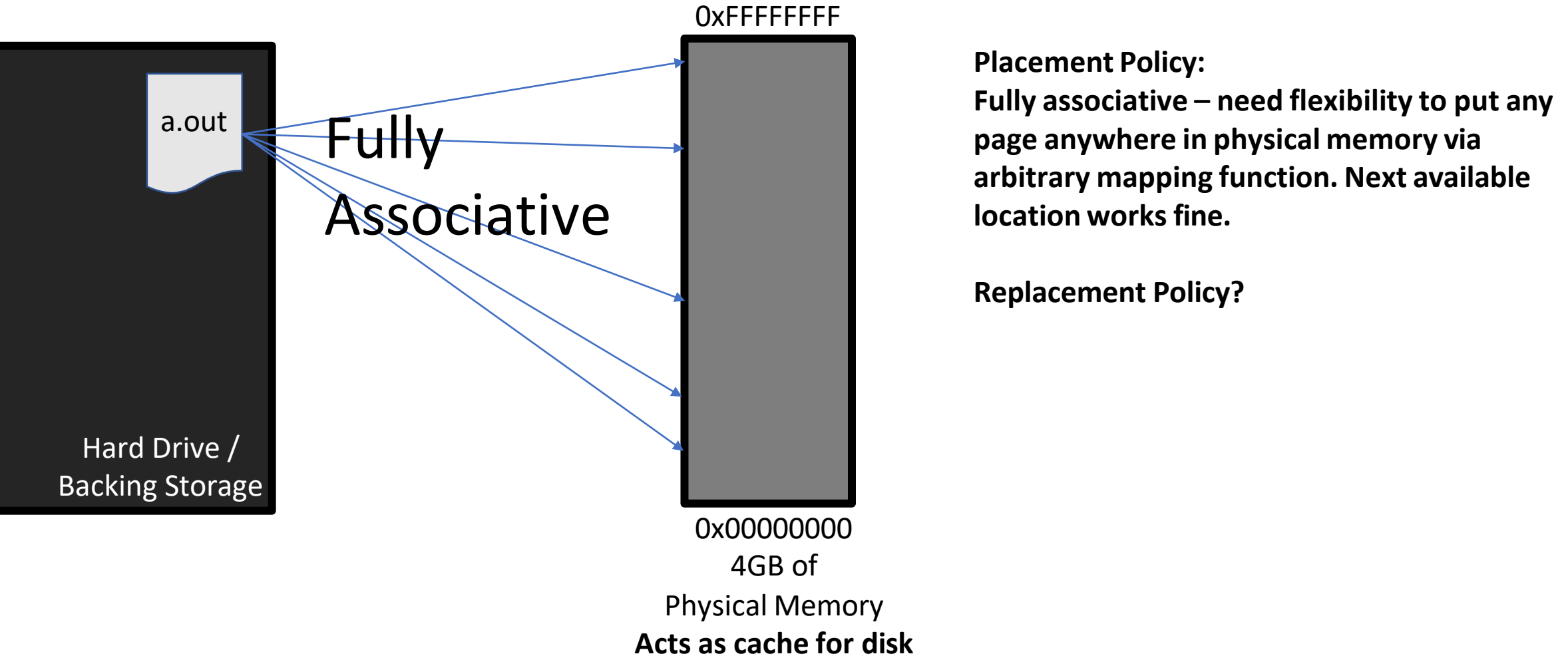


Page Cache Placement / Replacement

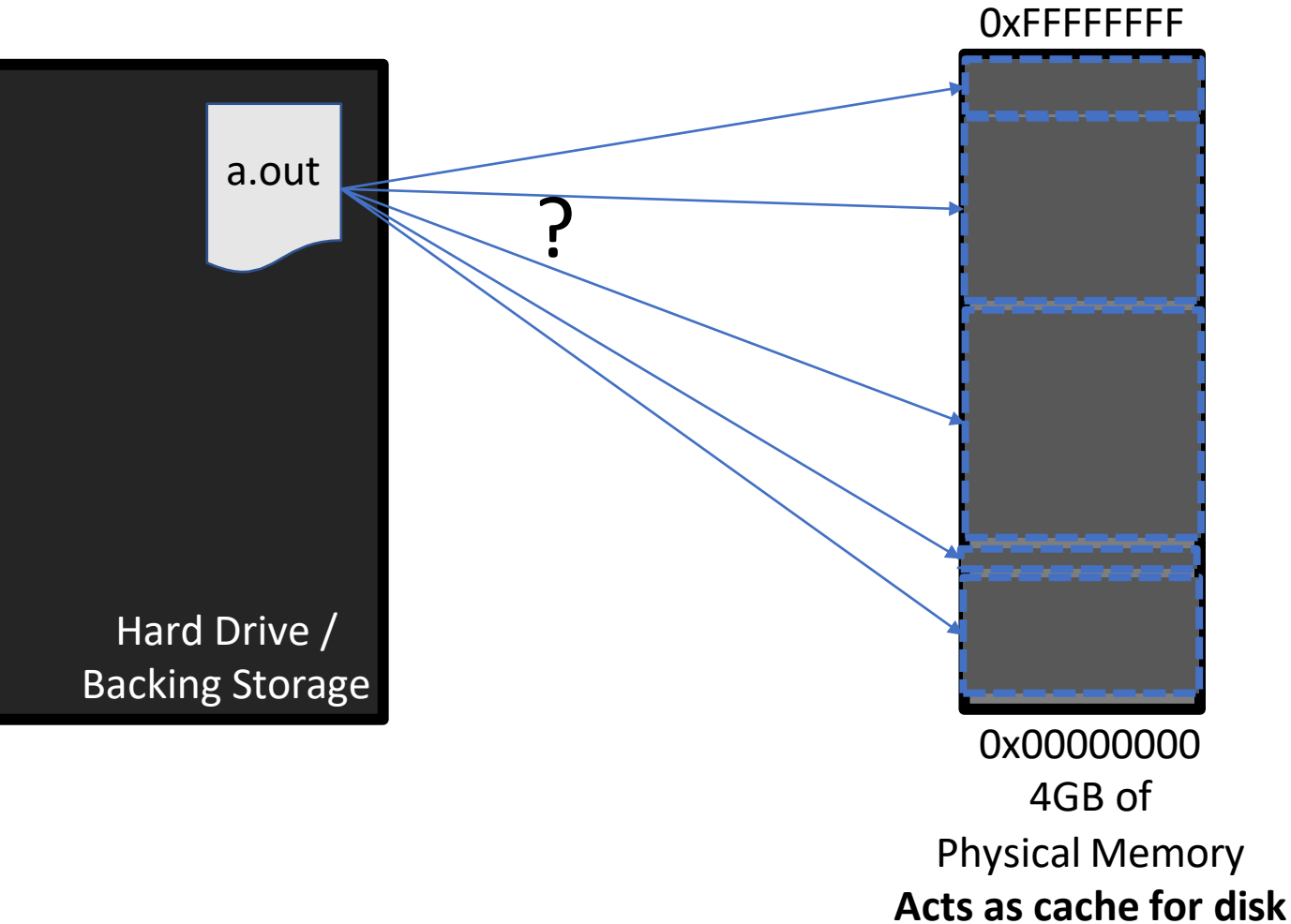
Physical Memory as a Cache of Data on Disk: Cache Placement / Replacement Policy



Physical Memory as a Cache of Data on Disk: Cache Placement / Replacement Policy



Physical Memory as a Cache of Data on Disk: Cache Placement / Replacement Policy

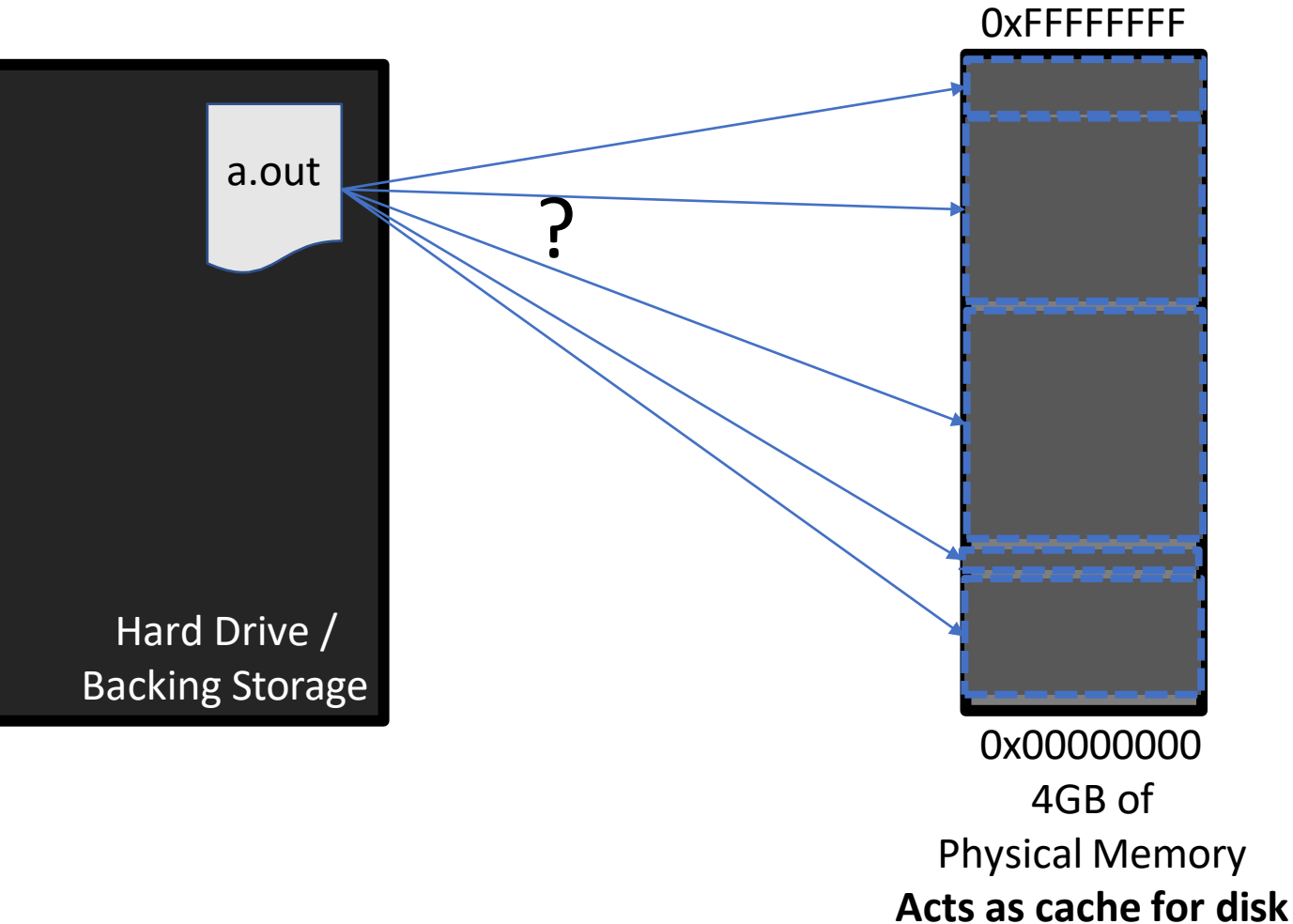


Placement Policy:

Fully associative – need flexibility to put any page anywhere in physical memory via arbitrary mapping function. Next available location works fine.

Replacement Policy?

Physical Memory as a Cache of Data on Disk: Cache Placement / Replacement Policy



Placement Policy:

Fully associative – need flexibility to put any page anywhere in physical memory via arbitrary mapping function. Next available location works fine.

Replacement Policy:

Can be complicated...

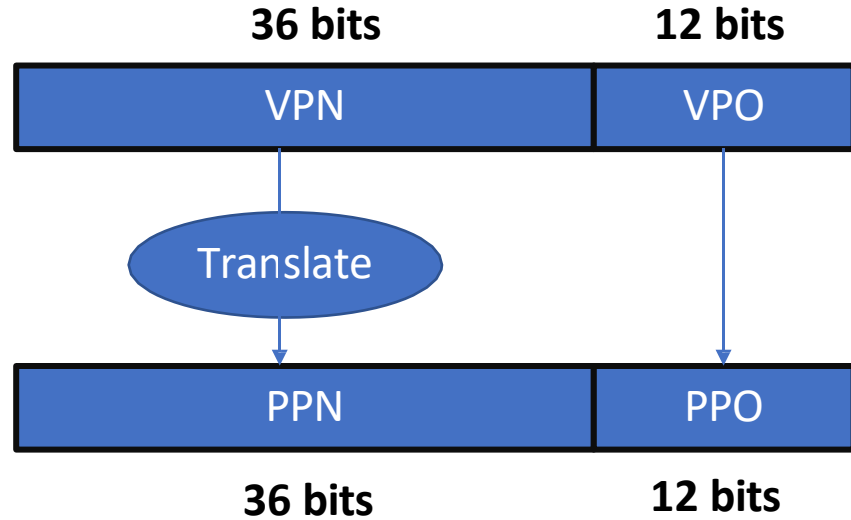
- Variants of LRU & approximations
- Not Recently Used (like Bit-PLRU)
- Not Frequently Used
- Re-reference-Distance-Based Policies

In Lab 3 we handle placement & replacement so you can focus on translation & mapping

Page Tables: Another Look at the Translation Function

Page Translation and Its Implementation

48-bit Virtual Address (like AMD)



Page Table Entry – 6 Bytes



Q: Why can store flags in lower 12 bits of PTE?

Page Table

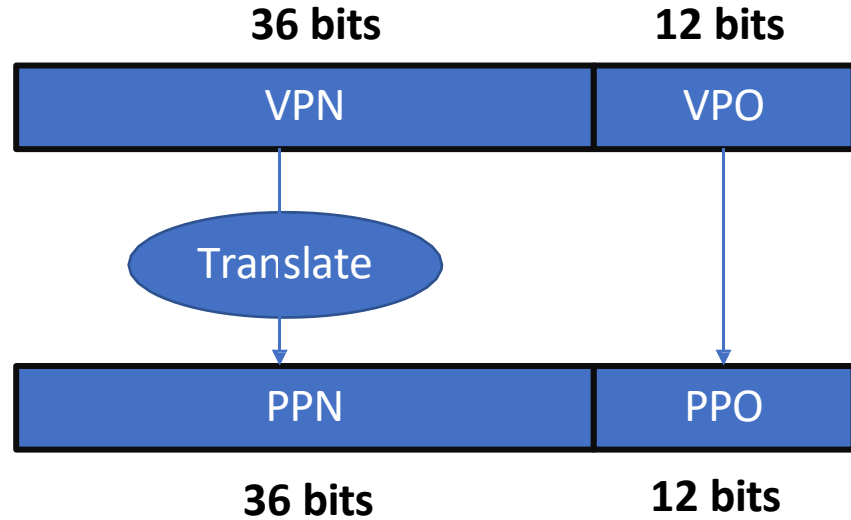
VPN	PPN	Perms/Flags
0x2000	0x123000	RW
0x3000	Paged Out: /swap/a.out	RWX
⋮		
0xF000	0xFFF0F000	RW
⋮		
0x 126F000	0x11212000	R
⋮		
0xFFFFFFFF	0x45454000	R / COW

Table stores 2^{36} entries in it for virtual pages 0x000000000000 up to 0xFFFFFFFFF000 which span the entire 48-bit address space.

Implementation Issues?

Page Translation and Its Implementation

48-bit Virtual Address (like AMD)



Page Table Entry – 6 Bytes



Page Table

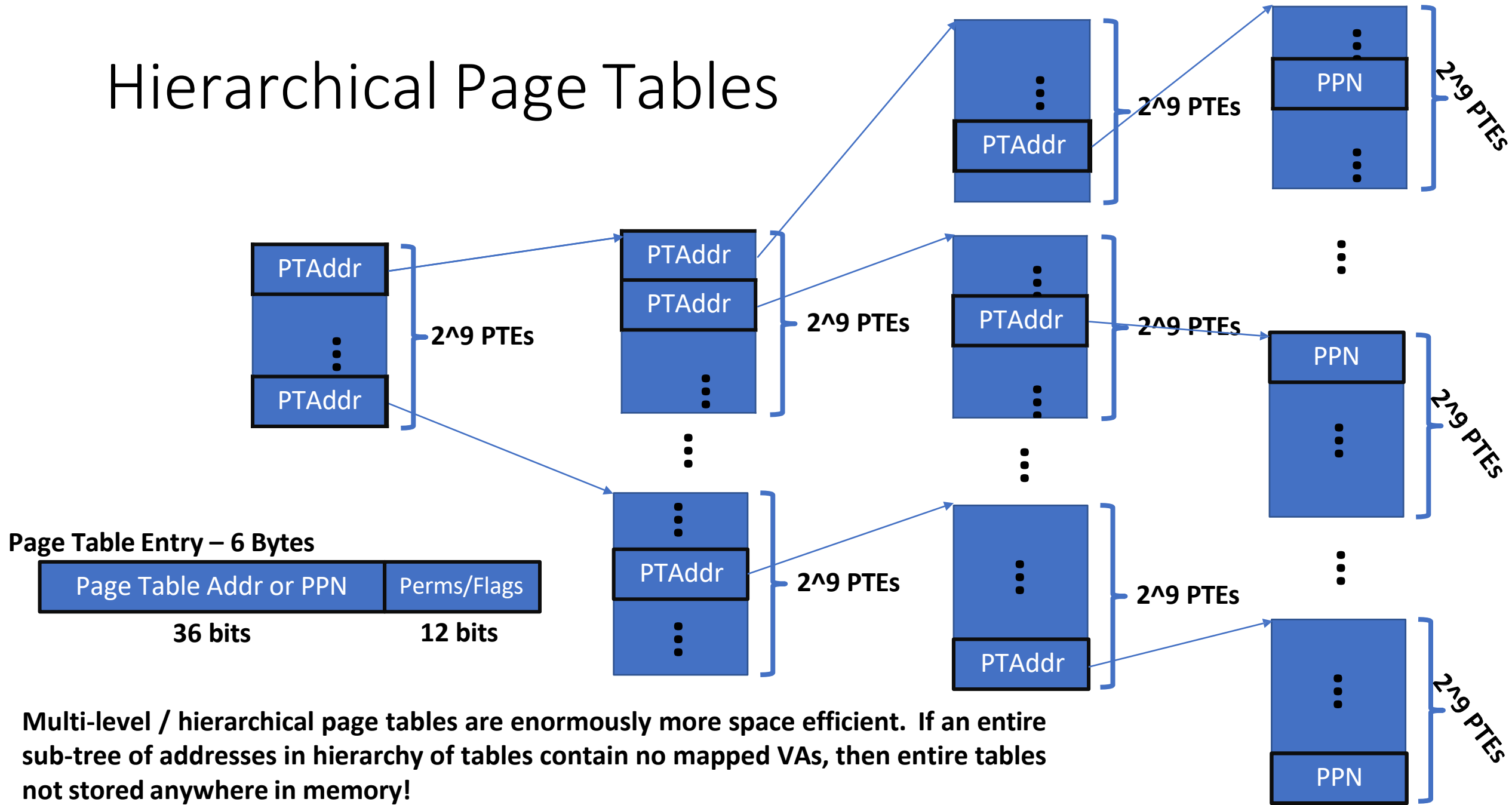
VPN	PPN	Perms/Flags
0x2000	0x123000	RW
0x3000	Paged Out: /swap/a.out	RWX
⋮		
0xF000	0xFFF0F000	RW
⋮		
0x 126F000	0x11212000	R
⋮		
0xFFFFFFFF	0x45454000	R / COW

Table stores 2^{36} entries in it for virtual pages 0x000000000000 up to 0xFFFFFFFFF000 which span the entire 48-bit address space.

