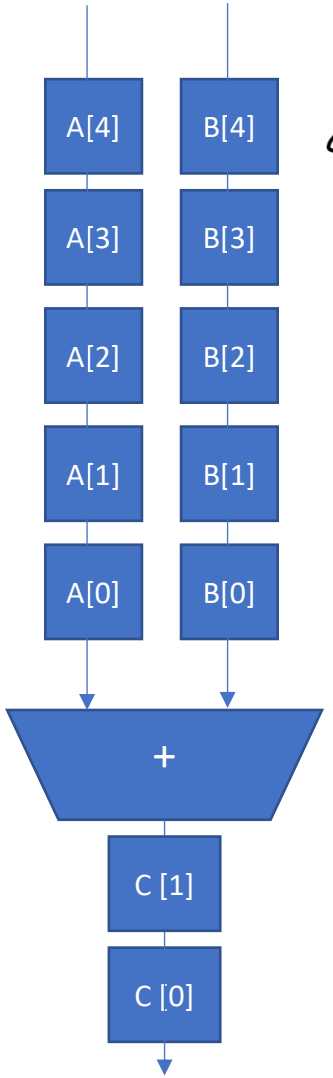
CMU 18-344: Computer Systems and the Hardware/Software Interface

Fall 2021, Prof. Brandon Lucia

Recap – Vector Machines

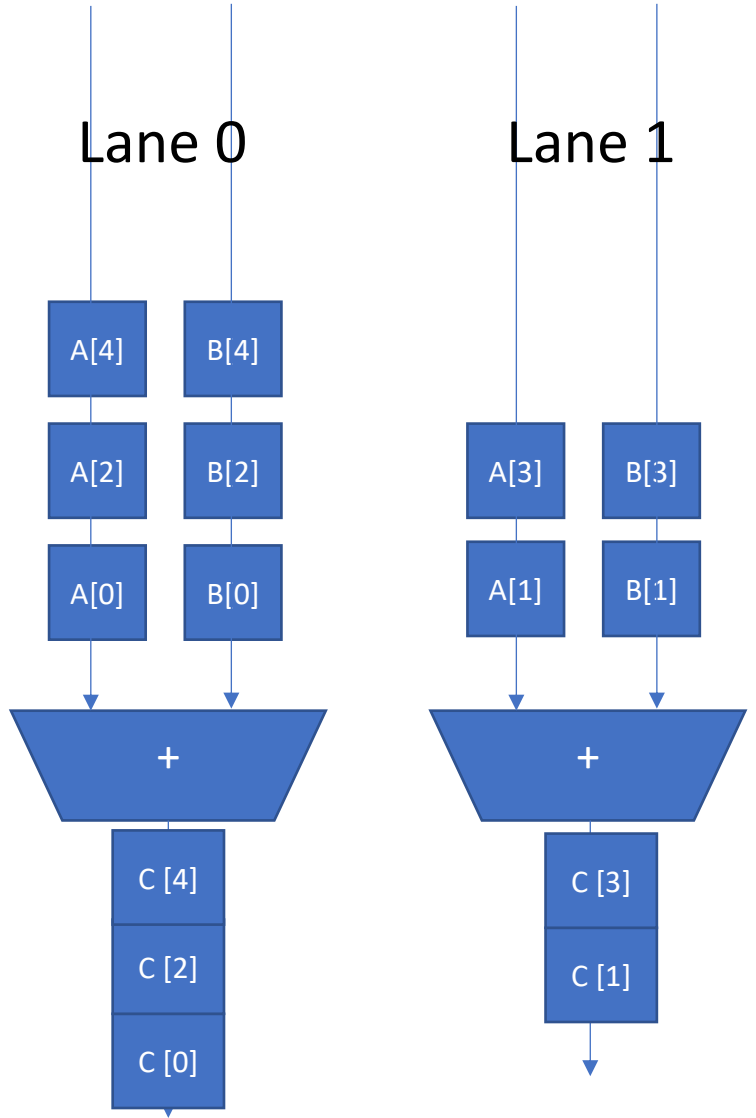
- SIMD is an execution model that lends itself easily to parallelism
- Vector machines have the abstraction of processing on long vectors
- Vector machines *amortize* instruction costs over many operations (one per data element in a vector of input data)
- Not free: need new interface (change programming) and some large structures (e.g., Vector Register File)

Vector Machines are Easily Parallelizable

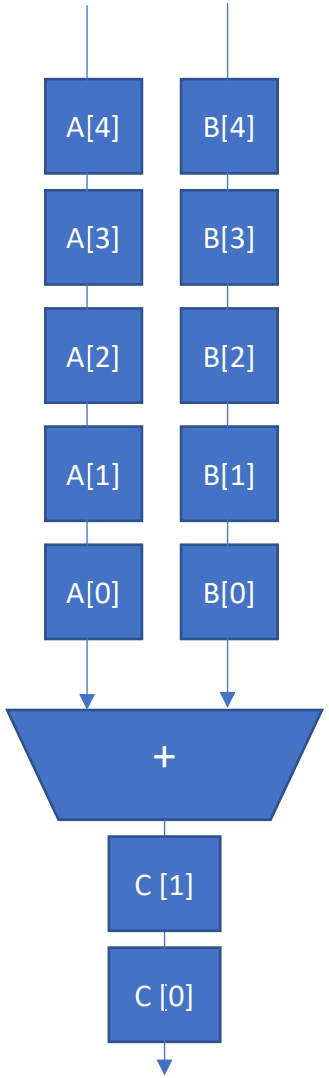


Abstraction: execute an instruction's operation over an entire vector of data

Implementation: Parallel functional units each process parts of a vector, producing a vector output. **Why simple?**



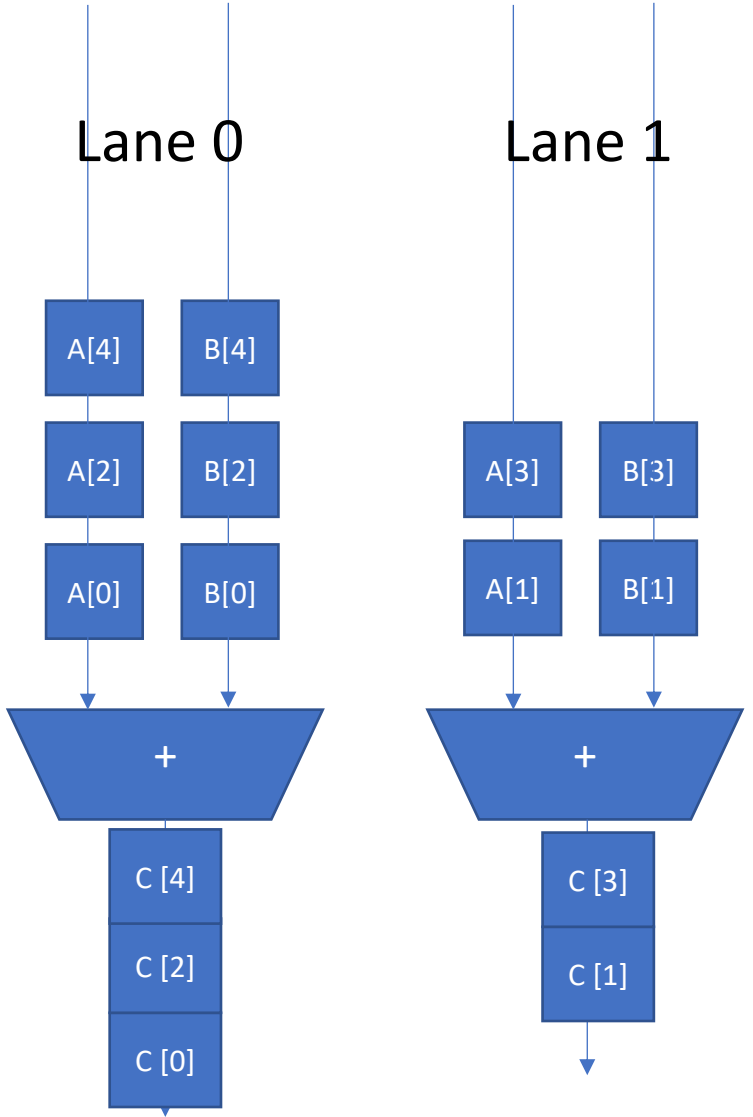
Vector Machines are Easily Parallelizable



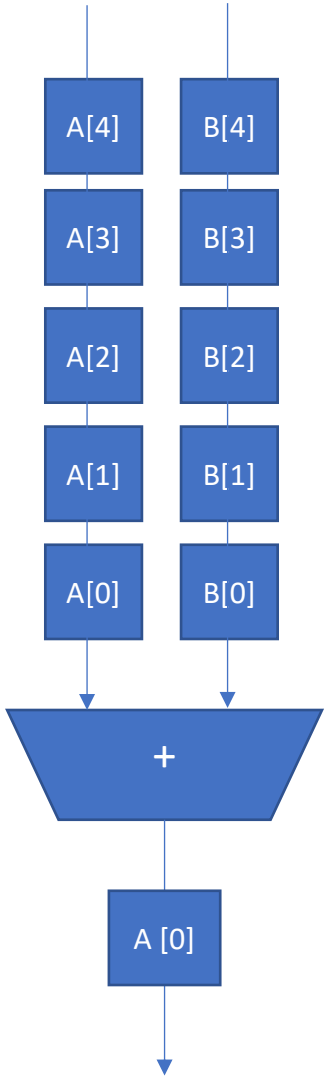
Simple: Vector instruction operates on v0[**i**] and v1[**i**] ***not*** v0[**i**] and elem *v.

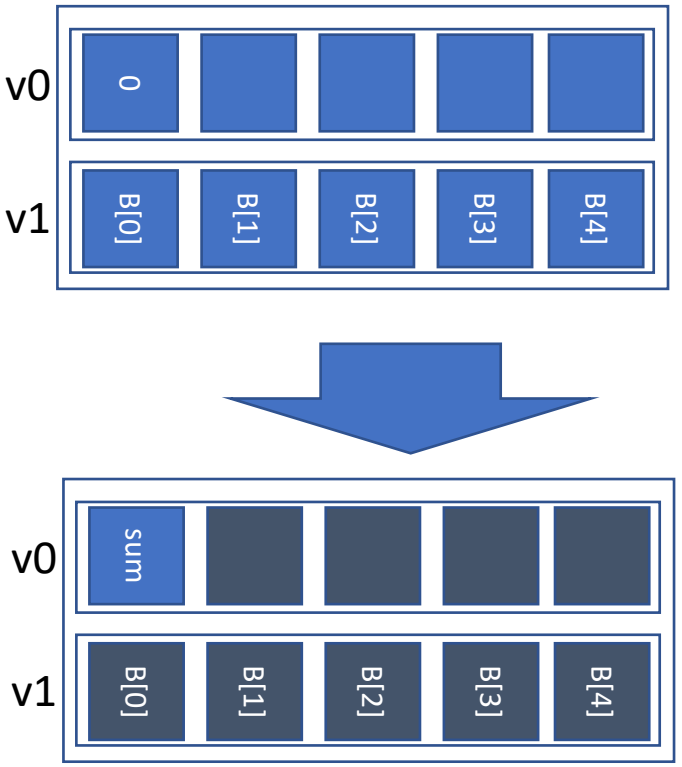
Very simple operand matching logic, no need to track complex producer consumer relationships across inputs of operations.

Primary cost?



Reduction Operations






```
setvl 5
vld v0, a
vld v1, b
vredsum v0, v0, v1, vm
```

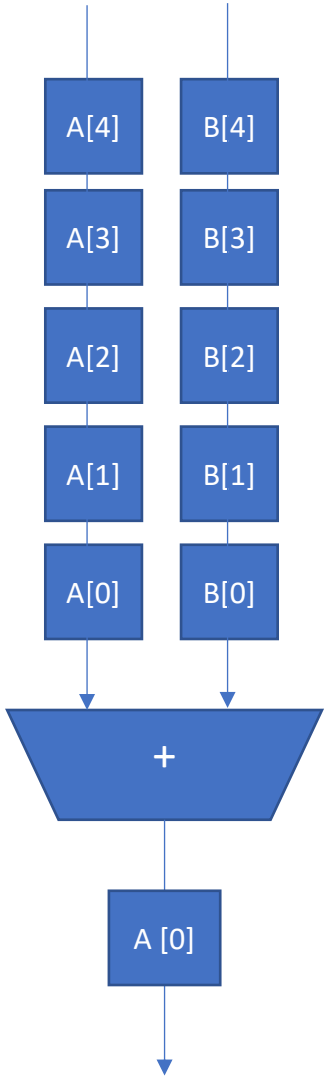
dest vreg [0]

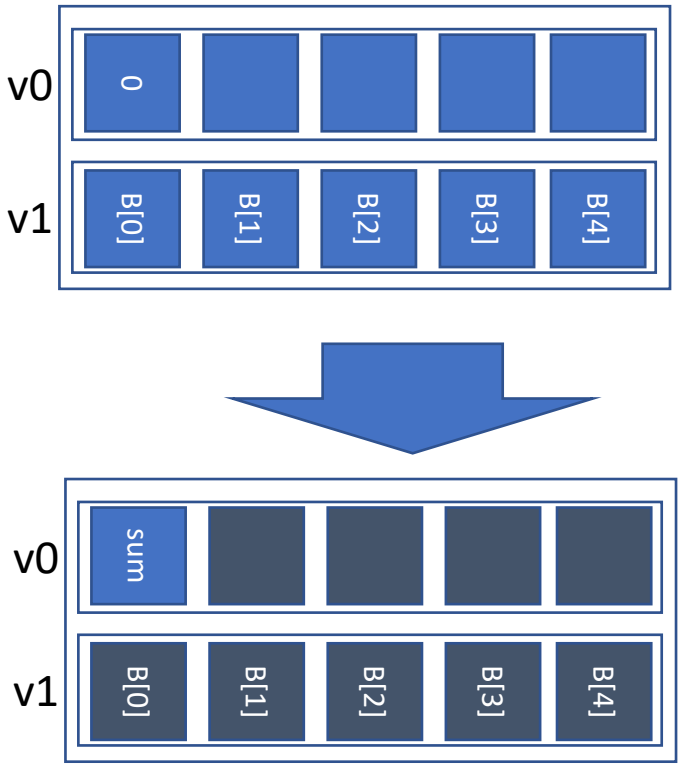
vreg to reduce

initial value

$v0[0] = v0[0] + \Sigma_i v1[i]$

Reduction Operations






```
setvl 5
vld v0, a
vld v1, b
vredsum v0, v0, v1, vm
```

v0[0] =

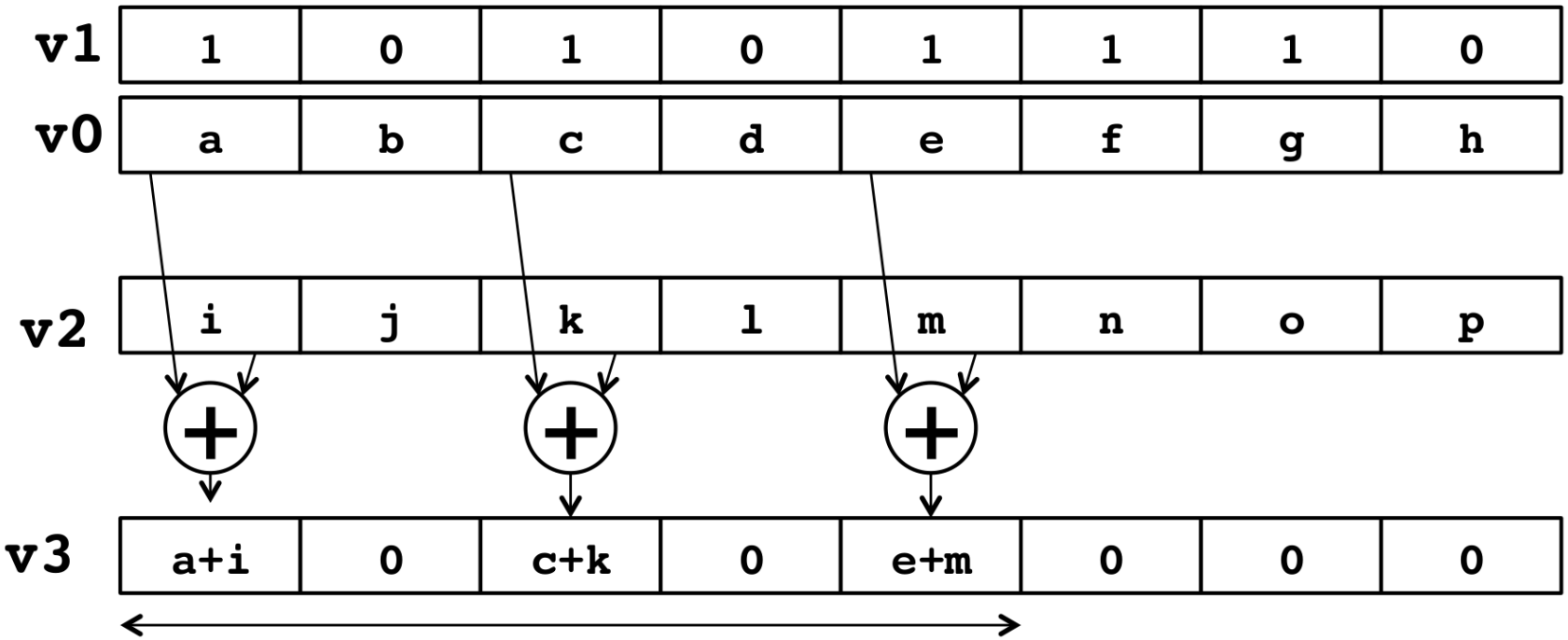
v0[0] + Σ_i v1[i]

Vector Masking

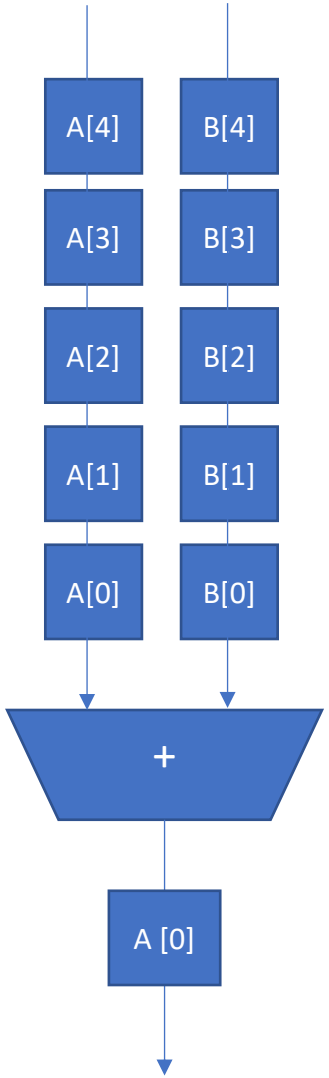
`vadd v3, v0, v2, v1.t`

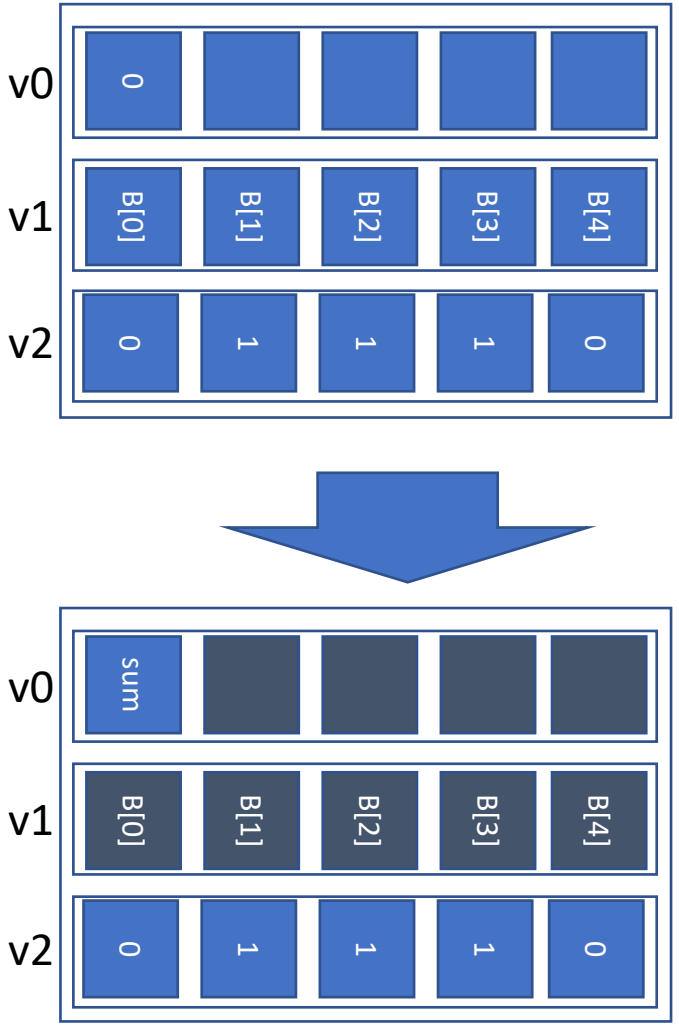
Behavior of a masked vector operation: For elements up to vl in v3, add elements from v0 and v2 if that element in v1's LSB is set to 1, set other v3 elems to 0

What high-level programming concept does this get used to implement?



Reduction Operations with a vector mask






```
setvl 5
vld v0, a
vld v1, b
vredsum v0, v0, v1, v2
```

input vec

mask

dst

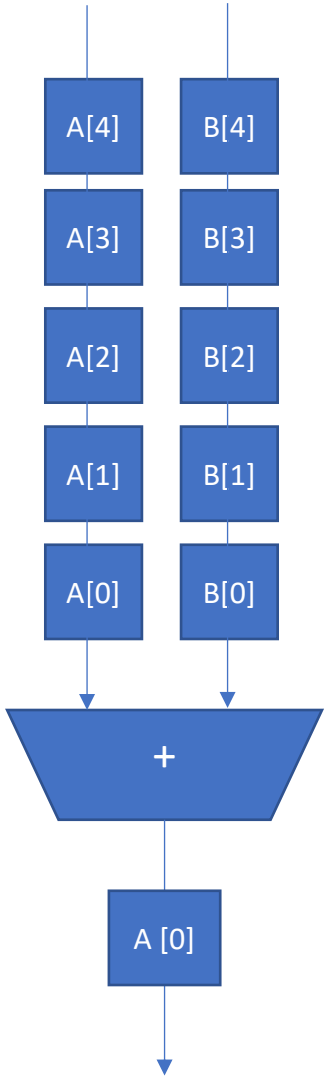
init val

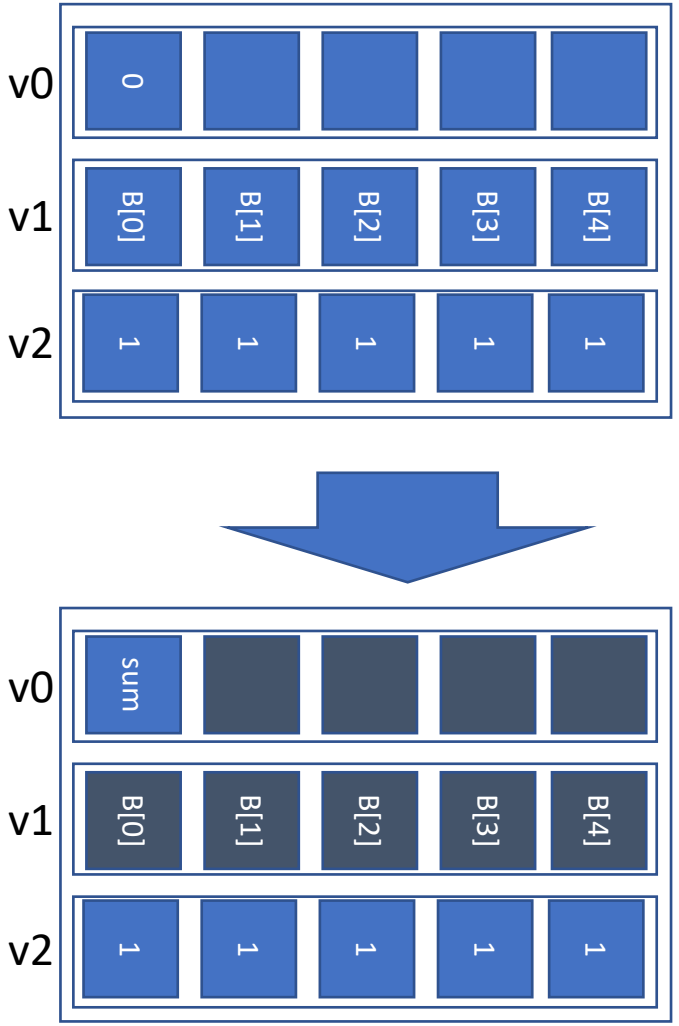
Reduction operations accumulate the result of an operation on a vector into the first element of a destination vector

Uses for reduction?

```
v0[0] =
v0[0] +
/*v1[0] +*/
v1[1] + v1[2] + v1[3]
/*+ v1[4]*/
```

Reduction Operations with a vector mask



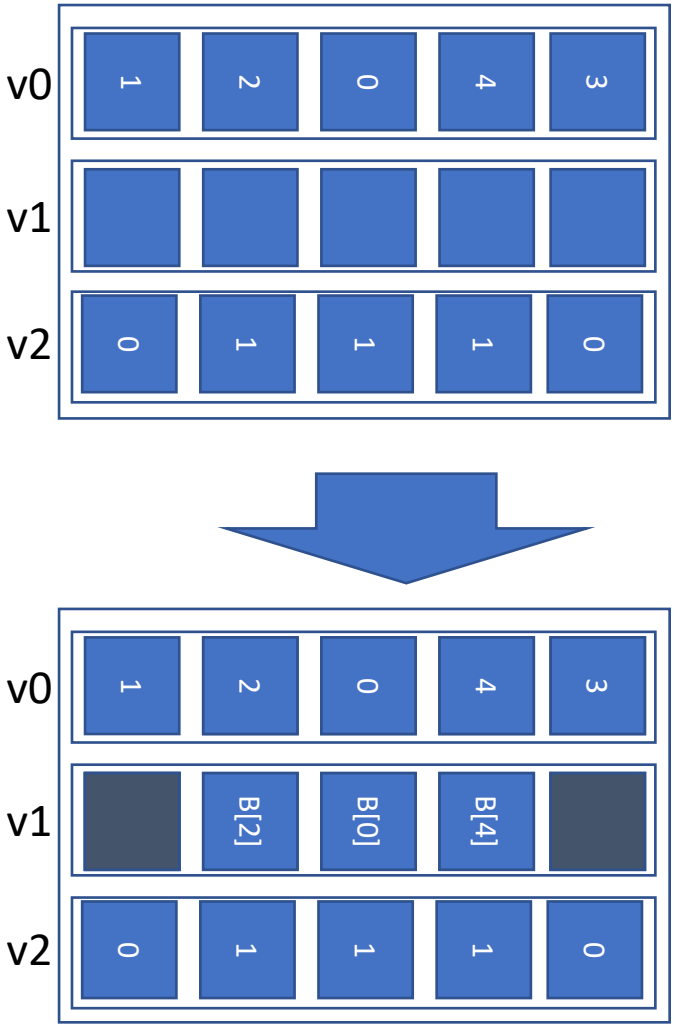


Uses for reduction?
Dot product, e.g.,

```
setvl 5
vld v0, a
vld v1, b
vmul v0, v1
vredsum v0, v0, v1, v2
```

```
for( i = 0..len){
  v[i] += a[i] * b[i]
}
```

Indexed Memory Accesses (Scatter/Gather)



index vector

dest

```
vluxei64 v1, (&B), v0, v2
```

base addr

mask

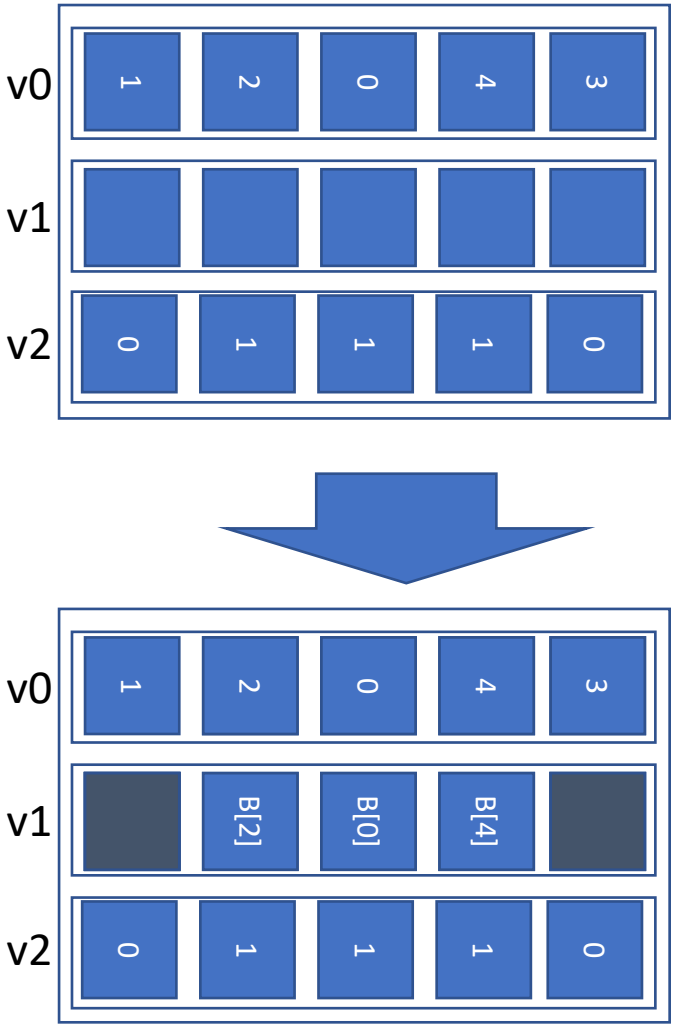
Indexed memory loads **"gather"** elements from all over memory into a contiguous vector register.

Indexed memory stores **"scatter"** elements from a contiguous vector register into locations all over memory

Uses?

```
v1[i] = v2[i] ? B[v0[i]] : v1[i]
```

Indexed Memory Accesses (Scatter/Gather)



dest

index vector

```
vluxei64 v1, (&B), v0, v2
```

base addr

mask

Common Use: indirect array accesses. Common in graph analytics

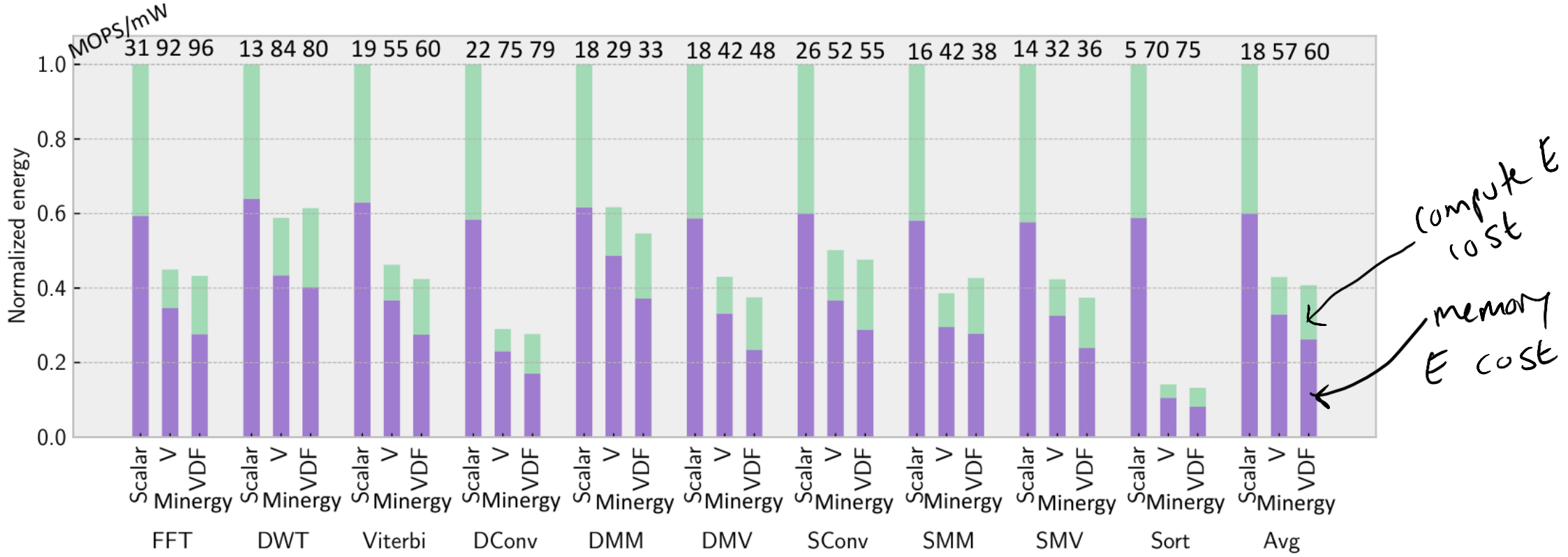
```
for( src in 0 .. n ){
  for( dst in 0..ind[src].len() ){
    data[ ind[src][dst] ]++;
  }
}
```

```
v1[i] = v2[i] ? B[v0[i]] : v1[i]
```

Summary of Benefits: Vector Architectures

- **Compared to scalar architectures:**
  - **Single instruction performs many operations:** one instruction is the equivalent of executing an entire loop of a program!
  - **Control is simpler:** no loops, no branches, no misprediction/misspeculation
  - **Vector interface makes data-independence across vector elements explicit:** simplifies implementations and eliminates complex dependence logic
  - **Dependence checking of vectors, not elements:** what dependence tracking is required pertains to entire vector registers, not individual elements, amortizing its cost significantly
  - **Easy to express data parallelism:** avoids software complexity of multithreading on a multiprocessor (i.e., MIMD)
  - **Maximize value of memory bandwidth:** contiguous/strided vector fetch operations are a good match for highly-banked memories
  - **Energy efficiency**: instruction & data fetch amortize costs across vector saving energy
  - **Require vector programming style, which means changing all of your code. Code doesn't match vector style well? Can't use the vector architecture without lots of extra work!**

Vector execution model saves energy (and time) over scalar processing



Taken from a very recent research project about optimizing for minimum energy by using a new vector processor (**V** bars in the plot) and a customized variant (**VDF** bars in the plot). **V/VDF** use RISCV vector insns., **scalar** plain RISCV insns.

Key take-away: vector processing cuts energy by ***more than half*** compared to scalar processing.

What did we just learn?

- We learned about how VLIW and Vector processing are two different takes on the hardware software boundary that admit more parallelism than SS/OoO's ILP focus allows
- VLIW did not take over, vector has been a consistent background hum
- Both approaches require the programmer and the compiler to make big changes to code to work well with these new hardware/software interfaces.

What to think about next?

- Lab 3 out Today
- Next we look at Virtual Memory as an abstraction
- Also look at the underlying mechanisms and options for implementing virtual memory in a modern CPU

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

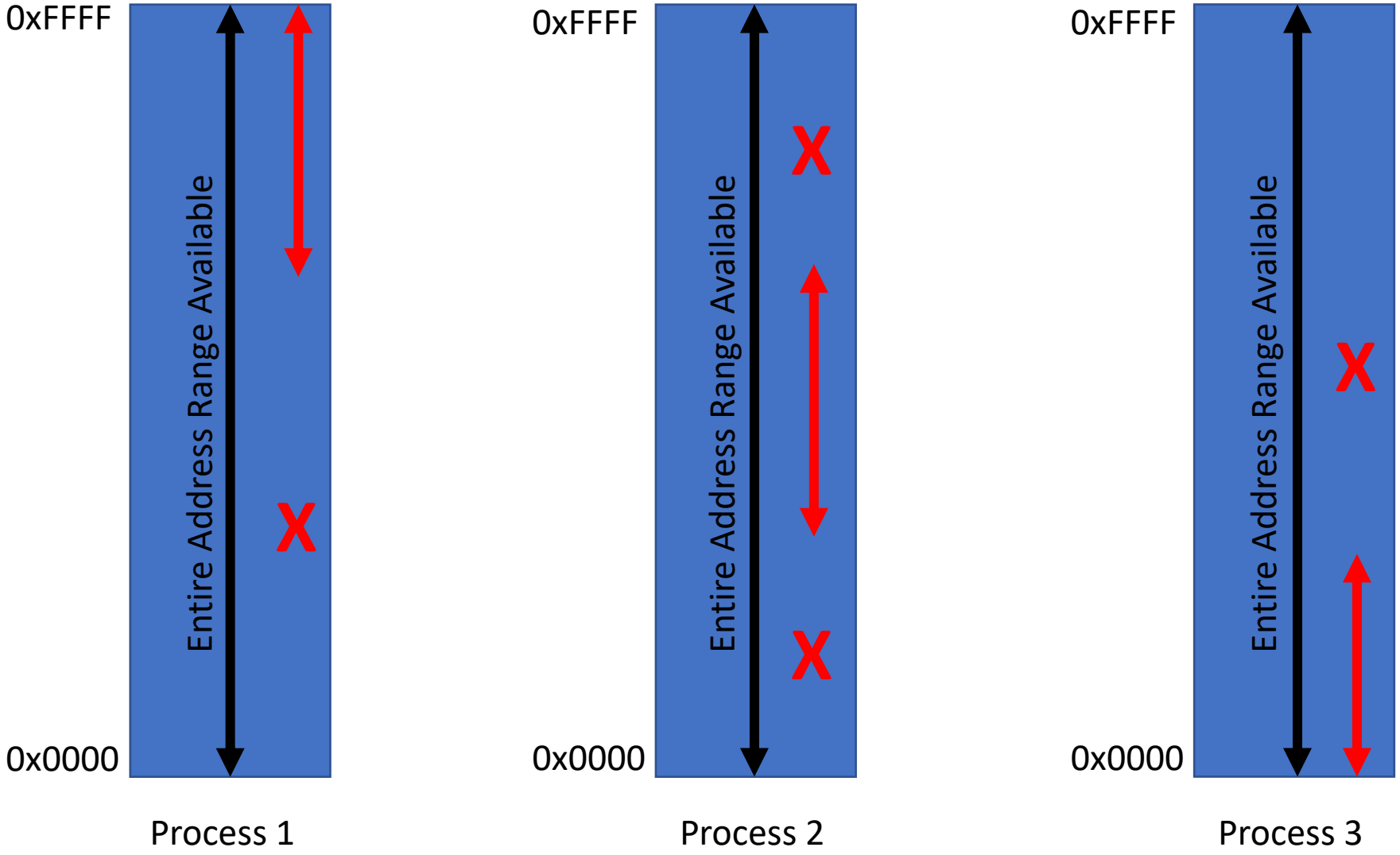
- Expose abstraction of abundant resource despite limited resource
- Expose abstraction of uniform resource despite heterogeneity of resource
- Expose abstraction of isolated resource despite sharing of resource

Virtualization – What resources?