Dense, linear table stores $2^{36} * 6\text{B}$ PTEs: 550.8GB of Page Tables

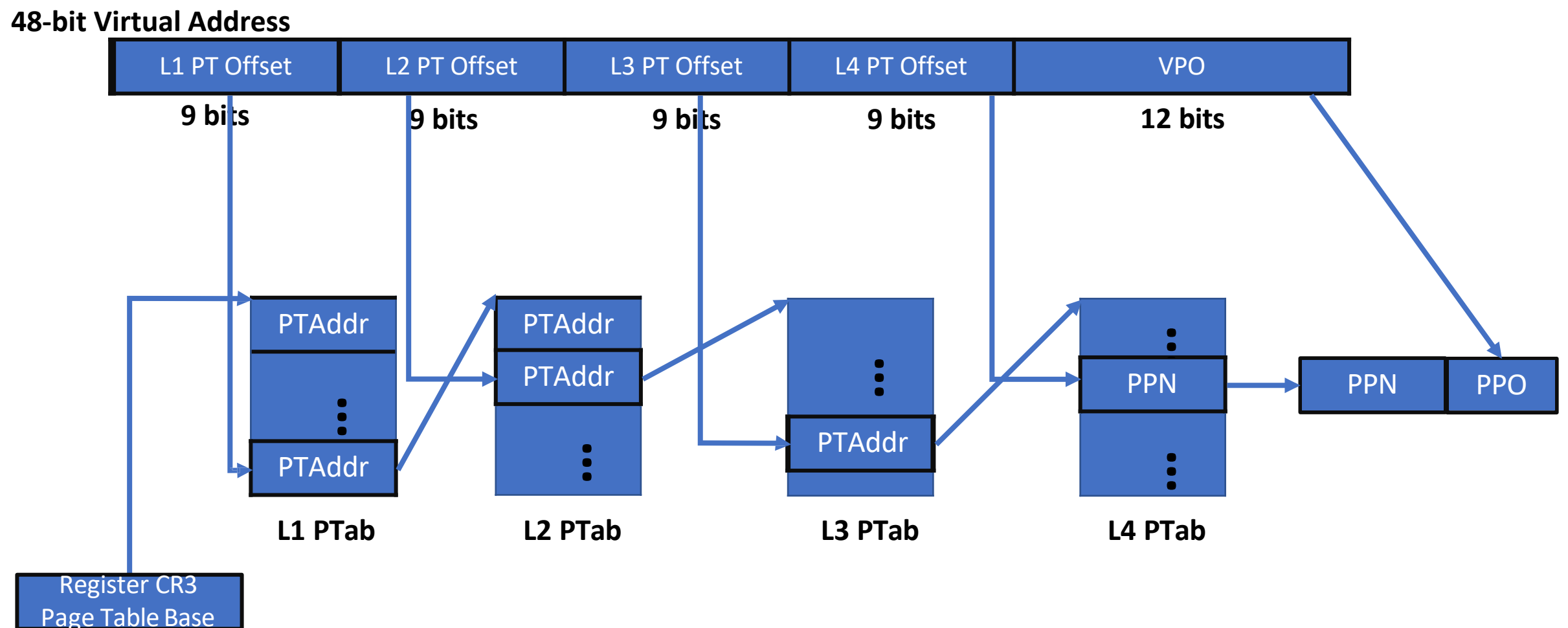
Most PTEs Empty!!!

Hierarchical Page Tables

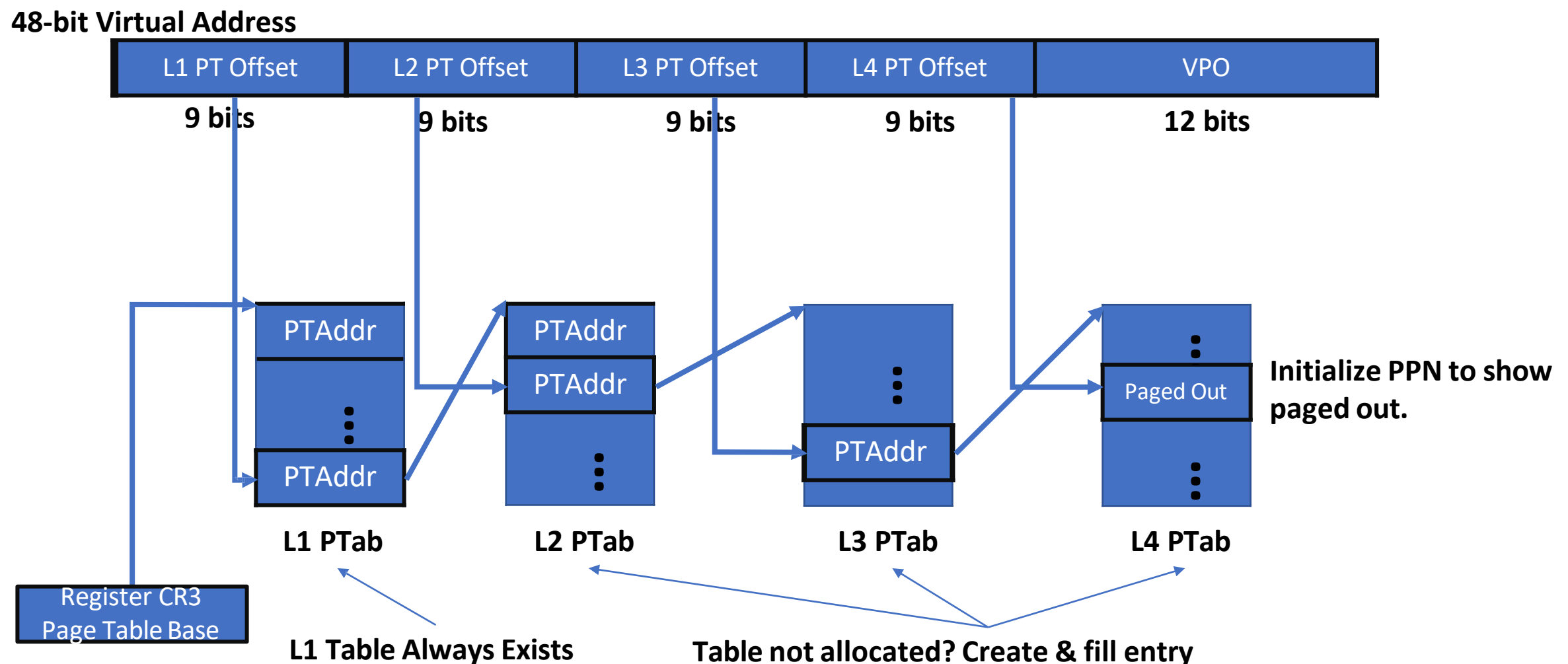


Multi-level / hierarchical page tables are enormously more space efficient. If an entire sub-tree of addresses in hierarchy of tables contain no mapped VAs, then entire tables not stored anywhere in memory!

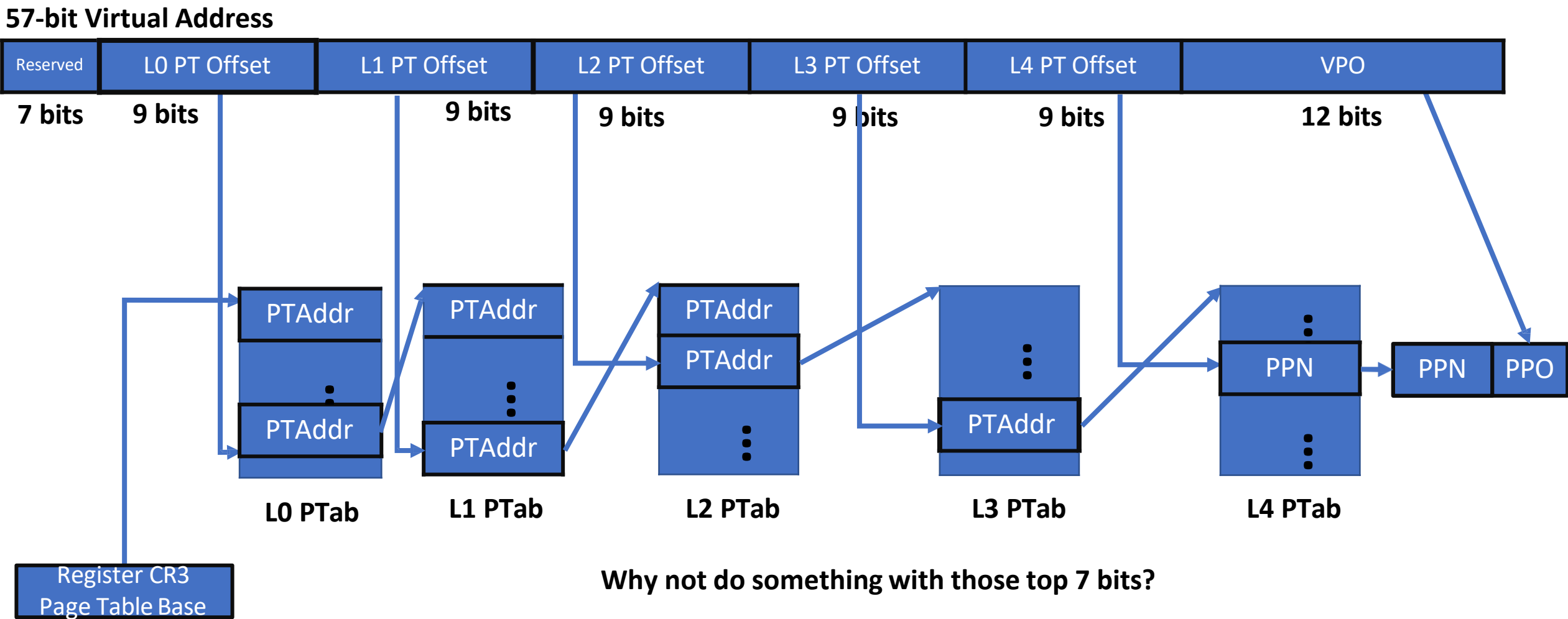
Translation Using Hierarchical Page Tables



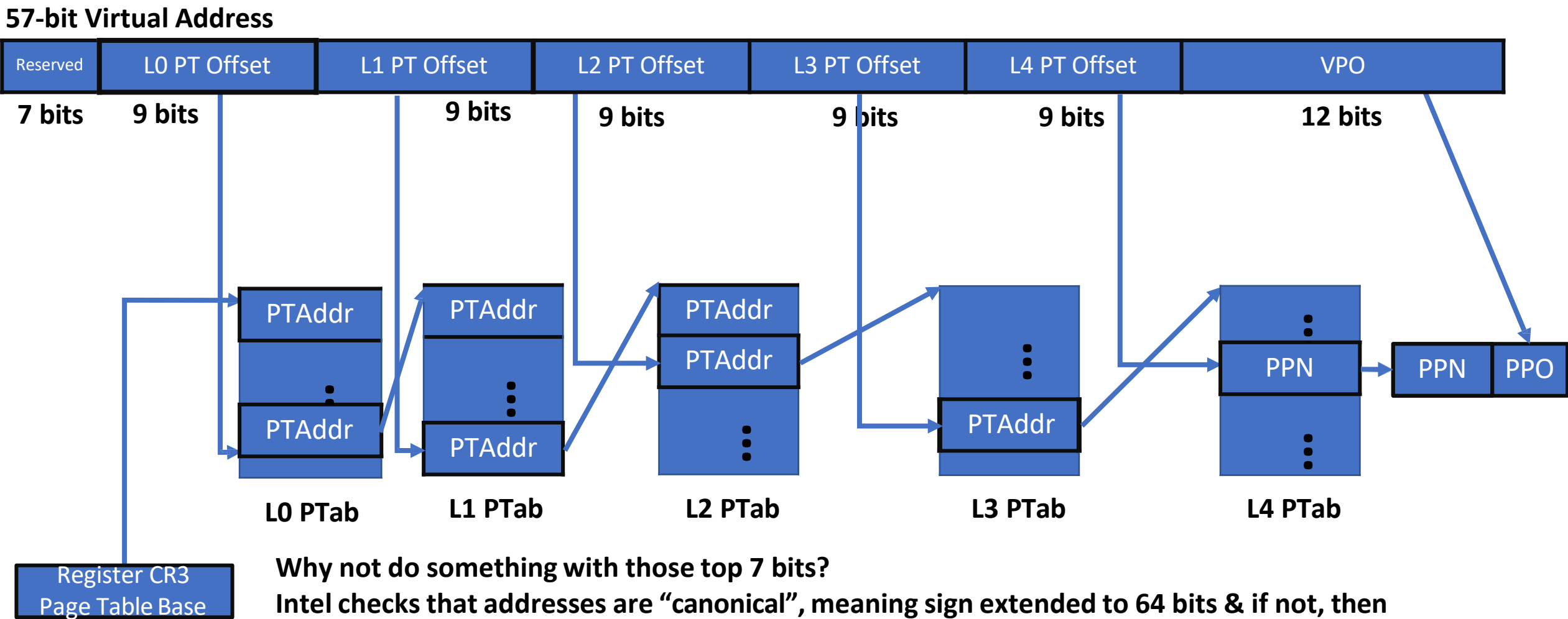
Mapping Using Hierarchical Page Tables



(New) Intel 57-bit Virtual, 52-bit Physical, 5-level Translation Using Hierarchical Page Tables (2019)



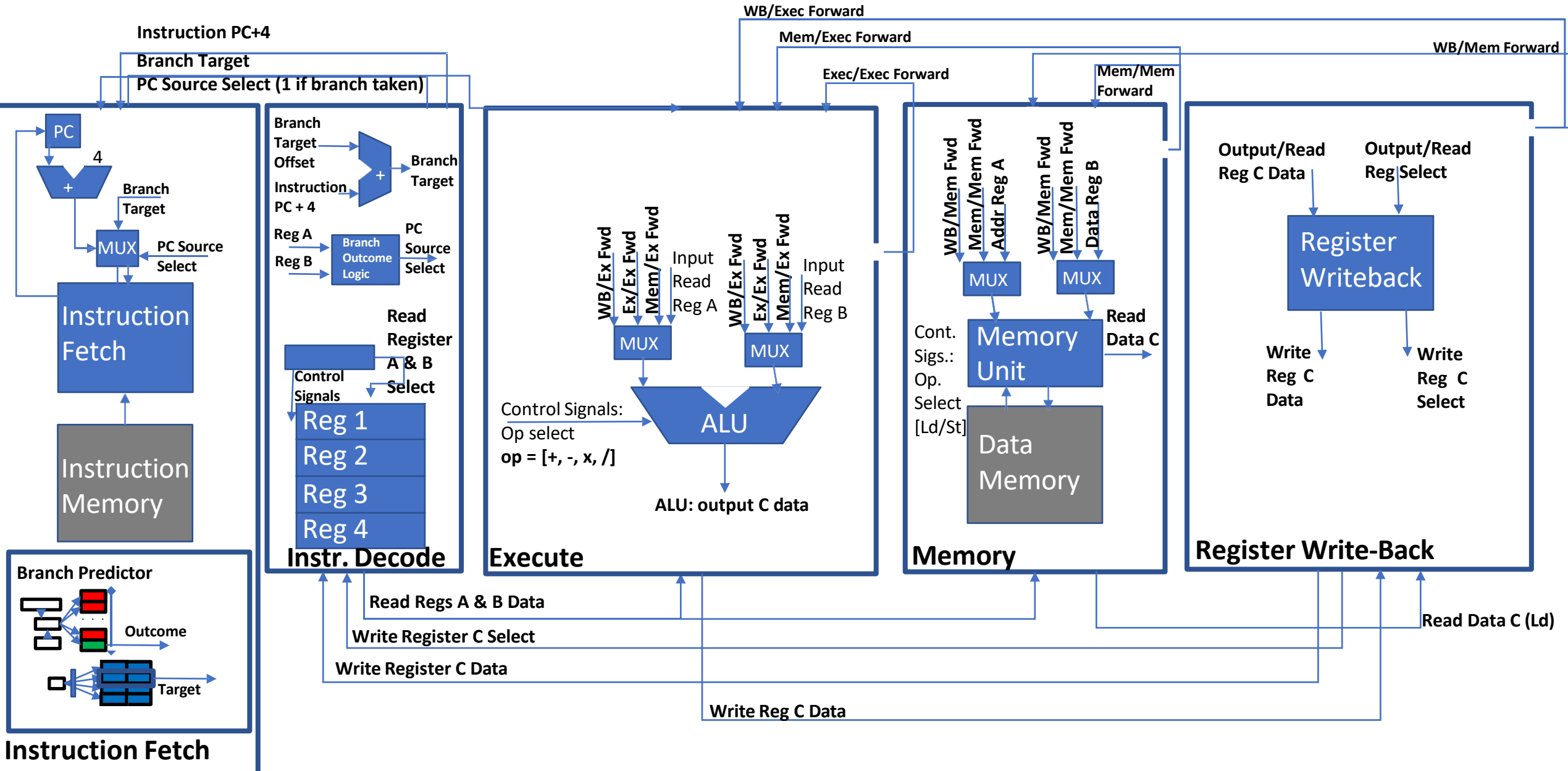
(New) Intel 57-bit Virtual, 52-bit Physical, 5-level Translation Using Hierarchical Page Tables (2019)