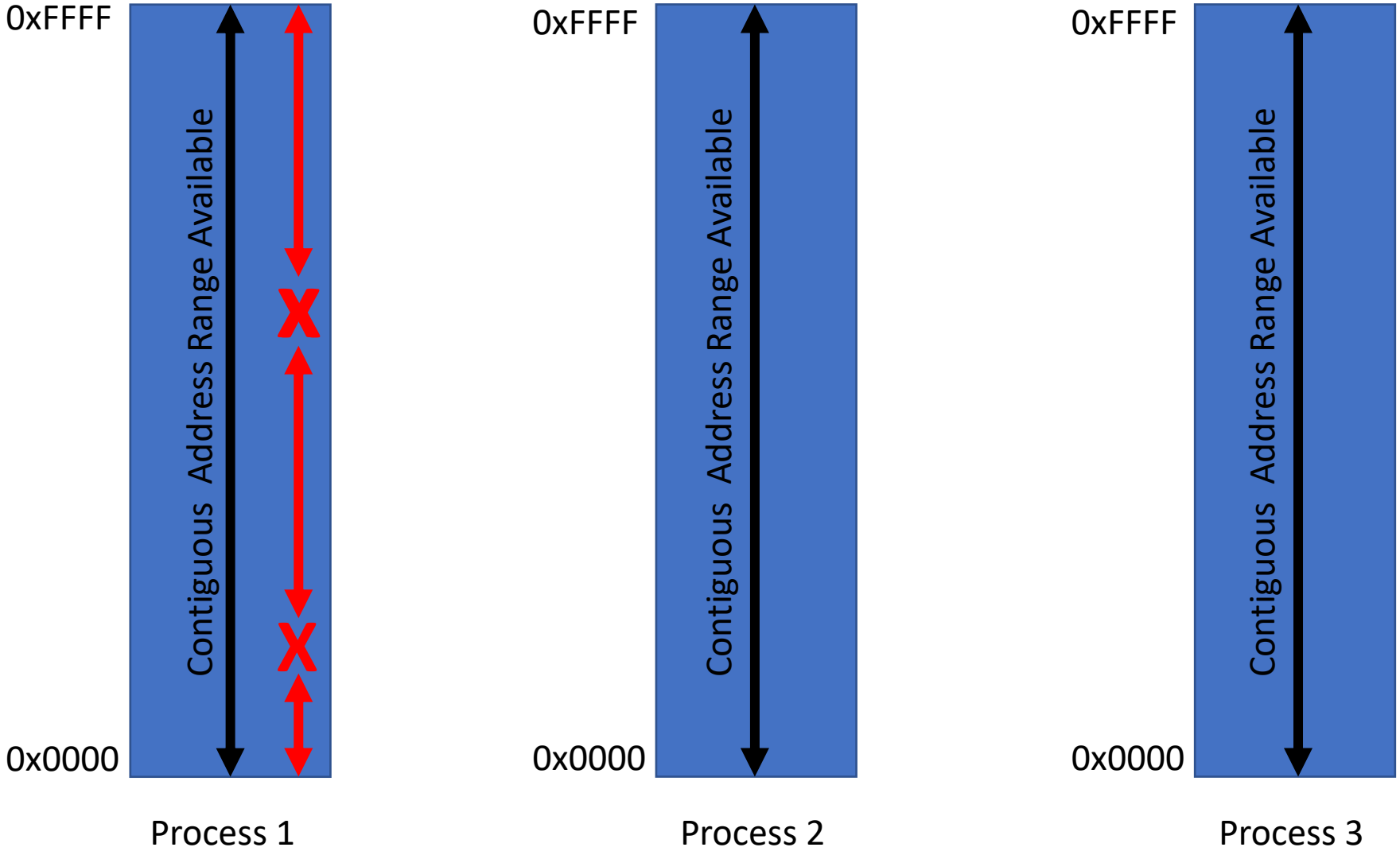
- Entire machines (VMMs)
- Storage (Disk controllers / Flash controllers)
- Memory (Virtual Memory)
- Network connectivity / bandwidth (Software-defined Networks)

Memory Virtualization

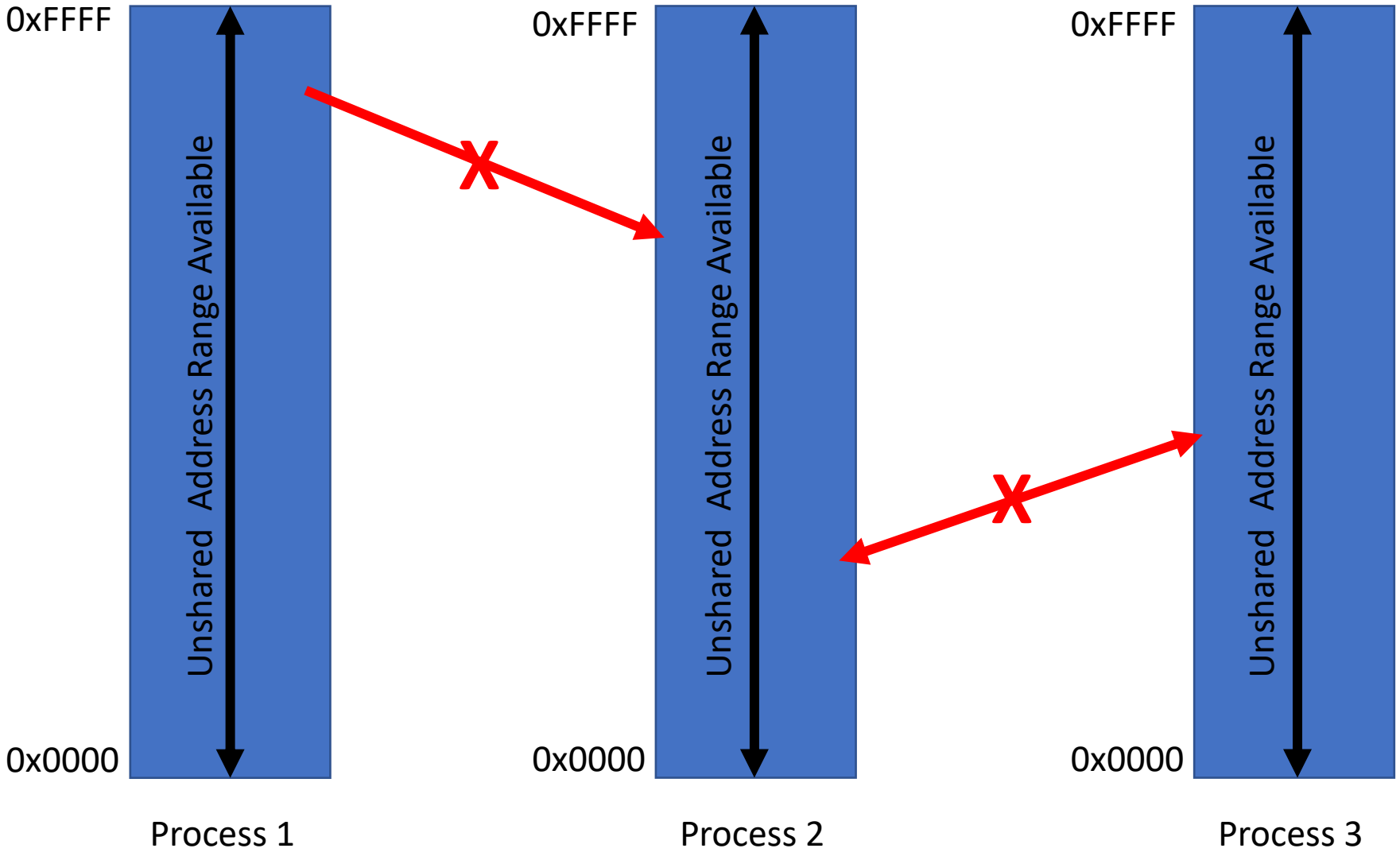
Virtual Memory – Abstraction of Abundance



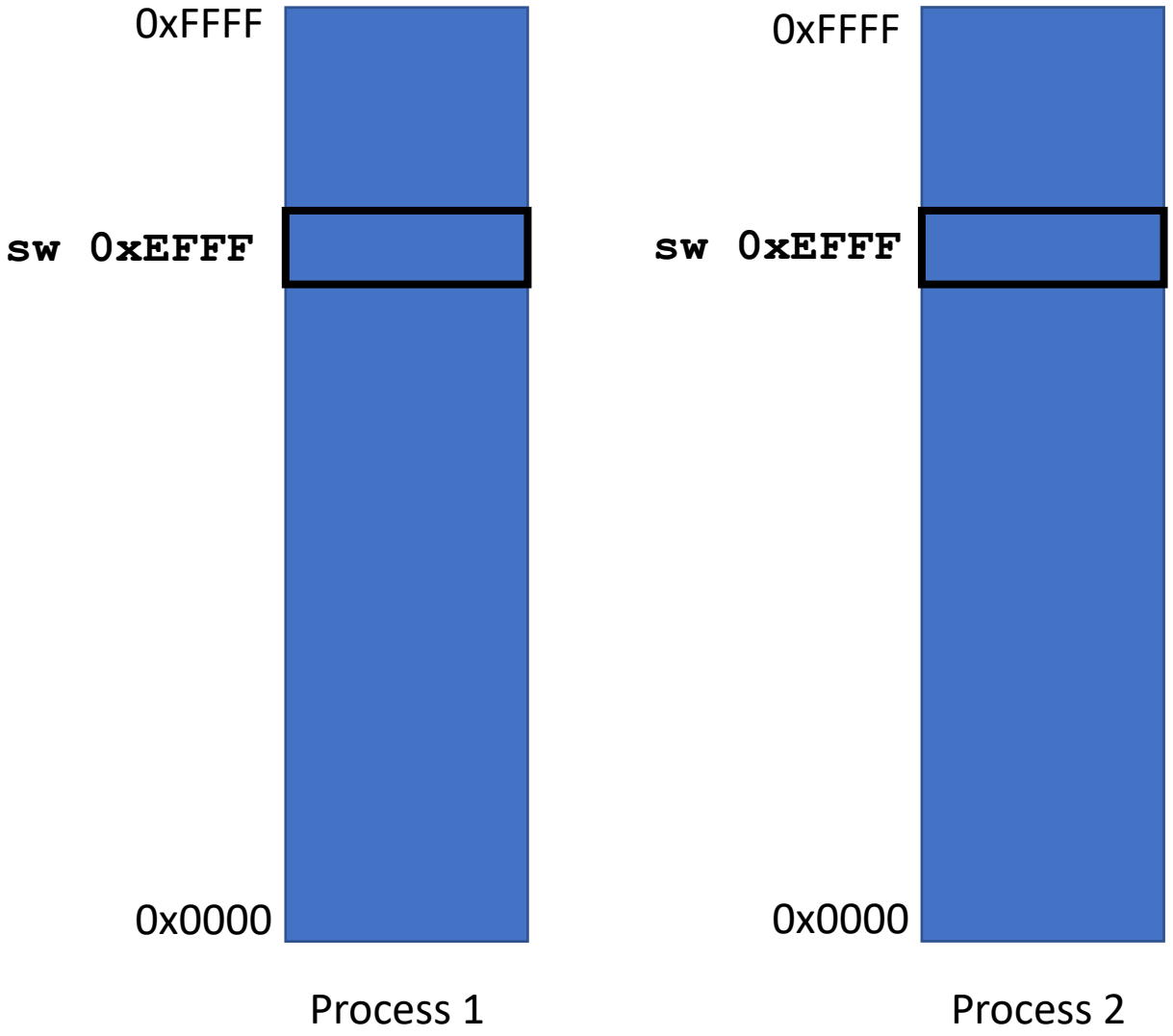
Virtual Memory – Abstraction of Uniformity



Virtual Memory – Abstraction of Isolation*



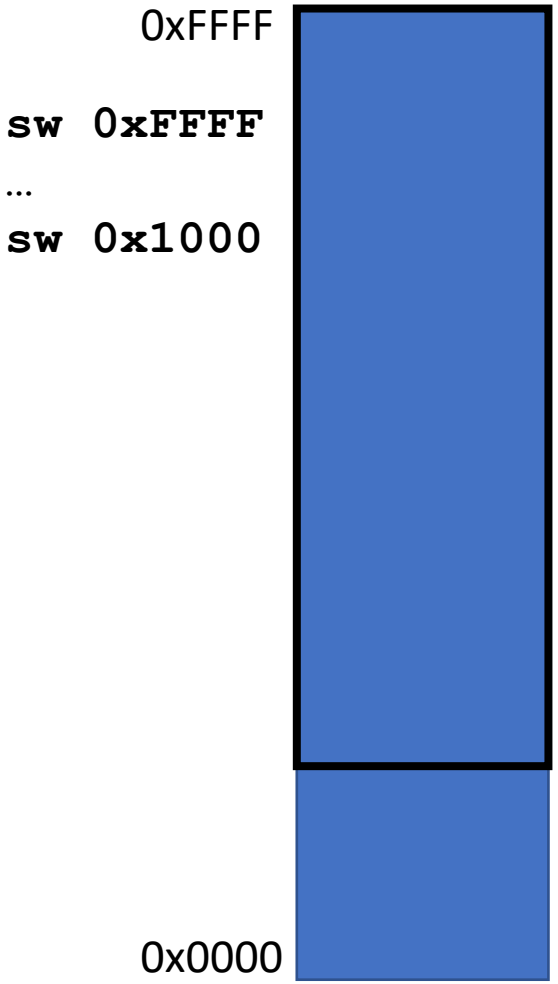
Virtual Memory – Thinking about mechanism



First obvious problem:

Two processes access same location
violates isolation abstraction

Virtual Memory – Thinking about Mechanism



Process 1

Process 2

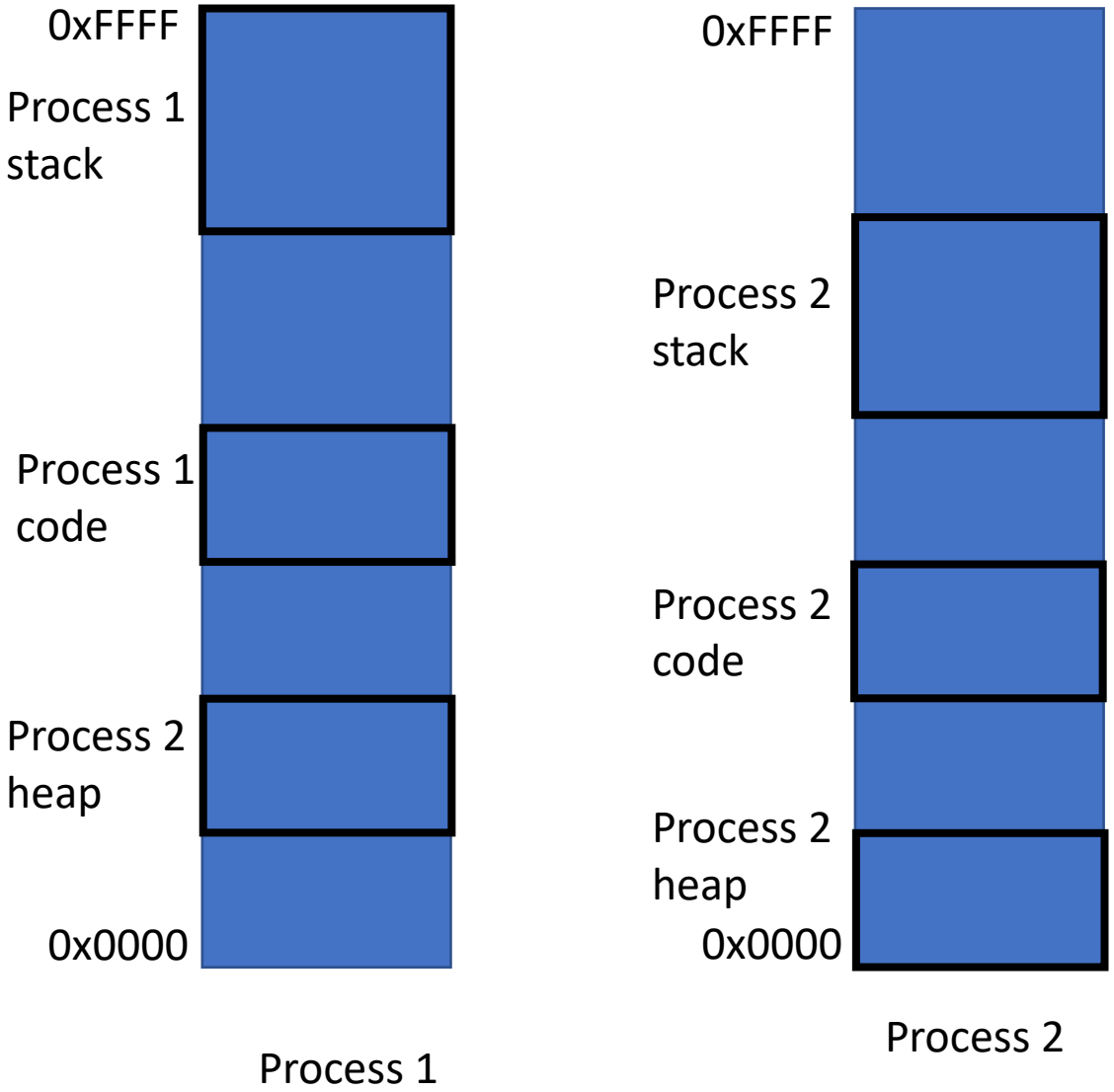
First obvious problem:

Two processes access same location
violates isolation abstraction

Second obvious problem:

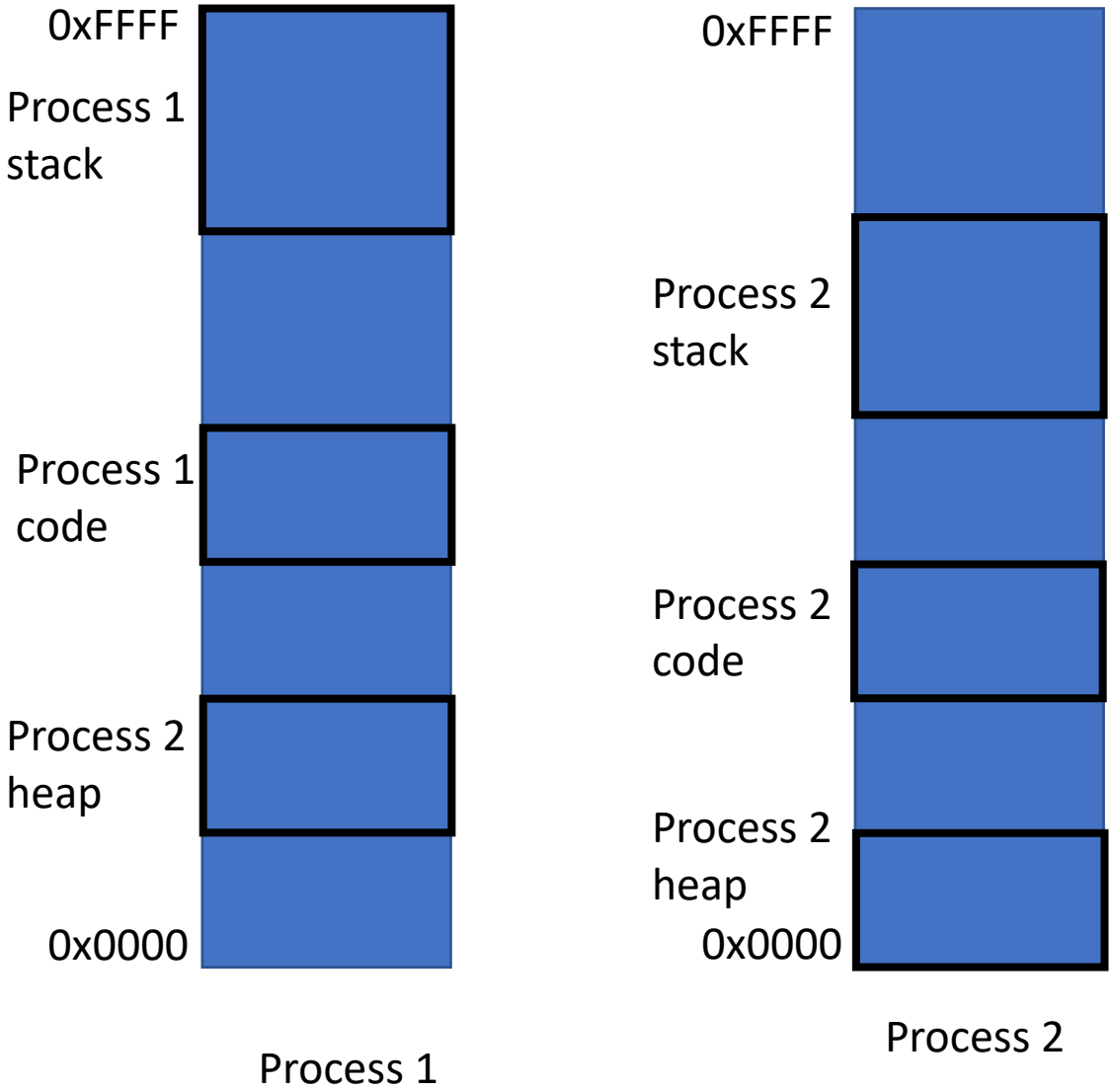
Two processes access #bytes > total memory size
violates abundance abstraction (and isolation)

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



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

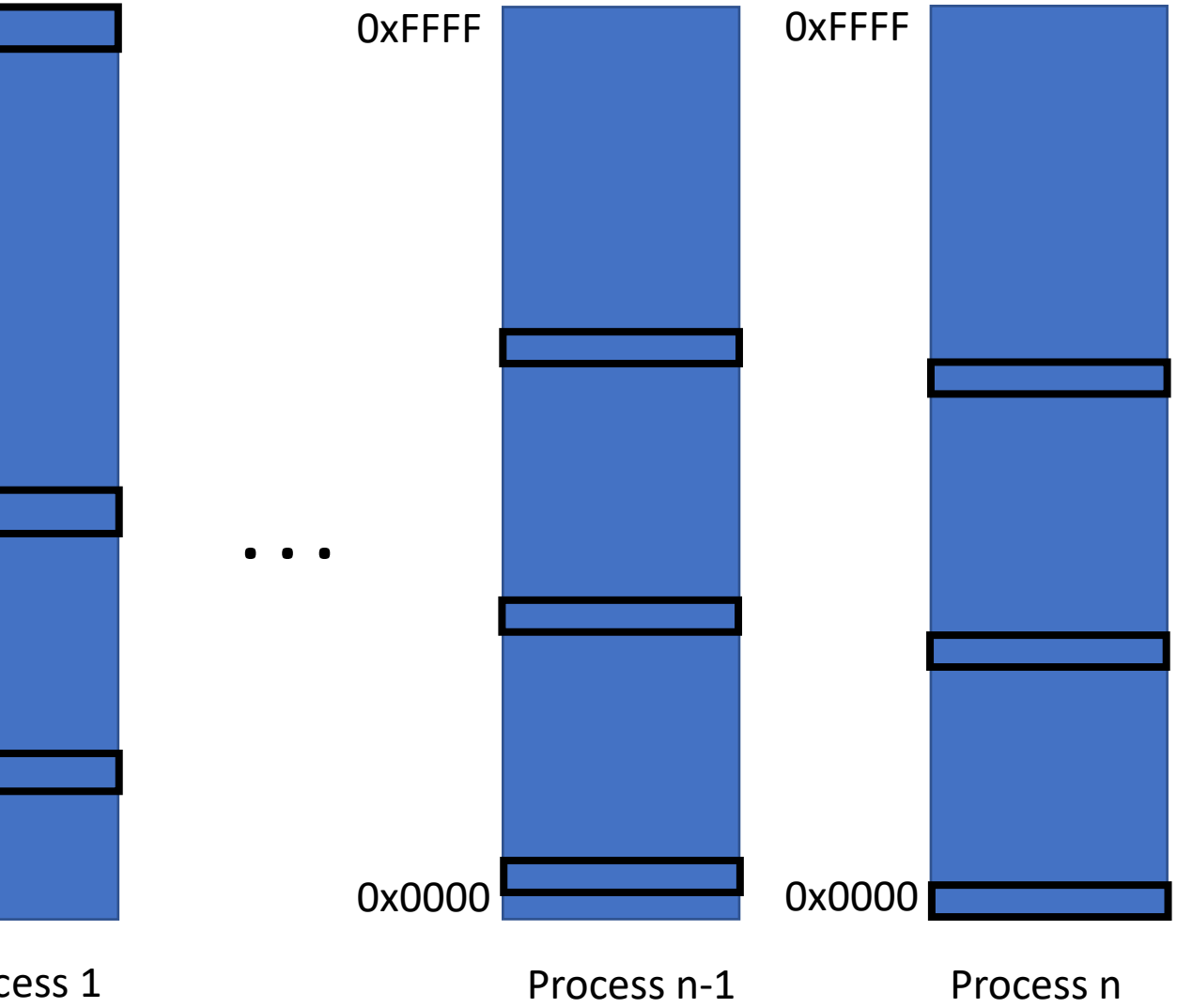
First Attempt: Static Partitioning



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



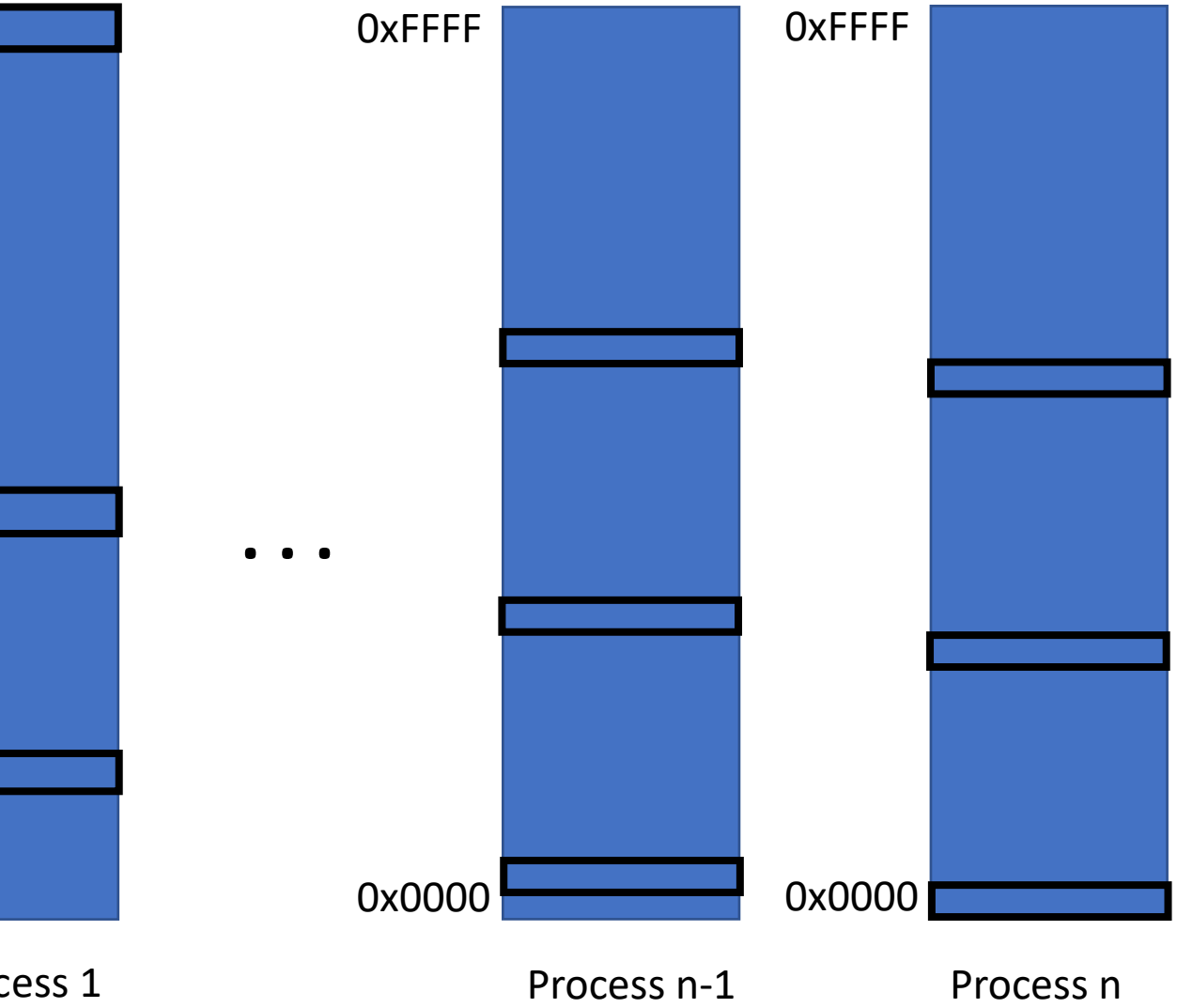
Statically partitioning the address space

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

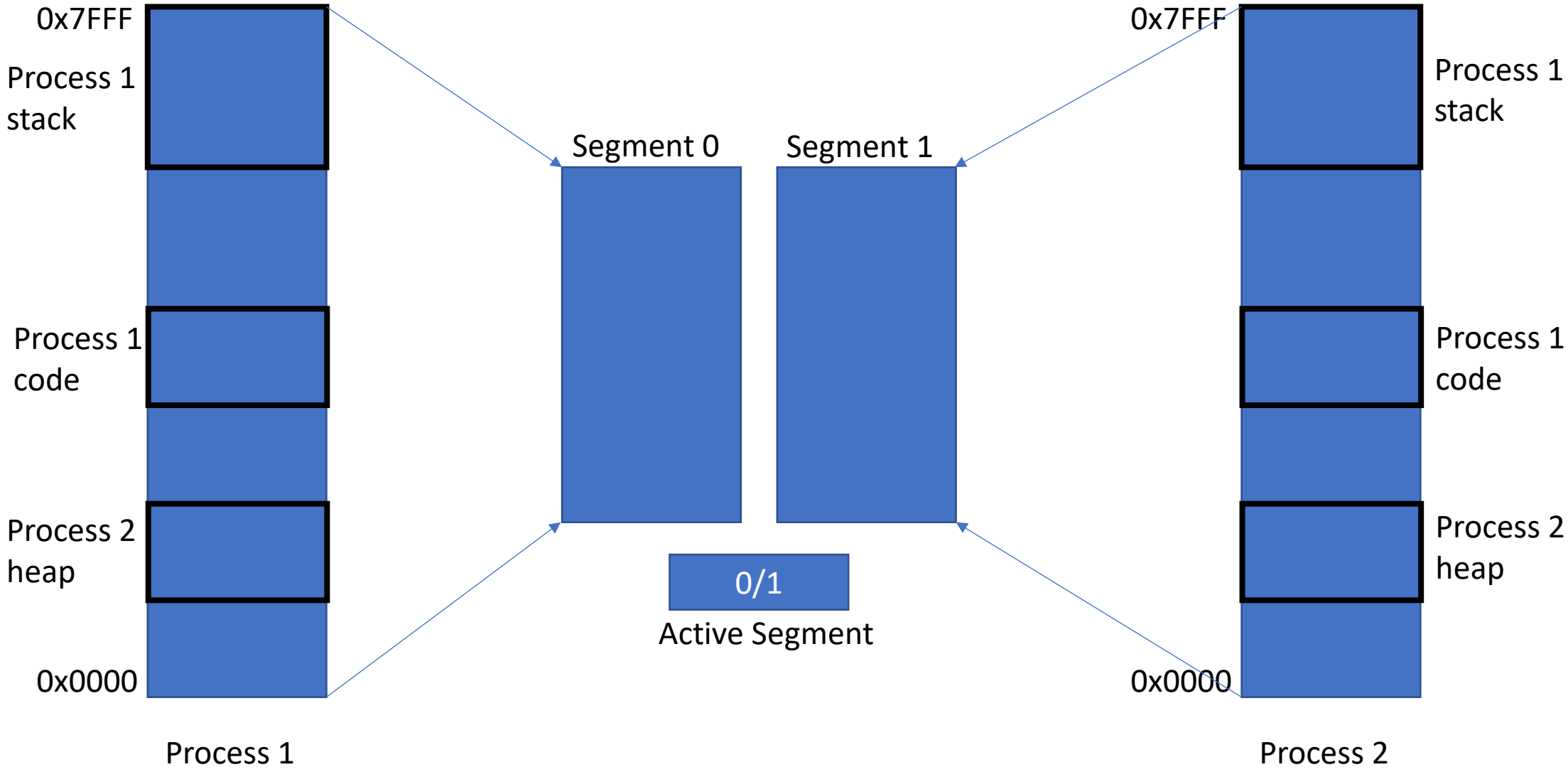
Need to be sure that neither process will go and mess around with the other process' address ranges (isolation)

Need to use increasingly tiny partitions per process (abundance)

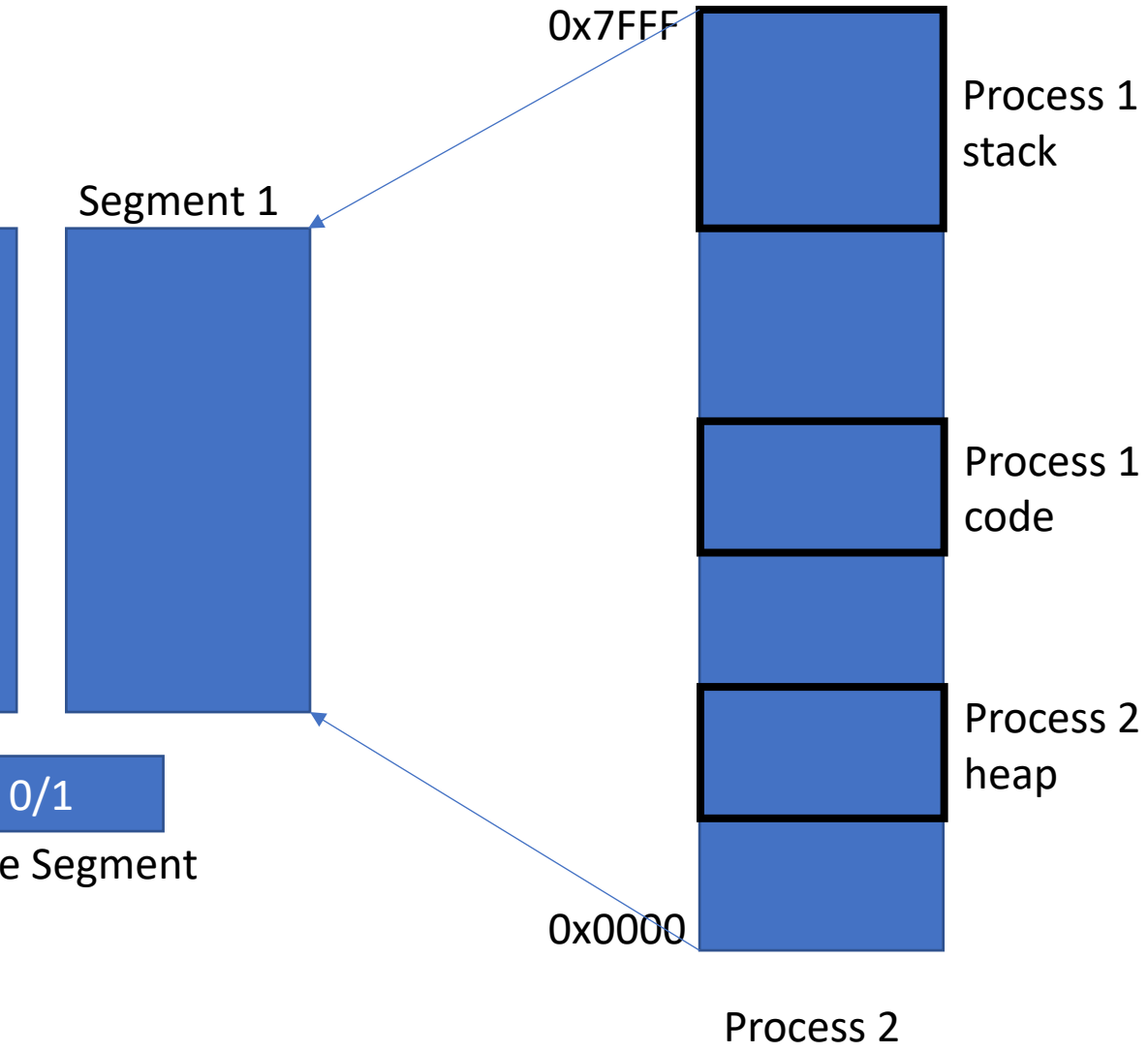
Need to know where your tiny partition starts so you can use it (uniformity).