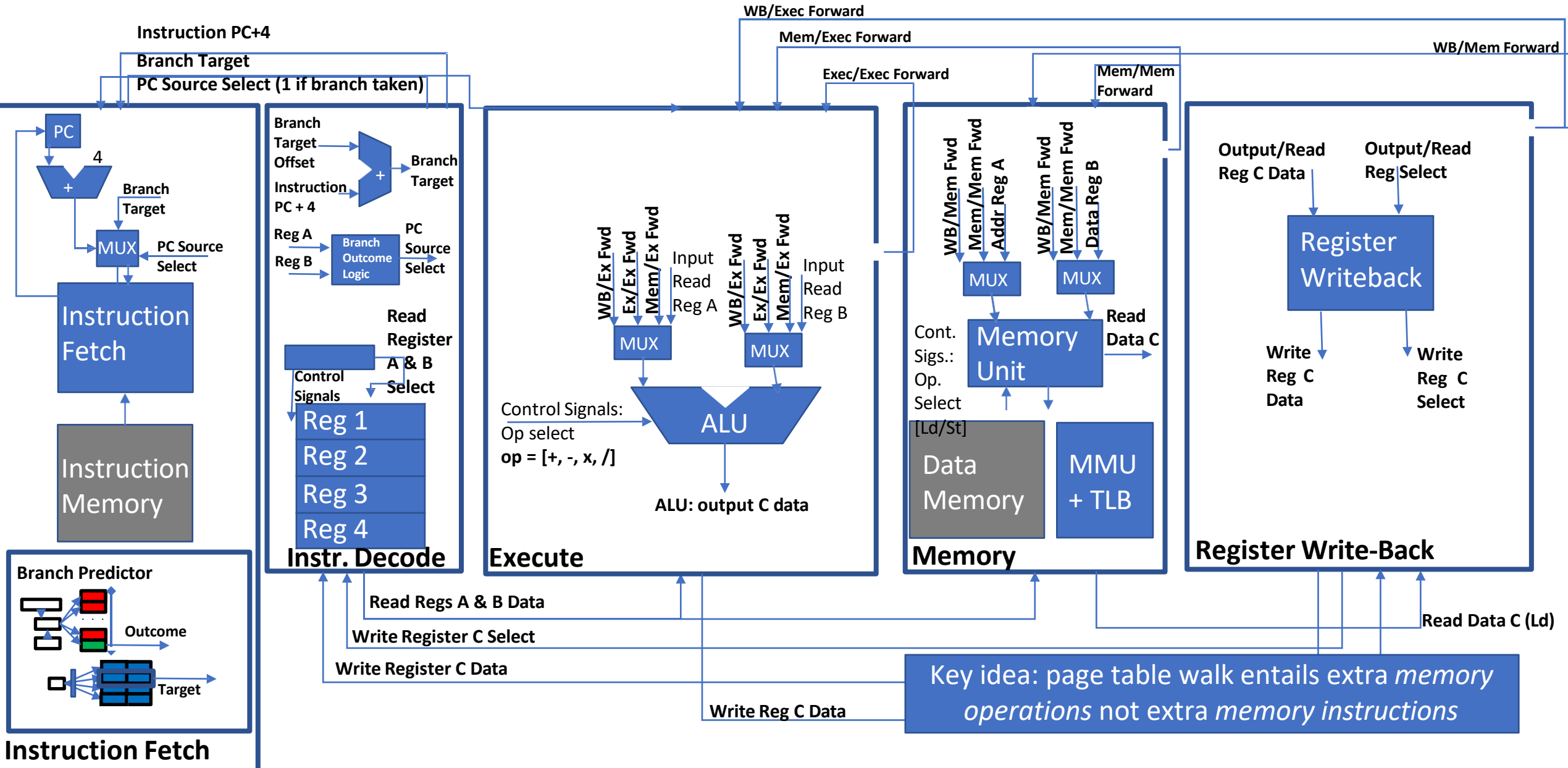
Why not do something with those top 7 bits?

Intel checks that addresses are “canonical”, meaning sign extended to 64 bits & if not, then SEGFAULT. Allows future architectures to use 64b addrs if they want to!

What part of the pipeline manipulates the page tables?

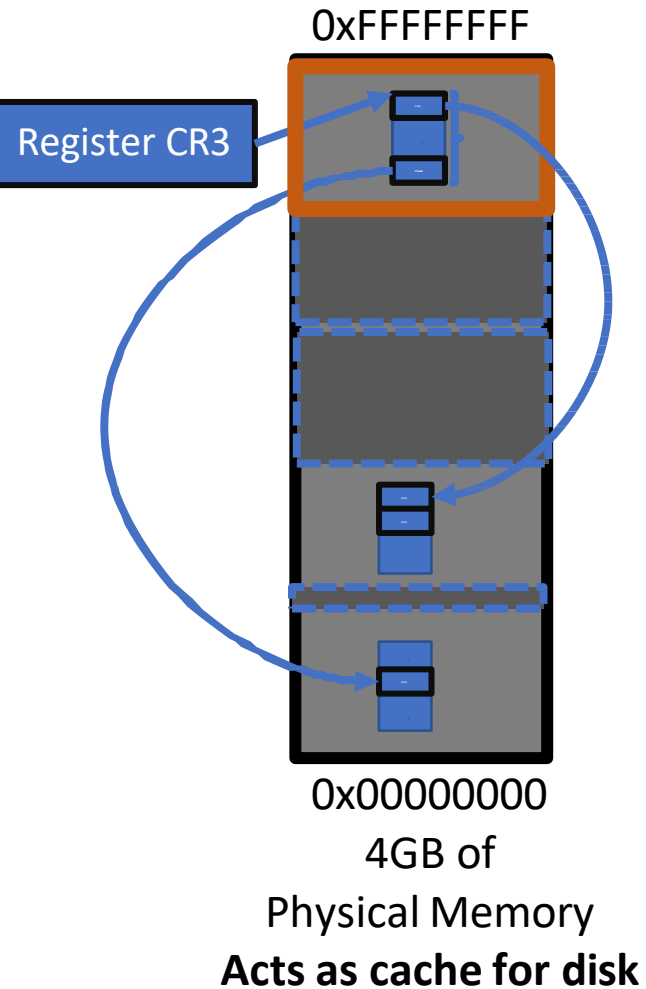


MMU has fast access to memory and TLB for translation



Performance and Storage Overhead Analysis of Translation with Page Tables

Page Tables Stored in Virtual Memory and Paged in and out Like Other Data

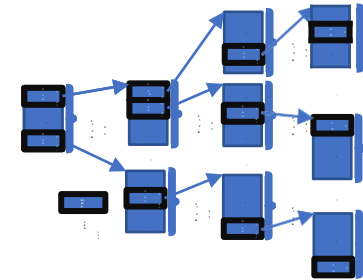


First level page table *always in physical memory at address in Register CR3.*
Other levels of page table can be paged out to make space for other data.

All paged & page table data moves through cache hierarchy like any other data

Question: How much space overhead to store hierarchical vs. linear page tables?

Question: How much time overhead to access hierarchical vs. linear page tables?

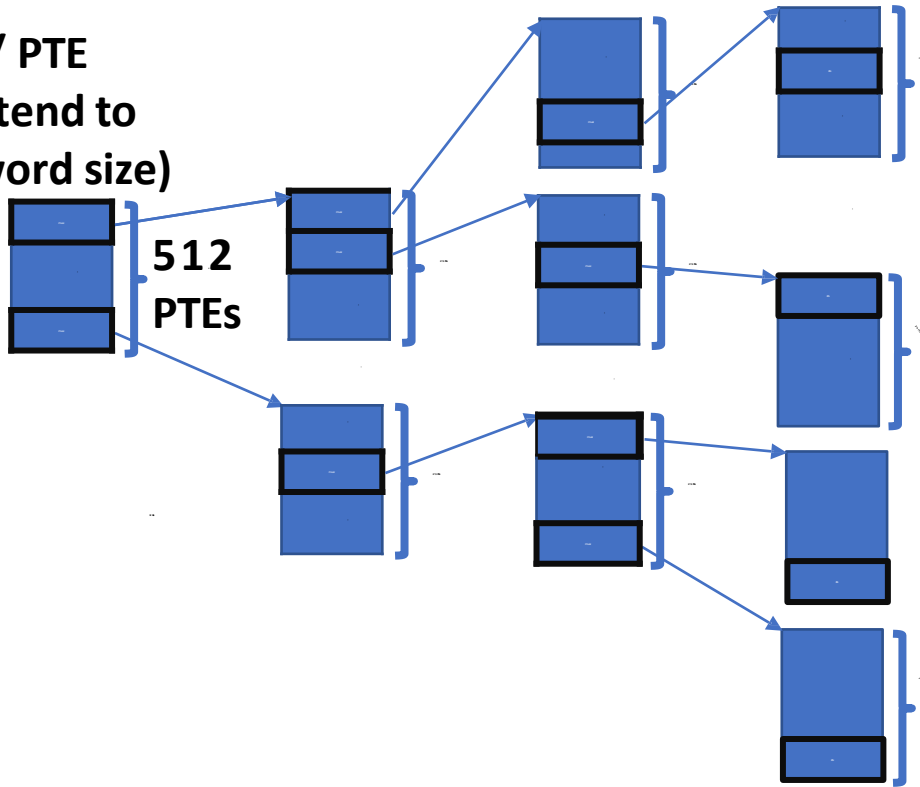


Space Overhead Analysis of Page Tables

4 Levels of Page Tables

8bytes / PTE
(sign extend to
64-bit word size)

512
PTEs



How much space for tables vs. mapped data? Compared to linear?

Space Overhead Analysis of Page Tables

4 Levels of Page Tables

8bytes / PTE
(sign extend to
64-bit word size)

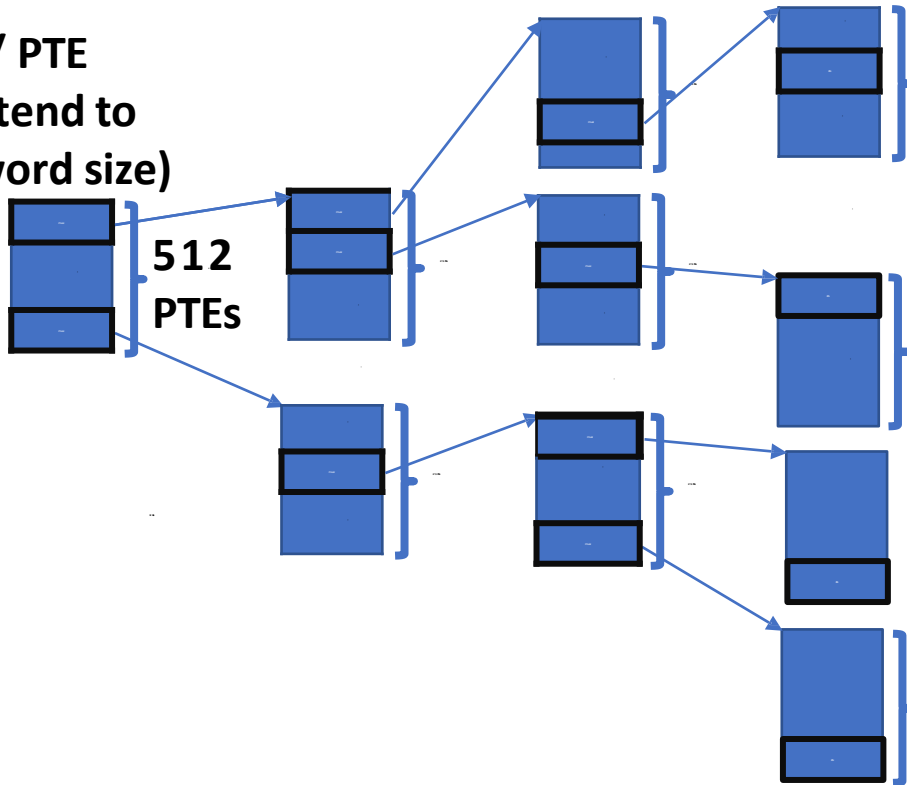


Table Size = Page Size

$$2^3 \text{ bytes / PTE} * 2^9 \text{ PTEs / Table} = 2^{12} = 4\text{kB / Table}$$

Pictured Example page tables size:

$$9 * 4\text{kB} = 36\text{kB of page tables}$$

Possible to map every page in last level PT:

$$4 \text{ last level tables exist} * 512 \text{ entries} * 4\text{kB / page} = 2^{23}\text{B mappable with just these page tables}$$

Overhead:

$$36\text{kB} / 2^{23}\text{B} = 36\text{kB} / 2^{13}\text{kB} = 0.004\text{x overhead}$$

Linear page tables:

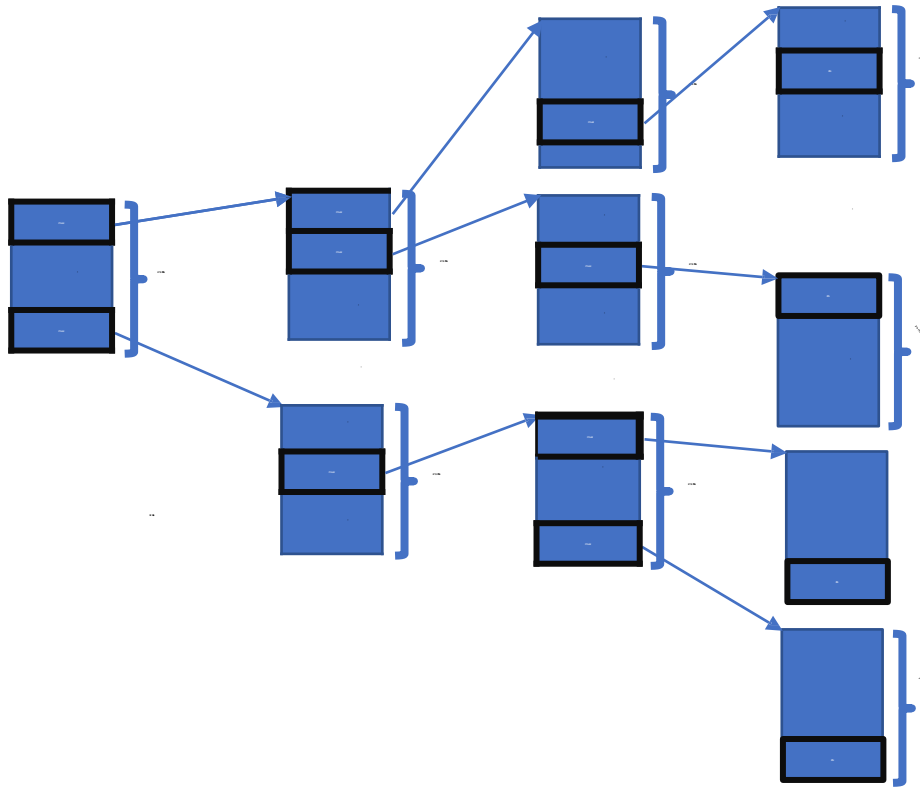
550GB of page tables

With 2^{23}B of data to map, **65565x overhead**

Performance Analysis of Page Tables

What is the time cost per memory access to use a hierarchical page table structure?

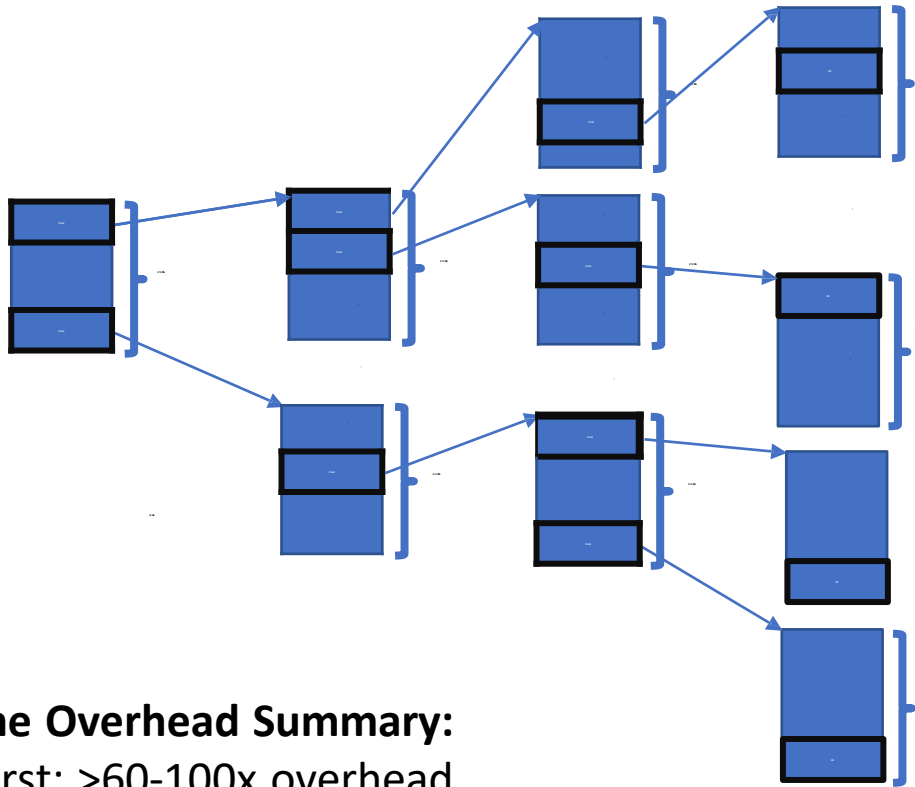
`sw 0x2000`



Performance Analysis of Page Tables

What is the time cost per memory access to use a hierarchical page table structure?

`sw 0x2000`



Time Overhead Summary:

Worst: >60-100x overhead

Best: ~2x overhead

Plus four more memory accesses:

- one memory access per page table level
- three of which levels may be swapped out / page fault
- all of which can be a cache miss

Worst case time overhead:

Word at 0x2000 is an L1 cache hit, all page table data except first level table misses in cache & page faults

1 cycle L1 hit becomes 3 cache misses & 3 page faults (DRAM ~20 cycles) = 60-100 (or more) cycles overhead on 1 cycle L1 hit

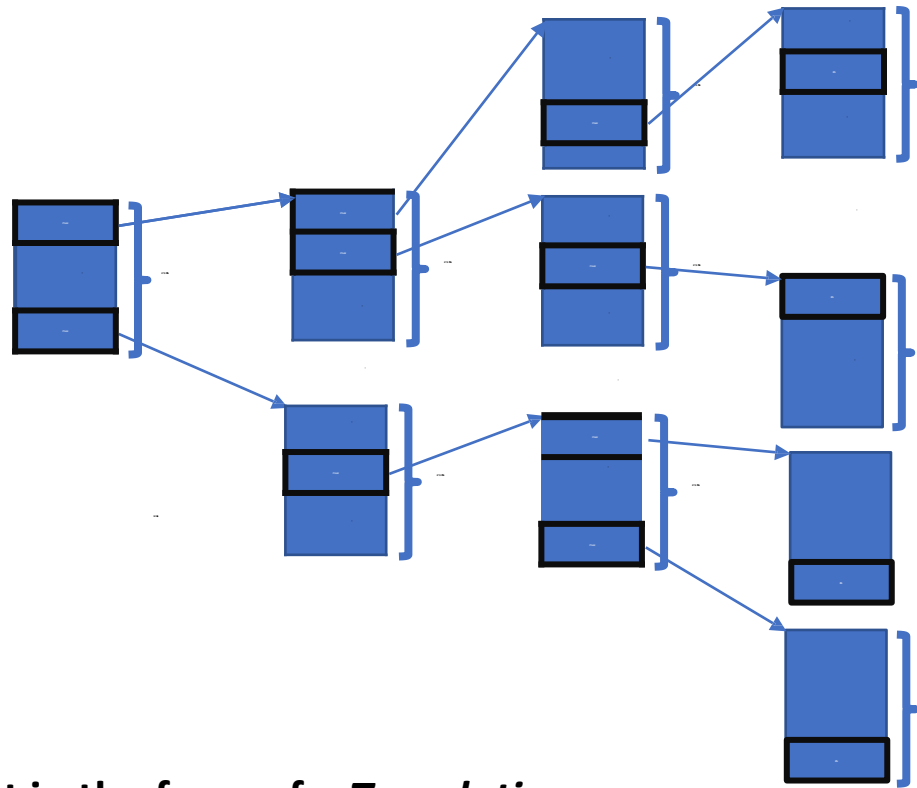
Minimum(-ish) Overhead:

Access to word at 0x2000 (L1 miss, say), all page table accesses hit in cache & no page faults

3 cycle L1 miss becomes 3 cache hits and a cache miss (1 cycle cache hits) = 3 cycles overhead on 3 cycle L1 miss

Hierarchical Page Tables Trade Time to Save Space

Time overhead:
From 2x constant time overhead
to a variable overhead that can
be upwards of 100x!

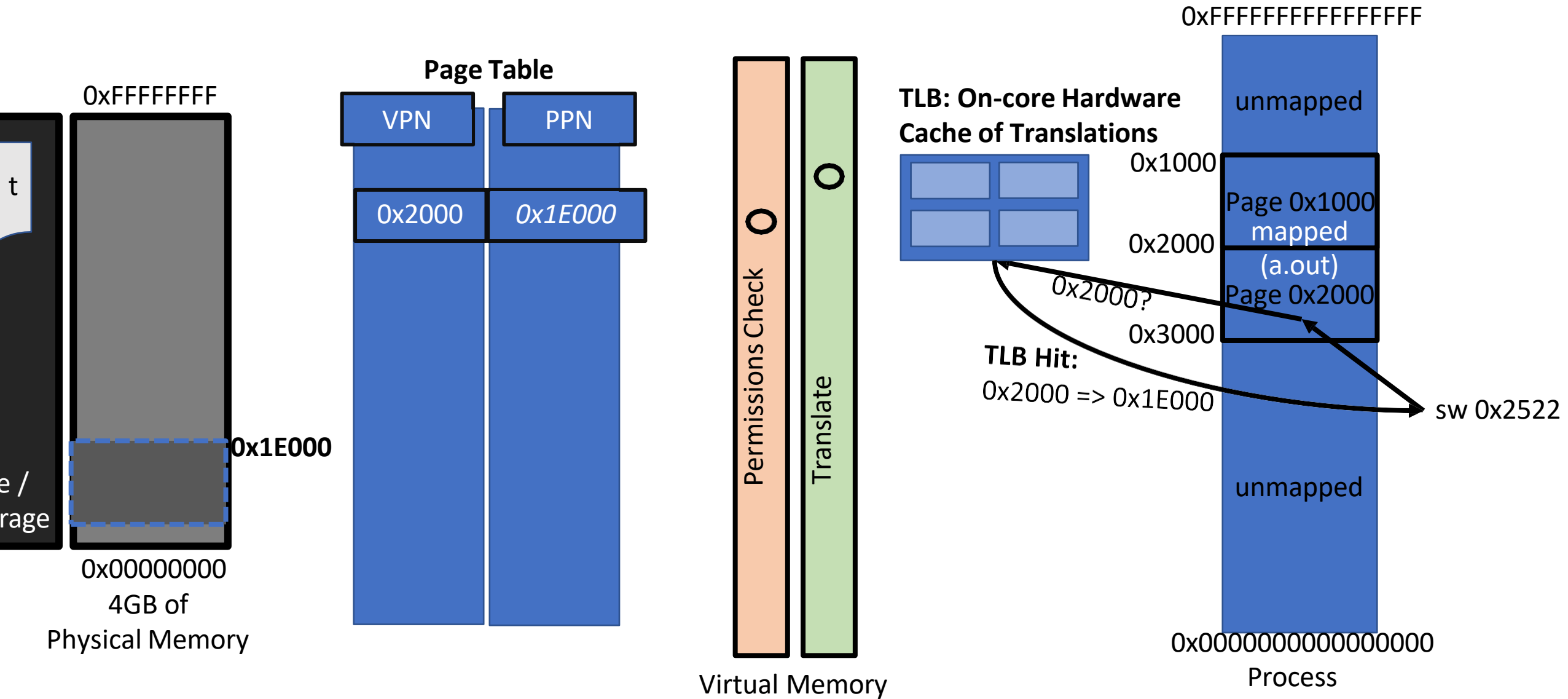


Space Savings:
From 65565x space overhead
to a 0.001x space overhead

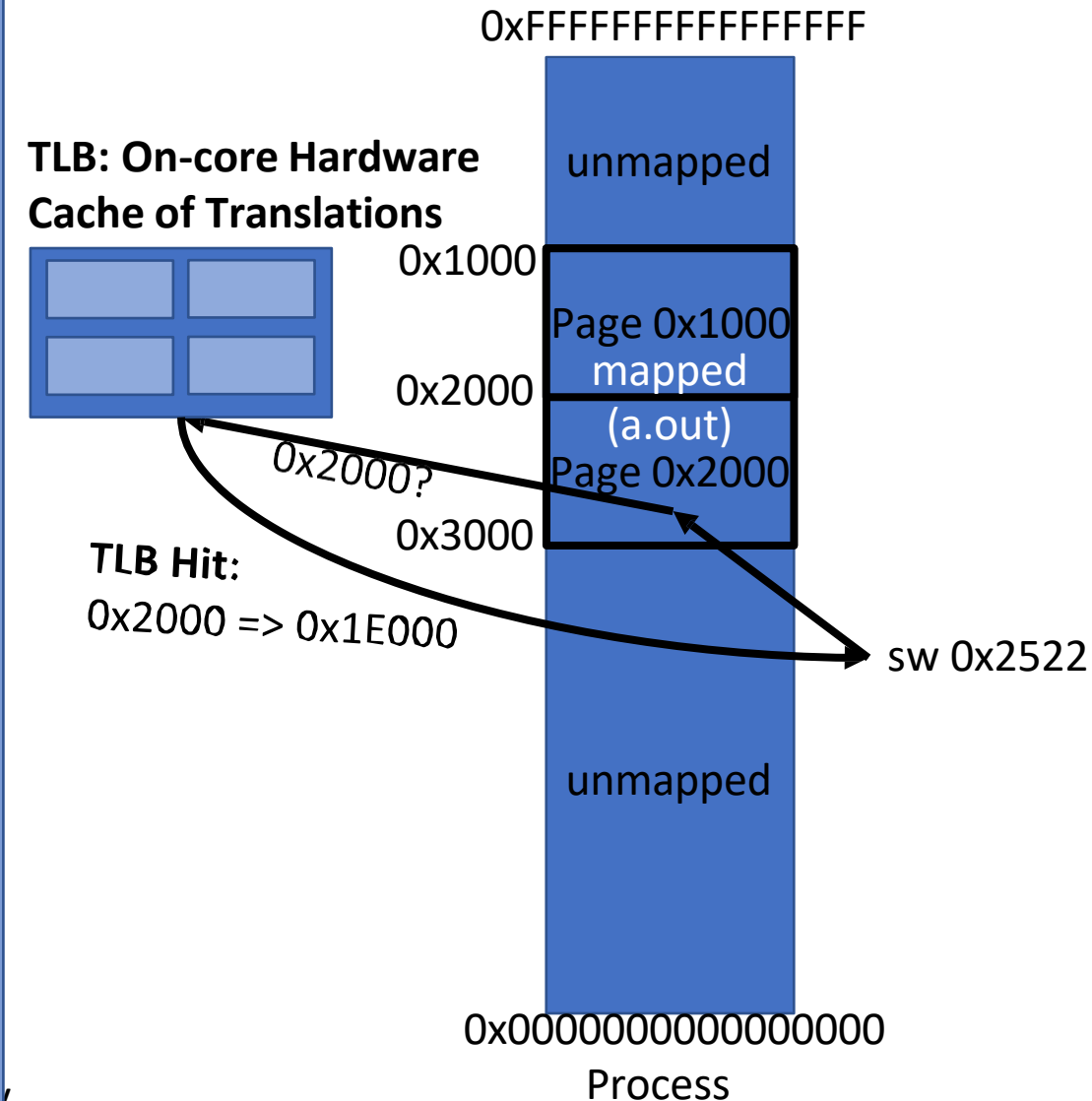
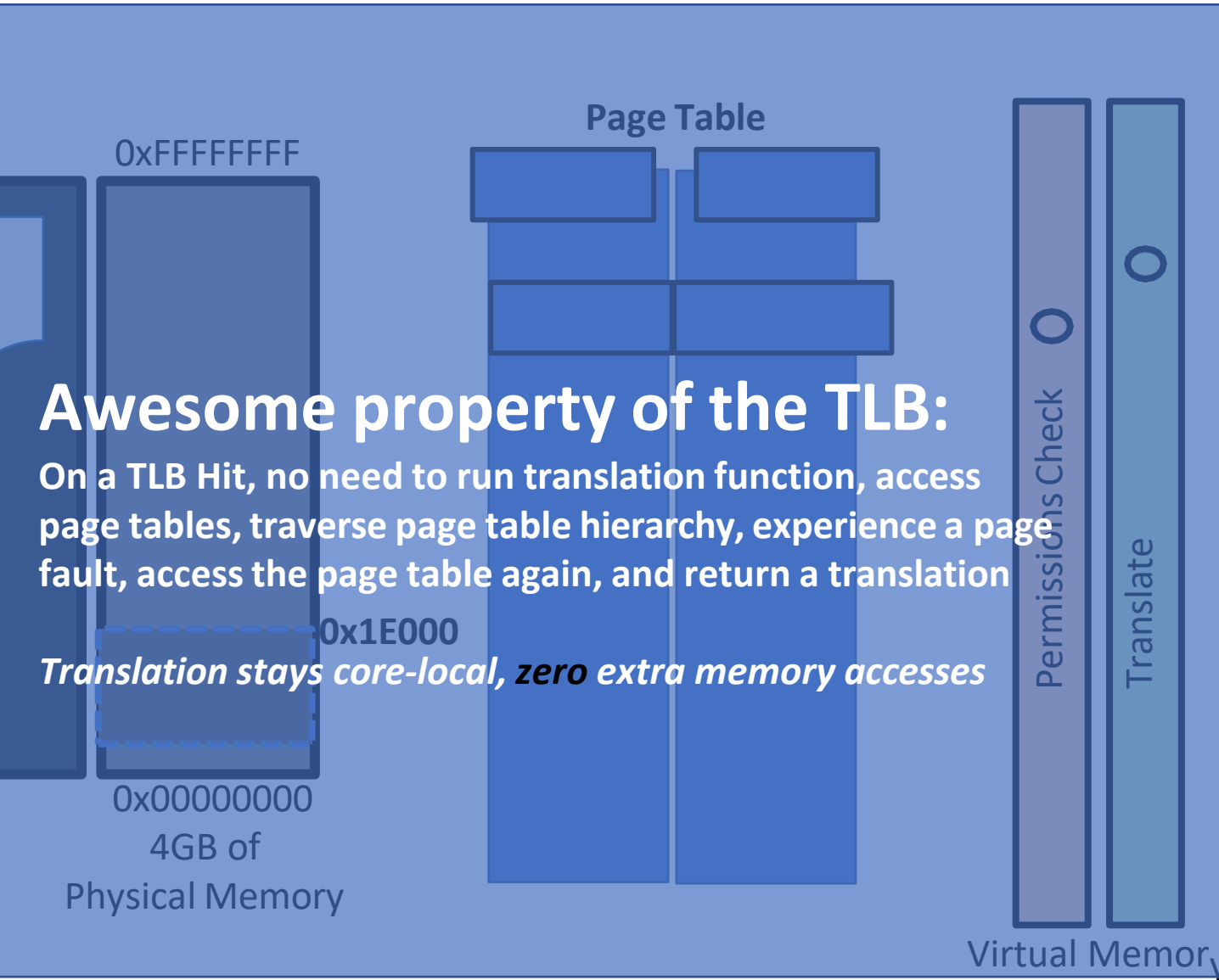
Key Insight:
Use microarchitectural support in the form of a *Translation Lookaside Buffer* to eliminate the time cost of *most* translations

Translation Lookaside Buffers: Hardware Support for Caching Page Address Translations