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.**

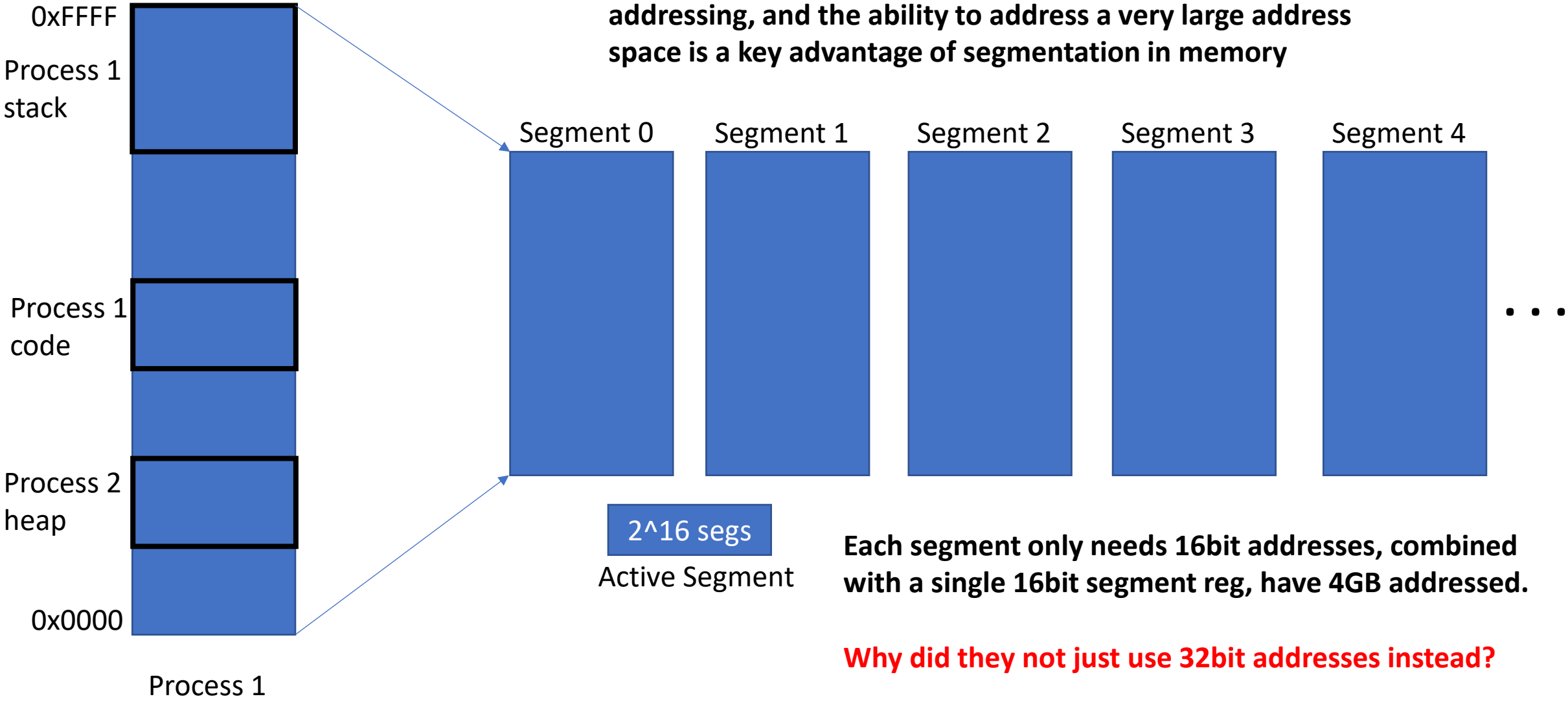
Benefit: Processes can choose a segment and use predictable addresses off of that segment. **Abstraction of uniformity.**

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

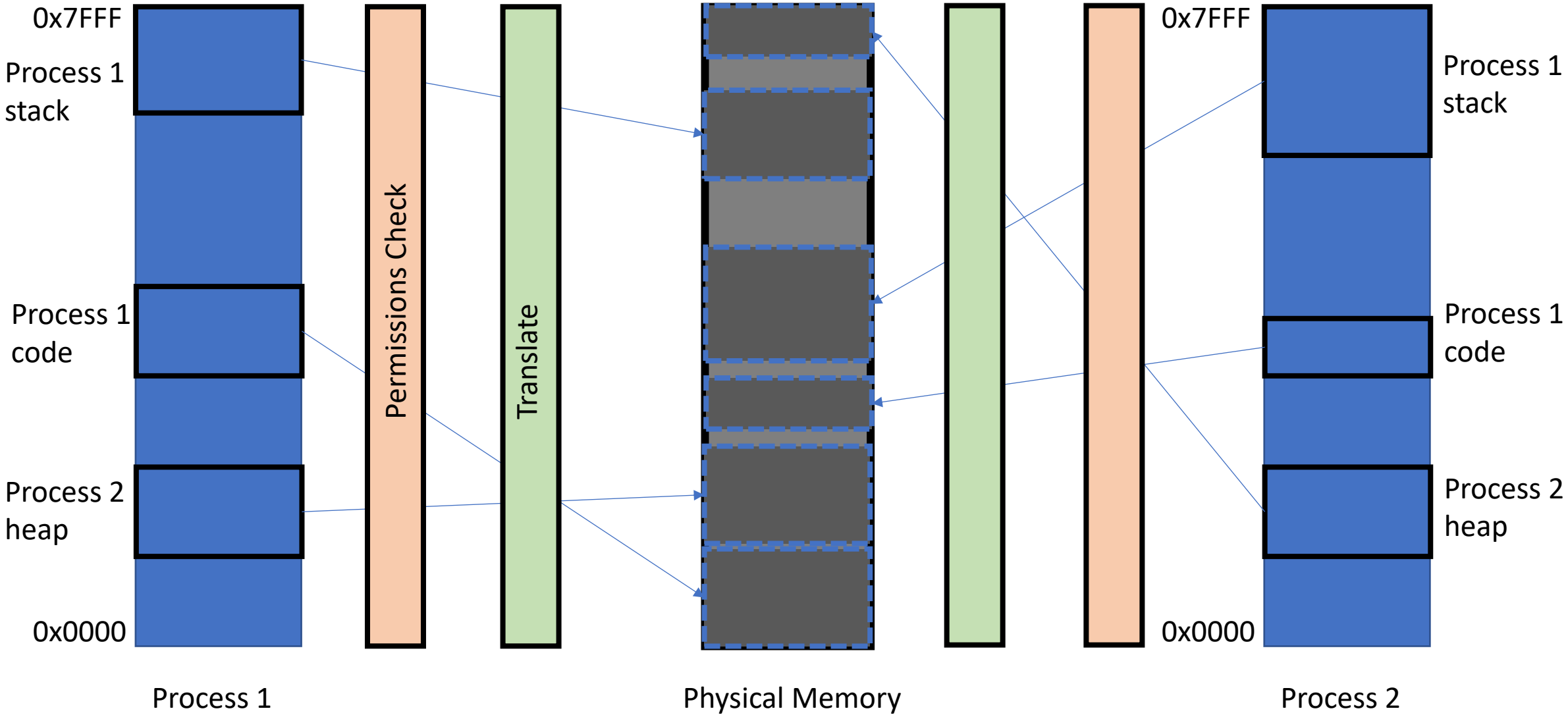


Abstraction of abundance, uniformity in segment relative addressing, and the ability to address a very large address space is a key advantage of segmentation in memory

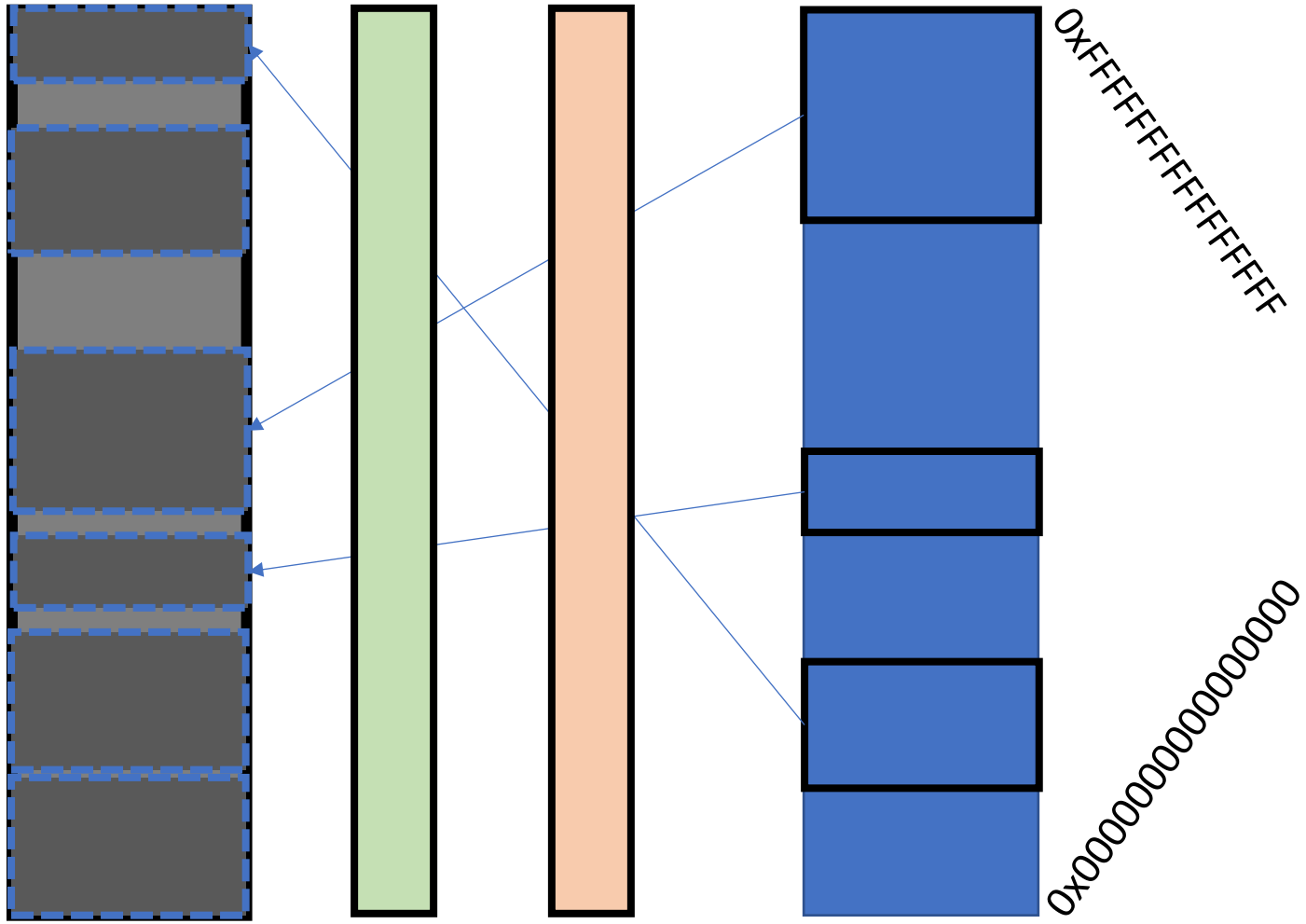
Each segment only needs 16bit addresses, combined with a single 16bit segment reg, have 4GB addressed.

Why did they not just use 32bit addresses instead?

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



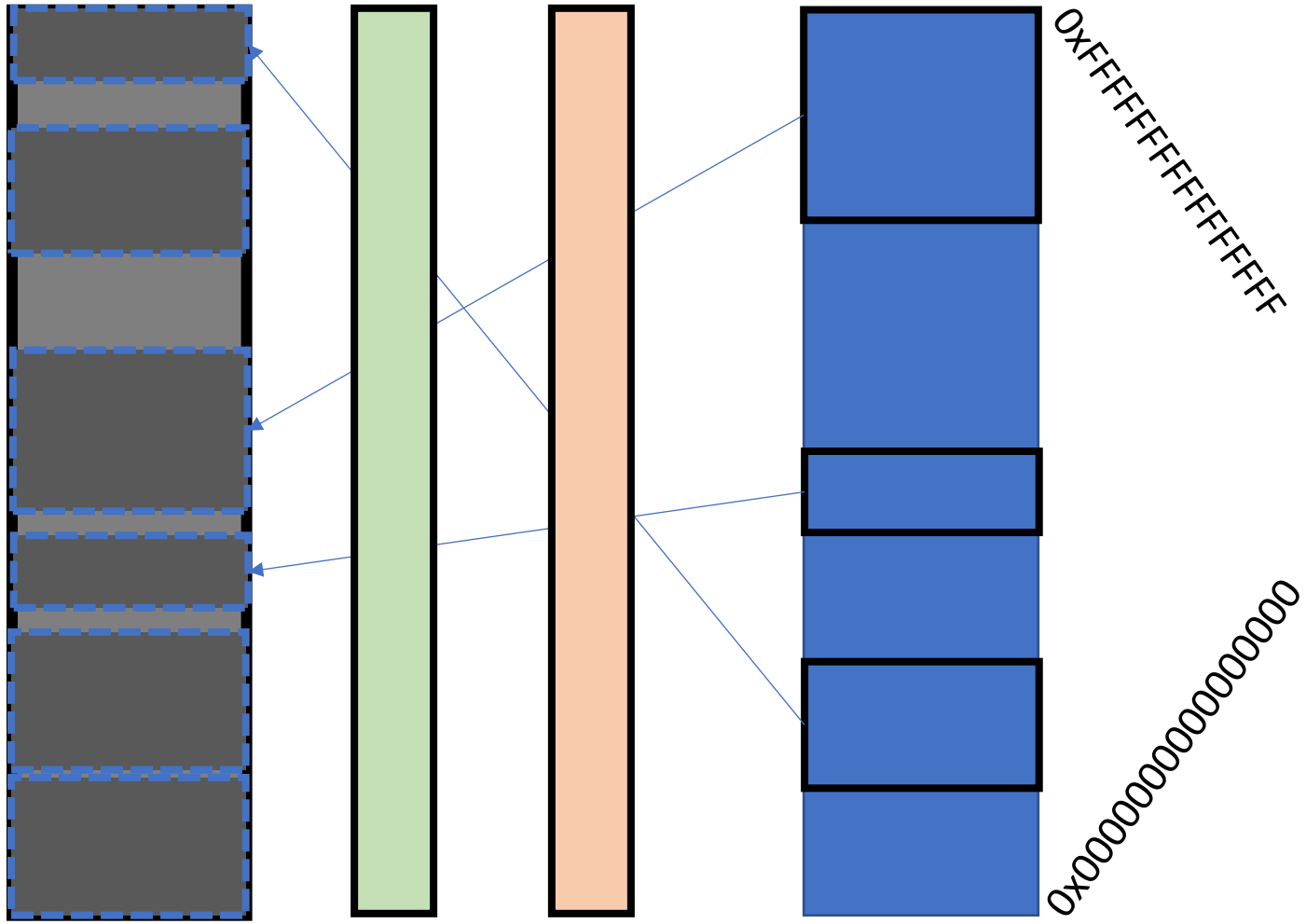
Virtual Memory: Software Dynamic Address Translation and Permission Checking



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.**

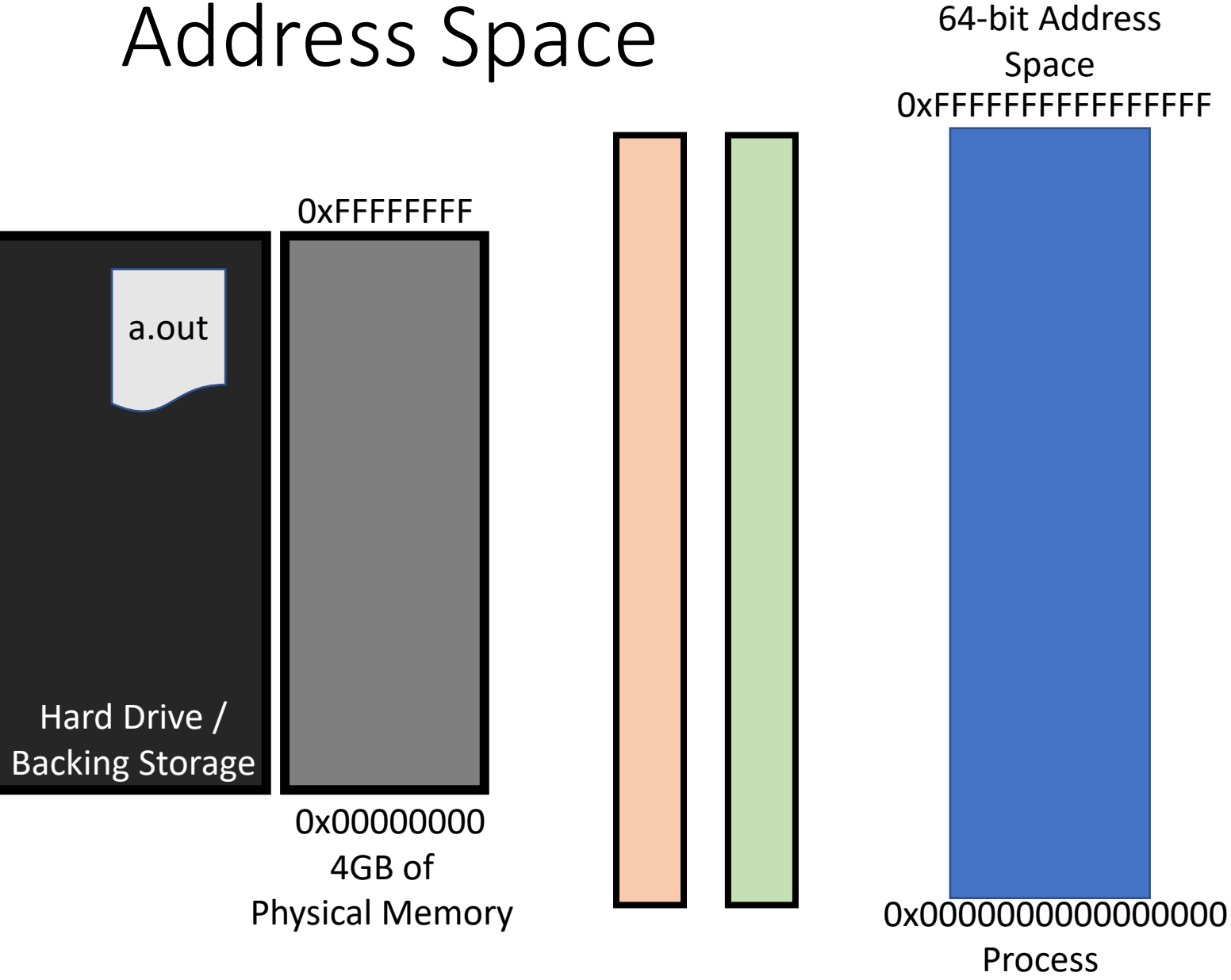
Caveat: need mechanism for mapping data in

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



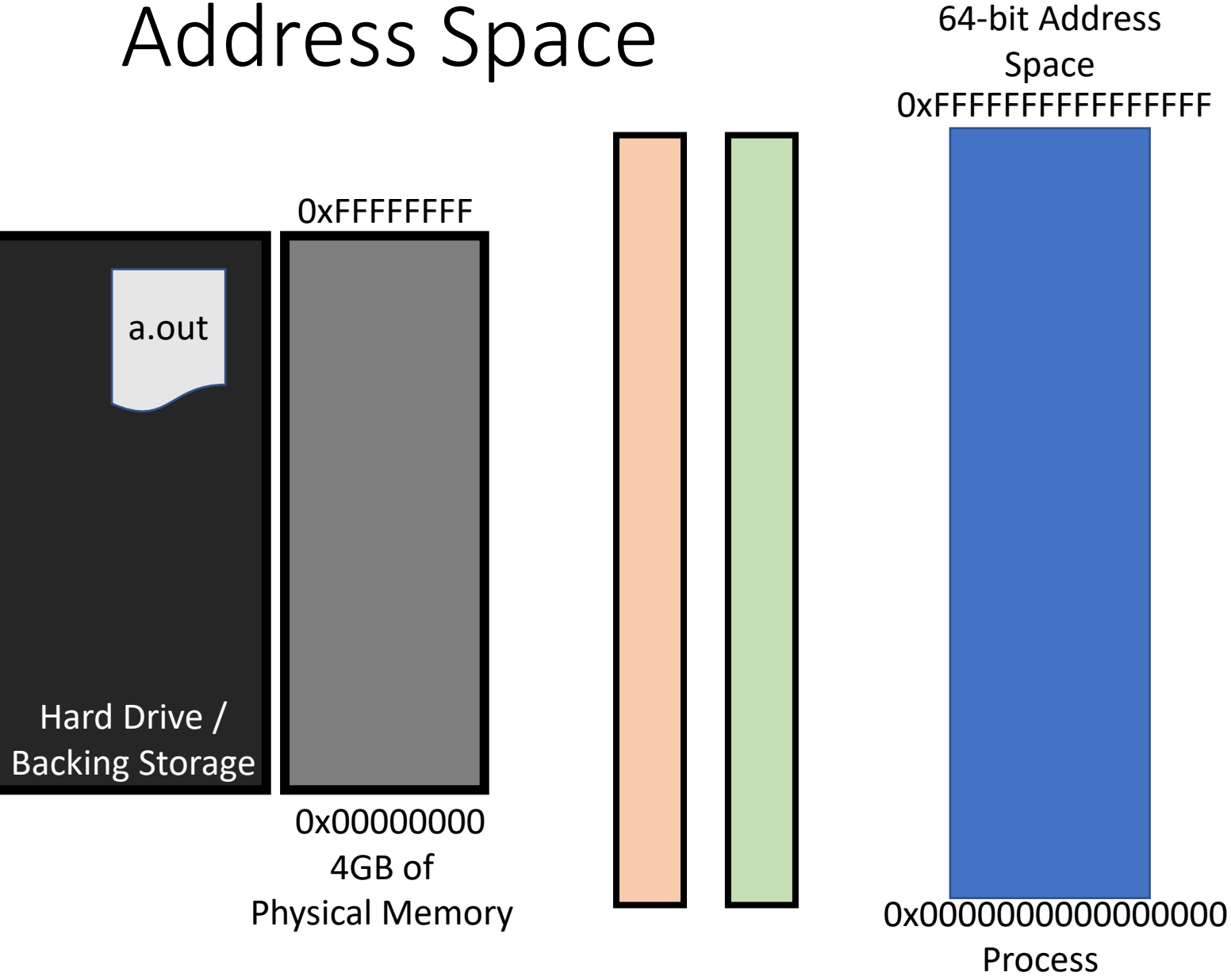
Define a new operation ***map(data, size, mode)***

Data: ID of file on disk to map into address space (or "anonymous" for blank memory)
Size: How many bytes in the address space to map
Mode: Readable, writeable, executable

Semantics of map: Mapped addresses in virtual address space *become accessible* and if file-backed, correspond to file's data

What does it mean to "become accessible"?

Virtual Memory: Mapping Data into Virtual Address Space



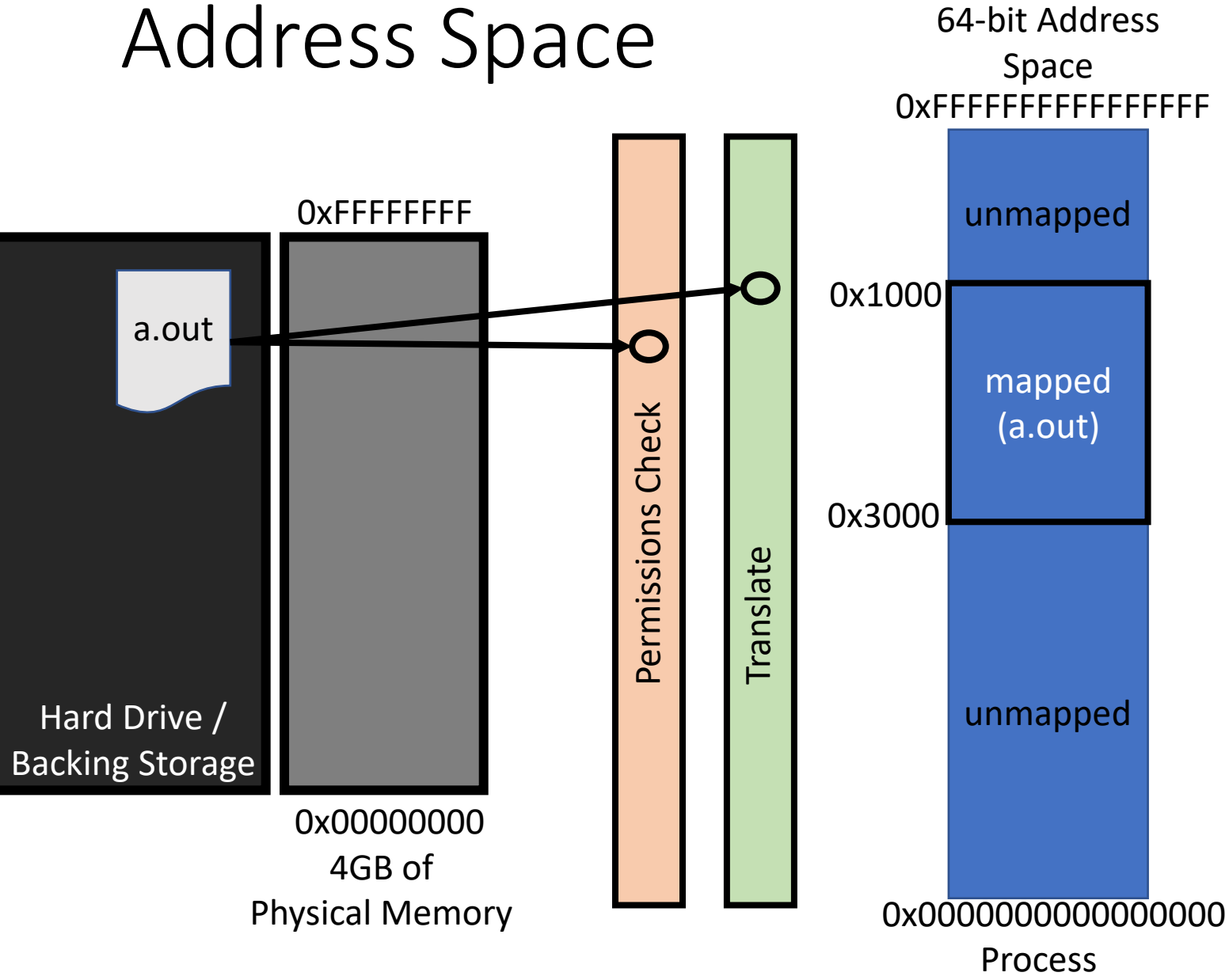
Define a new operation
map(addr, data, size, mode)

Data: file descriptor to map into address space (or "anonymous" for just memory)
Size: How many bytes to map
Mode: Readable, writeable, executable

Semantics of map: Mapped addresses in virtual address space *become accessible* and if file-backed, correspond to file's data

What does it mean to "become accessible"? executing memory access to address in range is no longer illegal; corresponds to data, either file or anonymous buffer

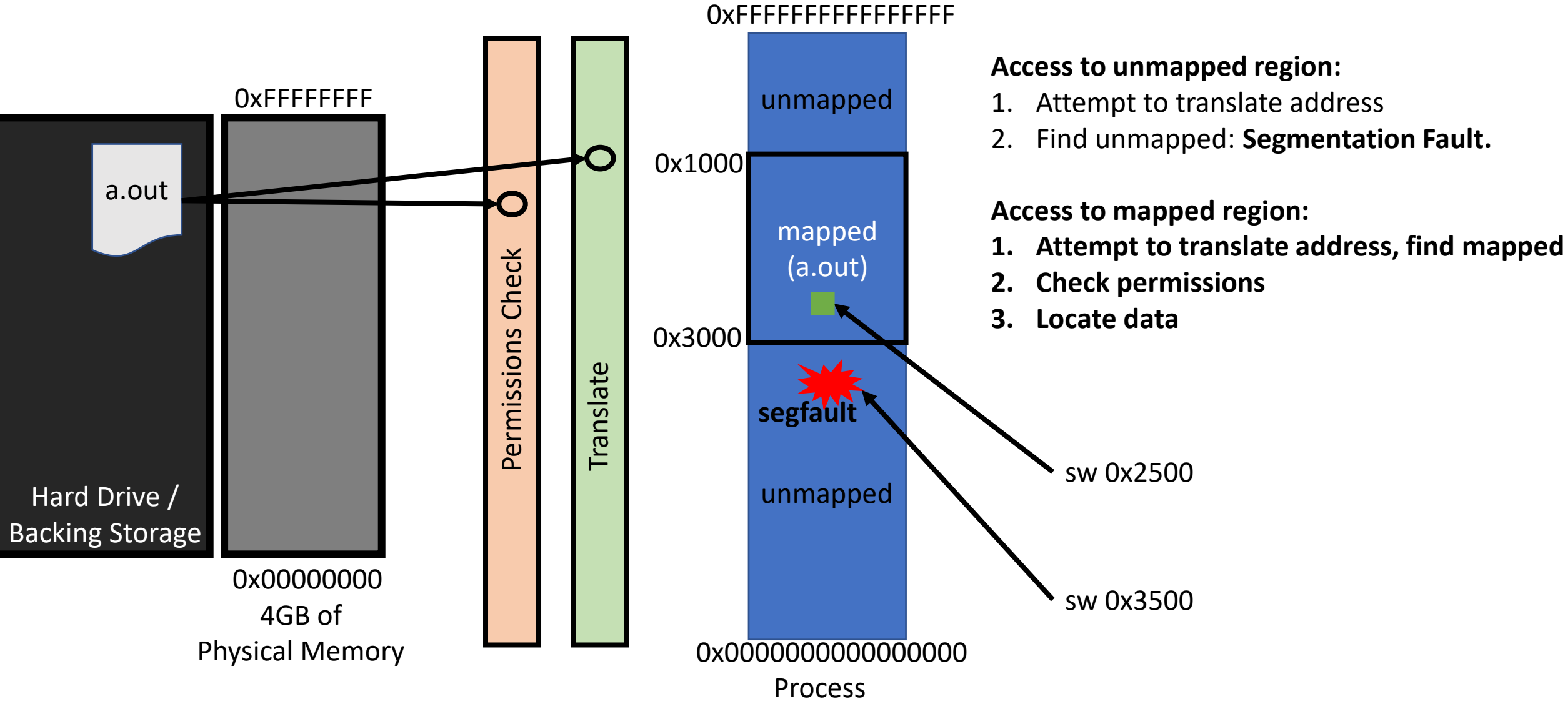
Virtual Memory: Mapping Data into Virtual Address Space



int fd; open(/foo/bar/baz/a.out);
Map(0x1000, fd, 7500 Bytes, RWX)

1. Update translation function
2. Update permissions entry
3. Reserve virtual address range
4. **DO NOT** move data anywhere

Virtual Memory: Accessing Data Mapped into the Virtual Address Space



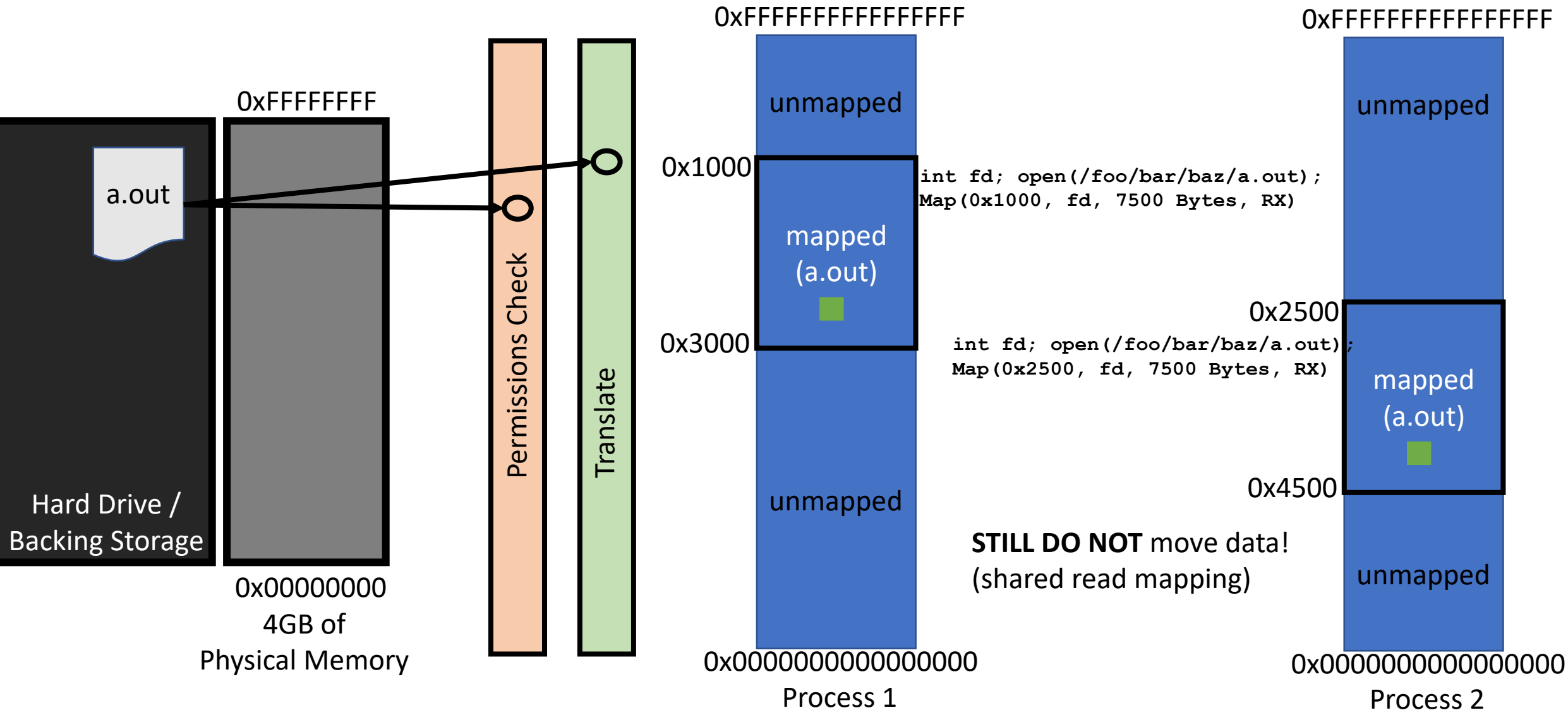
Access to unmapped region:

1. Attempt to translate address
2. Find unmapped: **Segmentation Fault.**

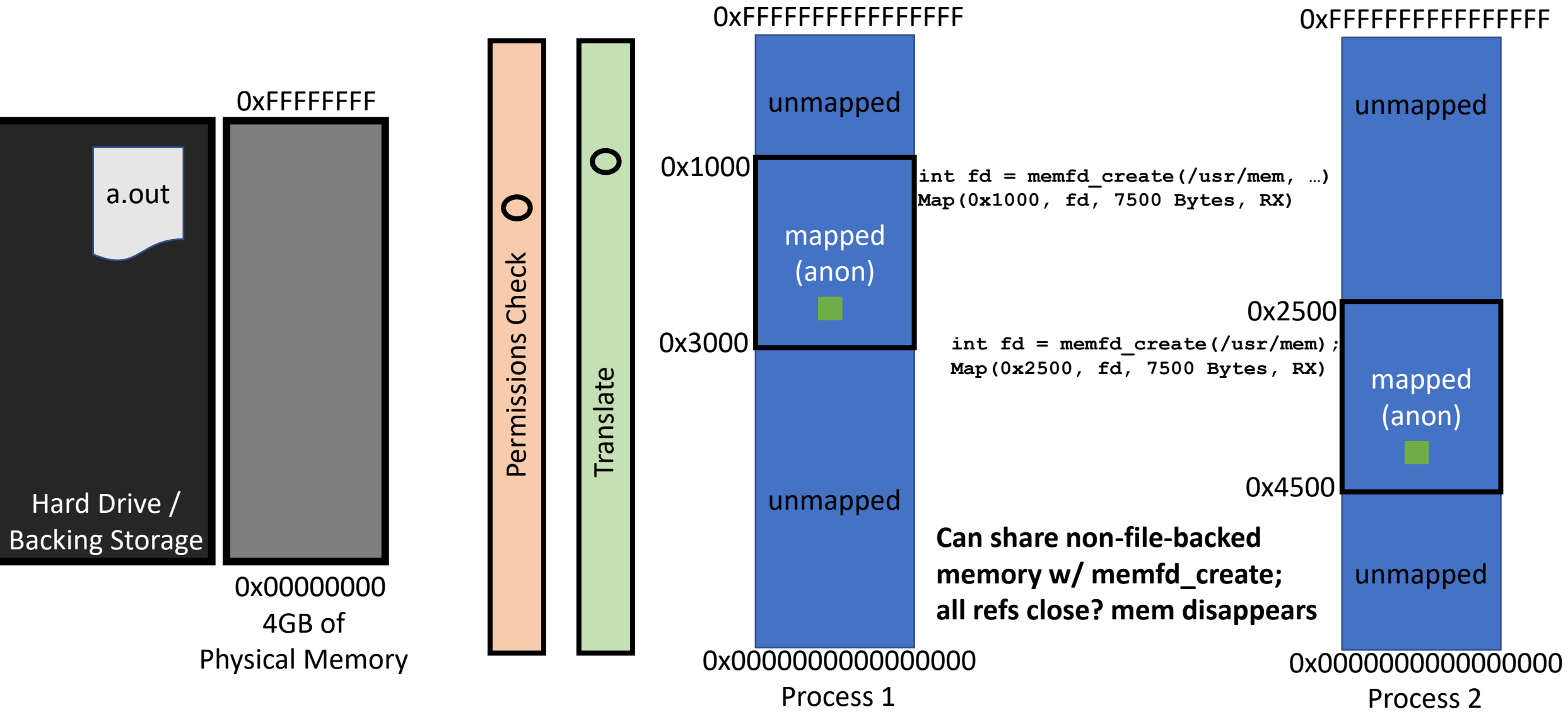
Access to mapped region:

1. **Attempt to translate address, find mapped**
2. **Check permissions**
3. **Locate data**

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

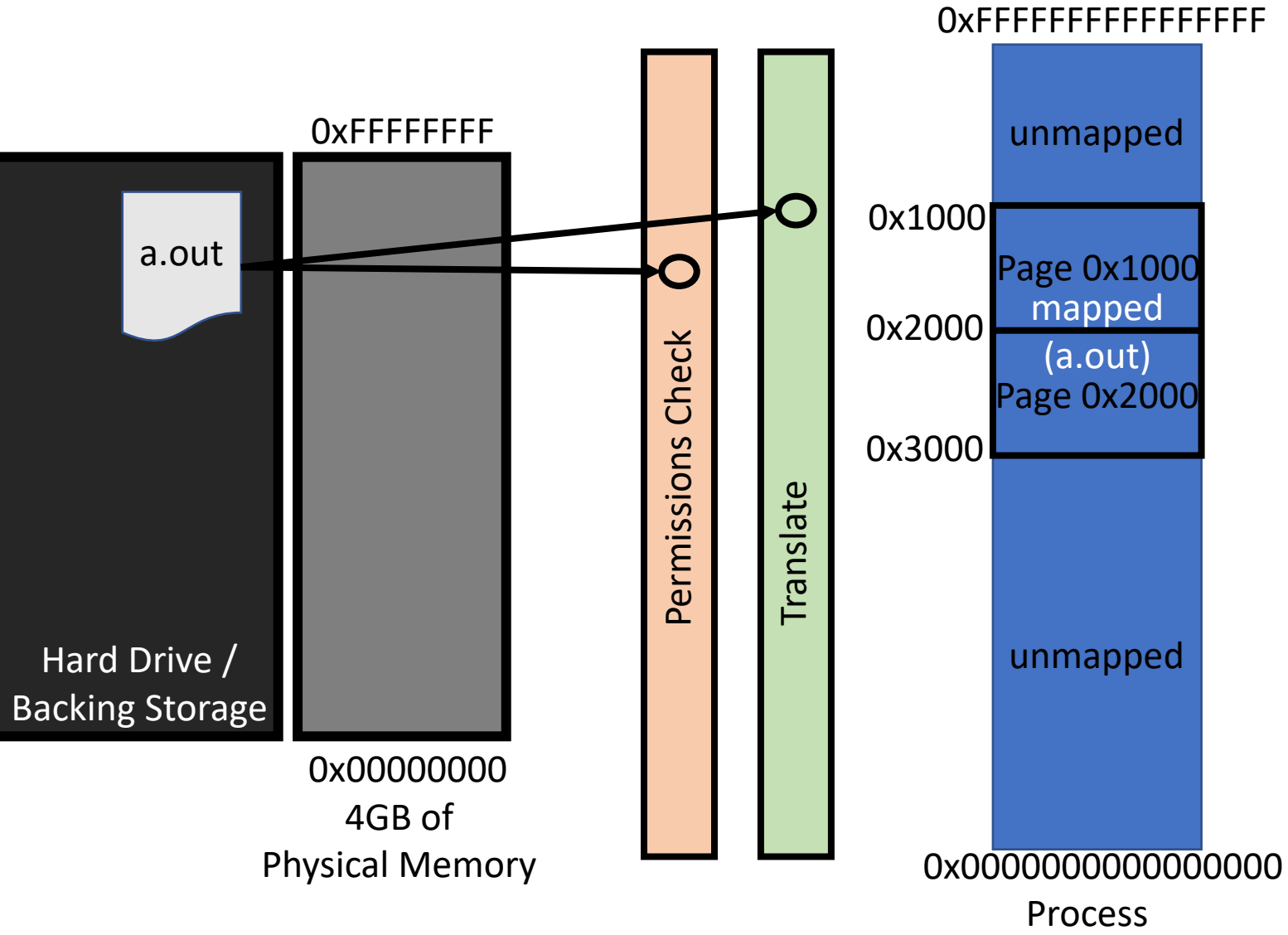


Virtual Memory: Shared Mapping of **Anonymous** Data into Address Space by Multiple Processes



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)

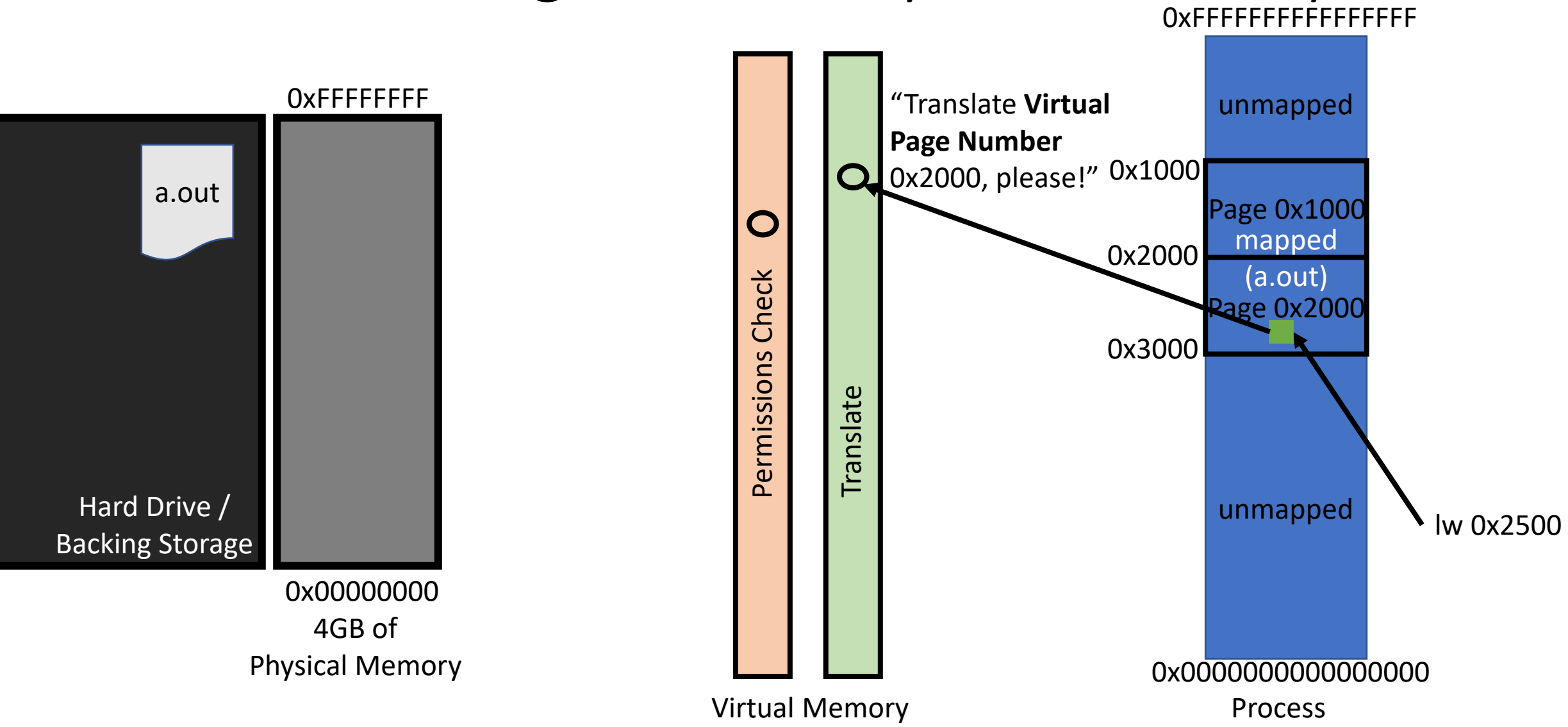
Translation happens at *page* granularity

Pages are usually 2^12 = 4096 bytes

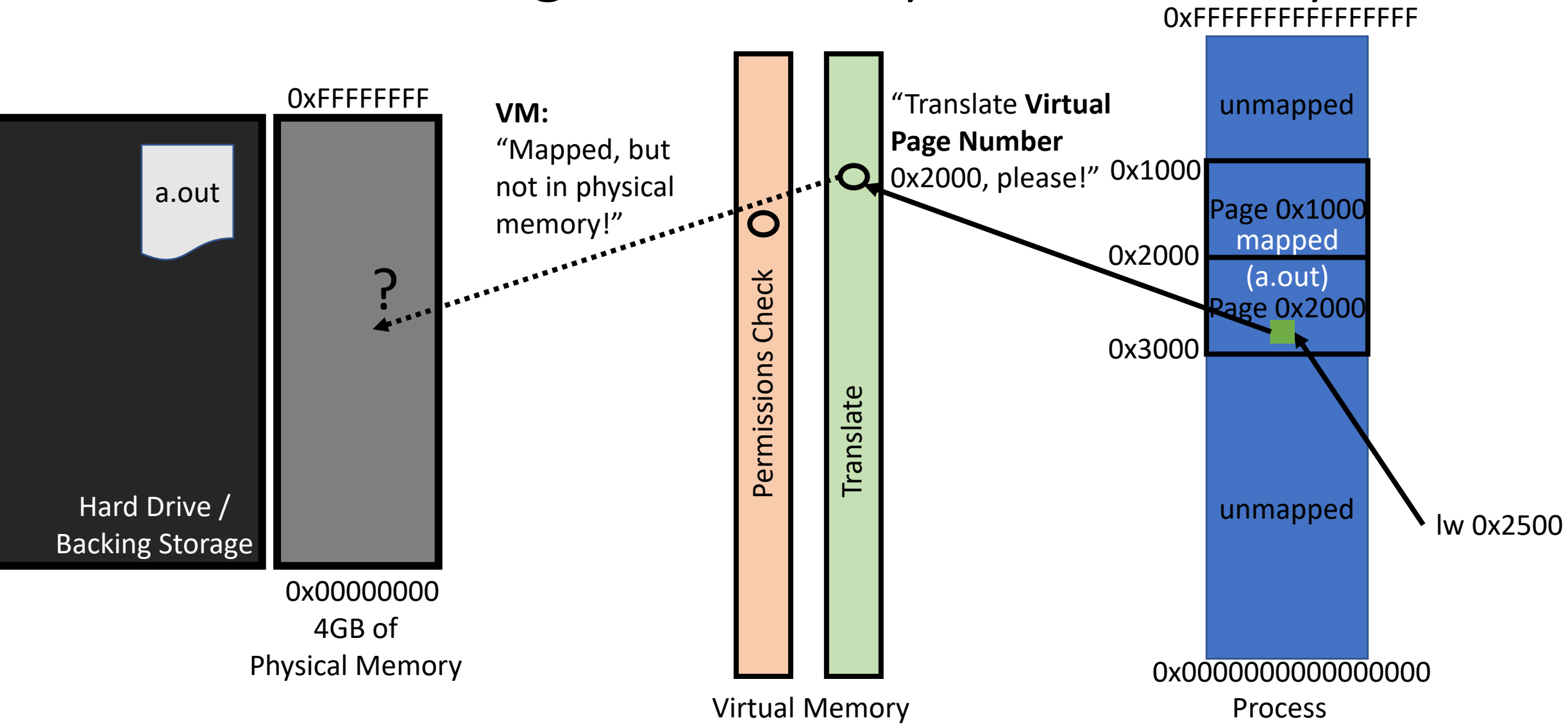
Memory access:

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

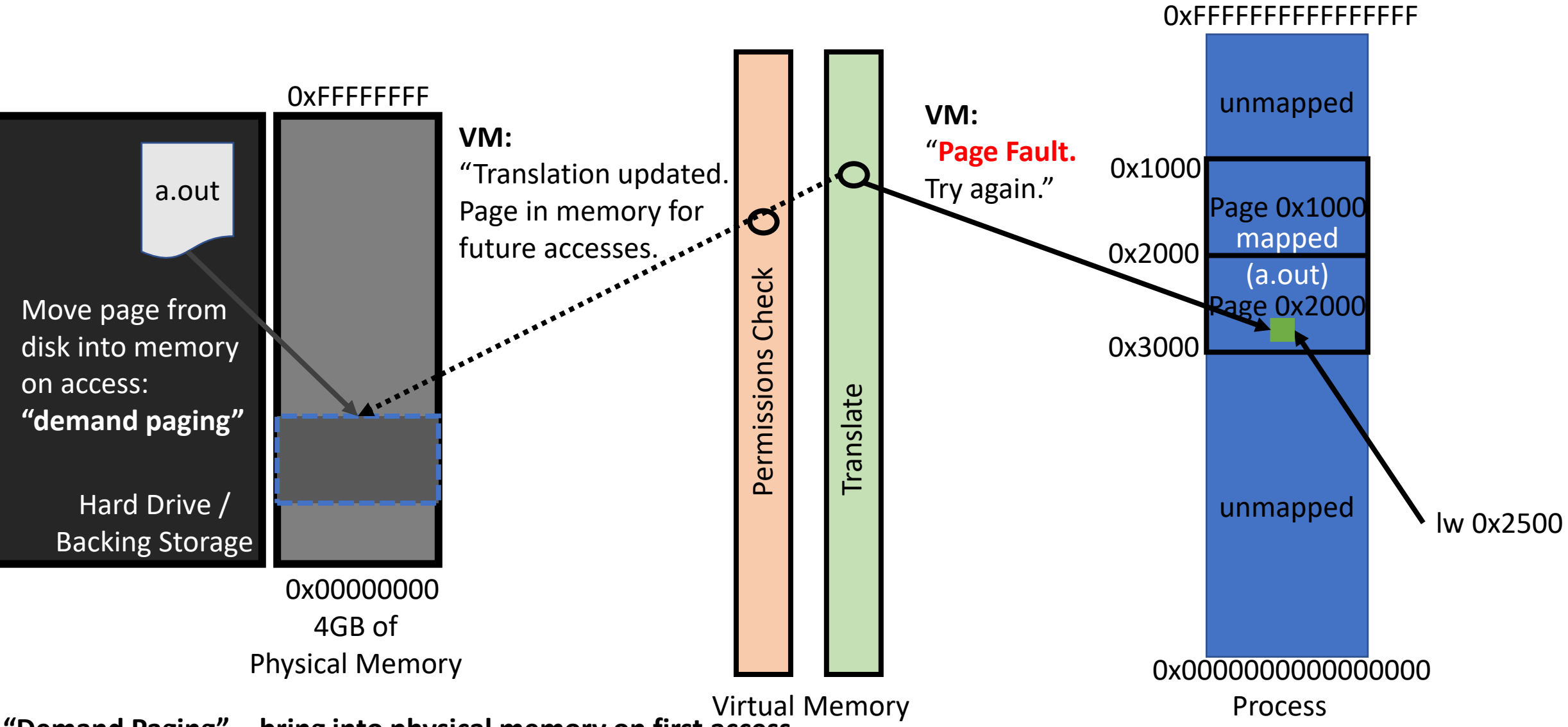


Virtual Memory: Translation and Finding Data Outcome #1: Page Not in Physical Memory



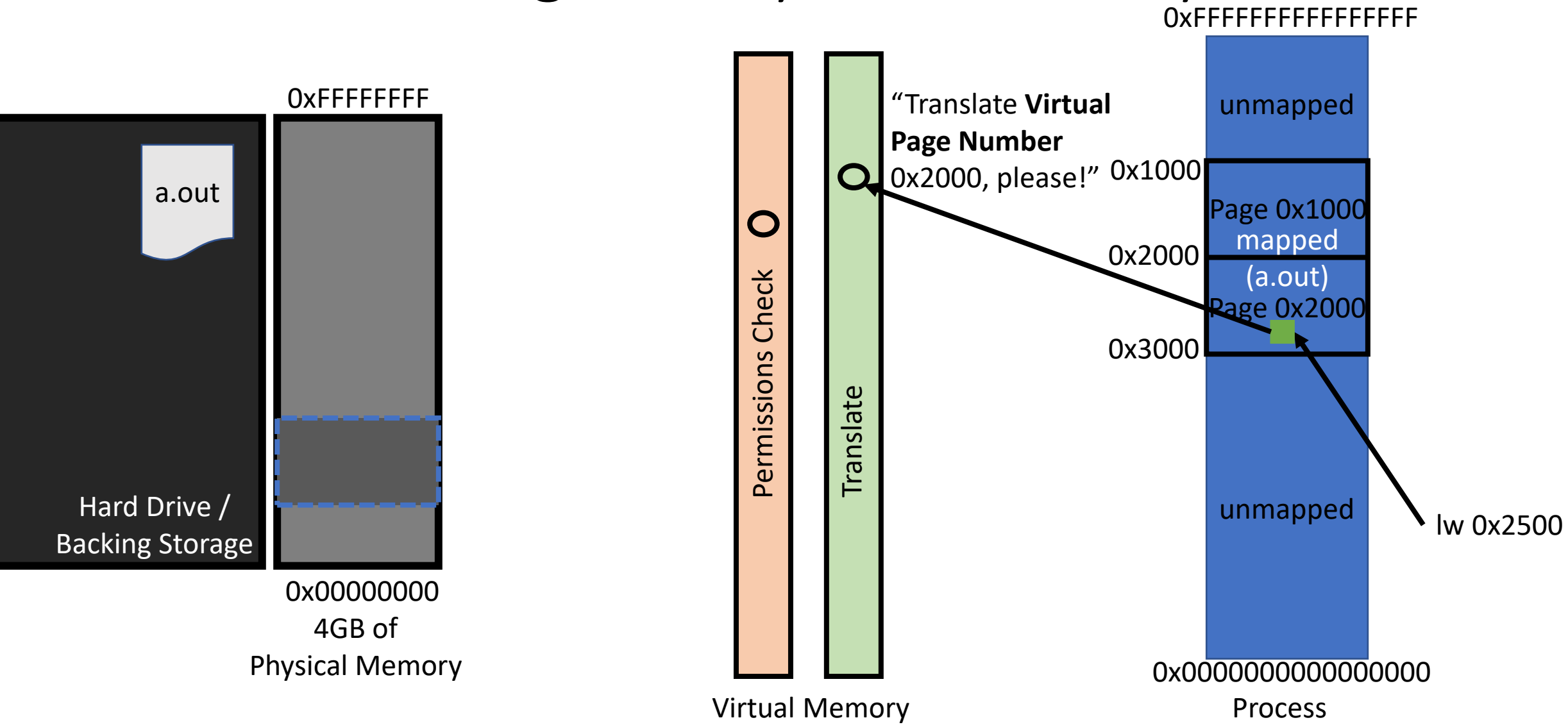
Page Fault: Basic Definition

Address exists in translation function but is not in physical memory

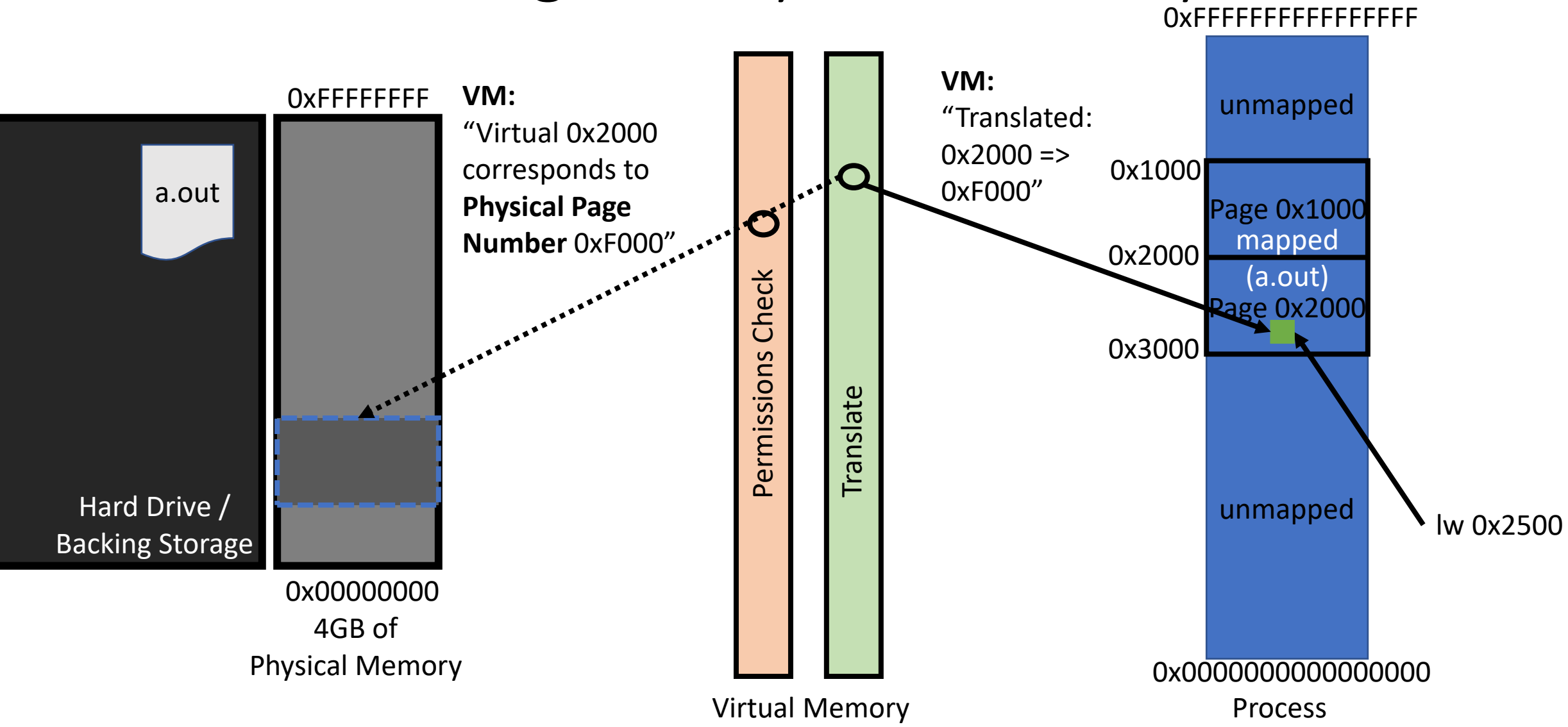


"Demand Paging" – bring into physical memory on first access

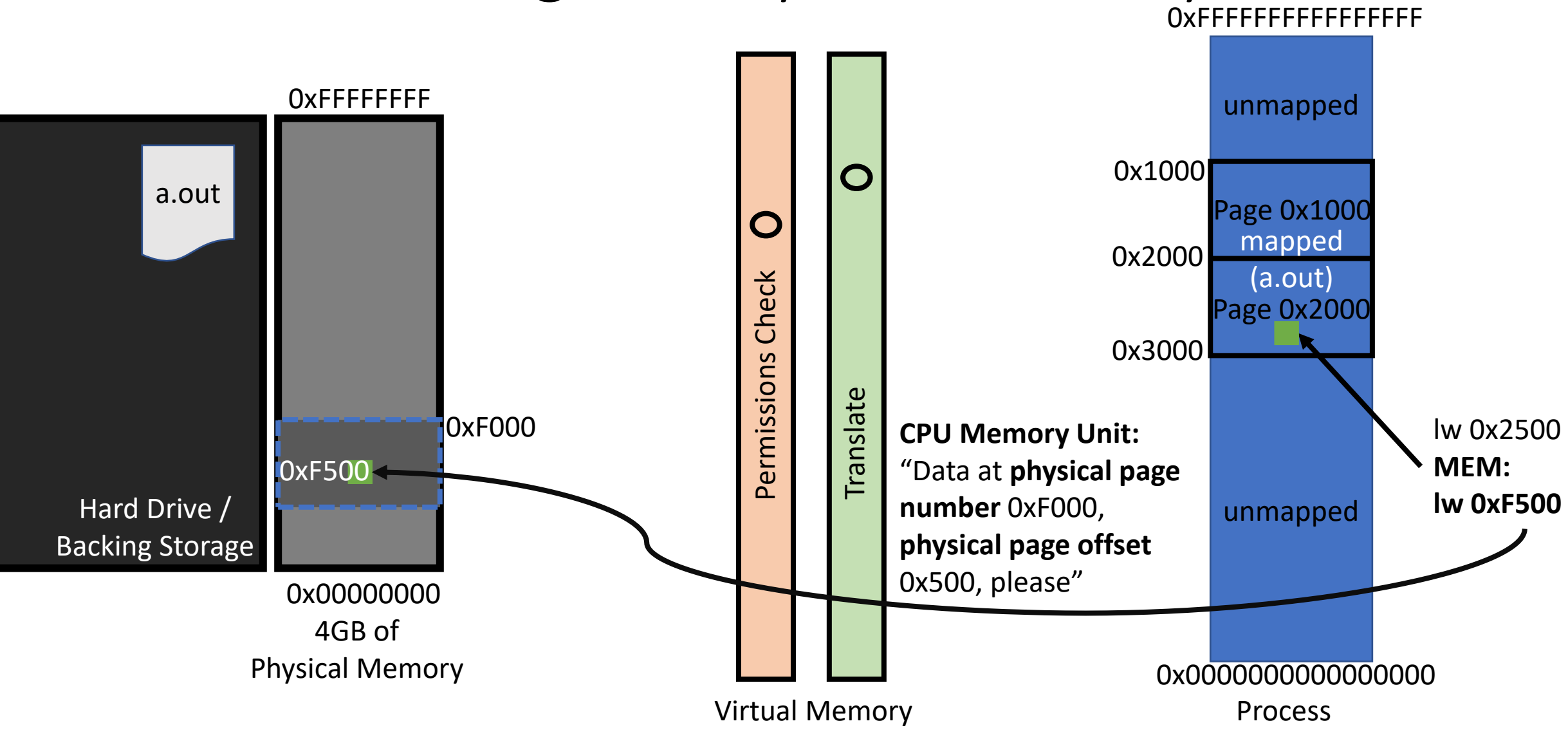
Virtual Memory: Translation and Finding Data Outcome #2: Page in Physical Memory



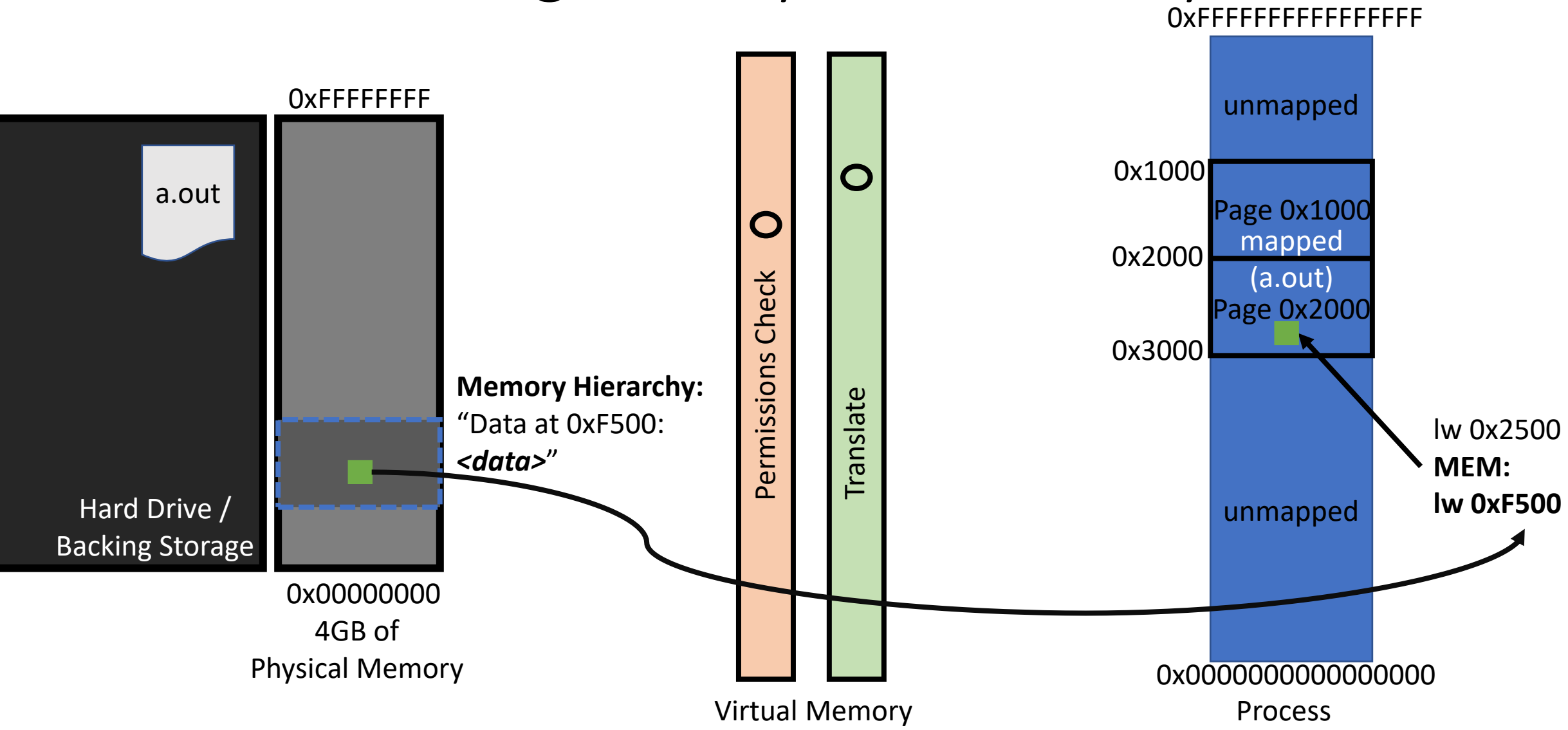
Virtual Memory: Translation and Finding Data Outcome #2: Page in Physical Memory



Virtual Memory: Translation and Finding Data Outcome #2: Page in Physical Memory

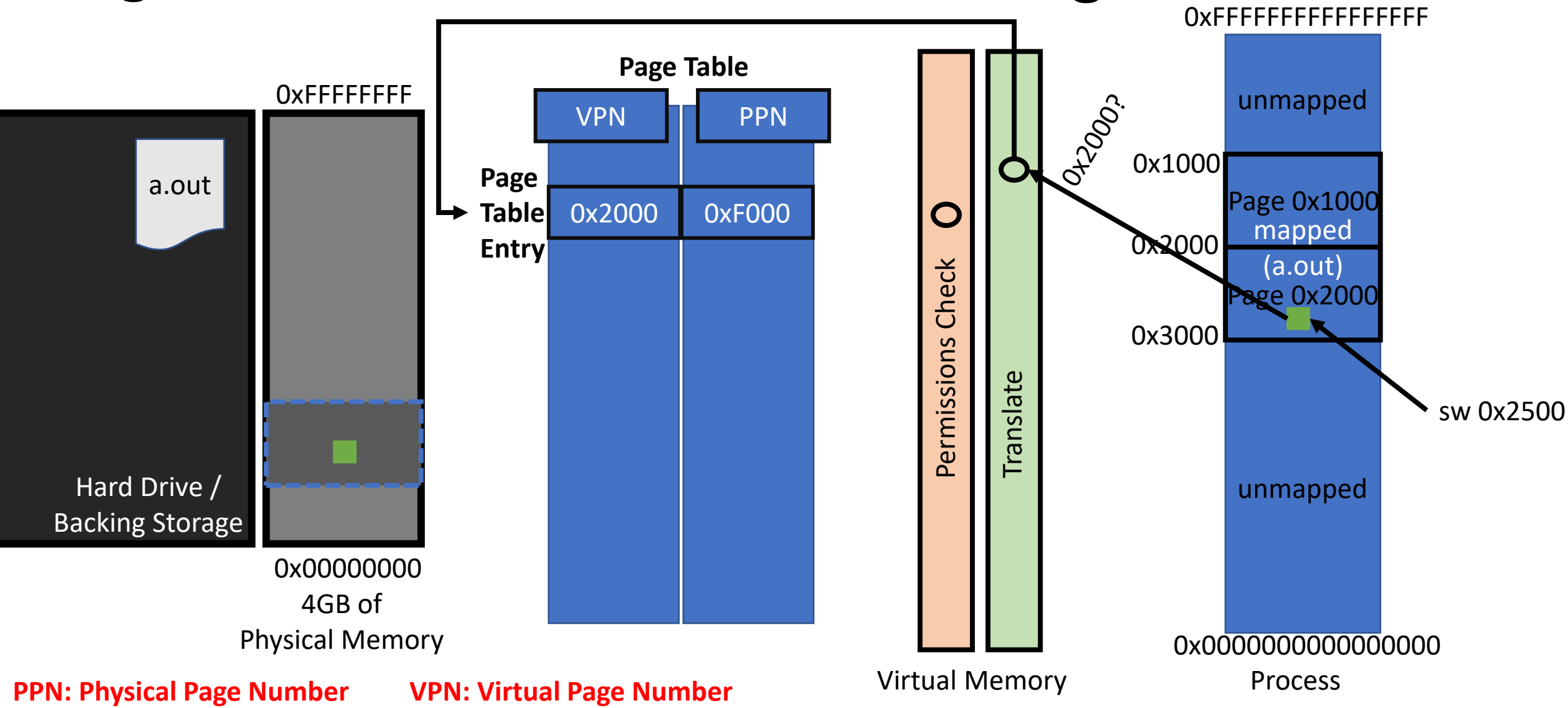


Virtual Memory: Translation and Finding Data Outcome #2: Page in Physical Memory



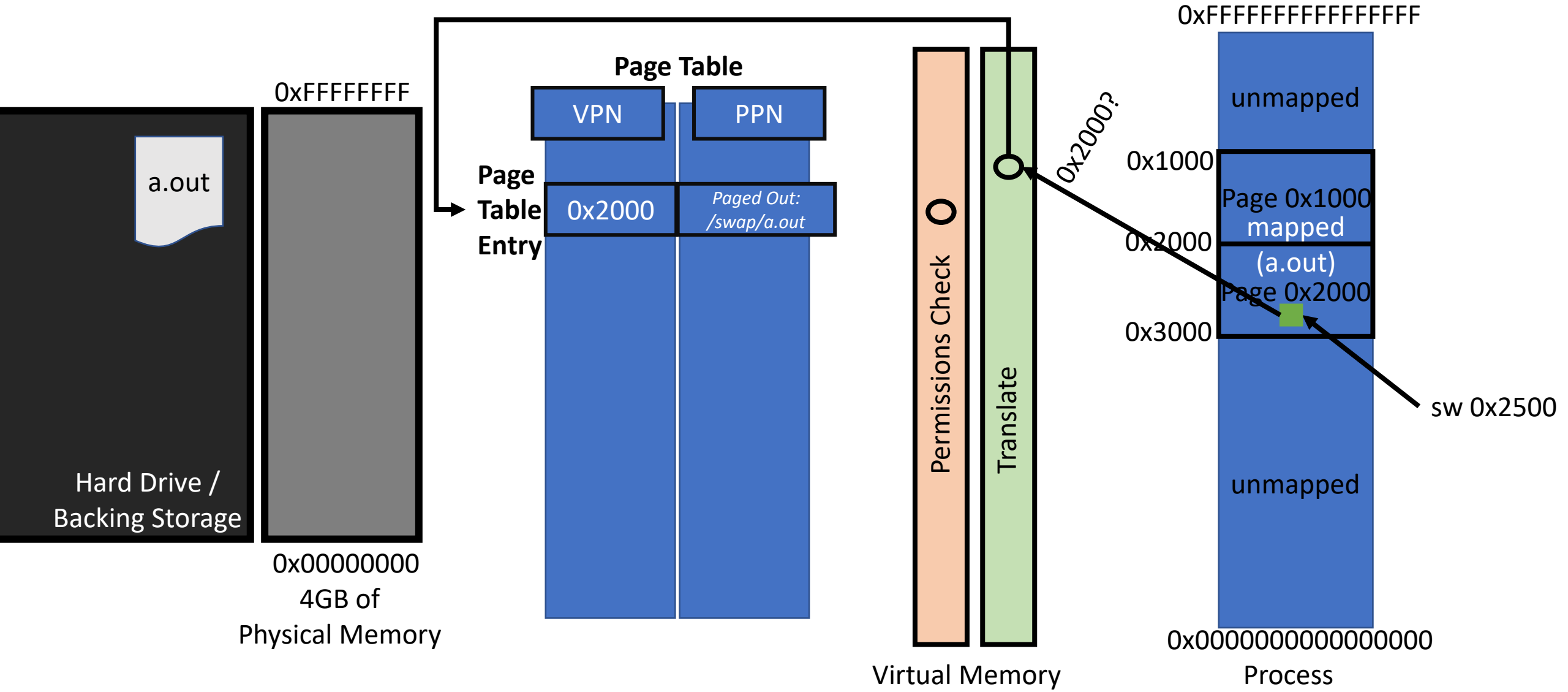
The Translation Function & Its Use

Virtual Memory: The Translation Function
Page Table Stores Translation for **Paged-In** Data