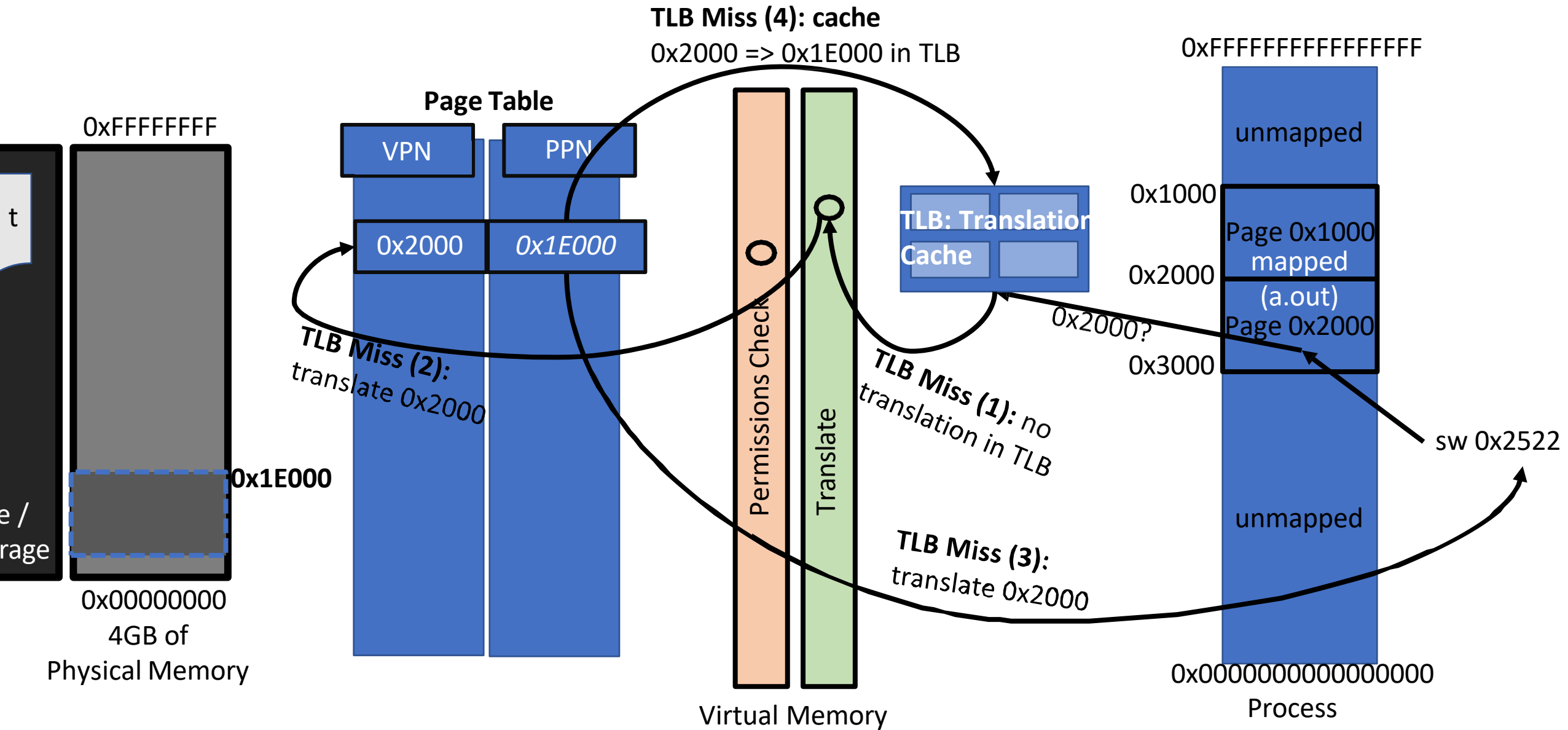
Translation Lookaside Buffer: Basic Idea (Hit)



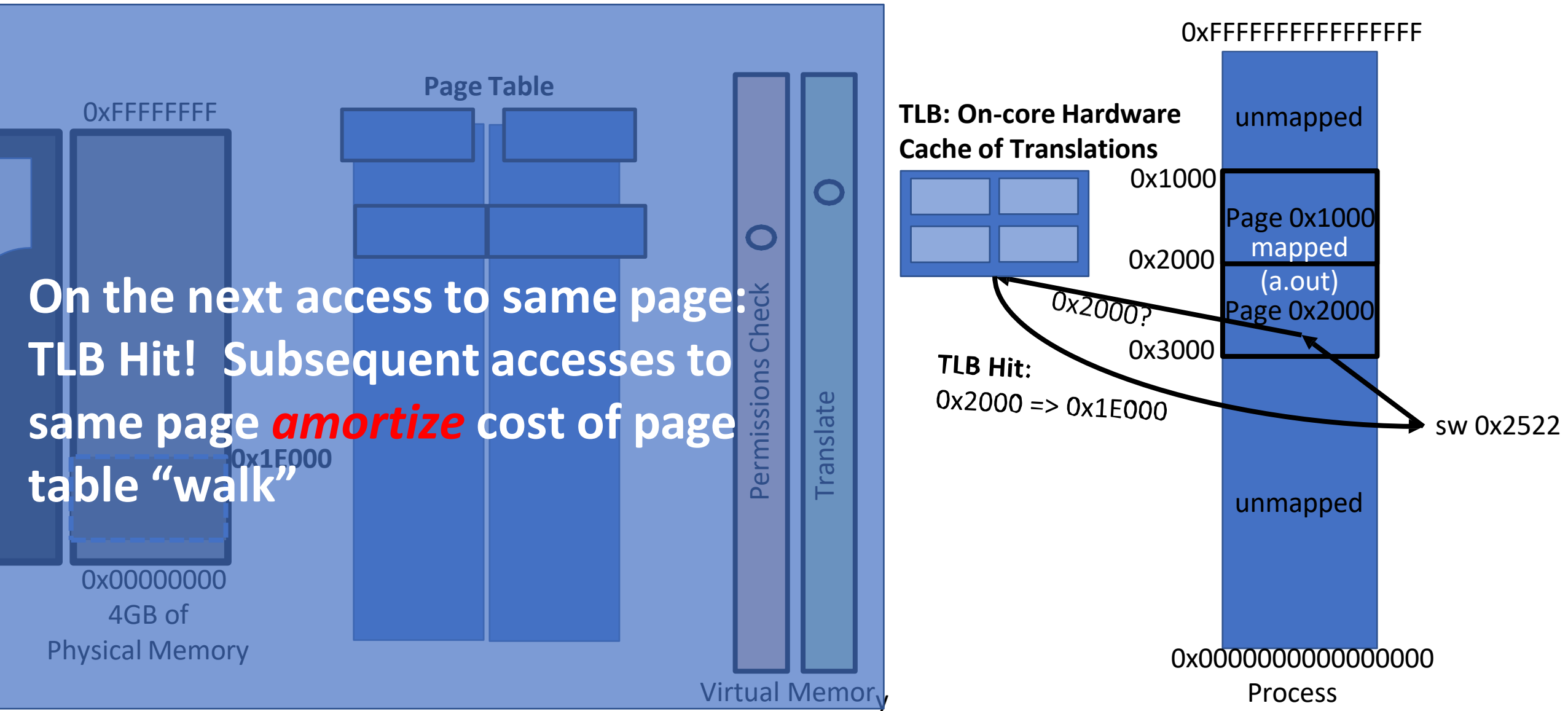
Translation Lookaside Buffer: Basic Idea (Hit)



Translation Lookaside Buffer: Basic Idea

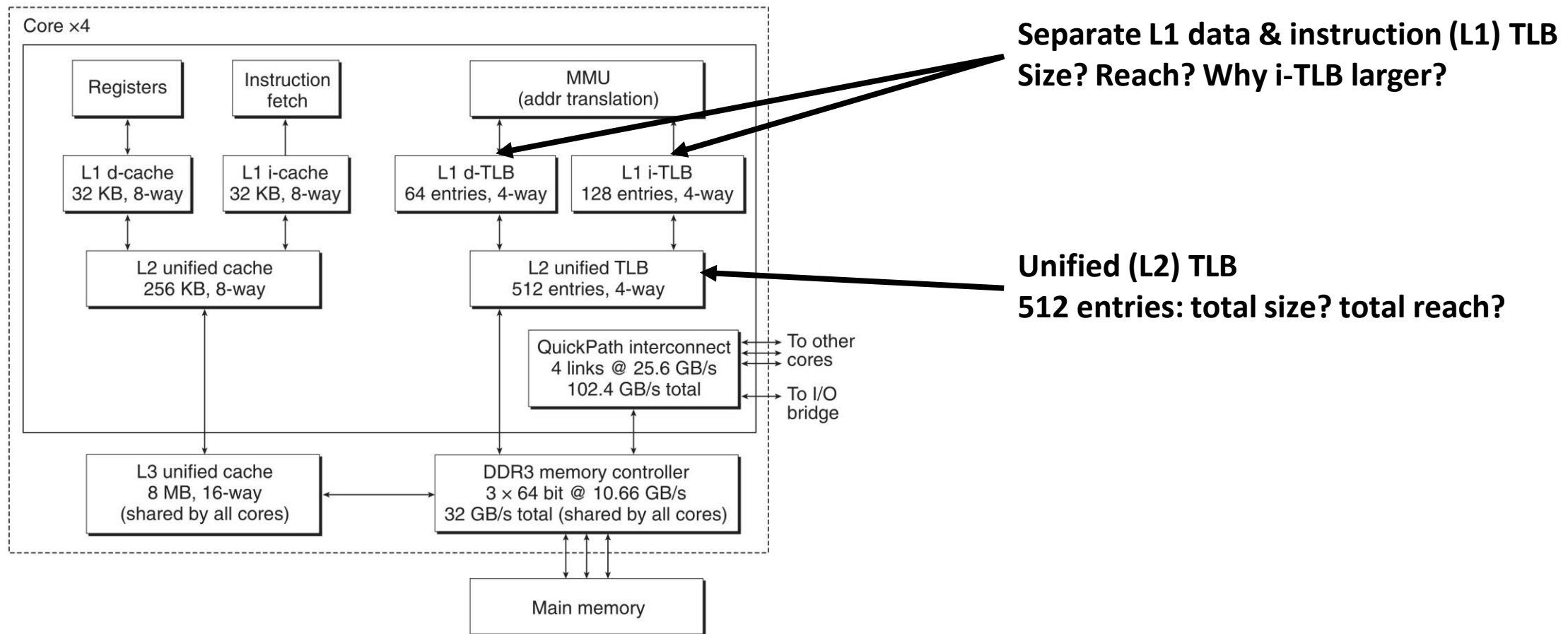


Translation Lookaside Buffer: Basic Idea (Hit)



Hardware Support for Virtual Memory: Translation Lookaside Buffers in Intel Core i7

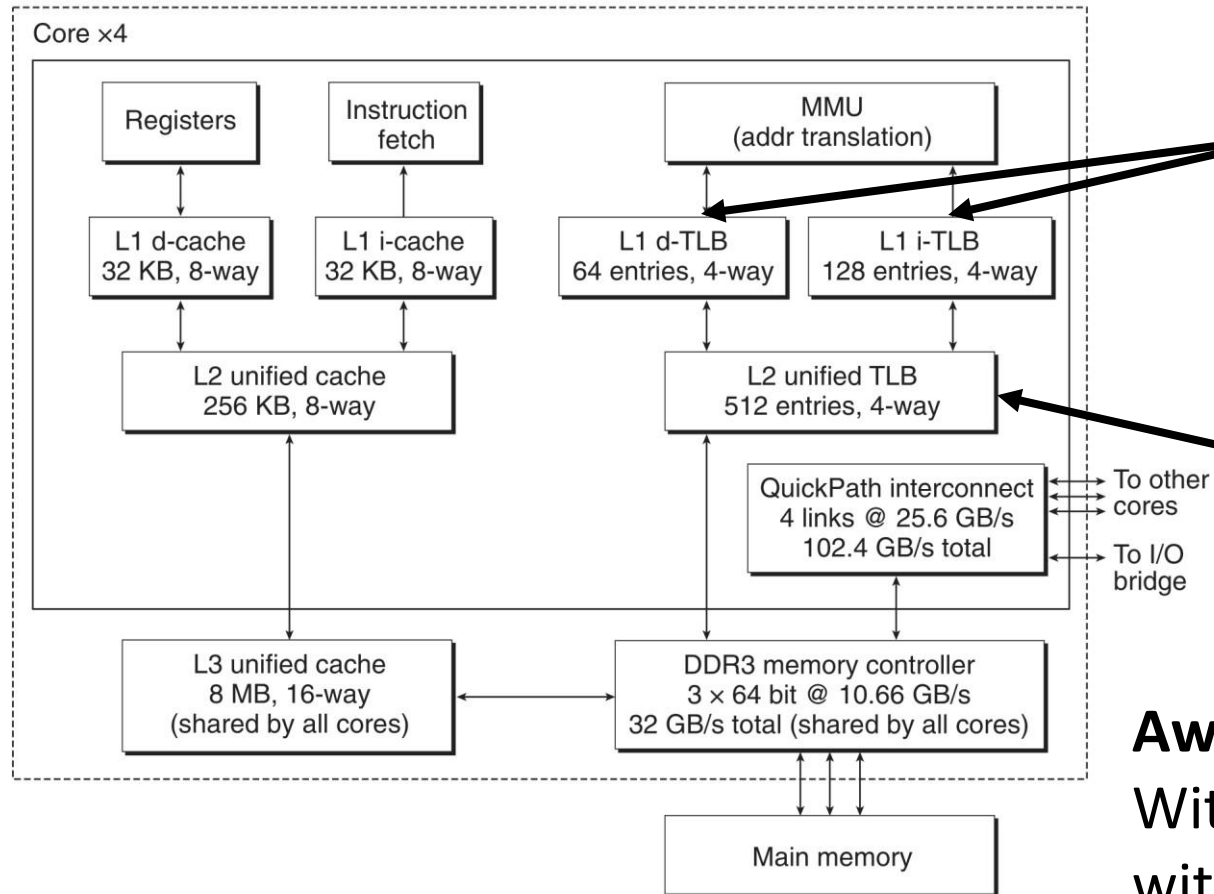
Processor package



The Core i7 memory system.

Hardware Support for Virtual Memory: Translation Lookaside Buffers in Intel Core i7

Processor package



Separate L1 data & instruction (L1) TLB
Size: $64 * 8B = 512B$; $128 * 8B = 1024B$
Data Reach: $64 * 4kB = 256kB$ data
Code Reach: $128 * 4kB = 512kB$ code

Unified (L2) TLB
Size: $512 * 8B = 4kB$
Reach: $512 * 4kB = 2^{21} = 2MB$ code+data

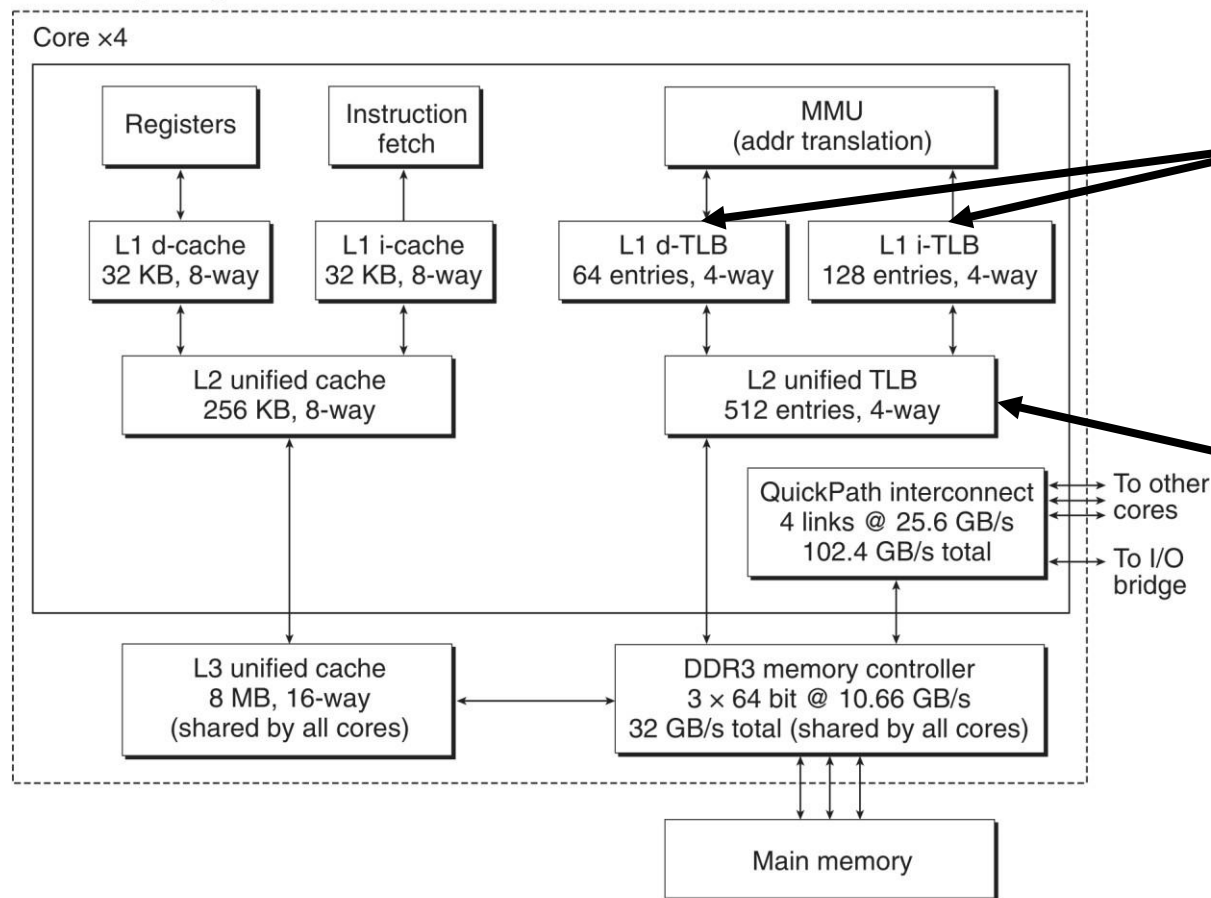
Awesome Property of the TLB:

With a 3MB-ish working set of data + code, can run without *ever* running virtual memory translation

The Core i7 memory system.