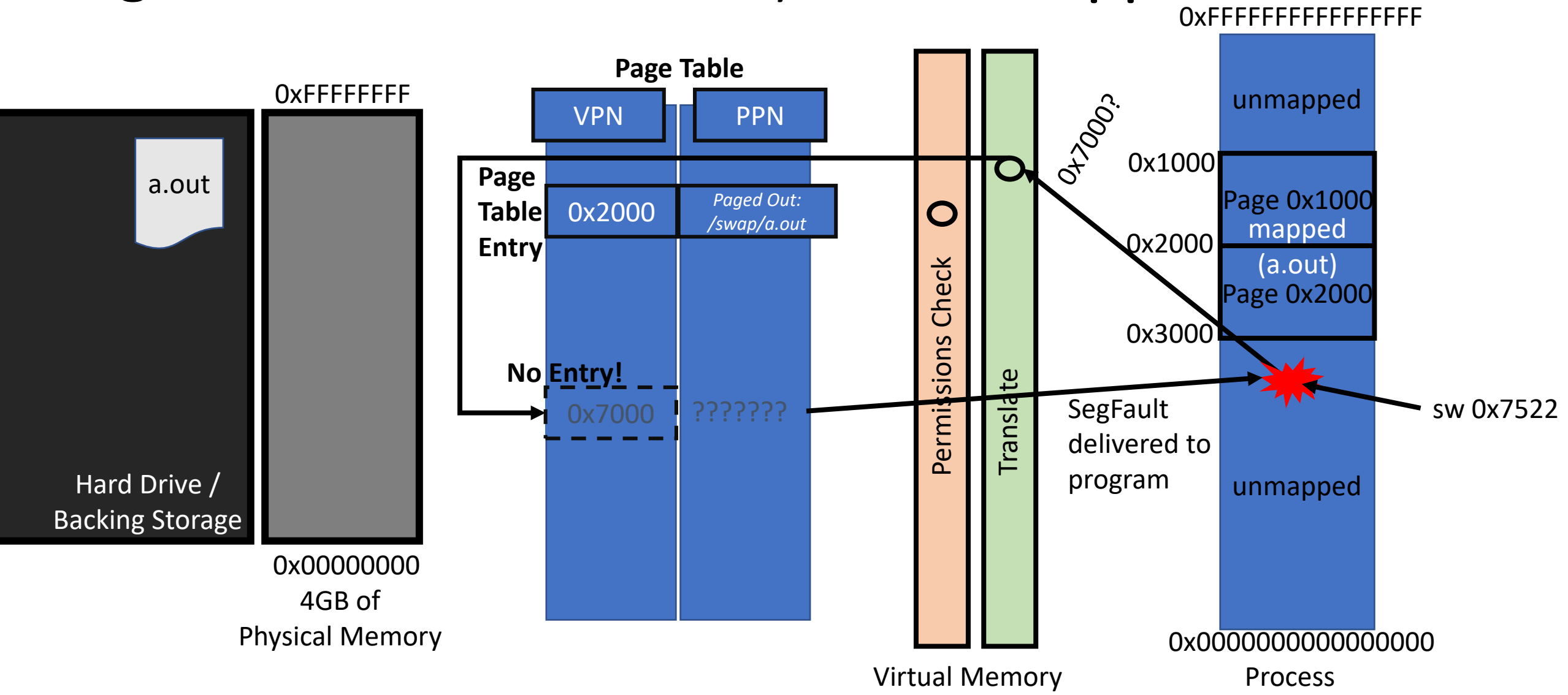


PPN: Physical Page Number **VPN: Virtual Page Number**

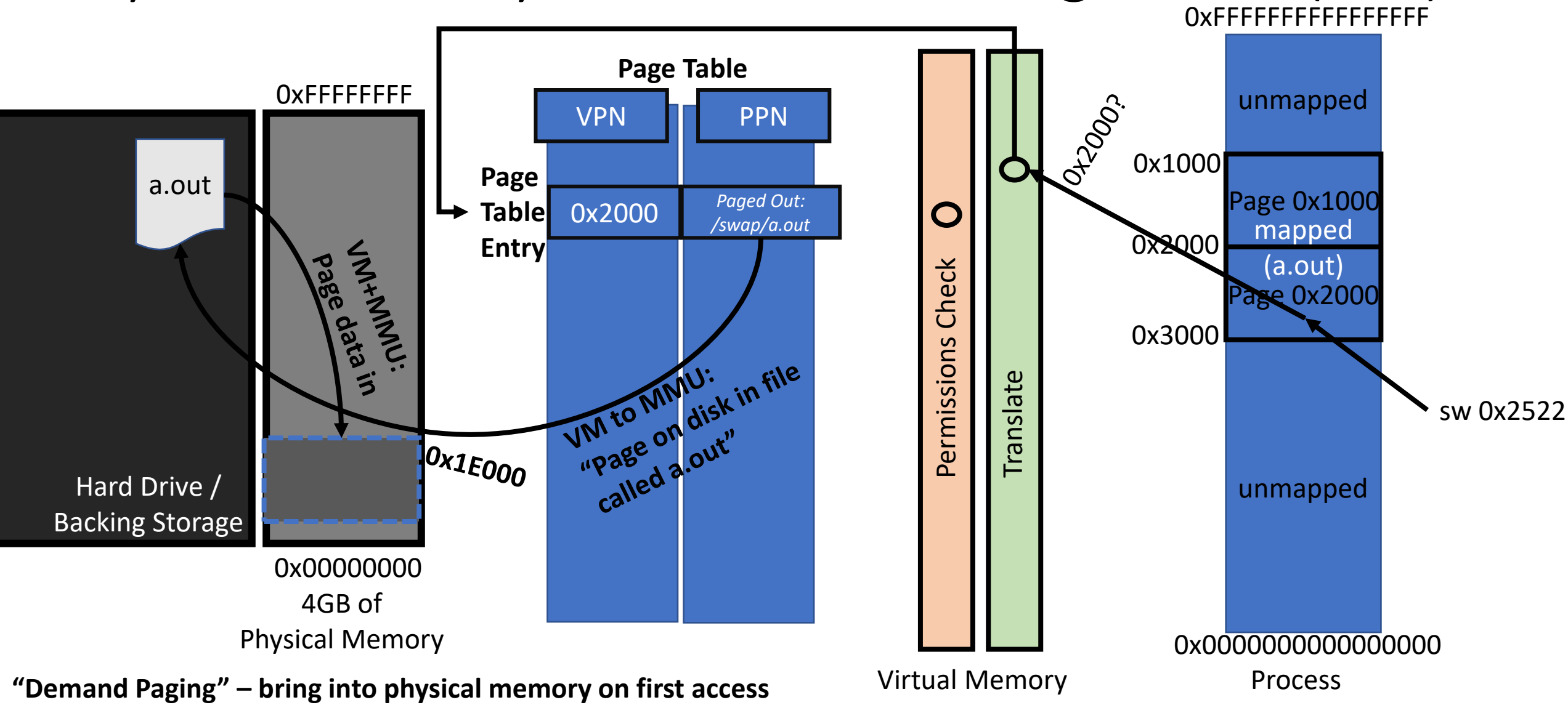
Virtual Memory: The Translation Function
Page Table Holds Disk Location for **Paged-Out** Data



Virtual Memory: The Translation Function
Page Table Holds **No** Entry for **Unmapped** Data

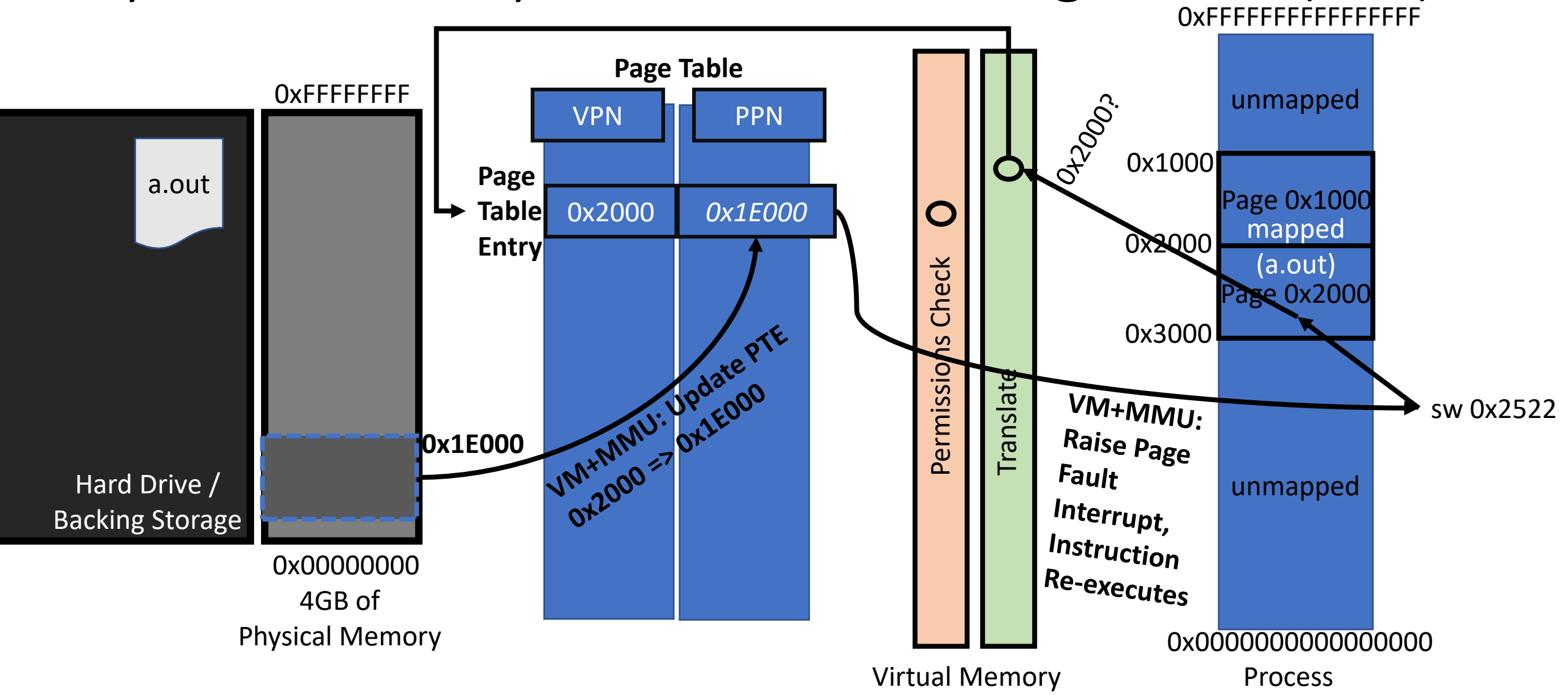


Physical Memory as a Cache of Data on Disk: Physical Memory Cache Miss == Page Fault (1/2)



"Demand Paging" – bring into physical memory on first access

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



Virtual Memory Translation Algorithmically

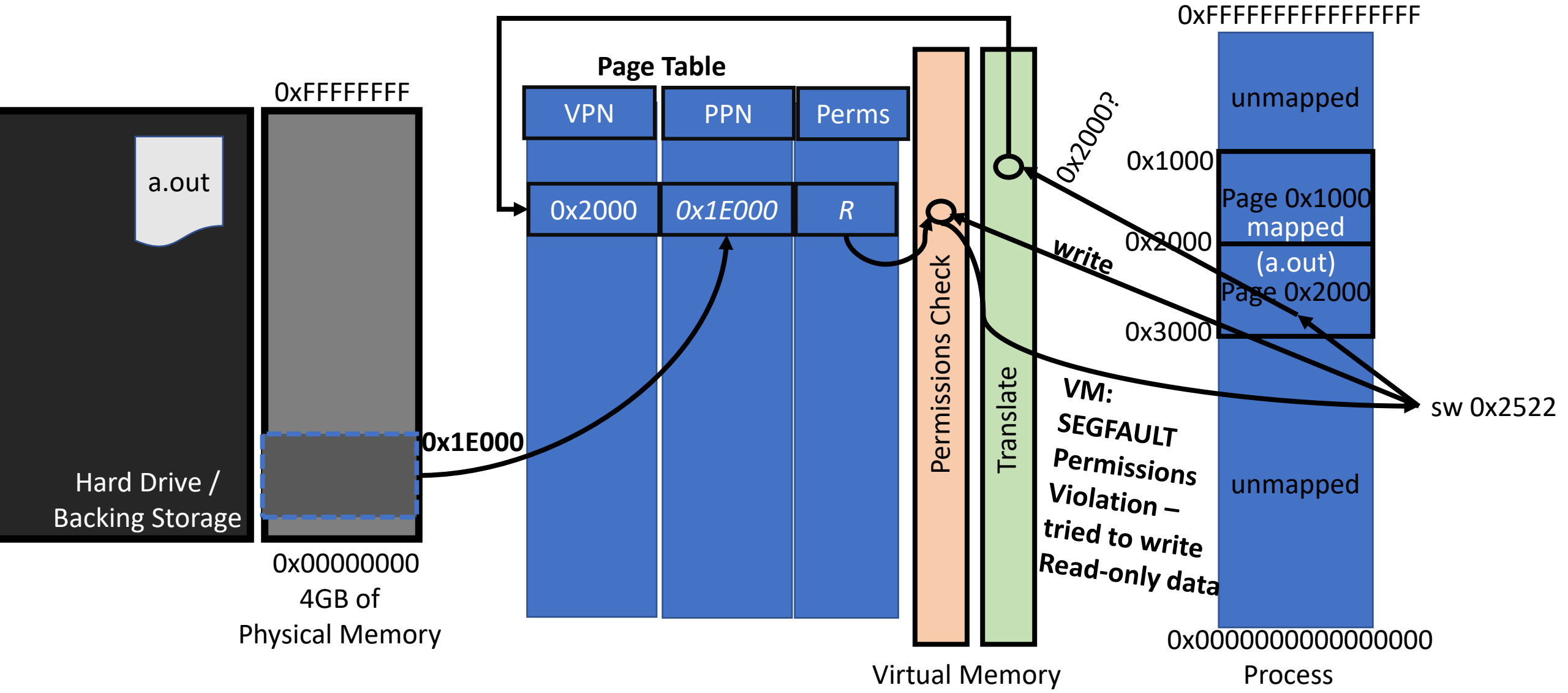
```
vmTranslate(vaddr){
  //Compute Virtual Page Number & Virtual Page Offset
  //from vaddr assuming 2^12 page size
  (VPN,VPO) = (vaddr[63:12],vaddr[11:0])
  PPN = PT.lookup(VPN)

  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
}
```

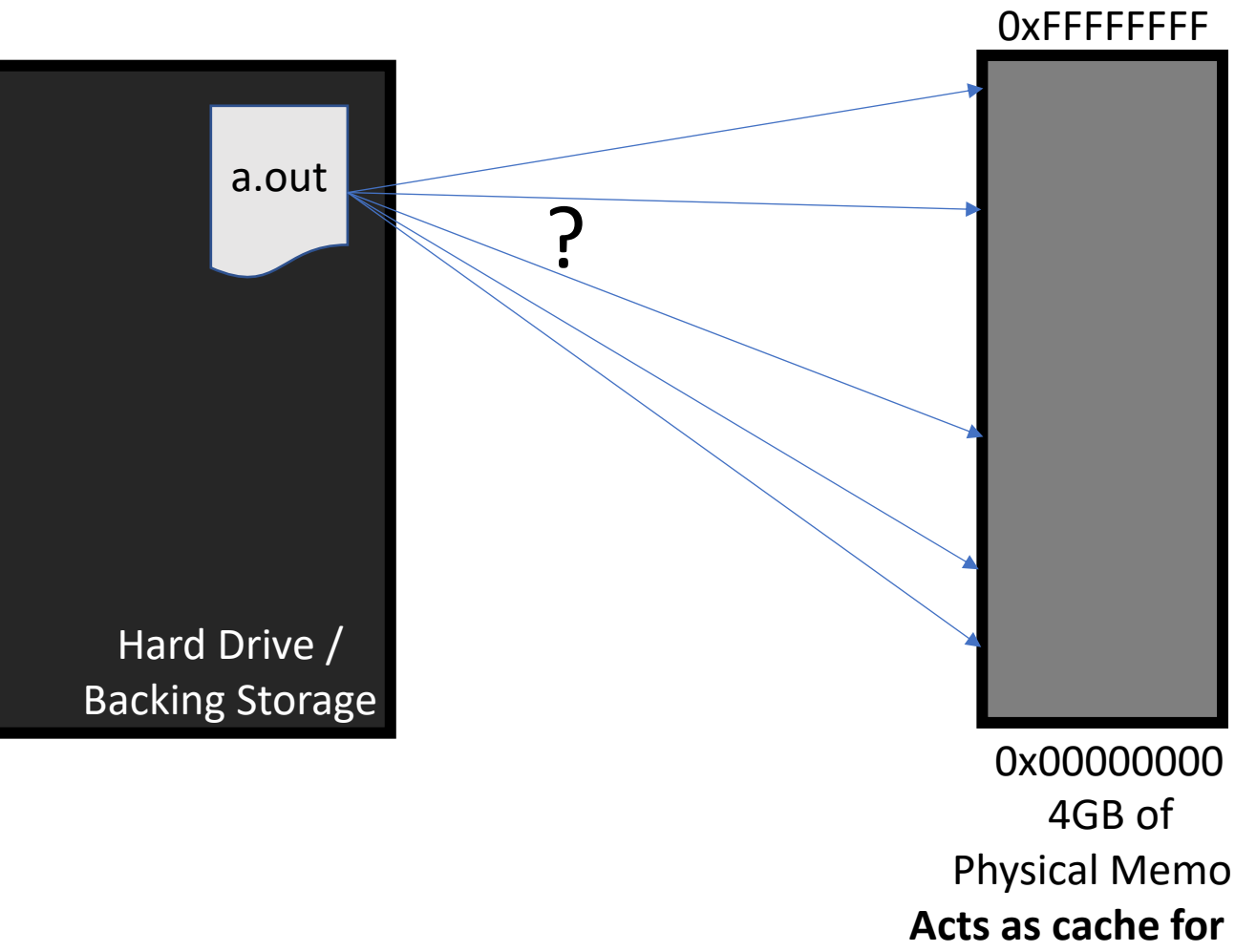
Permissions Checking

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



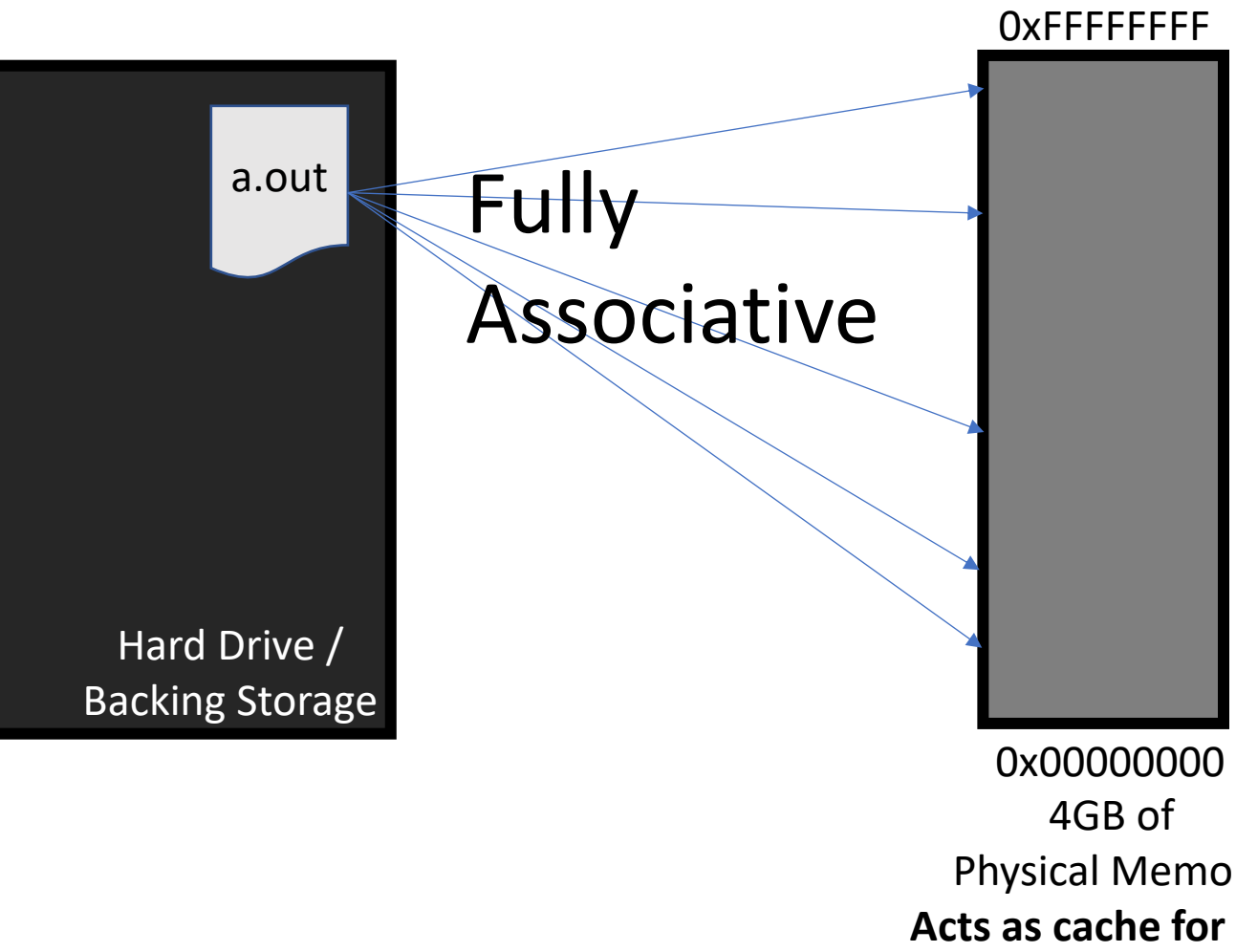
Page Cache Placement / Replacement

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



Placement Policy? Where to put a new page?

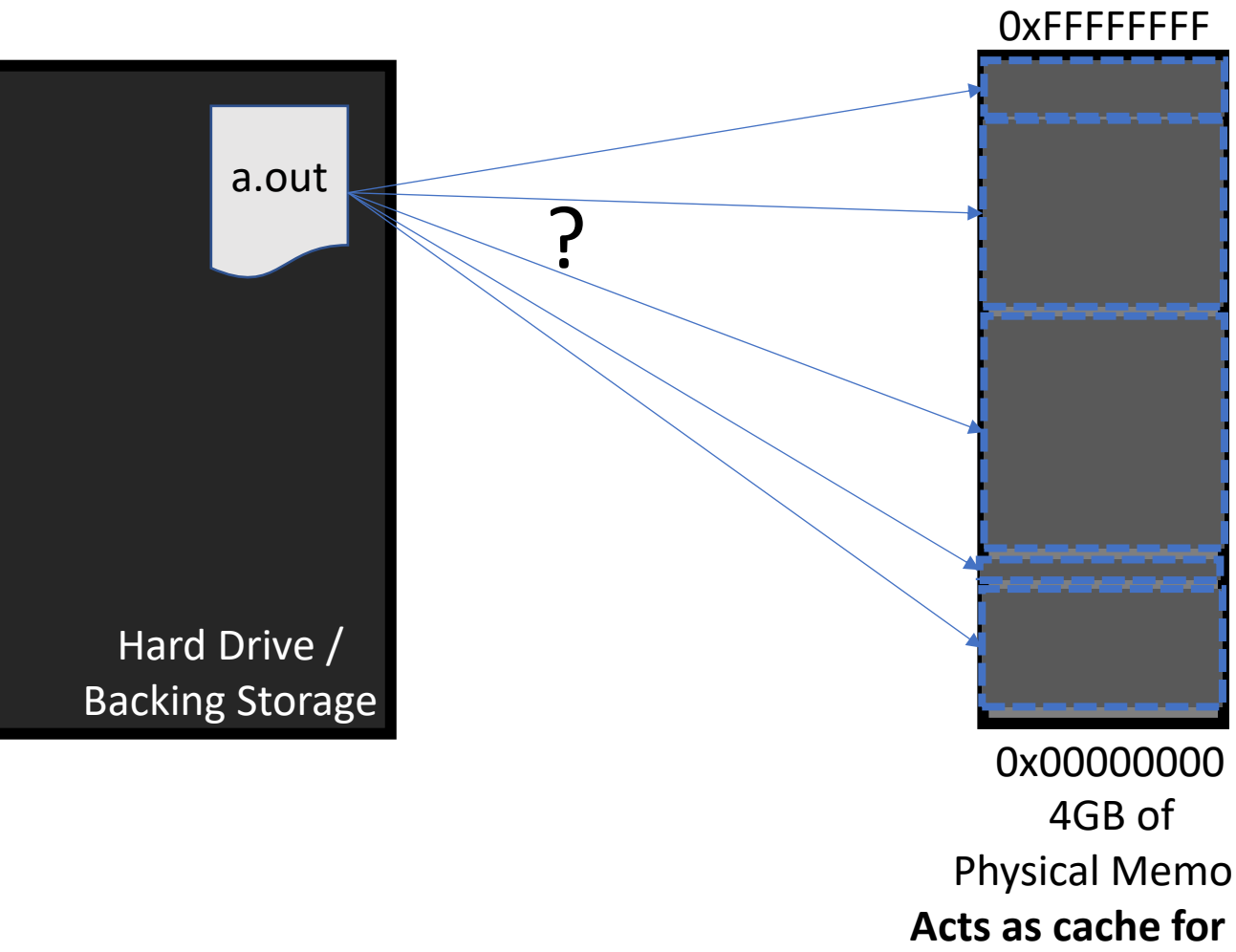
Physical Memory as a Cache of Data on Disk: Cache Placement / Replacement Policy



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?

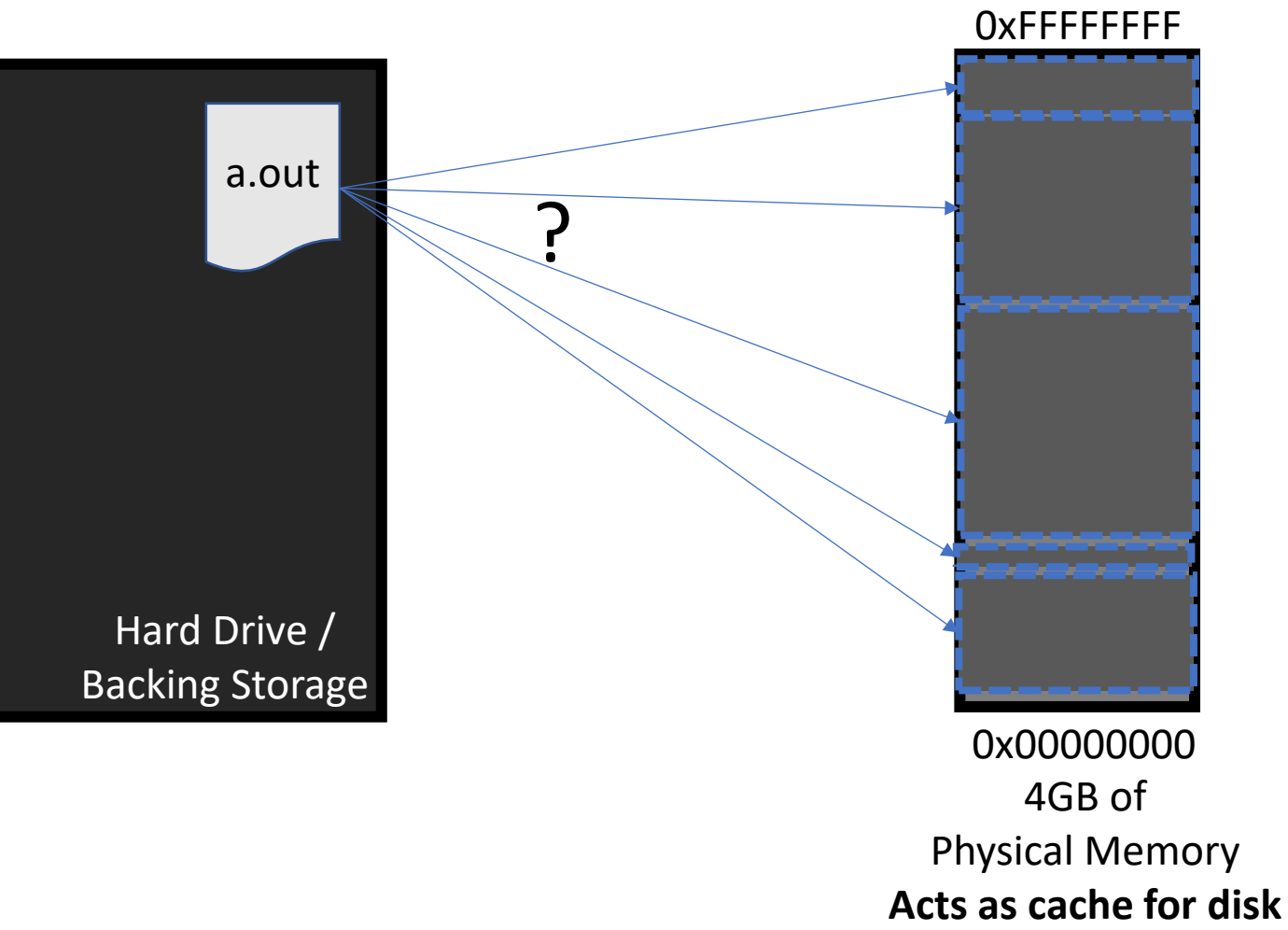
Physical Memory as a Cache of Data on Disk: Cache Placement / Replacement Policy



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?

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…

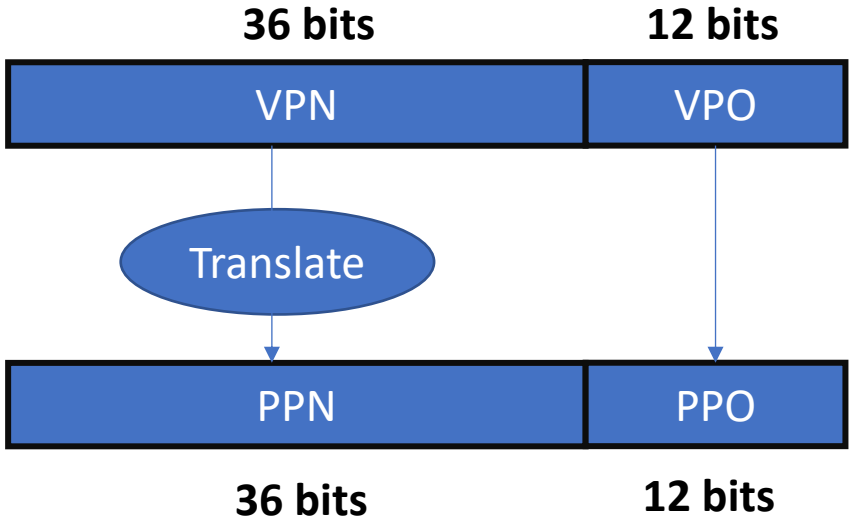
- **Variants of LRU & approximations**
- **Not Recently Used (like Bit-PLRU)**
- **Not Frequently Used**
- **Re-reference-Distance-Based Policies**

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

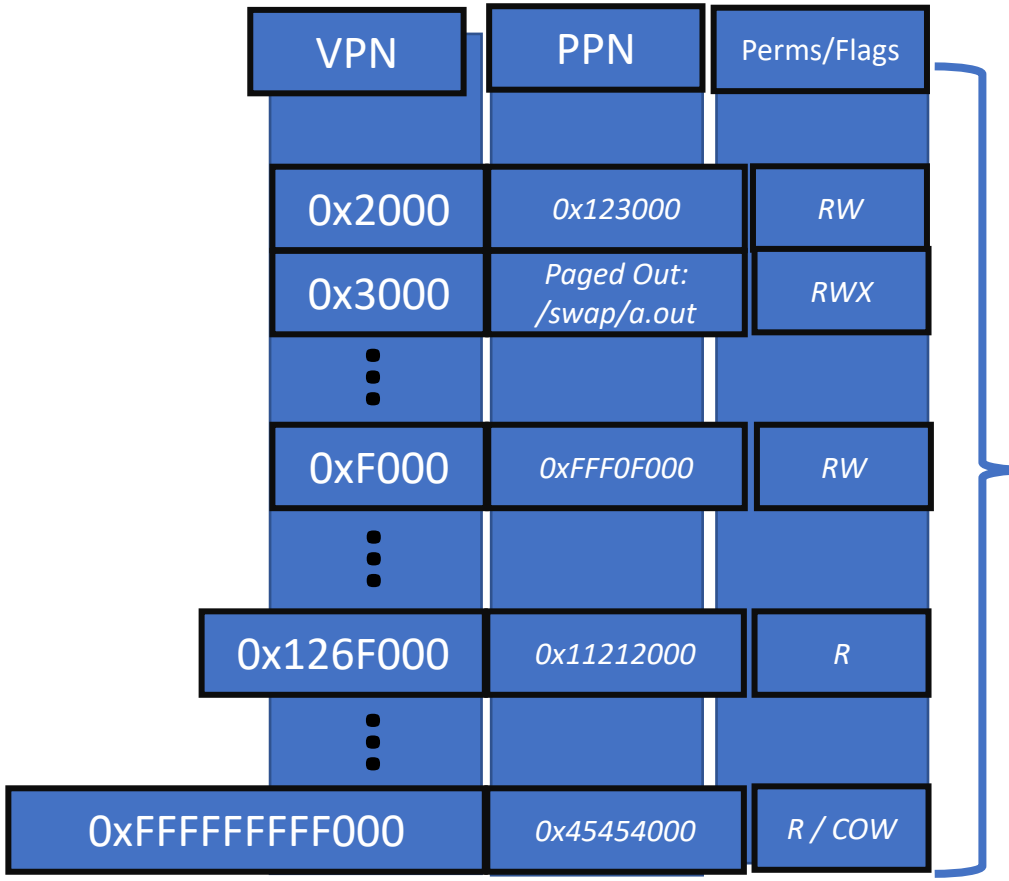
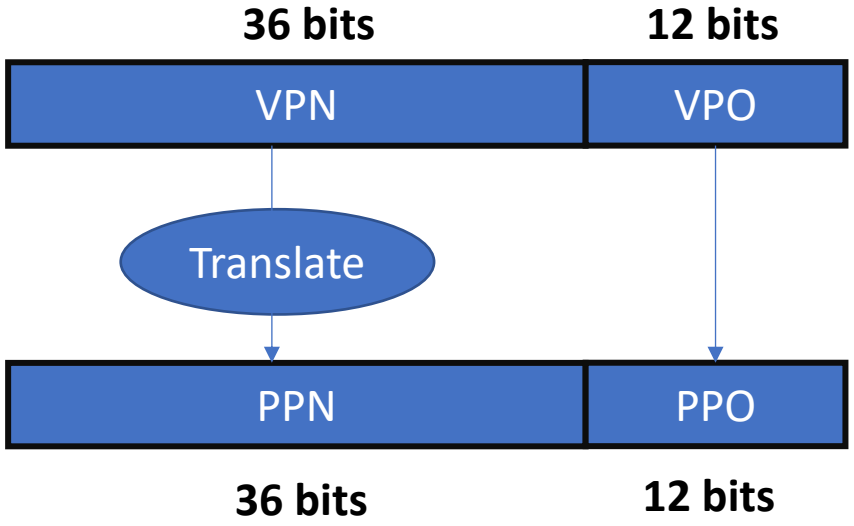


Table stores 2^36 entries in it for virtual pages 0x000000000000 up to 0xFFFFFFFFF000 which span the entire 48-bit address space.

Implementation Issues?

Page Translation and Its Implementation

48-bit Virtual Address (like AMD)



Page Table Entry – 6 Bytes

Page Table

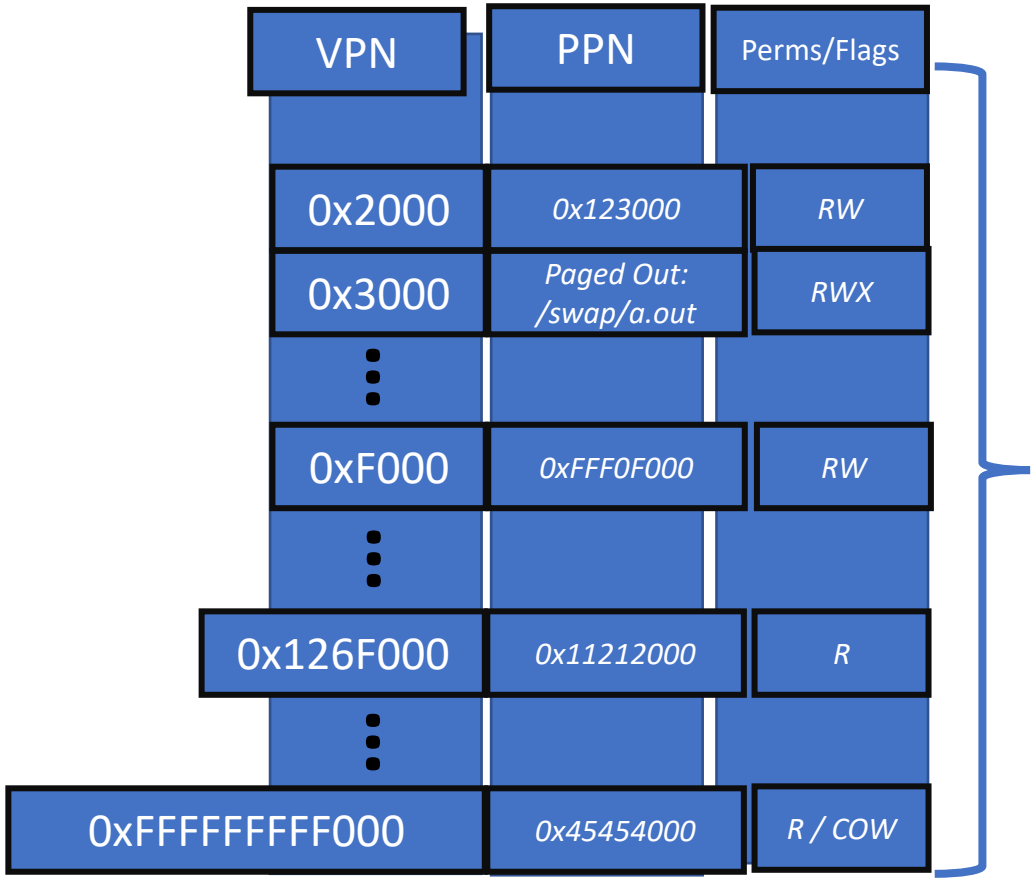
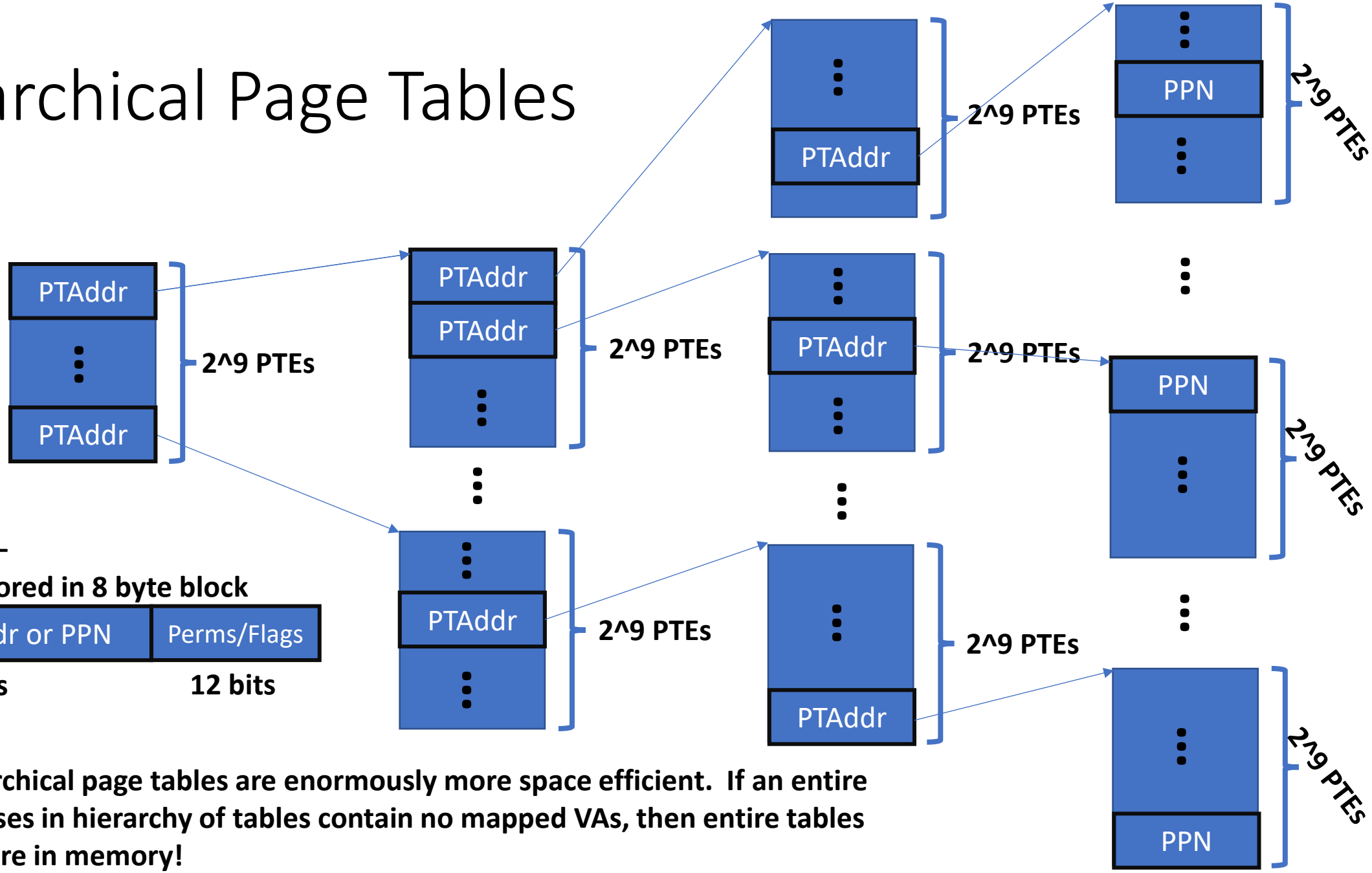


Table stores 2^36 entries in it for virtual pages 0x000000000000 up to 0xFFFFFFFFF000 which span the entire 48-bit address space.