Hardware Support for Virtual Memory: Translation Lookaside Buffers in Intel Core i7

Processor package



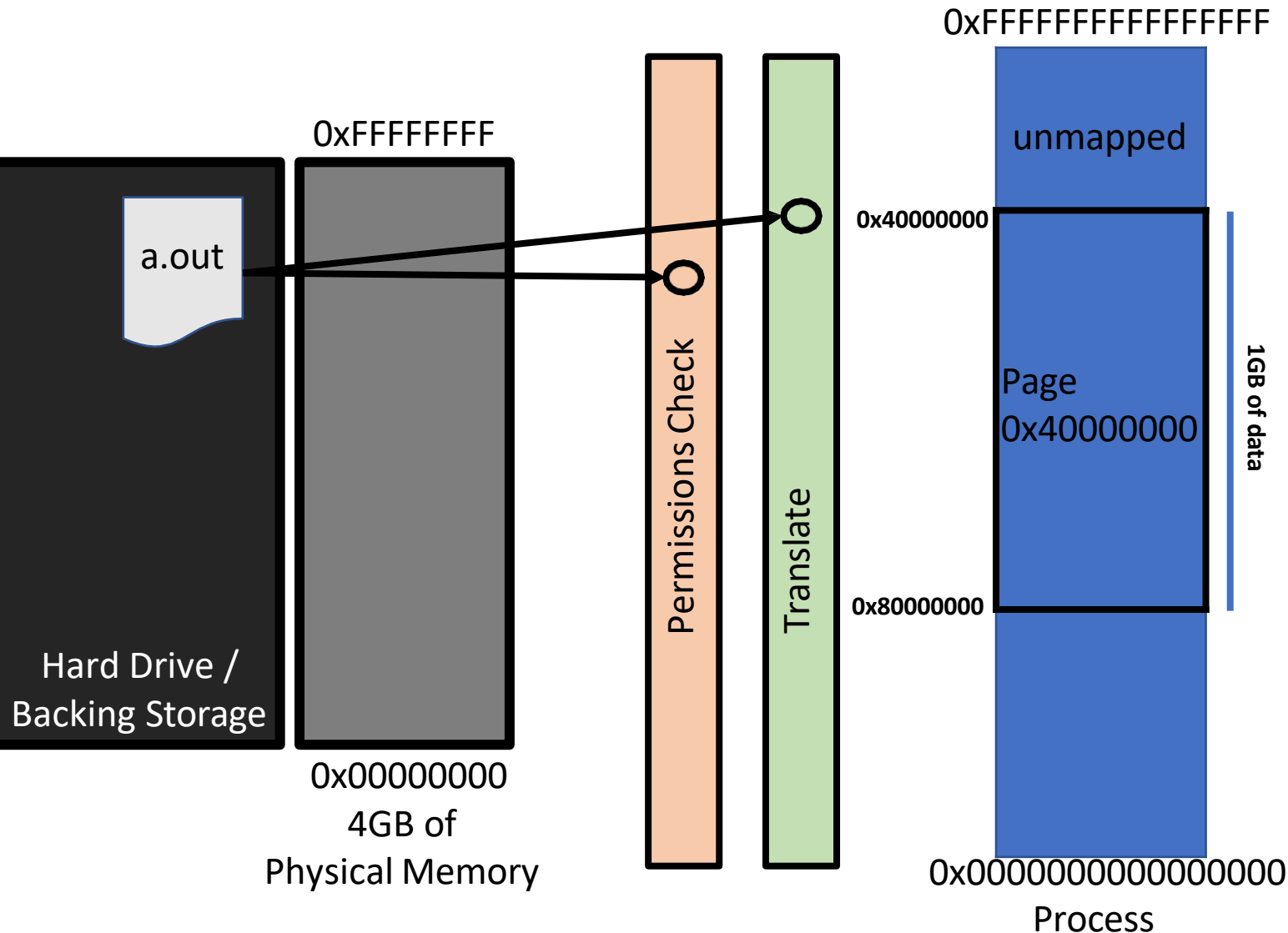
Separate L1 data & instruction (L1) TLB
Size: $64 * 8B = 512B$; $128 * 8B = 1024B$
Data Reach: $64 * 4kB = 256kB$ data
Code Reach: $128 * 4kB = 512kB$ code

Unified (L2) TLB
Size: $512 * 8B = 4kB$
Reach: $512 * 4kB = 2^{21} = 2MB$ code+data

Awesome Property of the TLB?
What if my working set is, like, 58GB?
Question: How do we increase TLB Reach?

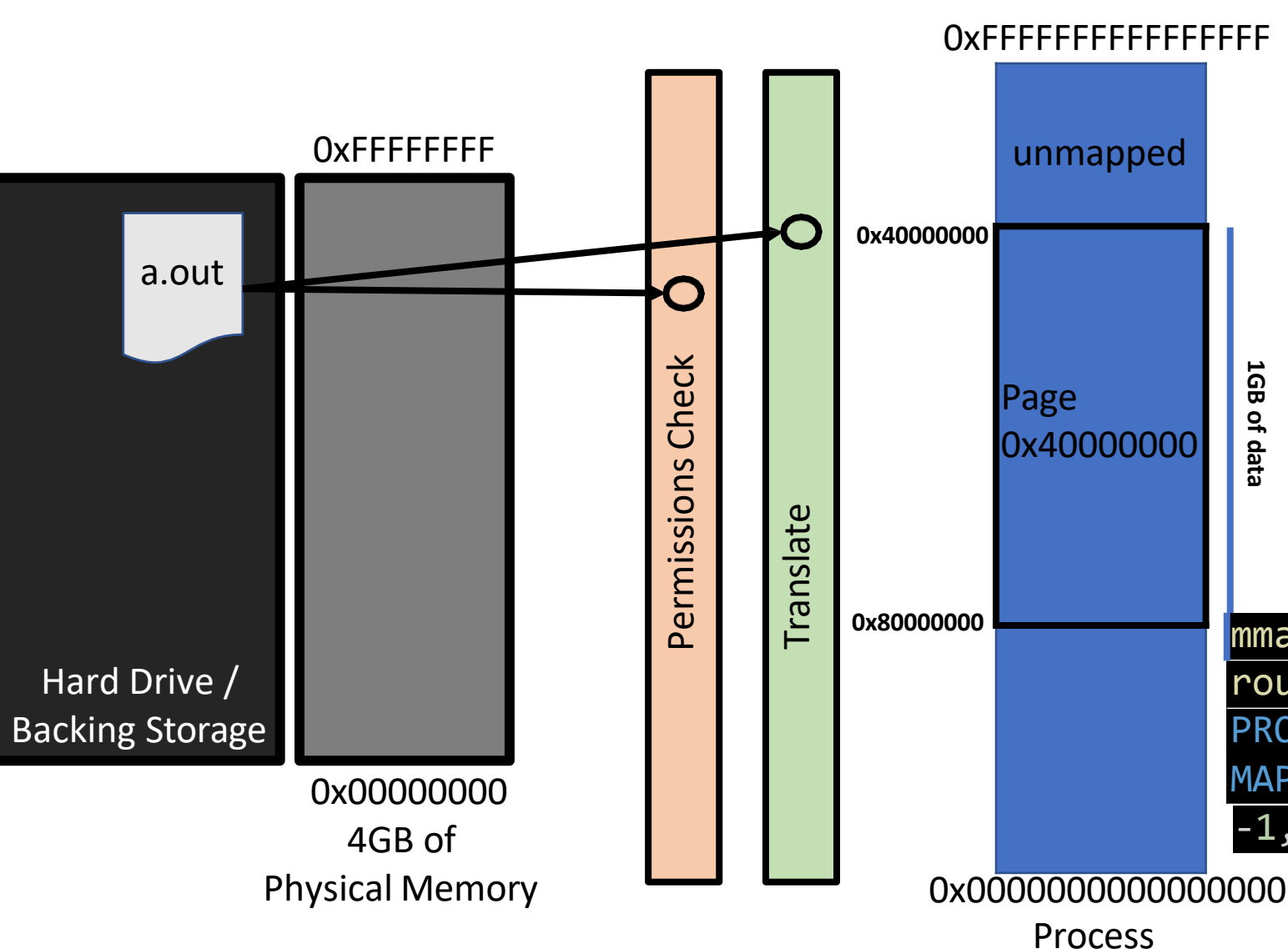
The Core i7 memory system.

Revisiting the Assumption of Page Granularity



What if a page were large (2MB) or huge (1GB)?

Increasing Page Size to Increase TLB Reach



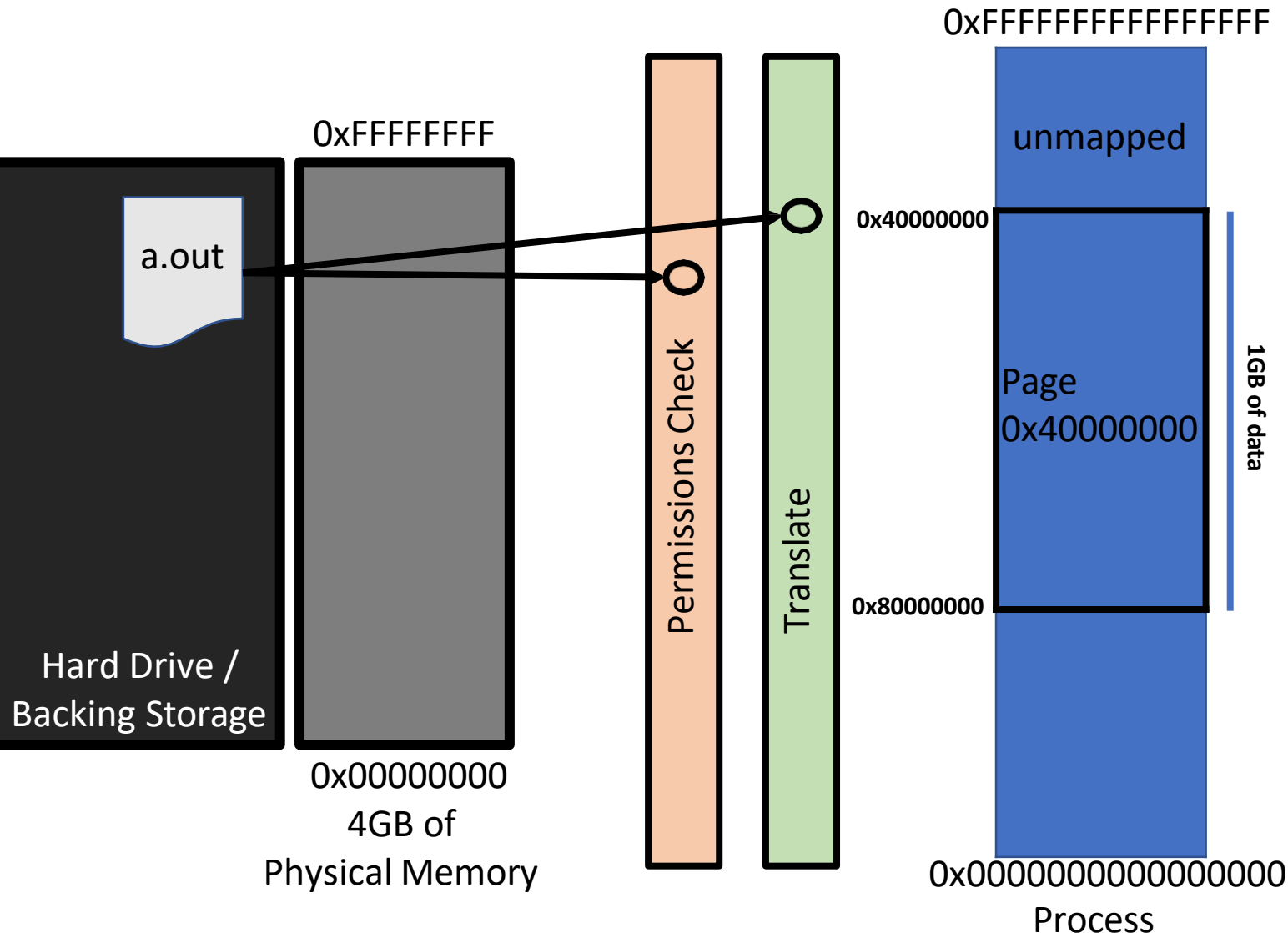
Modern OSes support large pages and huge pages via `mmap` (& other ways).

IA-64 (Itanium) had 8 page sizes 4kB – 256MB

Can mix different page sizes w/ hardware support

```
mmap((void*)0x0,  
round_to_huge_page_size(n * sizeof(T)),  
PROT_READ | PROT_WRITE,  
MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB,  
-1, 0));
```

Increasing Page Size to Increase TLB Reach



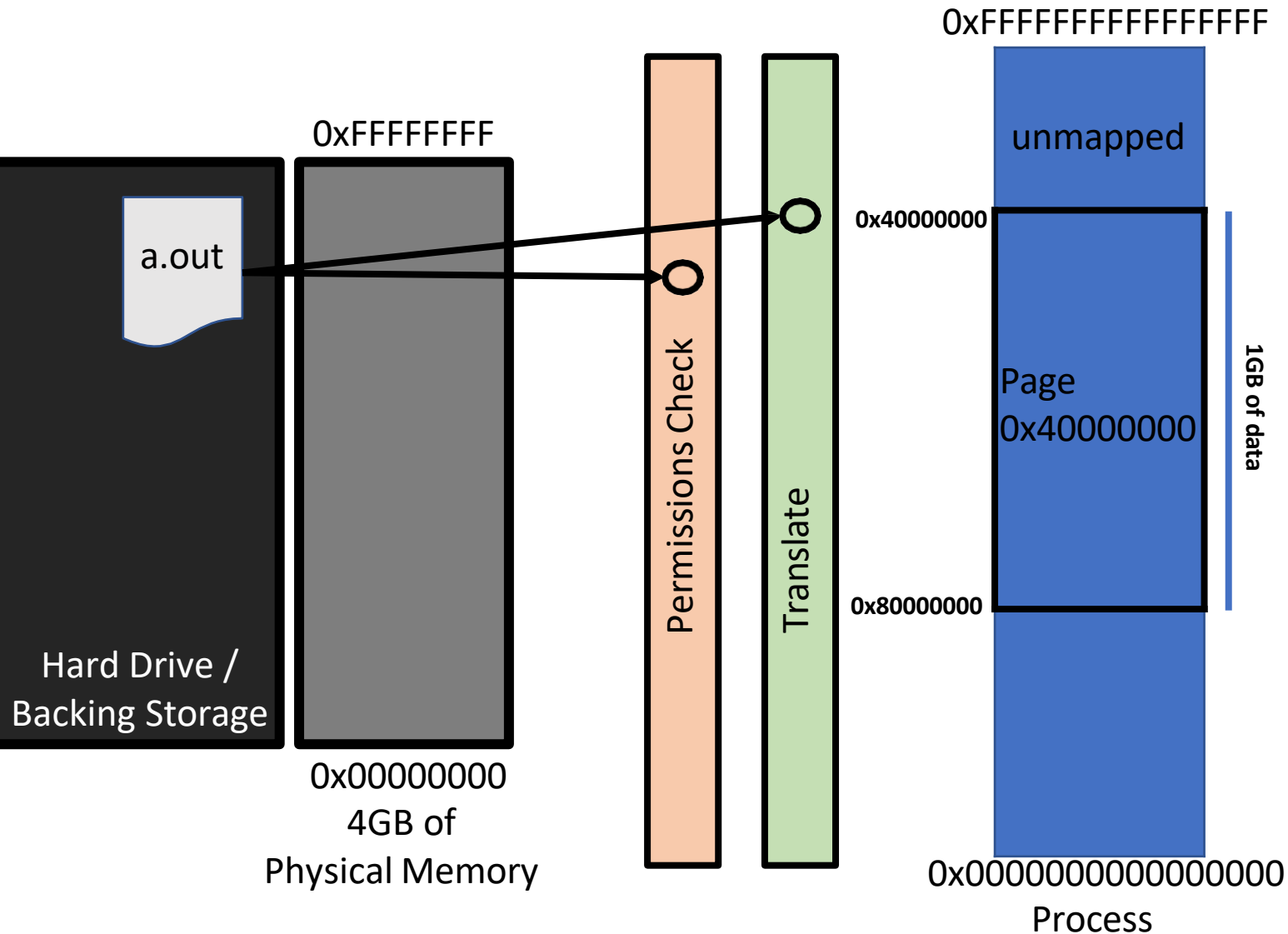
**“Transparent” huge pages
allow the OS to promote a
normal page to HUGE status**

```
advise(..., MADV_HUGEPAGE);
```

**Not guaranteed to Huge-ify.
If *aligned* more likely to be huge**

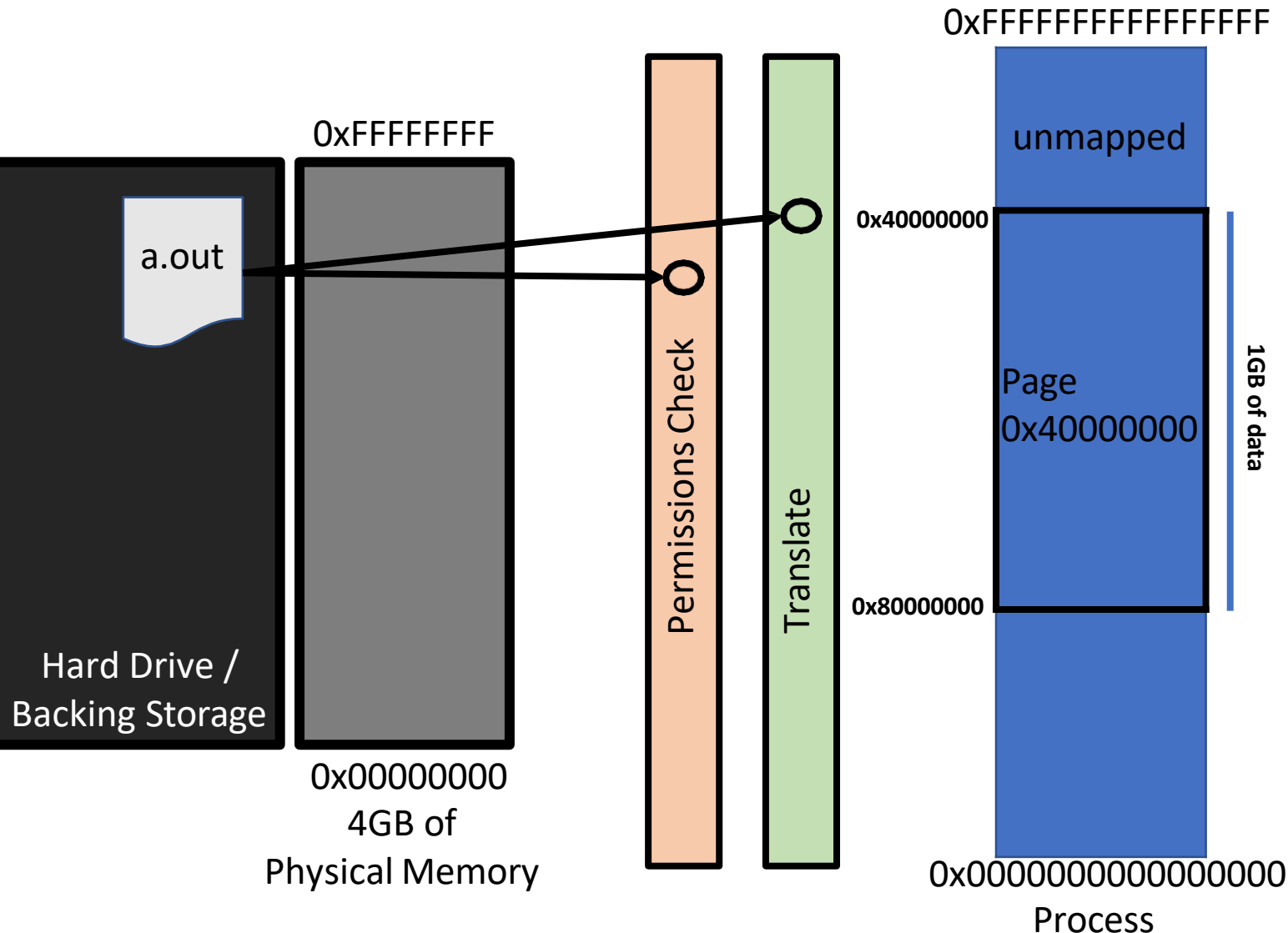
```
int posix_memalign(void **memptr,  
size_t alignment, size_t size);
```

Increasing Page Size to Increase TLB Reach



Risks / Costs of Increasing Page Size?