Dense, linear table stores 2^36 * 6B PTEs: 550.8GB of Page Tables

Most PTEs Empty!!!

Hierarchical Page Tables

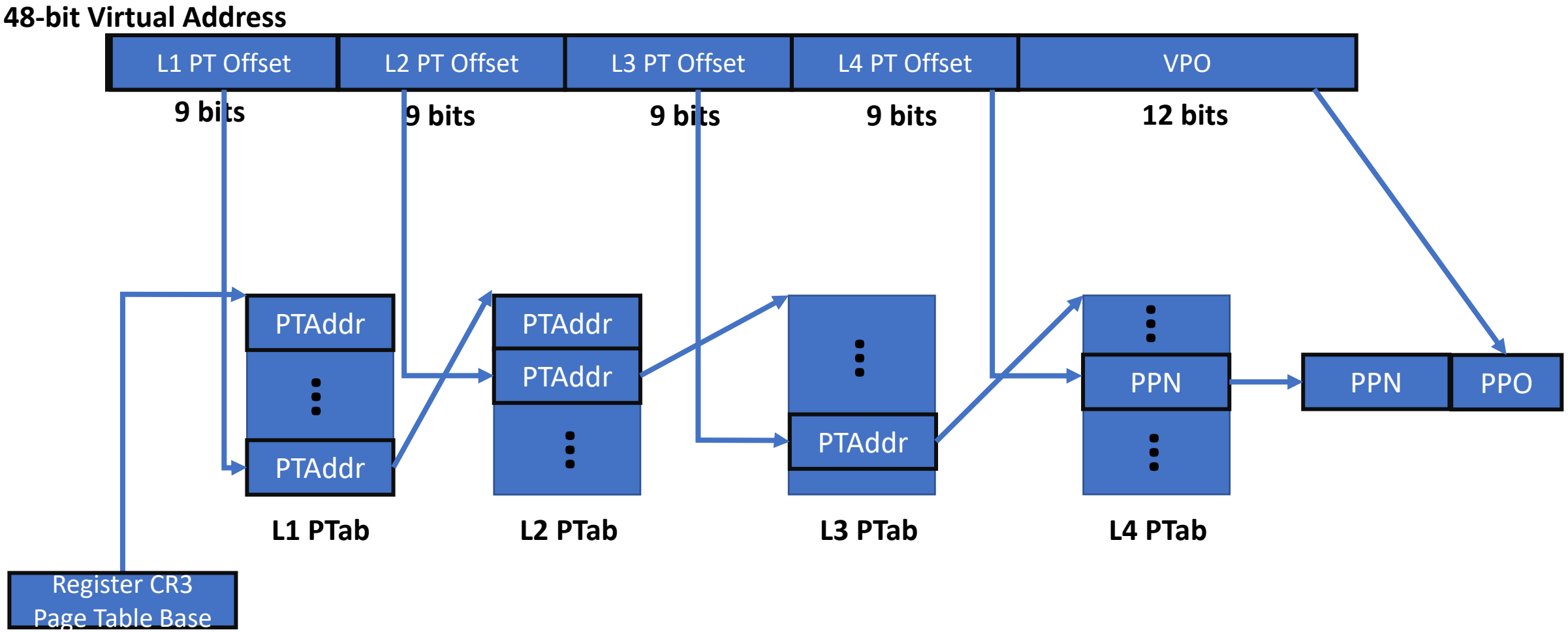


Page Table Entry –
6 Bytes of data stored in 8 byte block

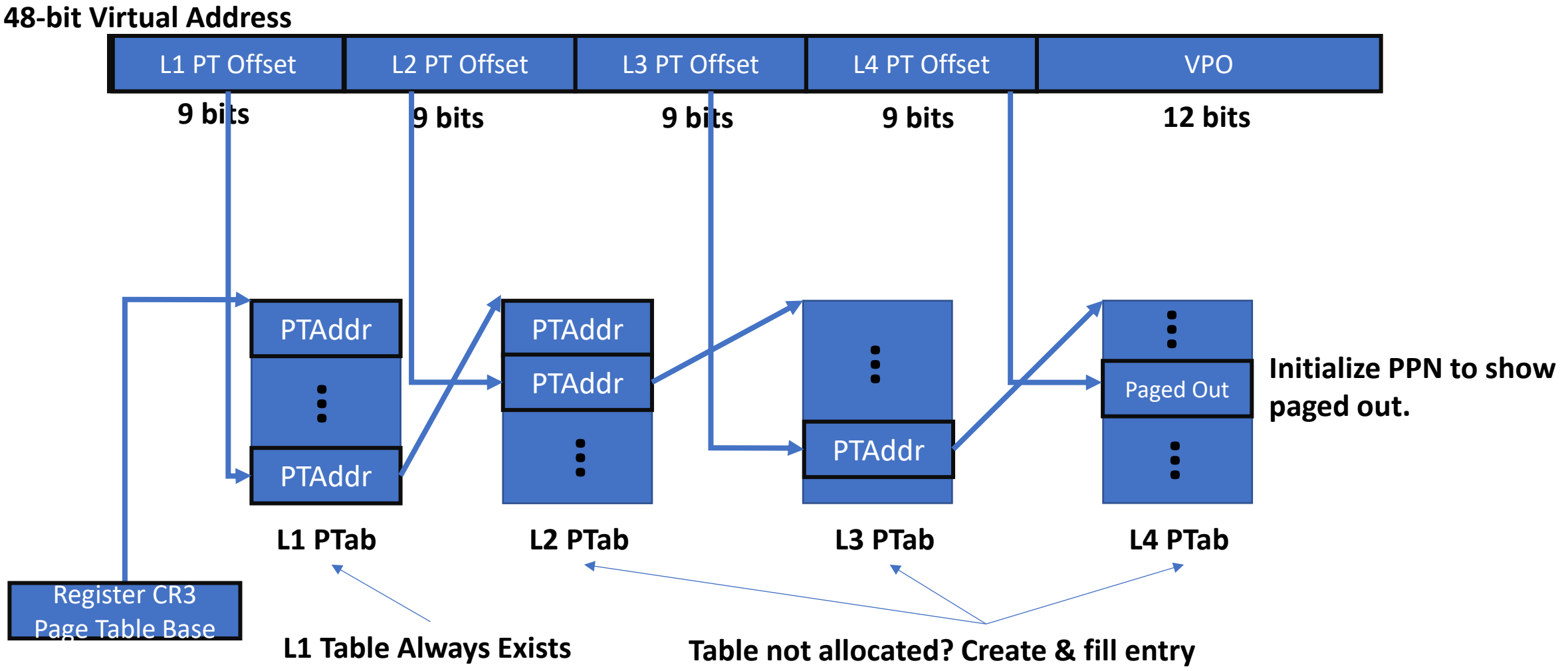
Page Table Addr or PPN	Perms/Flags
36 bits	**12 bits**

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!

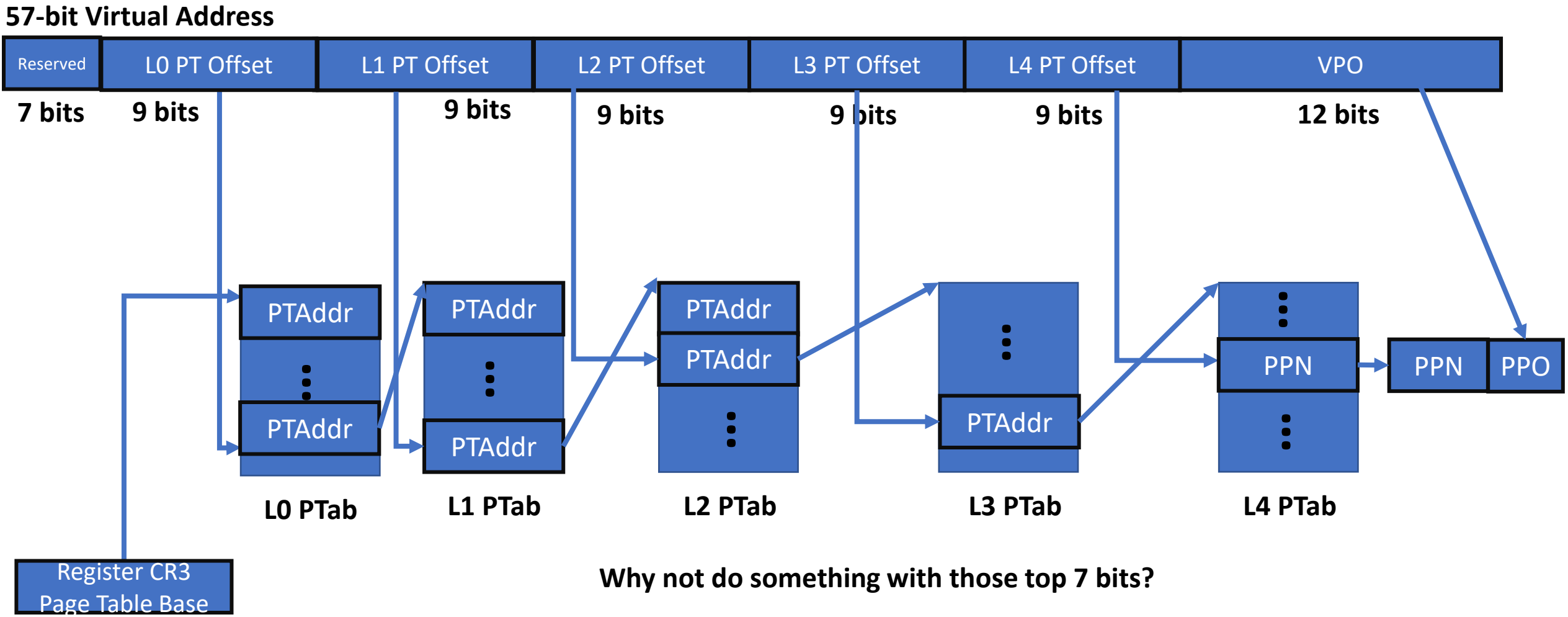
Translation Using Hierarchical Page Tables



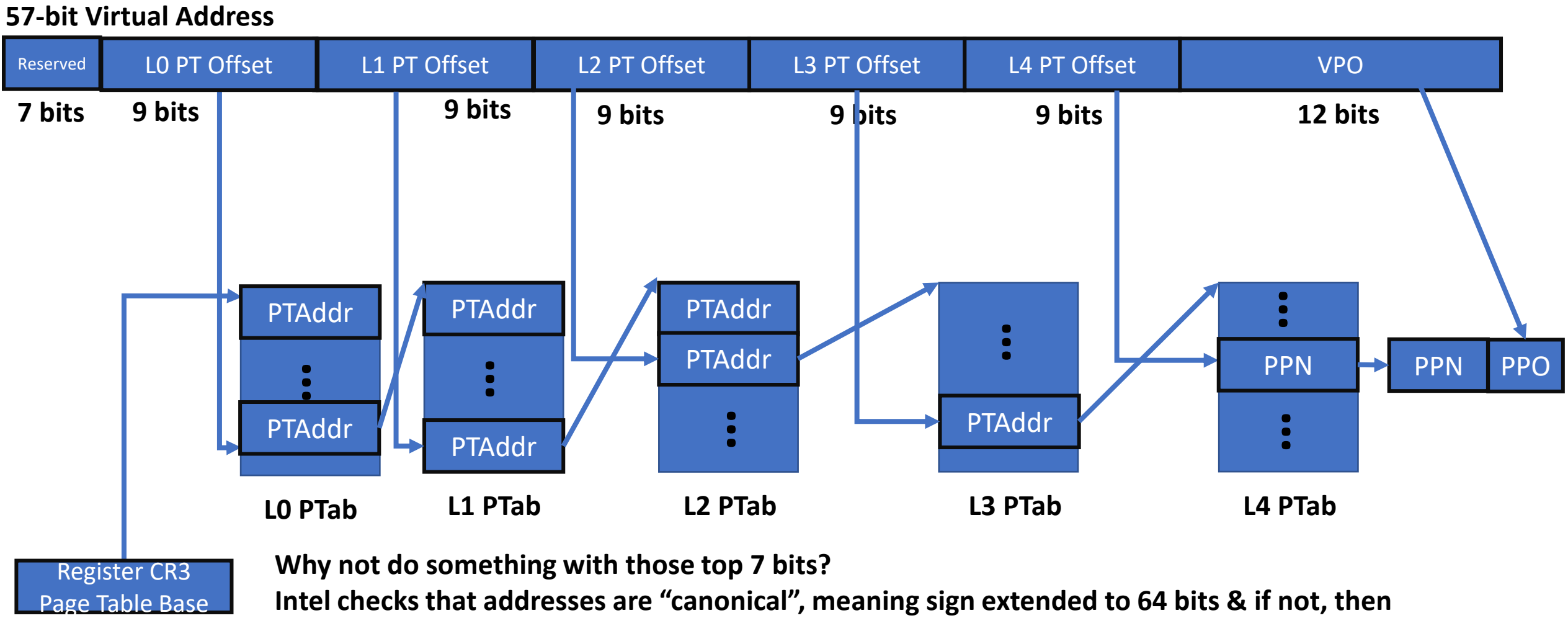
Mapping Using Hierarchical Page Tables



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

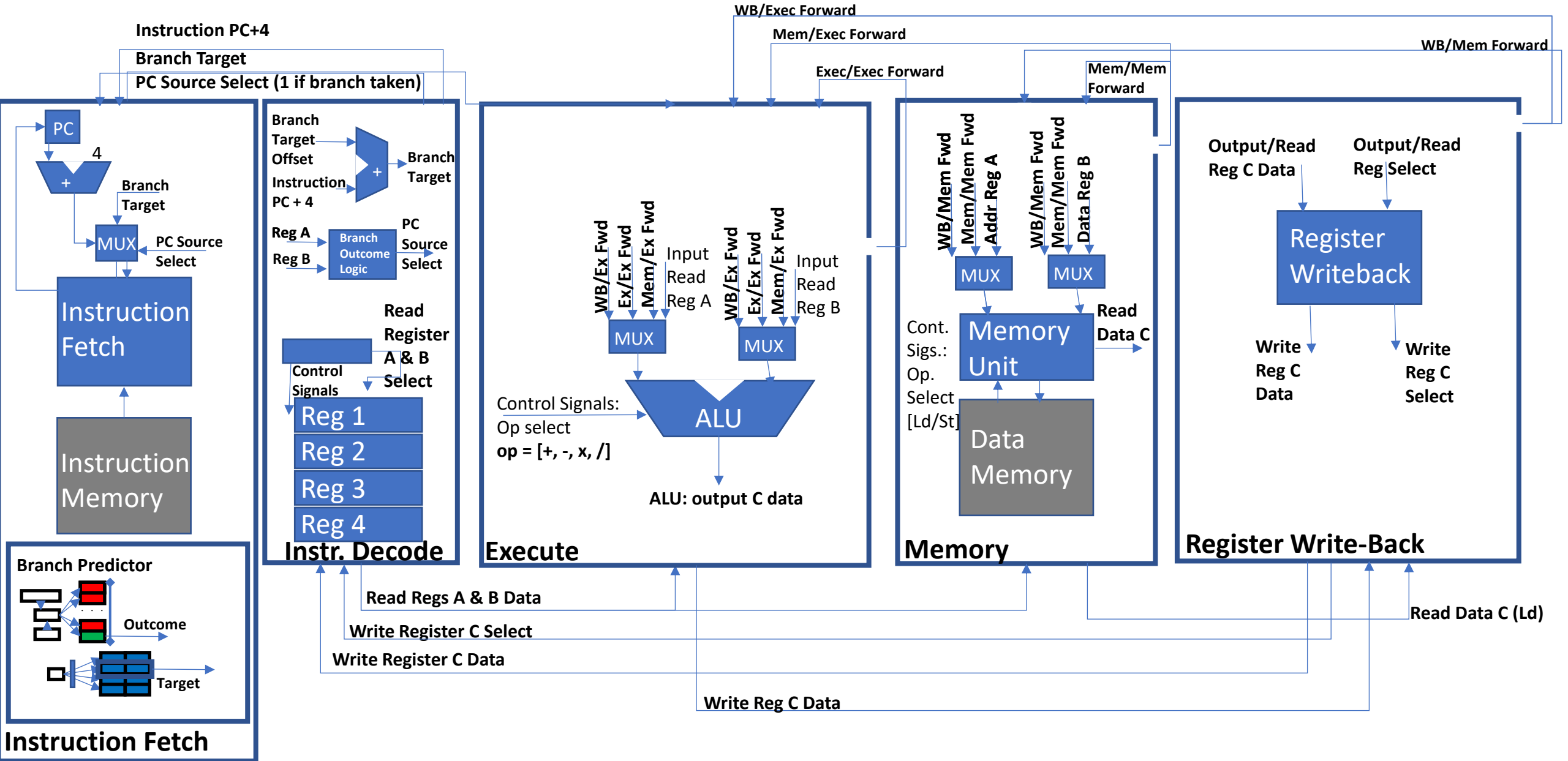


(New) Intel 57-bit Virtual, 52-bit Physical, 5-level Translation Using Hierarchical Page Tables (2019)

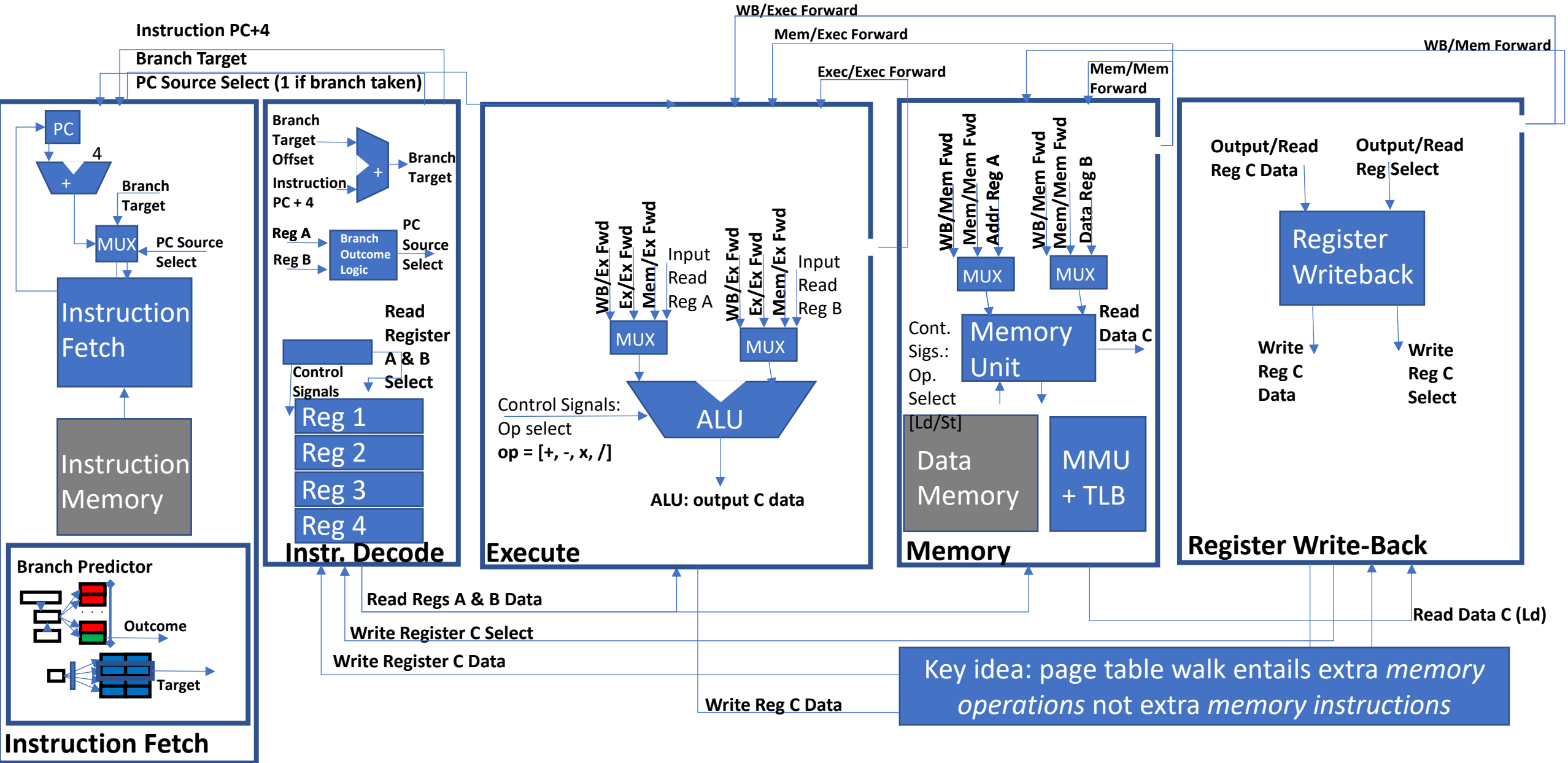


Why not do something with those top 7 bits?
Intel checks that addresses are "canonical", meaning sign extended to 64 bits & if not, then SEGFAULT. Allows future architectures to use 64b addrs if they want to!

What part of the pipeline manipulates the page tables?

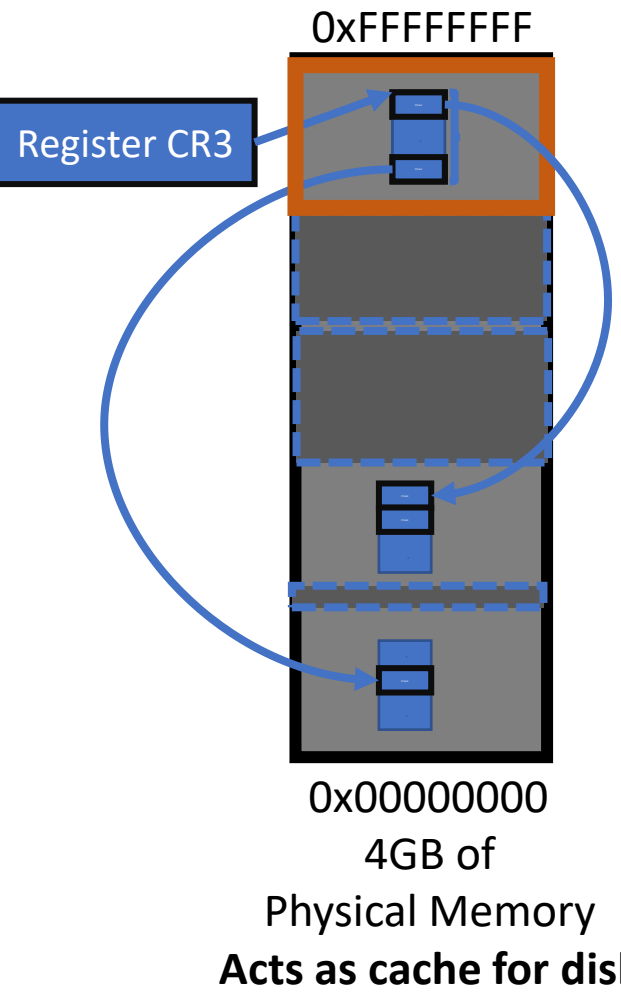


MMU has fast access to memory and TLB for translation



Performance and Storage Overhead Analysis of Translation with Page Tables

Page Tables Stored in Kernel Space of Virtual Memory & (all but first) Paged In & Out

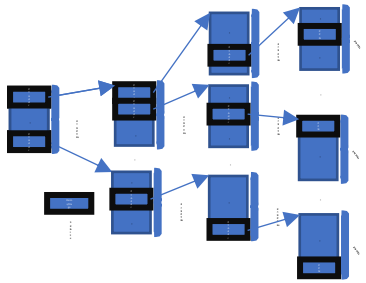


First level page table *always in physical memory at address in Register CR3*.
Other levels of page table can be paged out to make space for other data.

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?

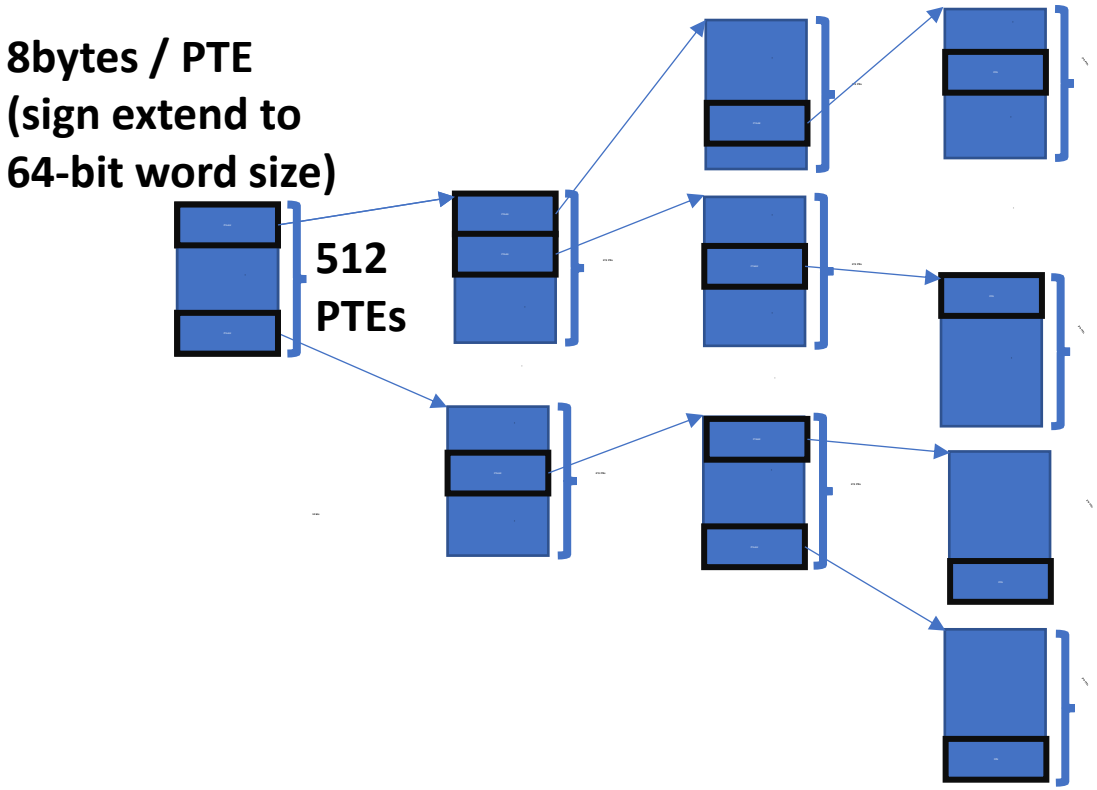
Question: How much time overhead to access hierarchical vs. linear page tables?



Space Overhead Analysis of Page Tables

4 Levels of Page Tables

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



Space Overhead Analysis of Page Tables

4 Levels of Page Tables

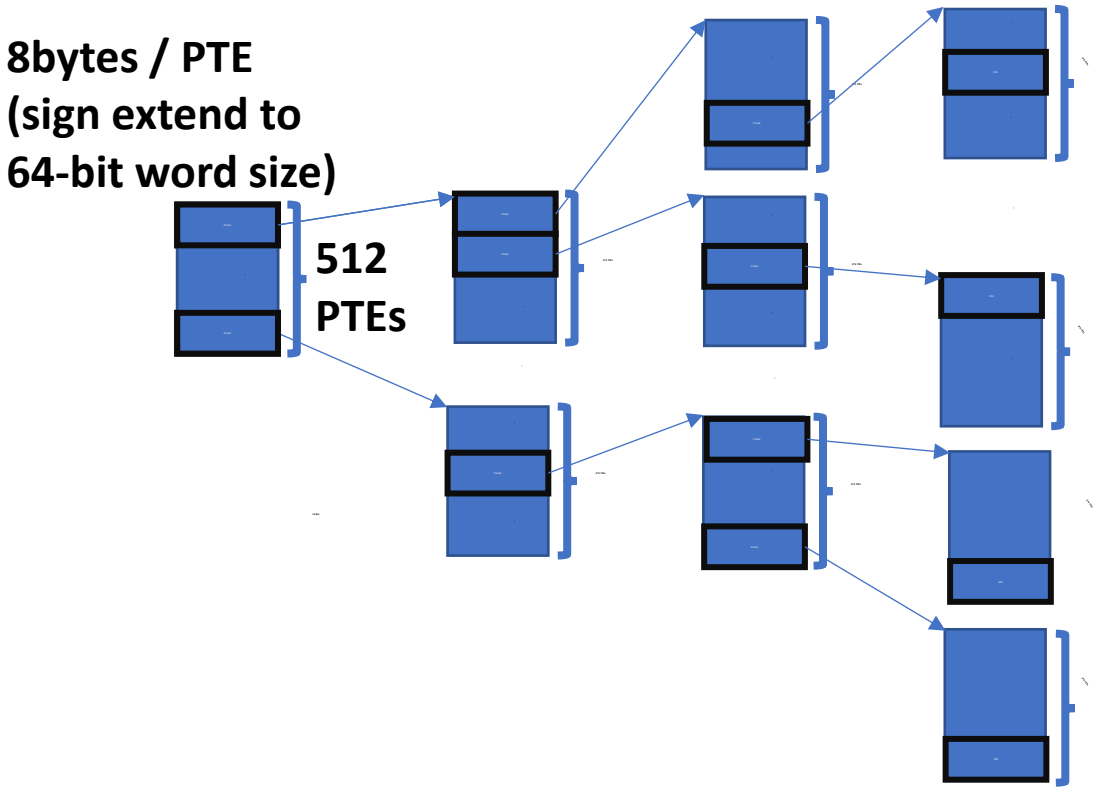


Table Size = Page Size
2^3 bytes / PTE * 2^9 PTEs / Table = 2^12 = 4kB / Table

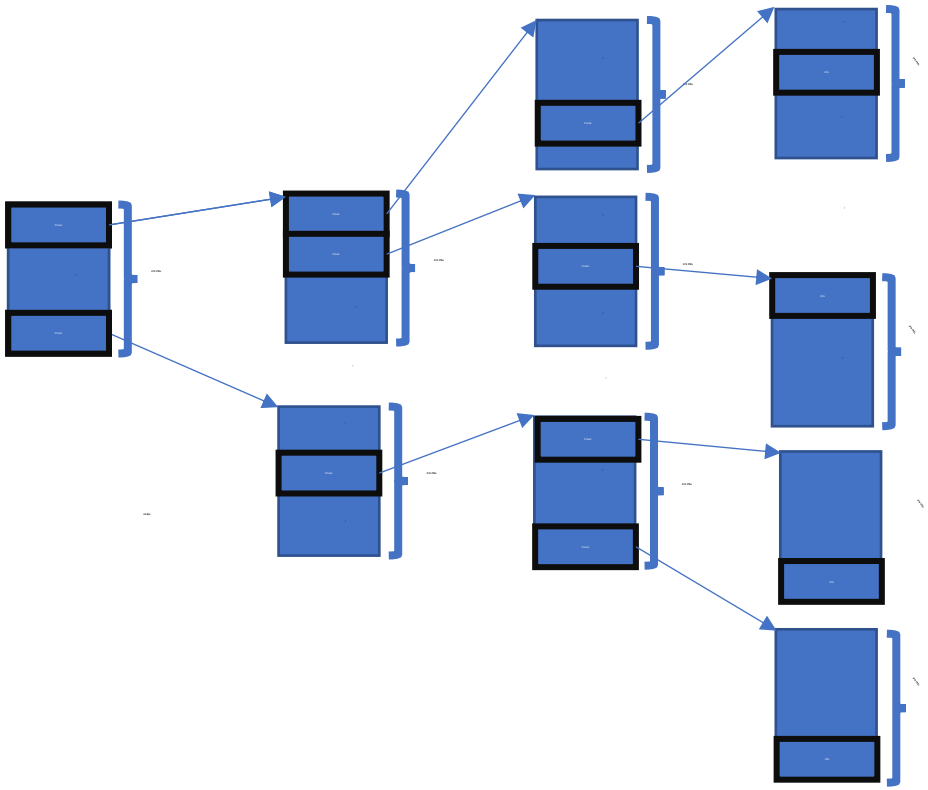
Pictured Example page tables size:
10 * 4kB = 40kB of page tables

Possible to map every page in last level PT:
4 last level tables exist * 512 entries * 4kB / page =
2^23B mappable with just these page tables

Hierarchical Page Table Overhead:
40kB / 2^23B = 40kB / 2^13kB = 0.005x overhead

Naïve Linear page tables:
550GB of page tables
With 2^23B of data to map, 65565x overhead

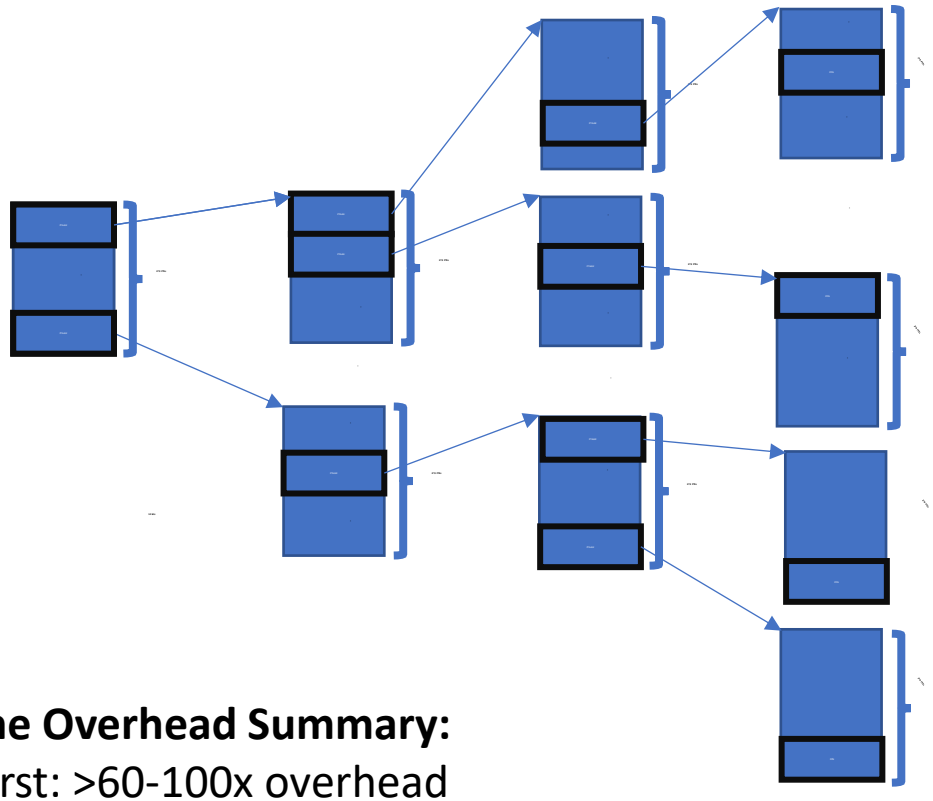
Performance Analysis of Page Tables



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

```
sw 0x2000
```

Performance Analysis of Page Tables



Time Overhead Summary:
Worst: >60-100x overhead
Best: ~2x overhead

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

```
sw 0x2000
```

Four extra 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:

L1 cache hit, all page tables miss in cache & page fault

1 cycle L1 hit becomes 4 cache misses & 3 page faults (DRAM ~20 cycles) = 60-100 cycles overhead for a 1 cycle L1 hit

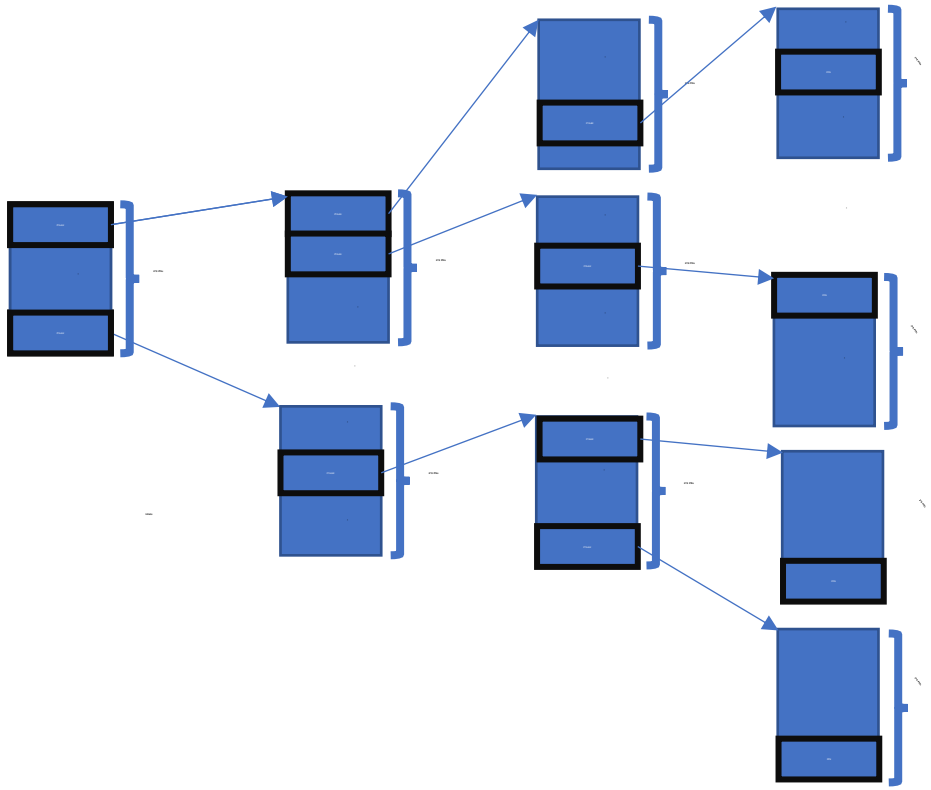
Minimum Overhead:

L1 miss, all page table accesses hit in cache & no page faults

3 cycle L1 miss becomes 4 cache hits (1 cycle cache hits) = 4 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!