Increasing Page Size to Increase TLB Reach



Risks / Costs of Increasing Page Size?

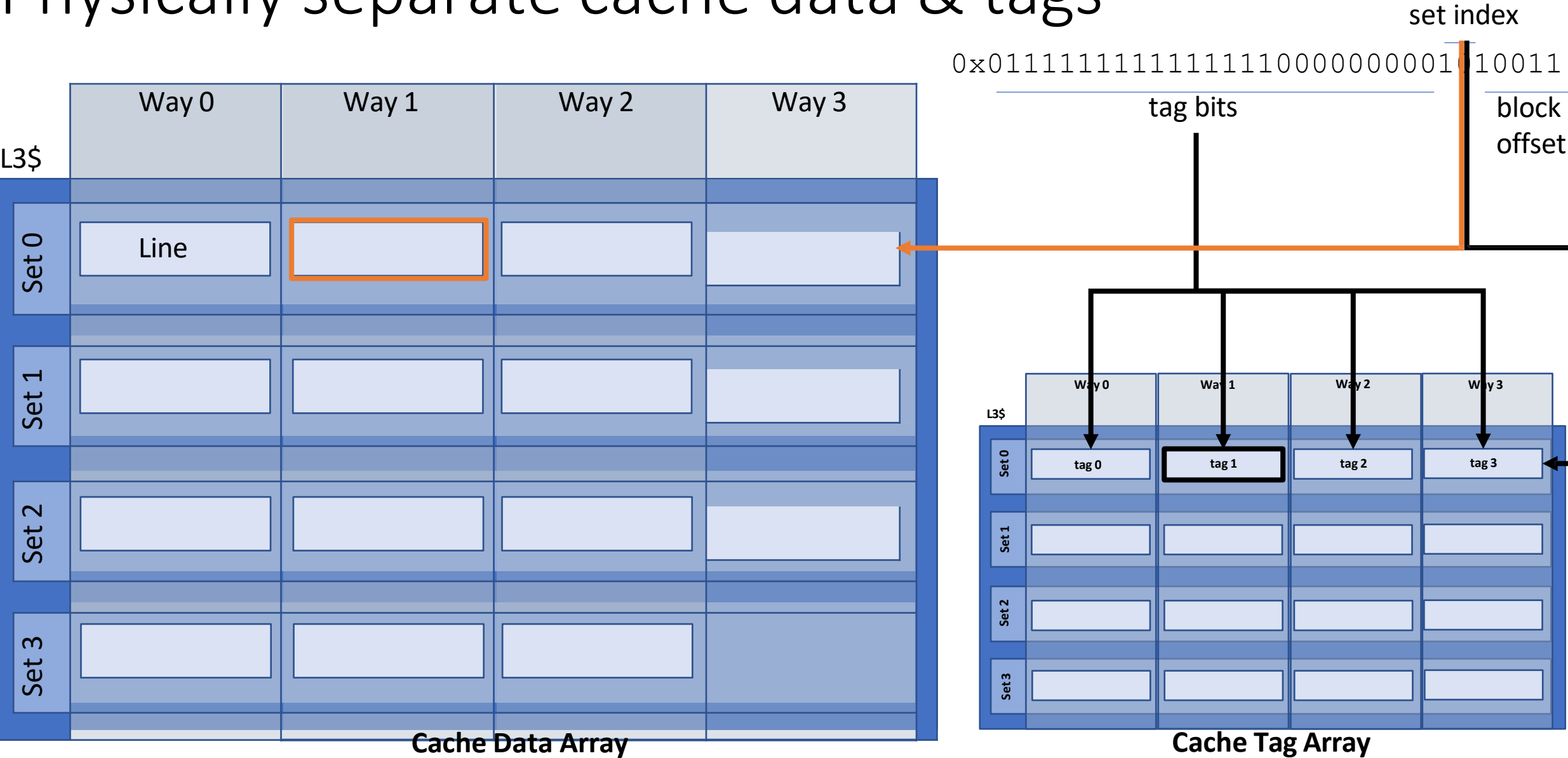
- High cost to page in/out on page fault (eek!)
- Wasting memory if hugeness is useless
- Internal page fragmentation
- Need HW to track page sizes
- Potential for programmer error w/ changing sizes
- High cost to zero a page

Use at your own risk! Try it out!

How Do Virtual Memory and Caching Interact?

Recall:

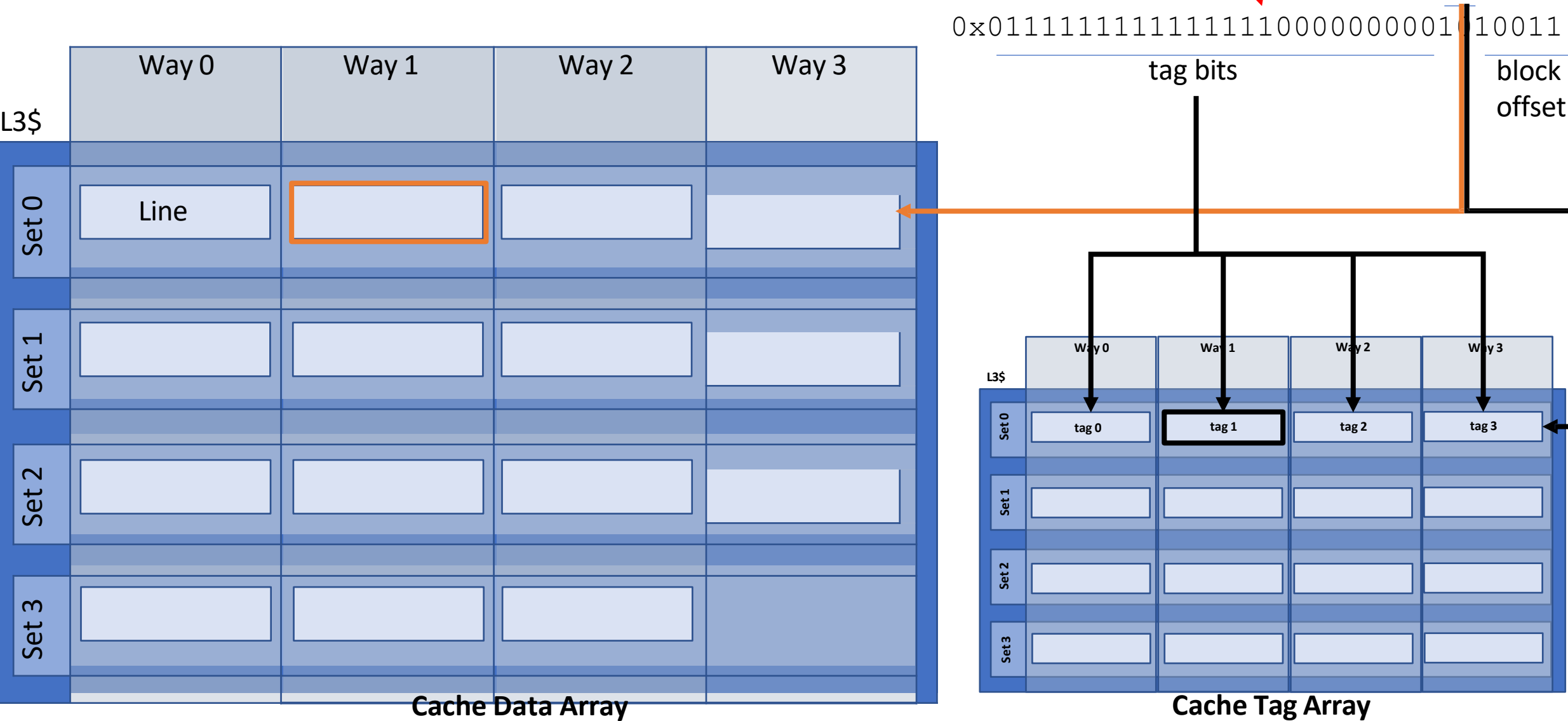
Physically separate cache data & tags



Recall:

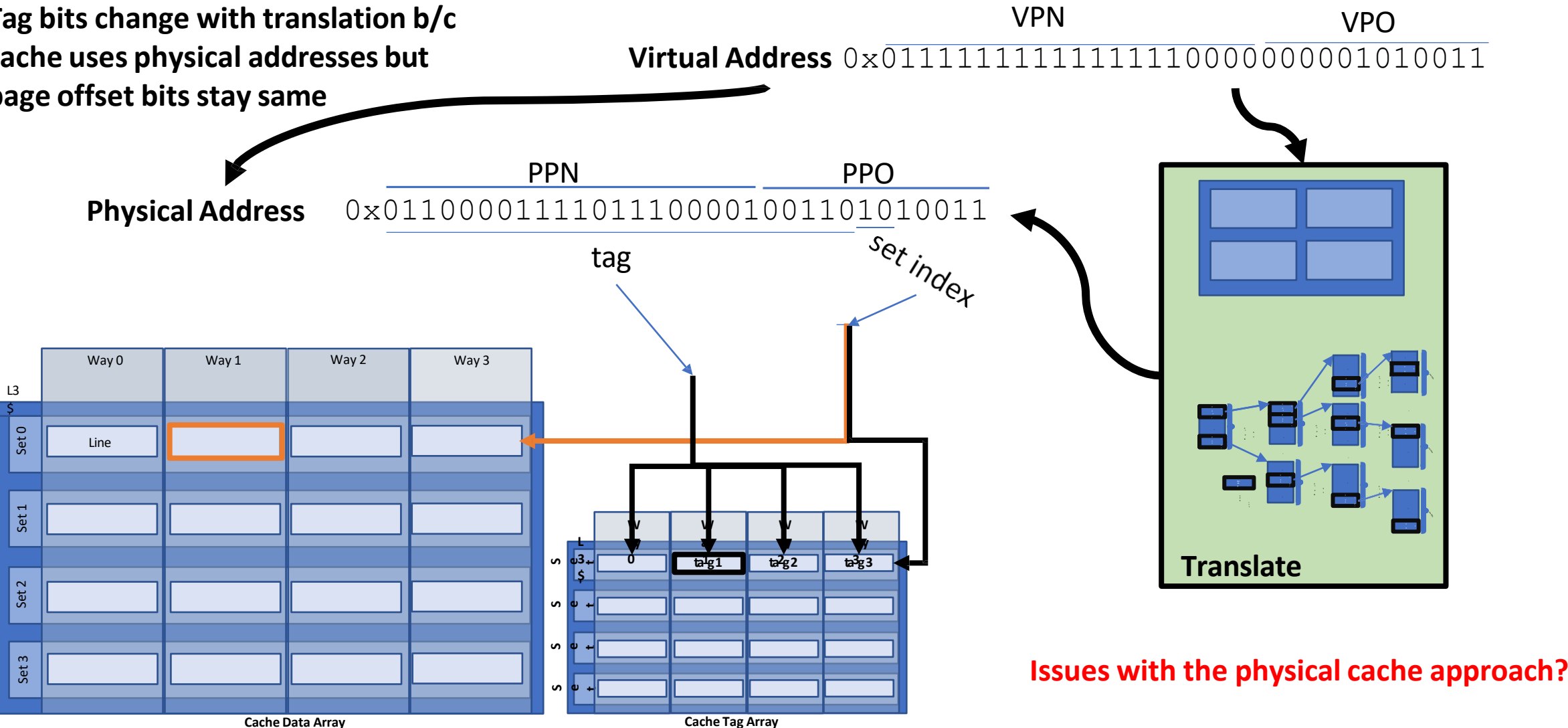
Physically separate cache data & tags

Question: Virtual or Physical Address?



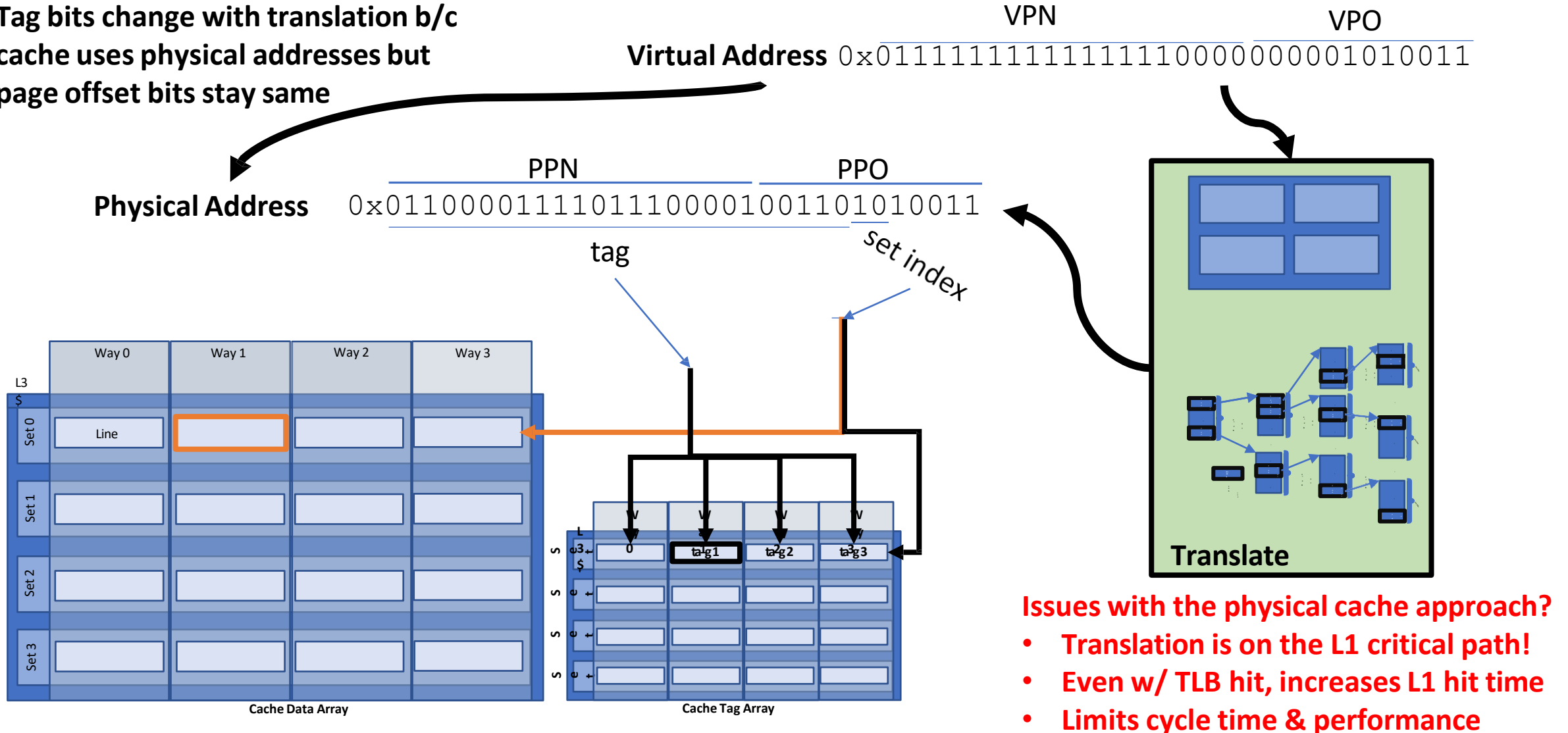
Physical Cache: Translate First Then Access Cache

Tag bits change with translation b/c
cache uses physical addresses but
page offset bits stay same



Physical Cache: Translate First Then Access Cache (PIPT: Physically Indexed, Physically Tagged)

Tag bits change with translation b/c
cache uses physical addresses but
page offset bits stay same



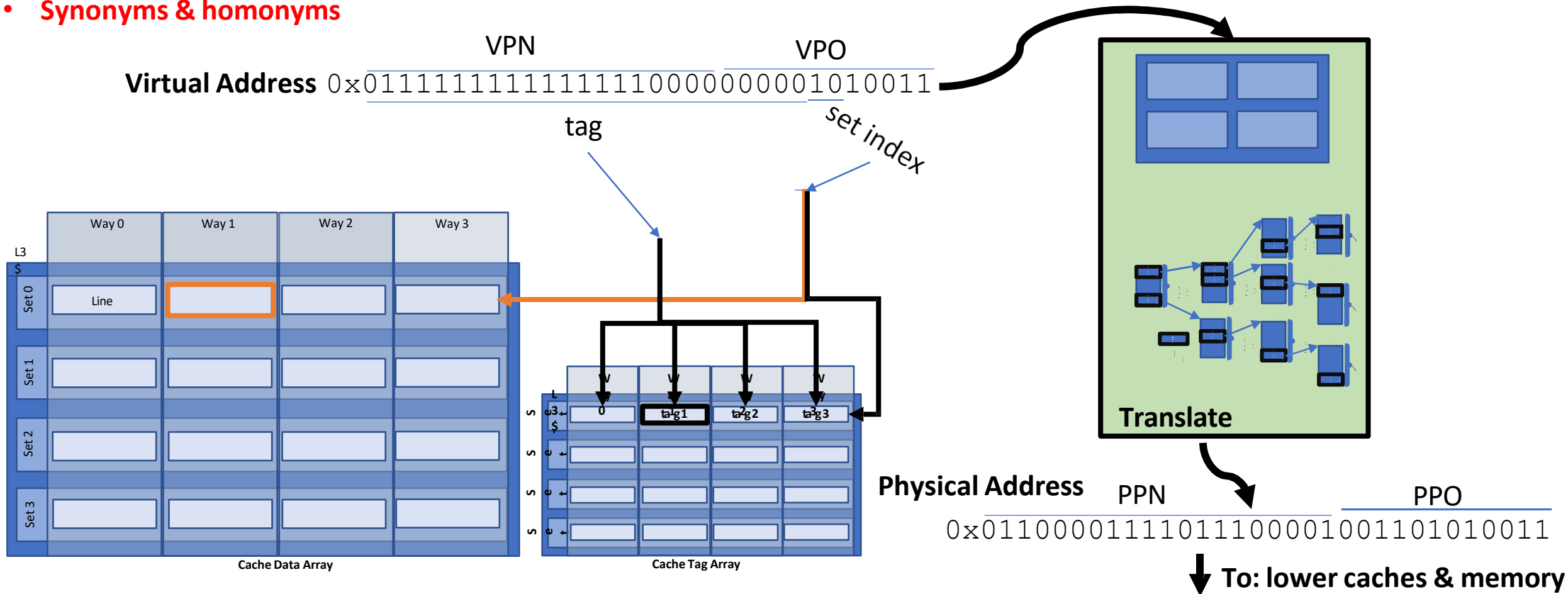
Virtual Cache: Access Cache Then Translate (VIVT: Virtually Indexed, Virtually Tagged)

Benefits of the virtual cache approach?

- Parallelize cache lookup & translate

Costs of the virtual cache approach?

- Synonyms & homonyms



Virtual Caches: The Synonym Problem

Process 1:

Virtual Address 1 VPN

VPO

0x011111111111111100001111000010010

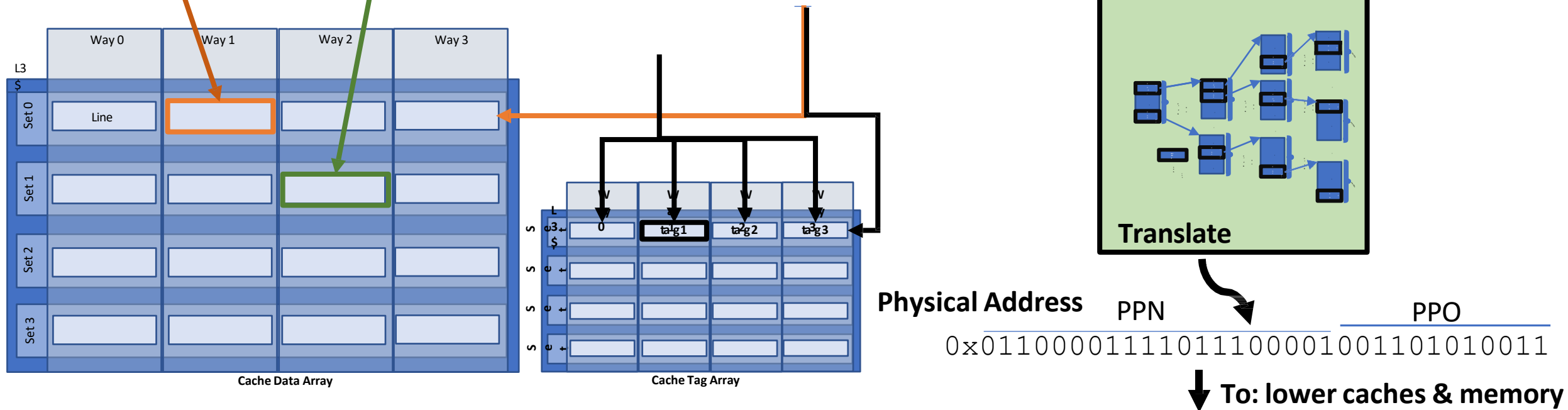
Process 2:

Virtual Address 1 VPN

VPO

```
0x01111111111111110000000000110011
```

Problem: two VAs refer to the same PA. Cached separately, in memory as a single block.



Virtual Caches: The Homonym Problem

Process 1:

Virtual Address 1

VPN

VPO

0x0111111111111111000011000010011

Process 2:

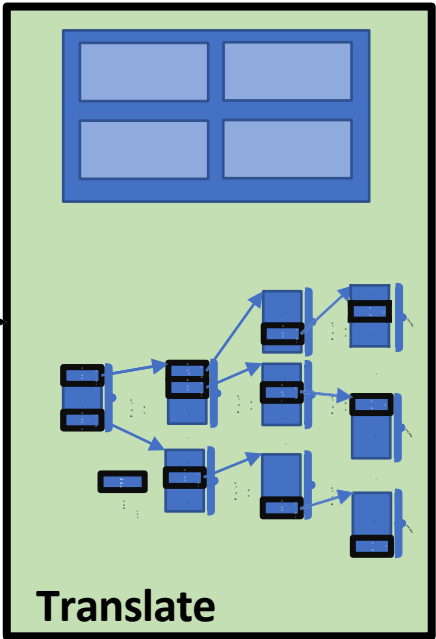
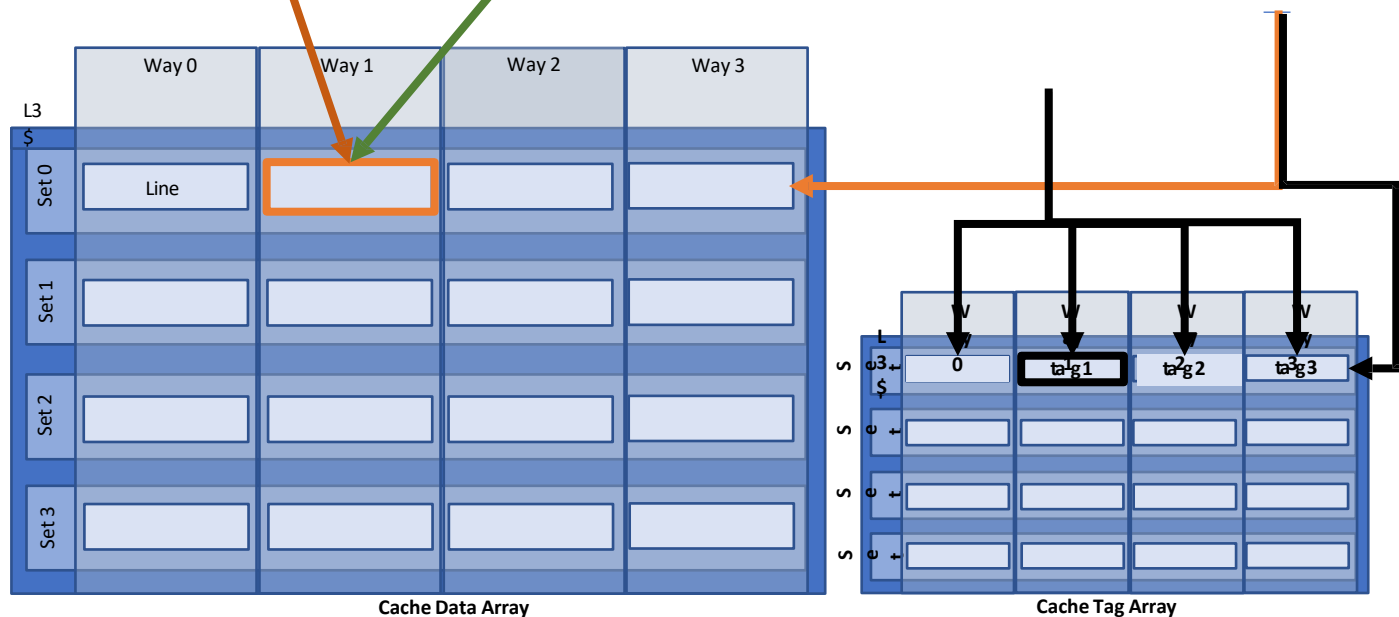
Virtual Address 2

VPN

VPO

0x01111111111111111000000000110011

Problem: *Same VA* refers to two different PAs. *Same block in cache, but distinct in memory.*



Physical Address 1

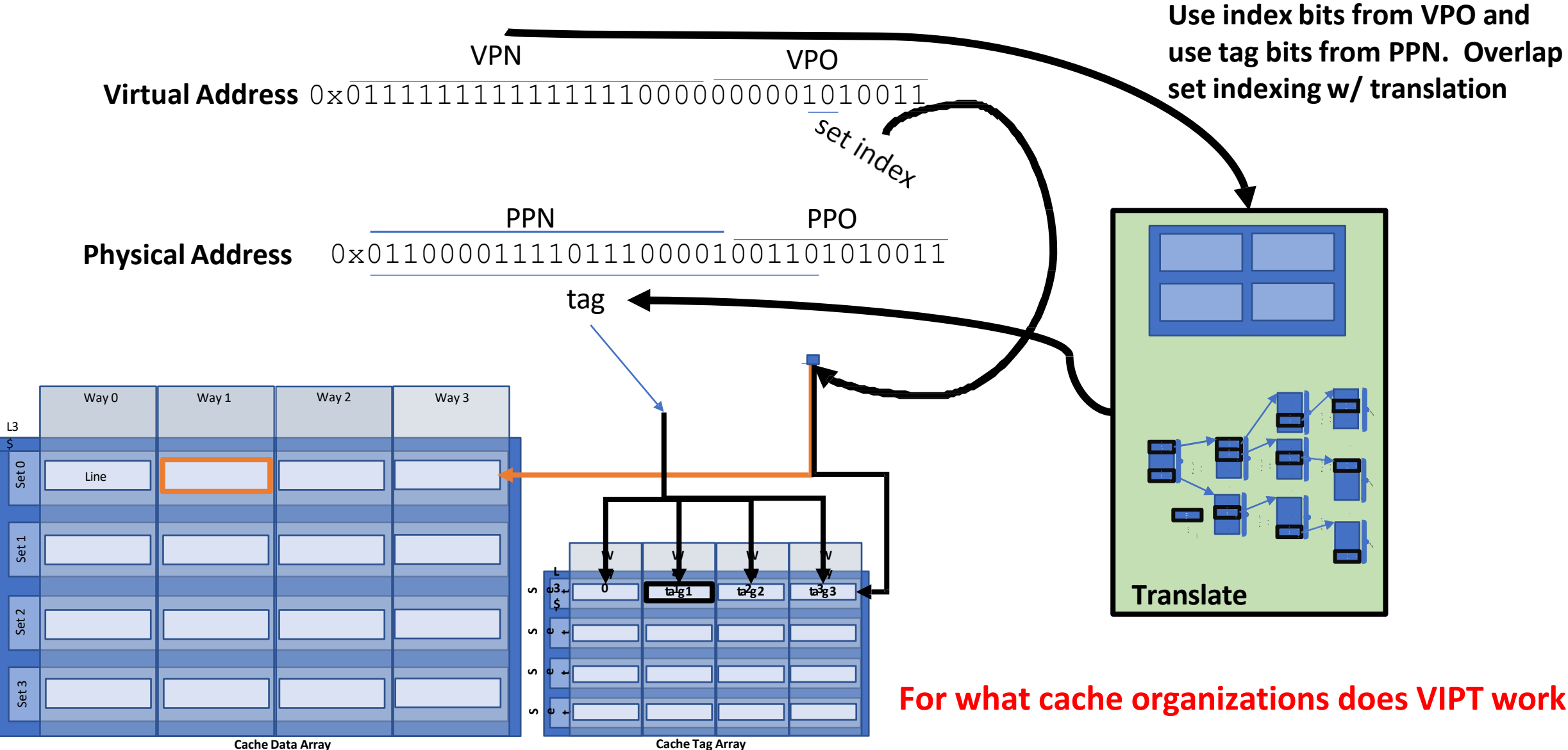
0x01100001111011100001001101010011

Physical Address 2

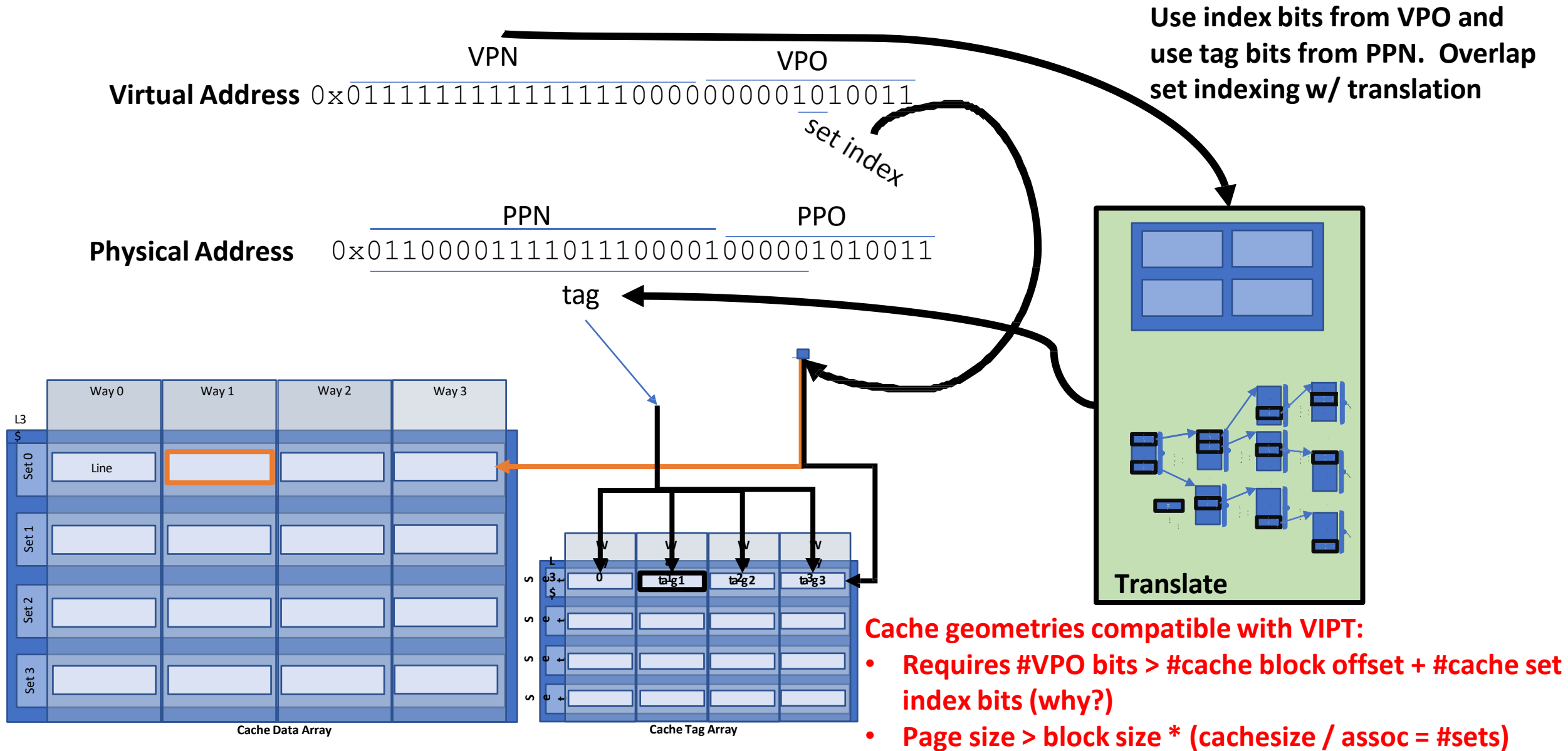
0x0100000011111000001001101010011

↓ To: lower caches & memory

Virtually Indexed, Physically Tagged Caches



Virtually Indexed, Physically Tagged Caches



Virtual Caches vs. Physical Caches

- Virtual Cache: uses virtual address to do cache lookups
- Physical Cache: uses physical address to do cache lookups
- Virtually-Indexed, Physically-Tagged (VIPT): uses virtual set index bits to do set lookup, uses physical tag bits to do tag comparison

What did we just learn?

- Virtual memory, from the ground up
- Partitioning & segmentation: partial solutions
- Dynamic, software mapping, translation, and permissions checking
- Page tables & hierarchical page tables
- TLBs for accelerating translation
- Caches & VM together