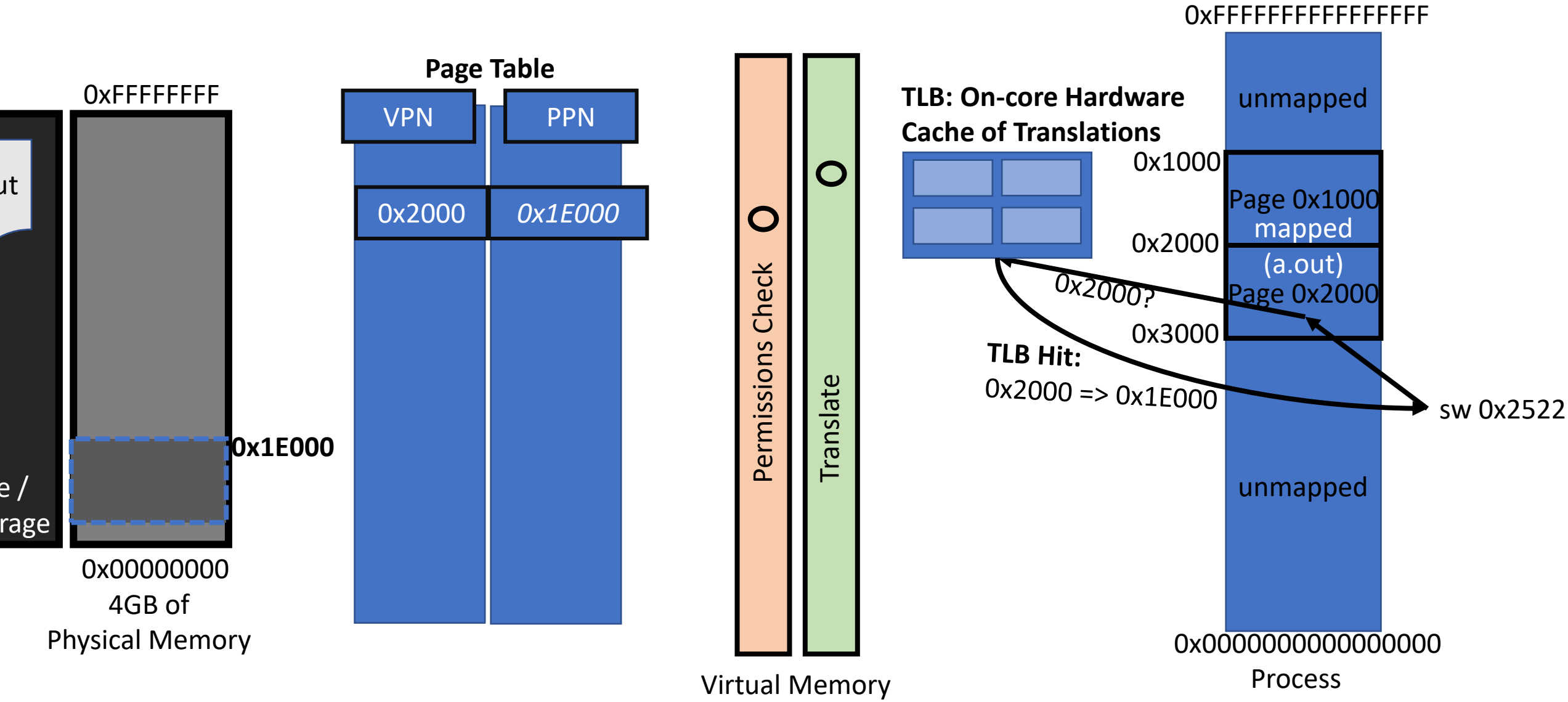


Space Savings:
From 65565x space overhead to a 0.005x space overhead

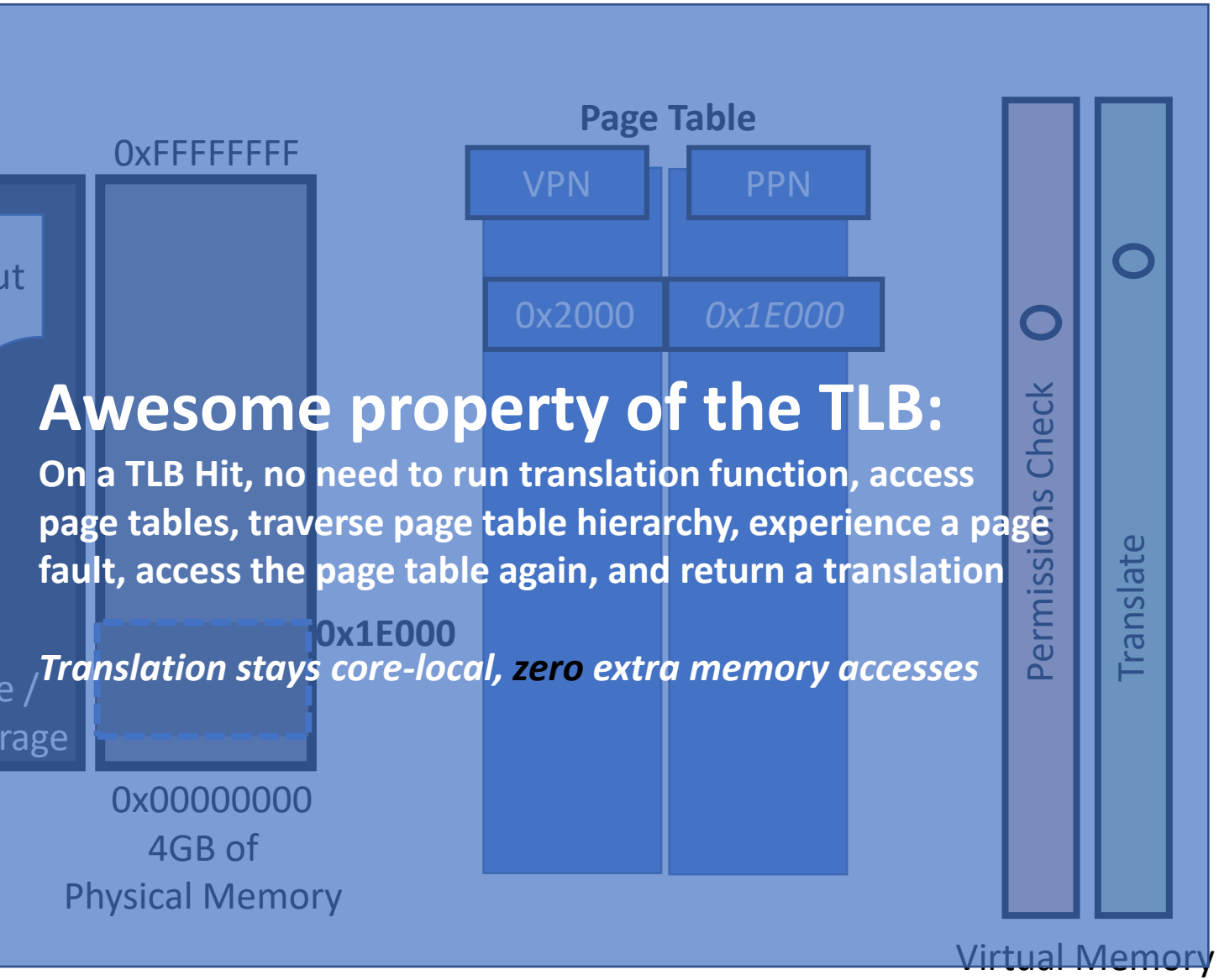
Key Insight:
Use microarchitectural support in the form of a *Translation Lookaside Buffer* to eliminate the time cost of *most* translations

Translation Lookaside Buffers: Hardware Support for Caching Page Address Translations

Translation Lookaside Buffer: Basic Idea (Hit)



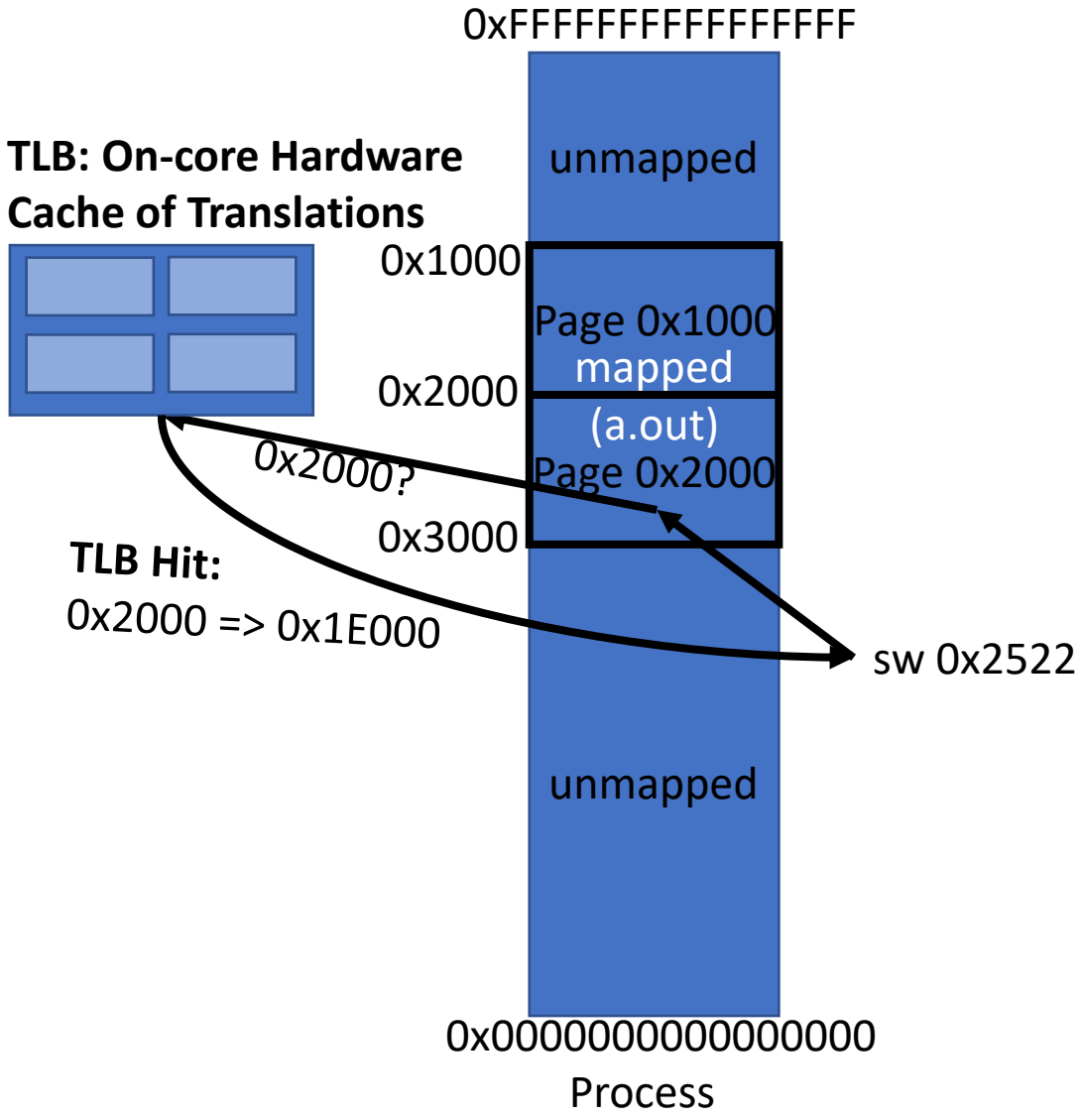
Translation Lookaside Buffer: Basic Idea (Hit)



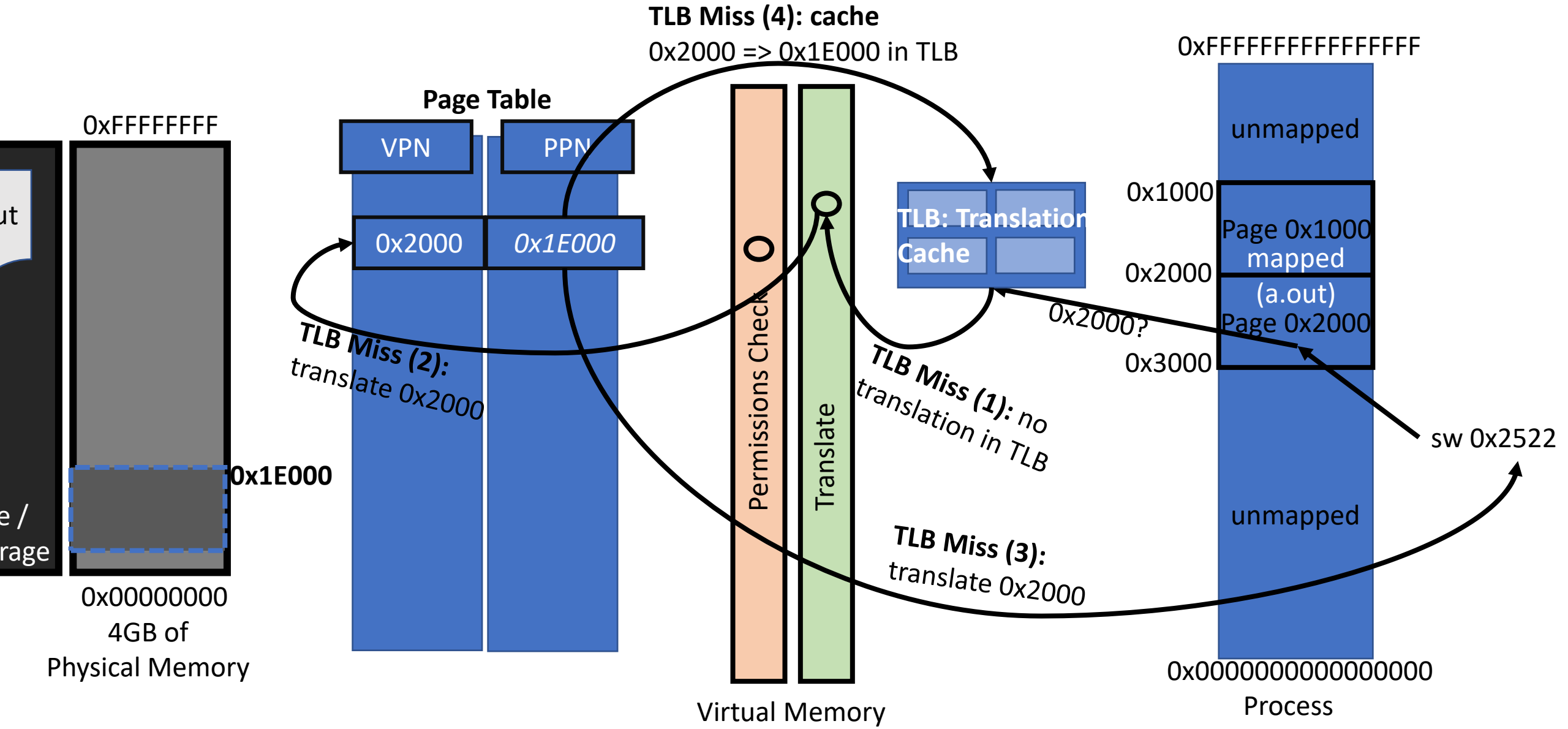
Awesome property of the TLB:

On a TLB Hit, no need to run translation function, access page tables, traverse page table hierarchy, experience a page fault, access the page table again, and return a translation

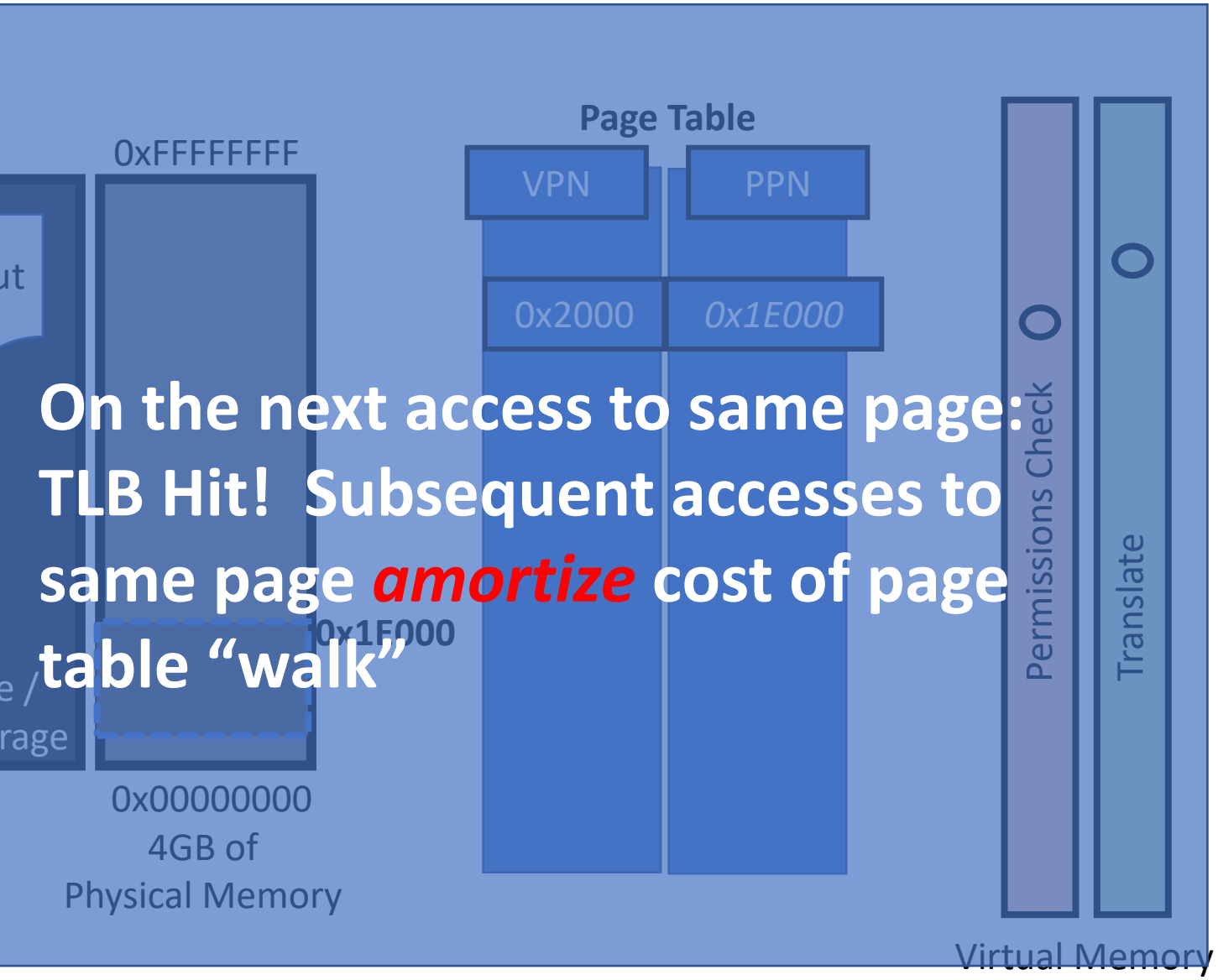
Translation stays core-local, zero extra memory accesses



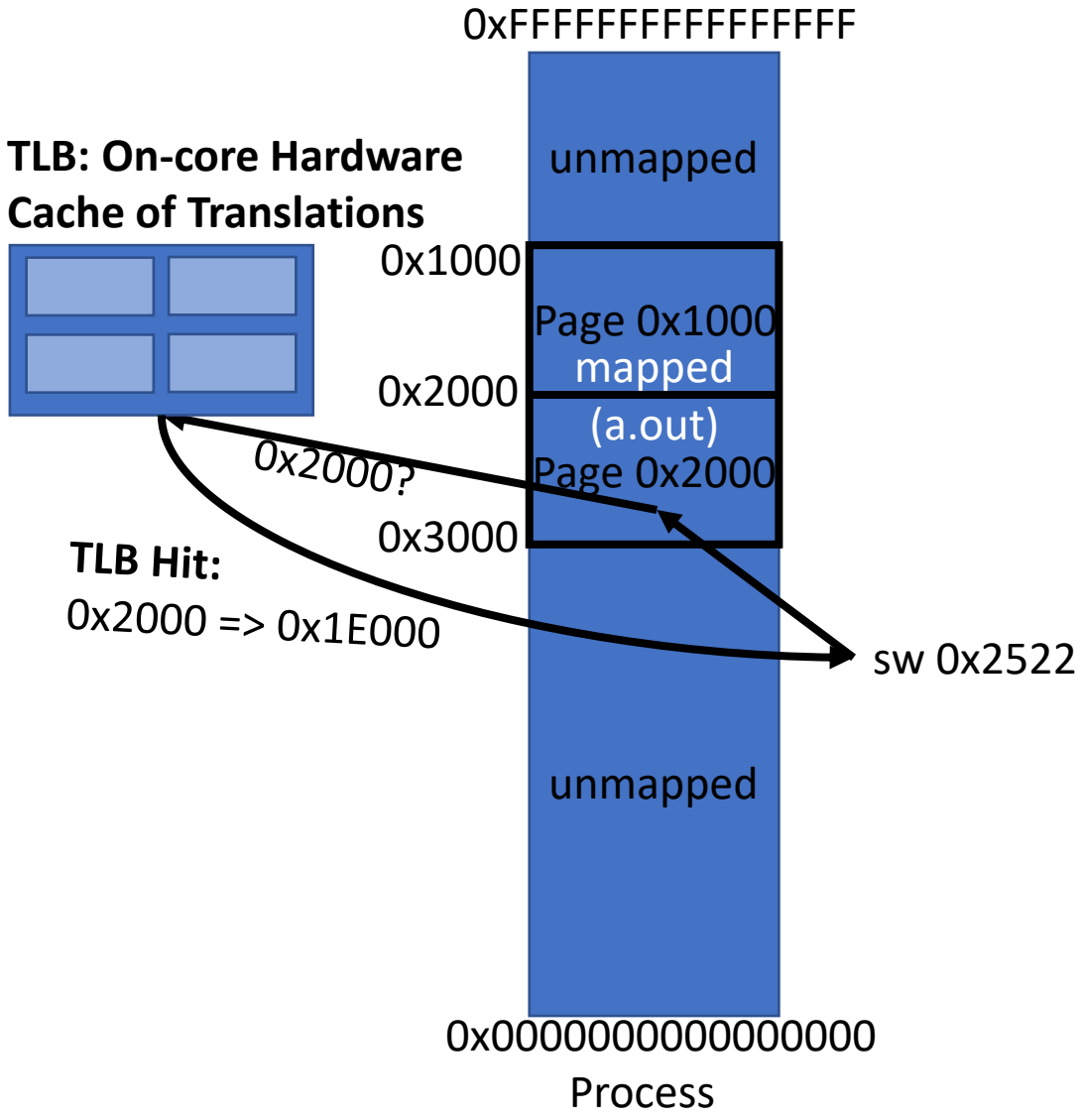
Translation Lookaside Buffer: Basic Idea



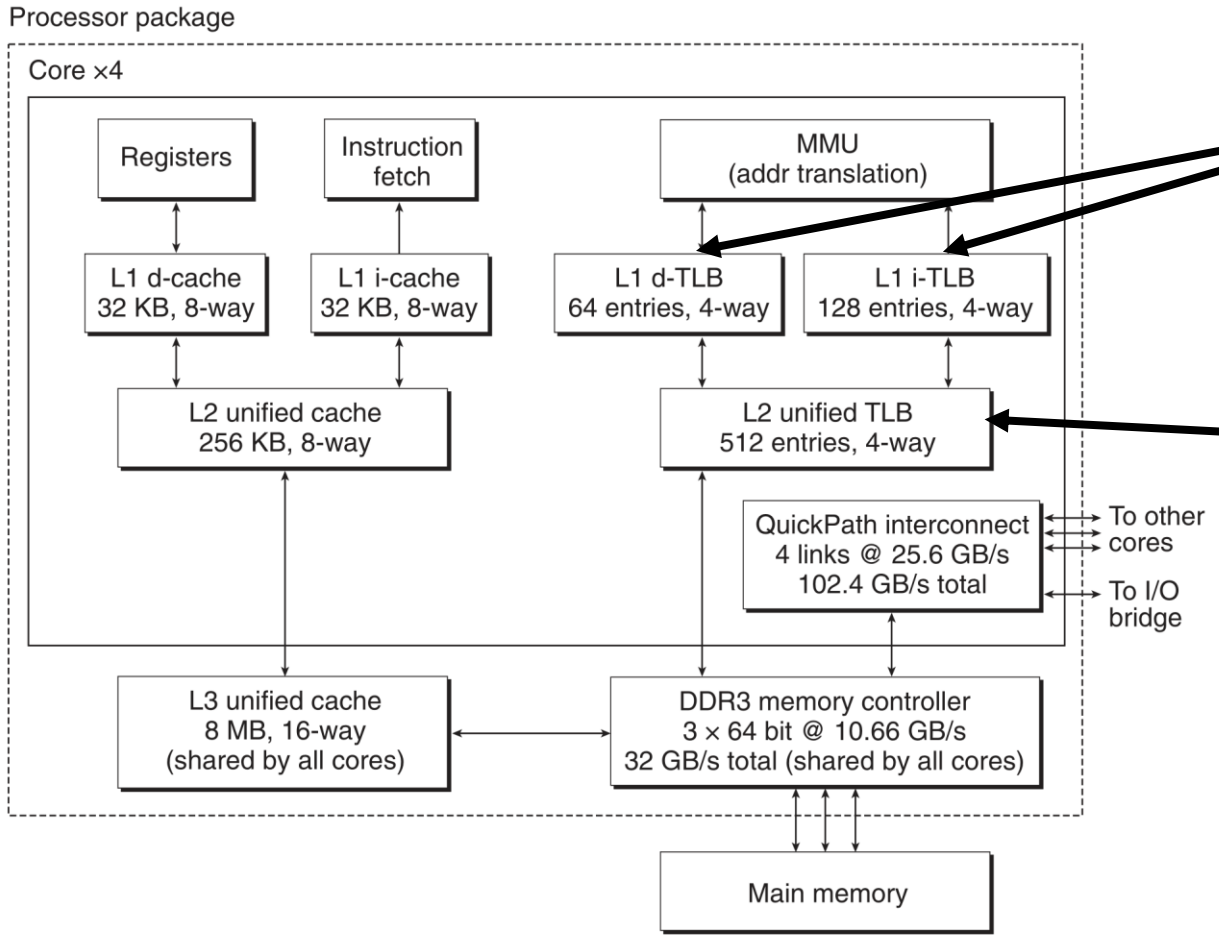
Translation Lookaside Buffer: Basic Idea (Hit)



On the next access to same page: TLB Hit! Subsequent accesses to same page *amortize* cost of page table "walk"



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

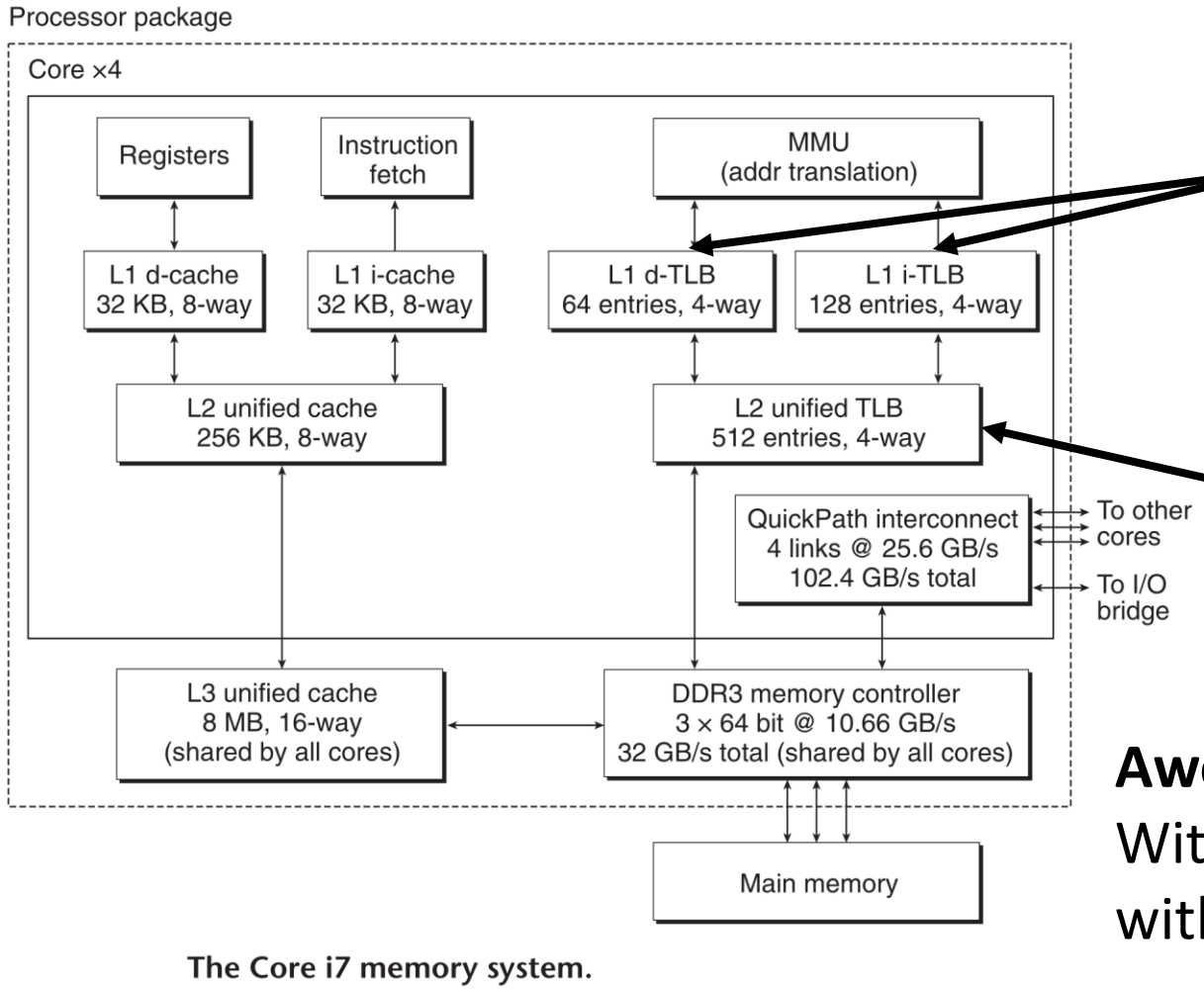


The Core i7 memory system.

Separate L1 data & instruction (L1) TLB
Size? Reach? Why i-TLB larger?

Unified (L2) TLB
512 entries: total size? total reach?

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



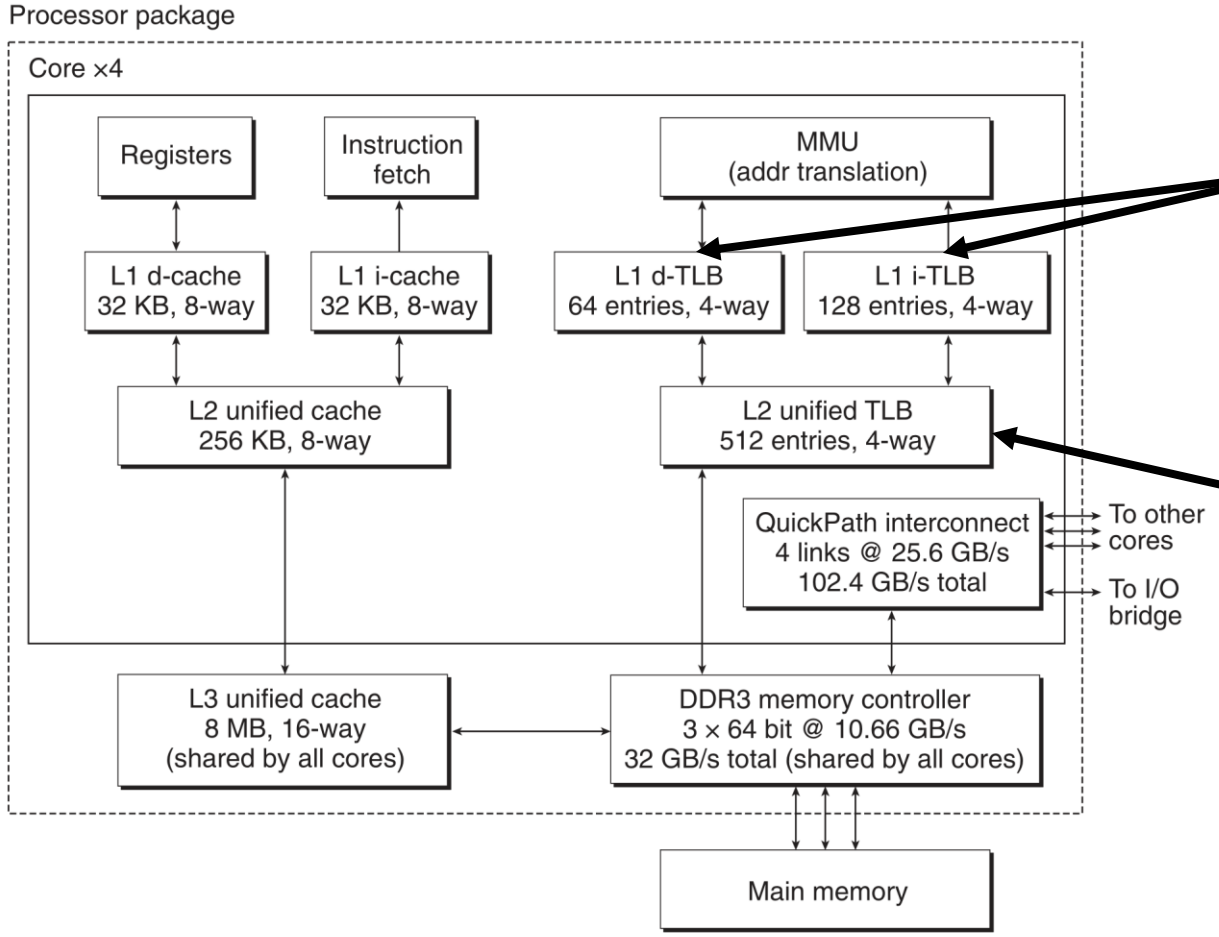
The Core i7 memory system.

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

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



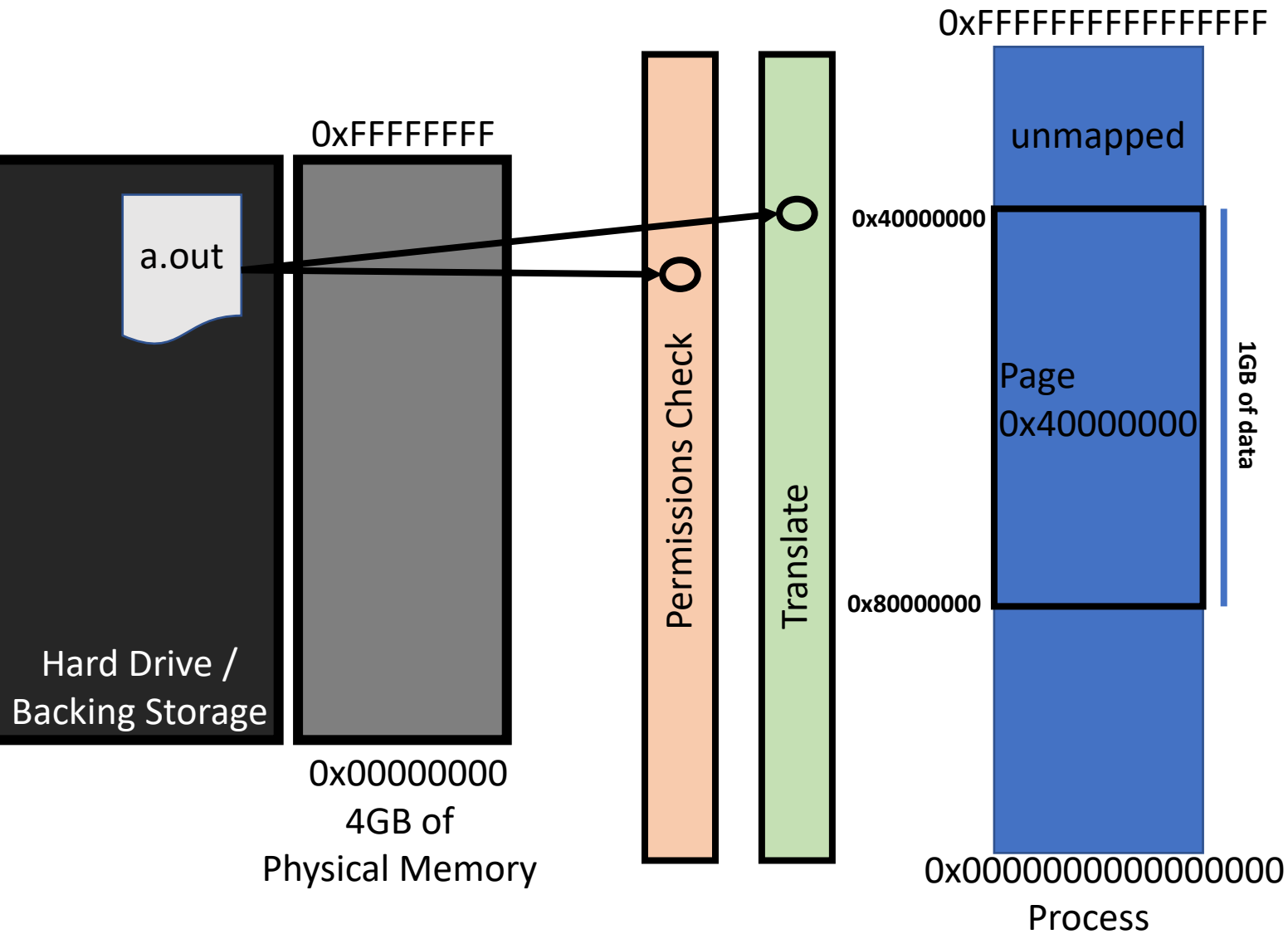
The Core i7 memory system.

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

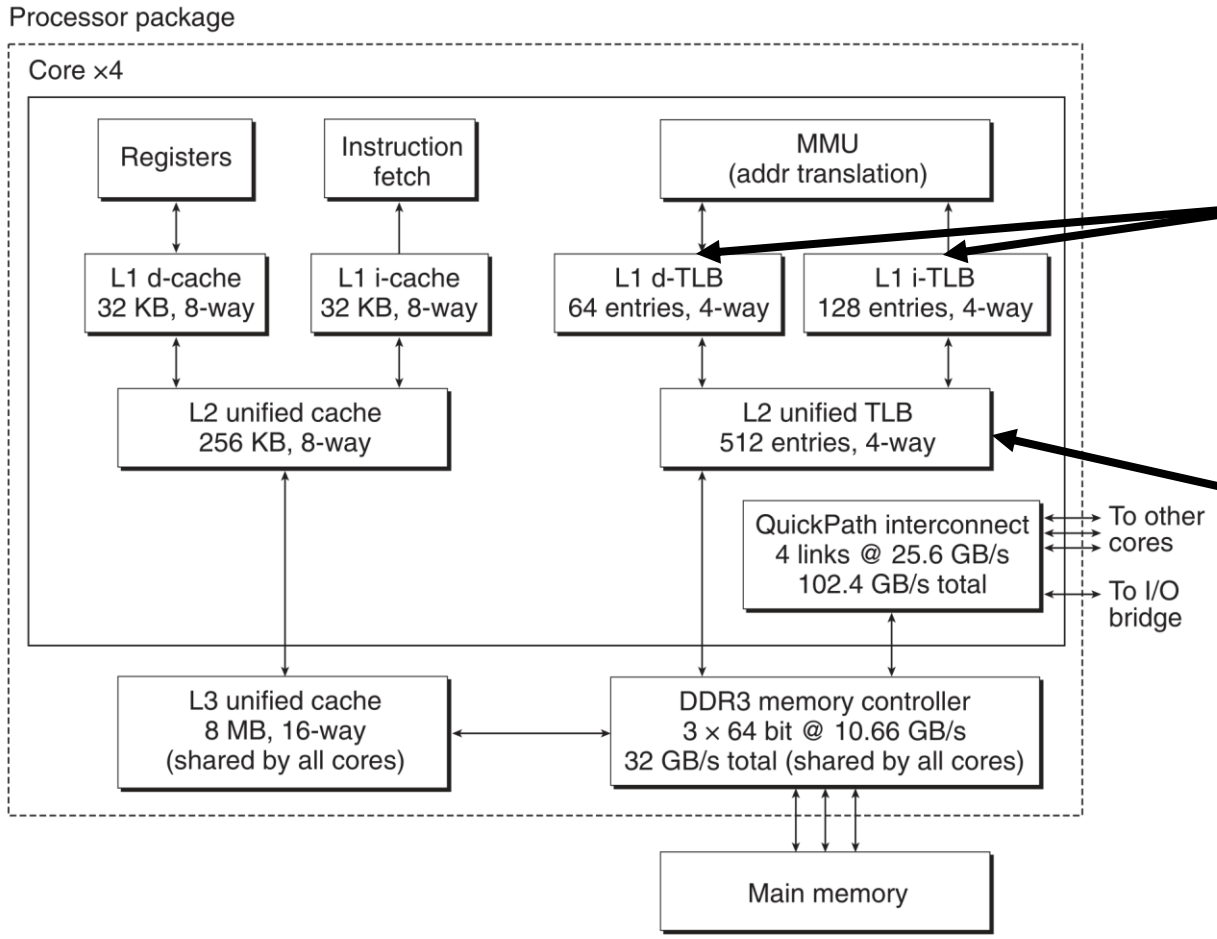
Awesome Property of the TLB?
What if my working set is, like, 58GB?
Question: How do we increase TLB Reach?

Revisiting the Assumption of Page Granularity



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

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



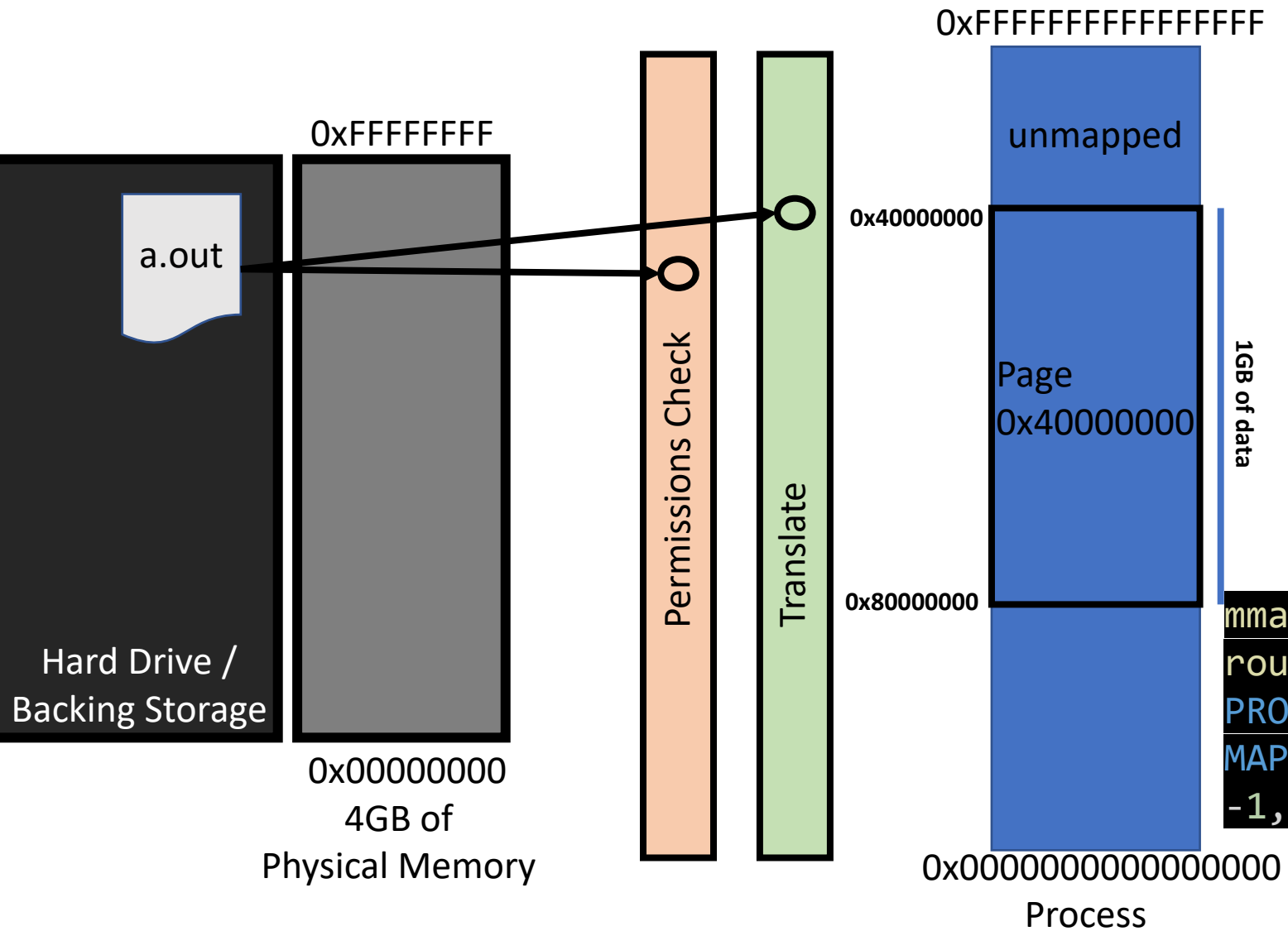
The Core i7 memory system.

Separate L1 data & instruction (L1) TLB
Size: 64 * 8B = 512B; 128 * 8B = 1024B
Max Data Reach: 64 * 1GB = 64GB data
Code Reach: 128 * 4kB = 512kB code
Implication of this TLB organization

Unified (L2) TLB
Size: 512 * 8B = 4kB
Max Reach: 512 * 1GB = 512GB code+data

Awesome Property of the TLB!
Huge pages make reach of L1TLB be 64GB!

Increasing Page Size to Increase TLB Reach



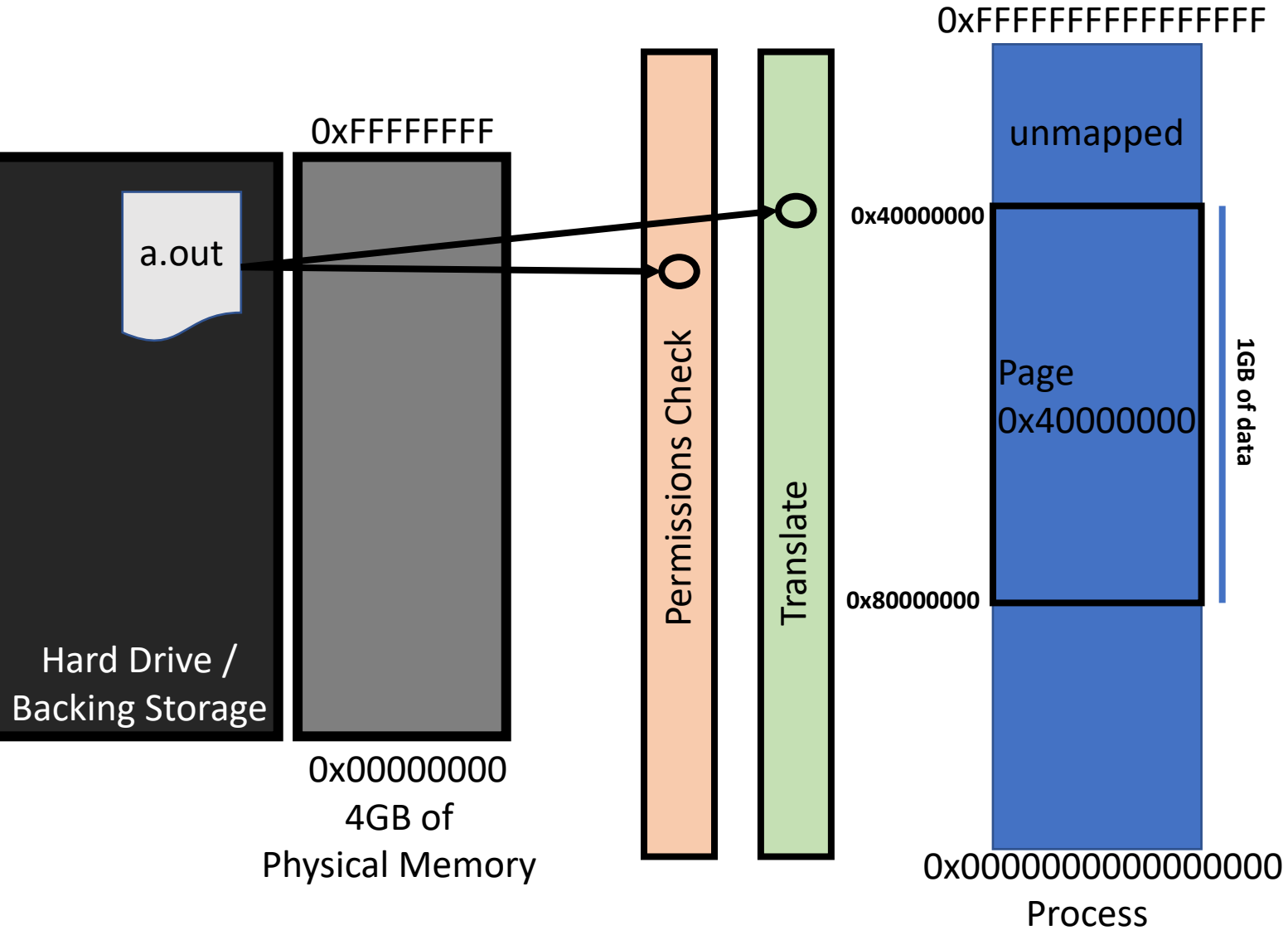
Modern Oses support large pages and huge pages via mmap (& other ways).

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



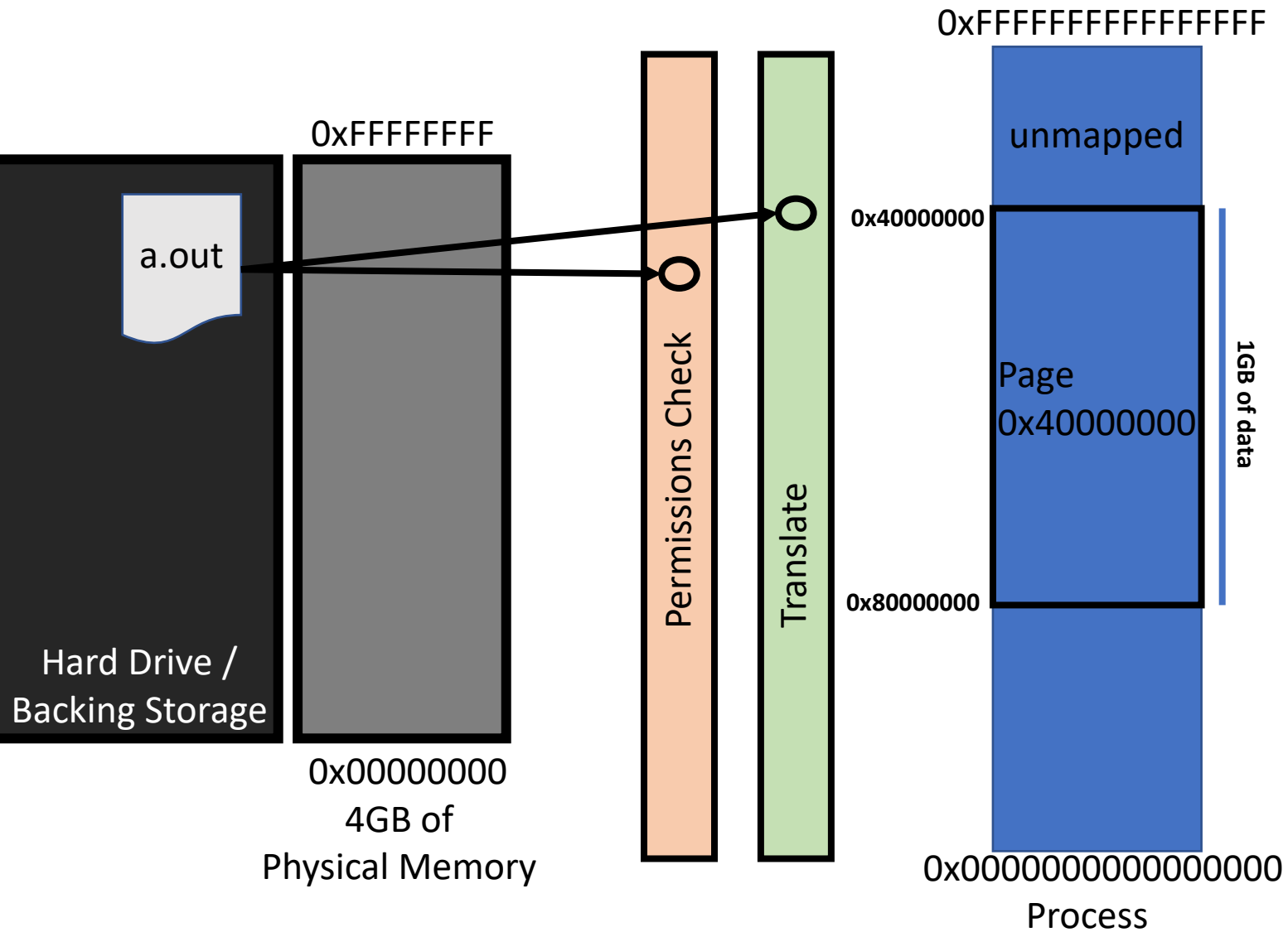
"Transparent" huge pages allow the OS to promote a normal page to HUGE status

```
madvise(..., MADV_HUGEPAGE);
```

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

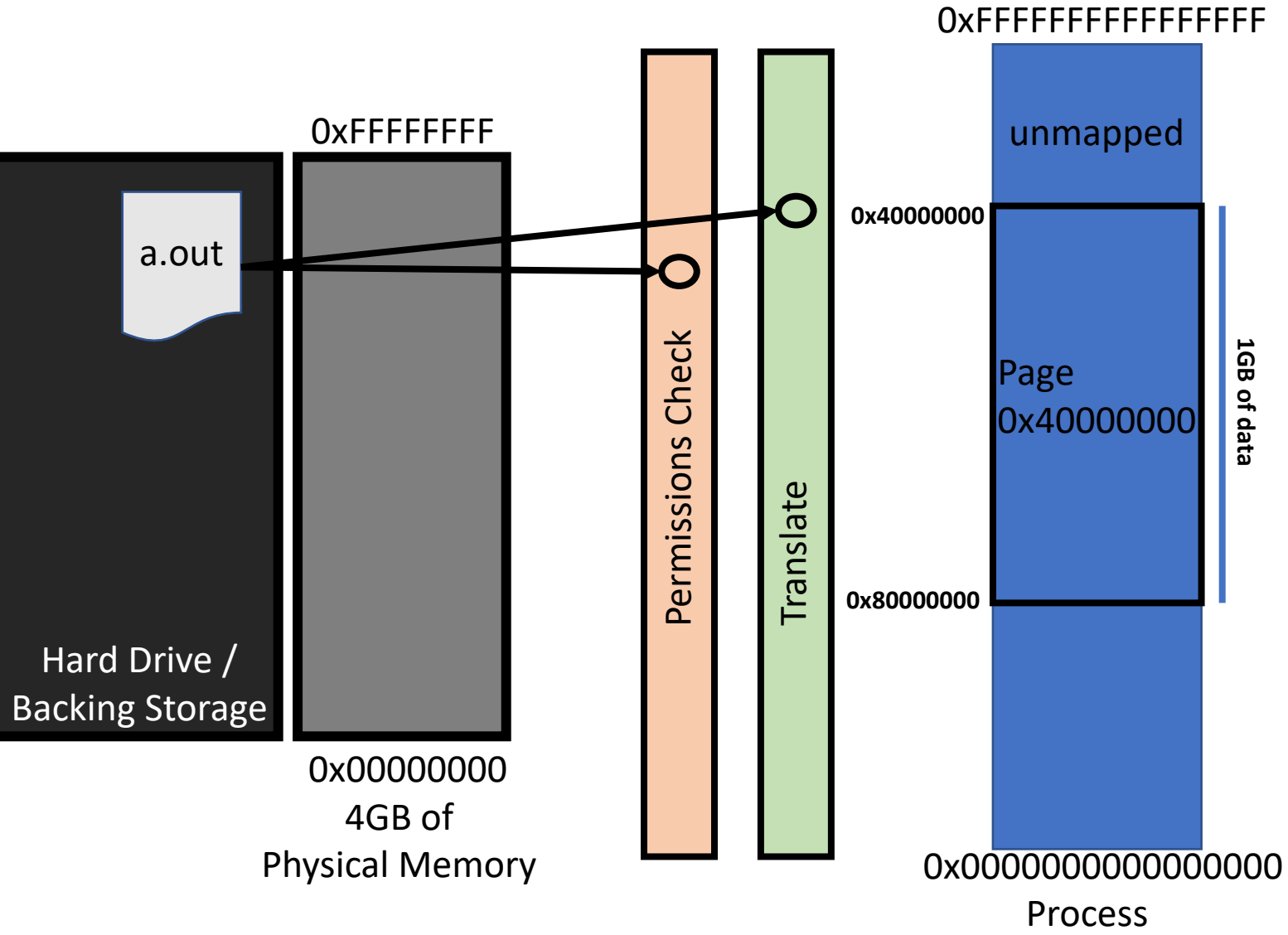
```
int posix_memalign(void **memptr,
size_t alignment, size_t size);
```

Increasing Page Size to Increase TLB Reach



Risks / Costs of Increasing Page Size?

Increasing Page Size to Increase TLB Reach



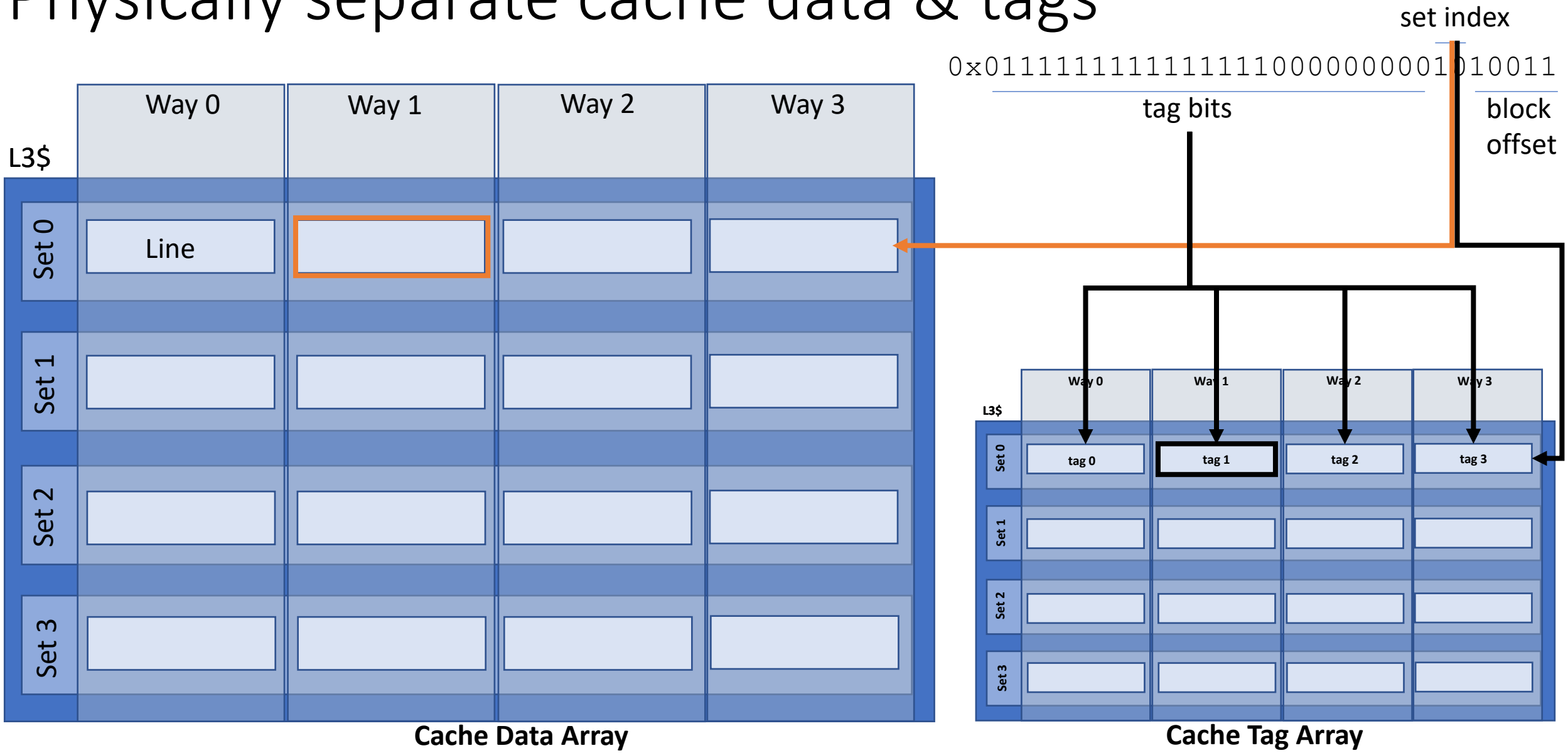
Risks / Costs of Increasing Page Size?

- **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!

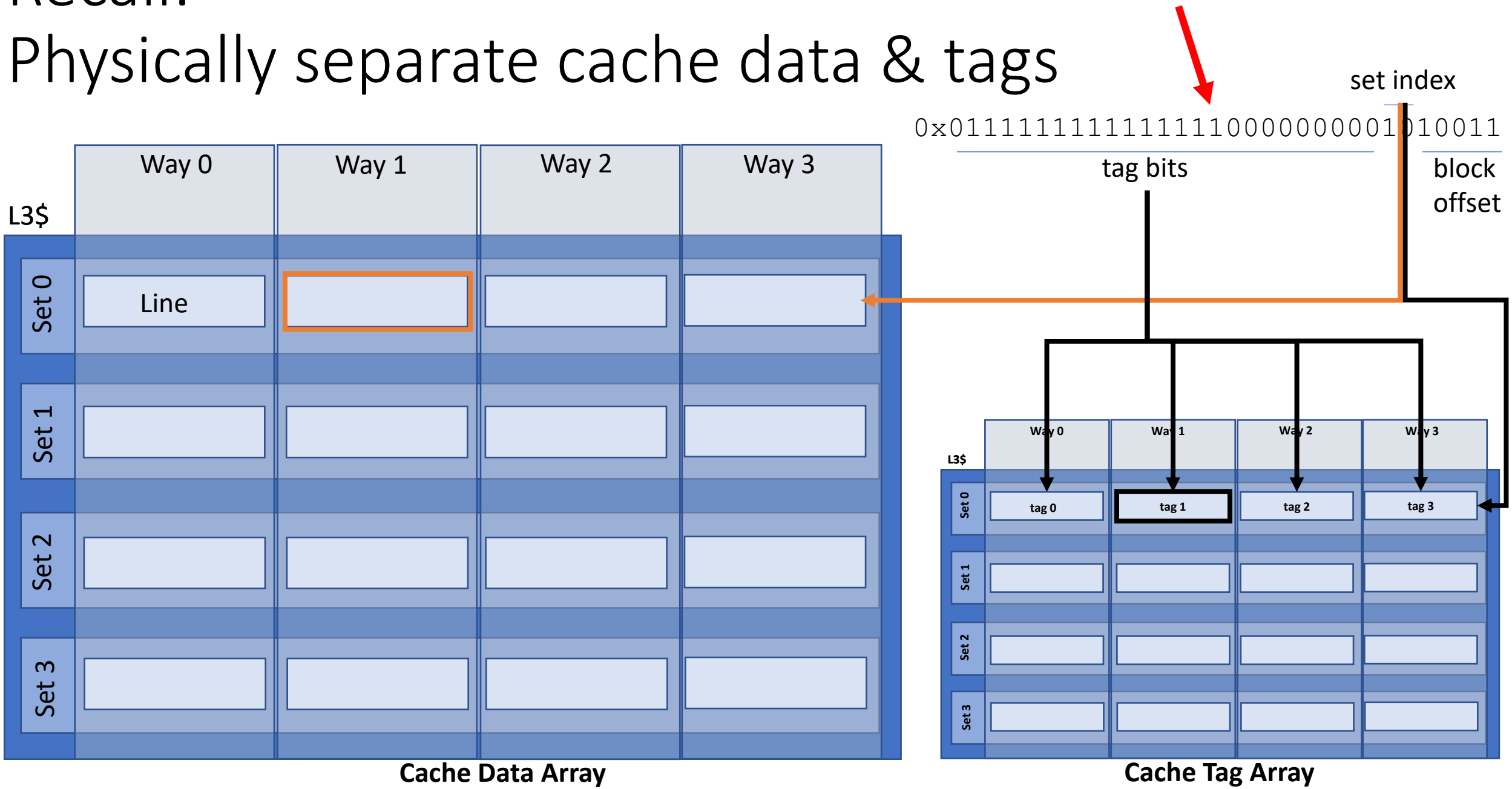
How Do Virtual Memory and Caching Interact?

Recall:
Physically separate cache data & tags



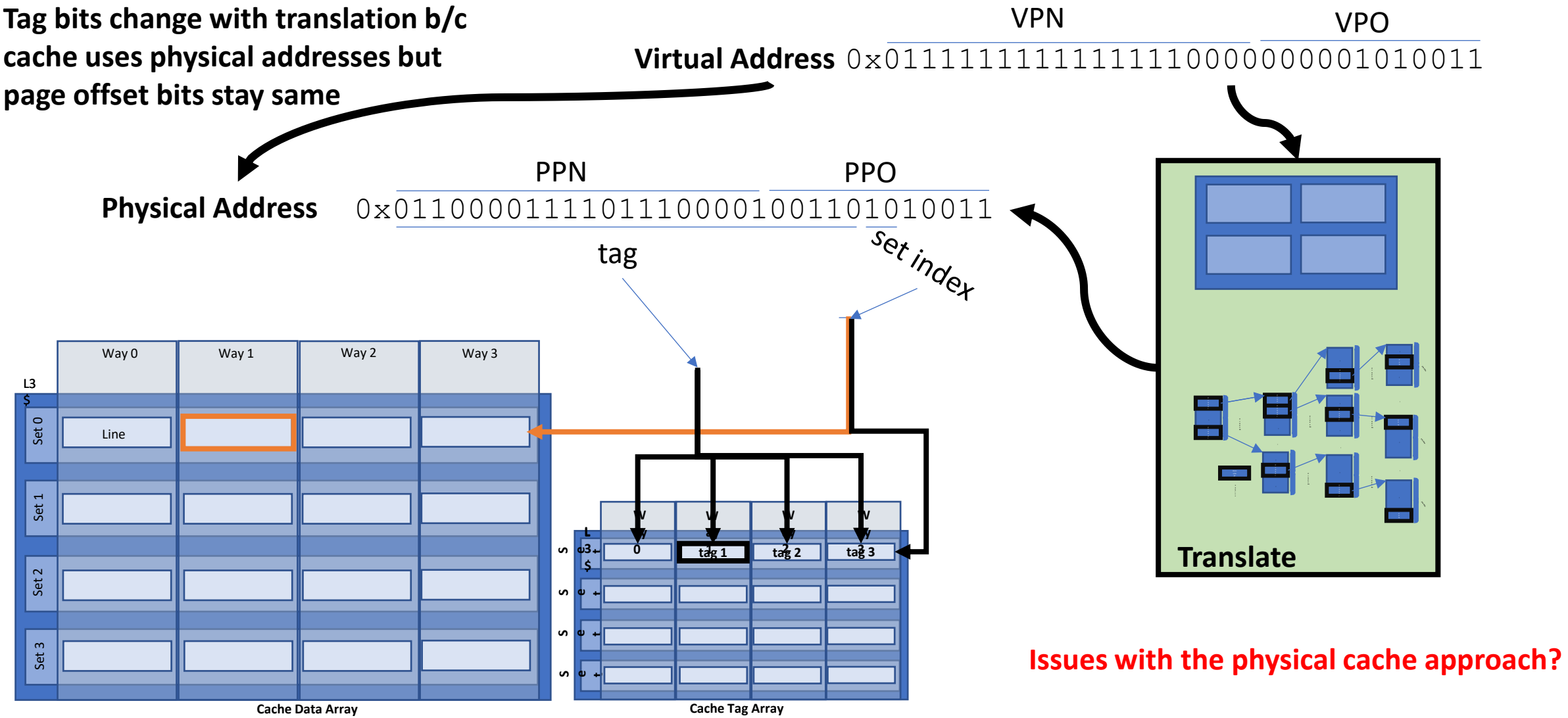
Recall:
Physically separate cache data & tags

Question: Virtual or Physical Address?



Physical Cache: Translate First Then Access Cache

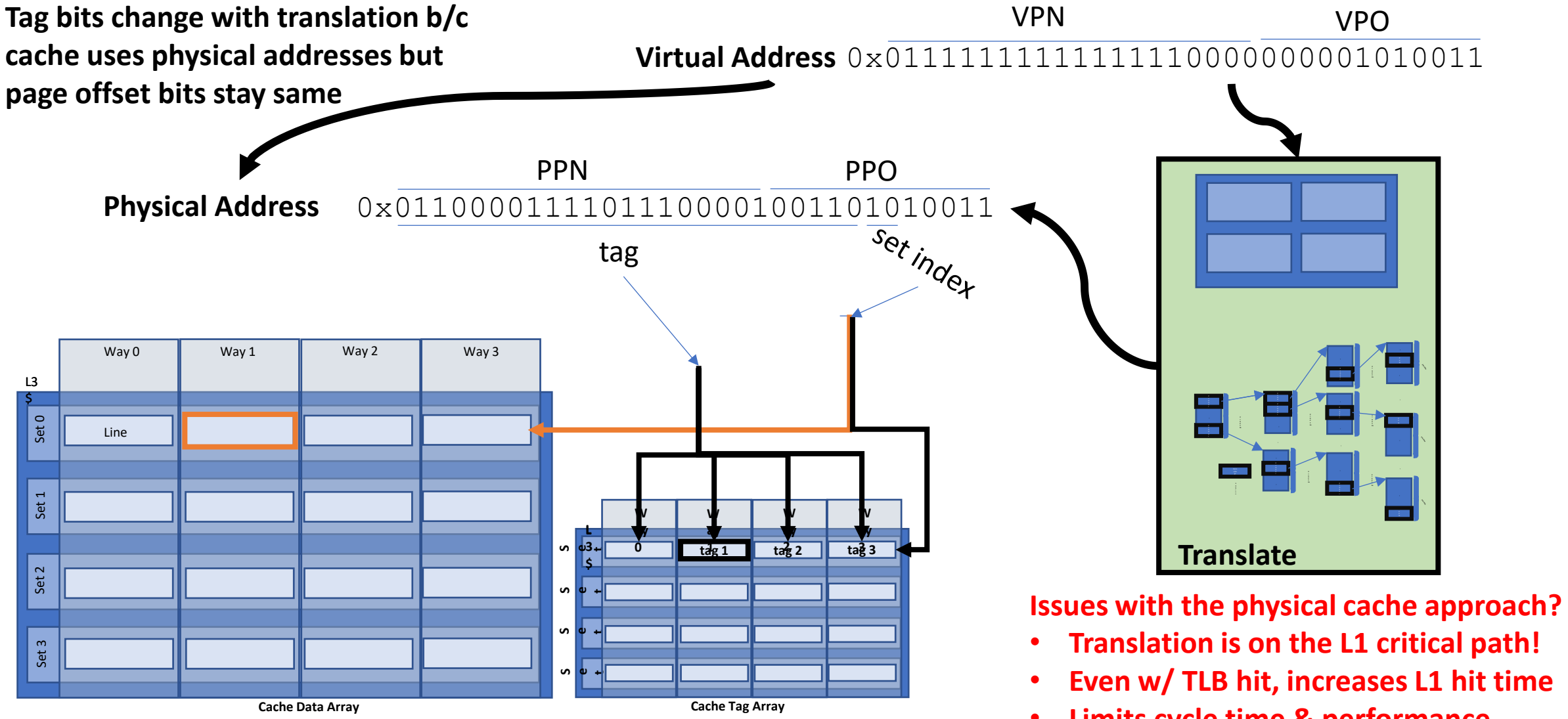
Tag bits change with translation b/c cache uses physical addresses but page offset bits stay same



Issues with the physical cache approach?

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



Issues with the physical cache approach?

- **Translation is on the L1 critical path!**
- **Even w/ TLB hit, increases L1 hit time**
- **Limits cycle time & performance**

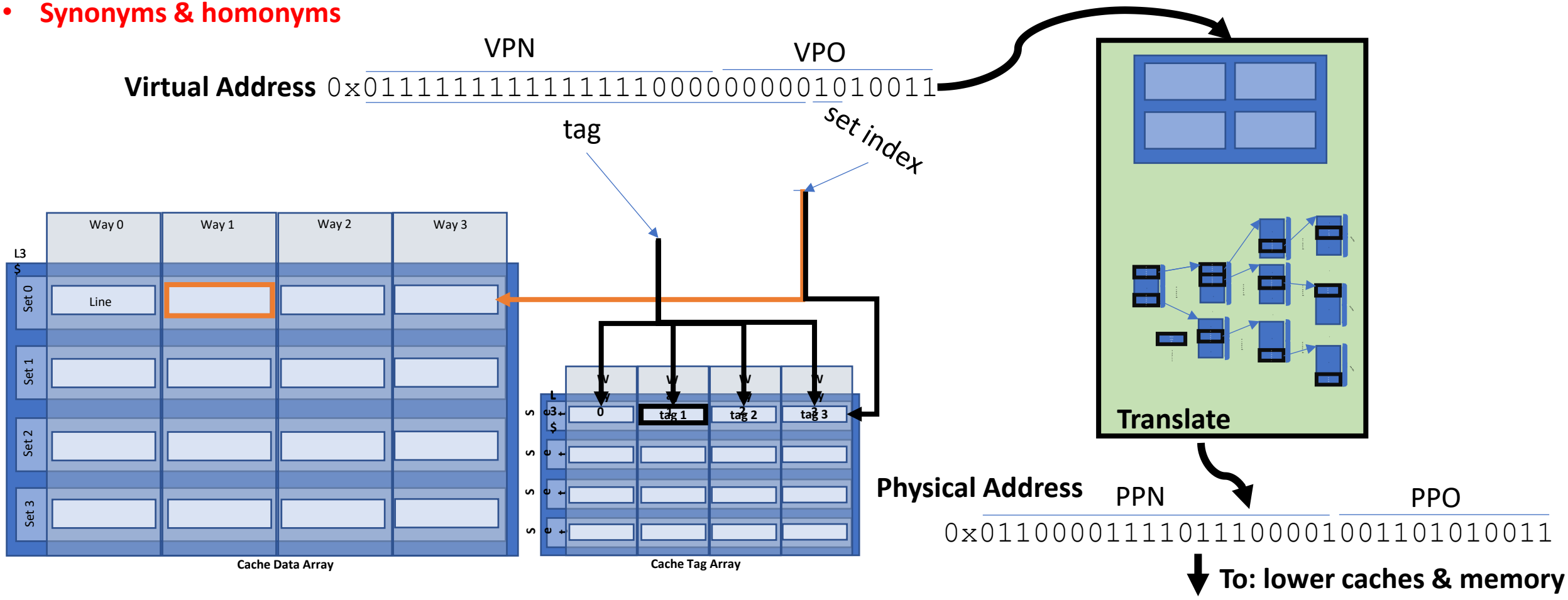
Virtual Cache: Access Cache Then Translate (VIVT: Virtually Indexed, Virtually Tagged)

Benefits of the virtual cache approach?

- **Parallelize cache lookup & translate**

Costs of the virtual cache approach?

- **Synonyms & homonyms**

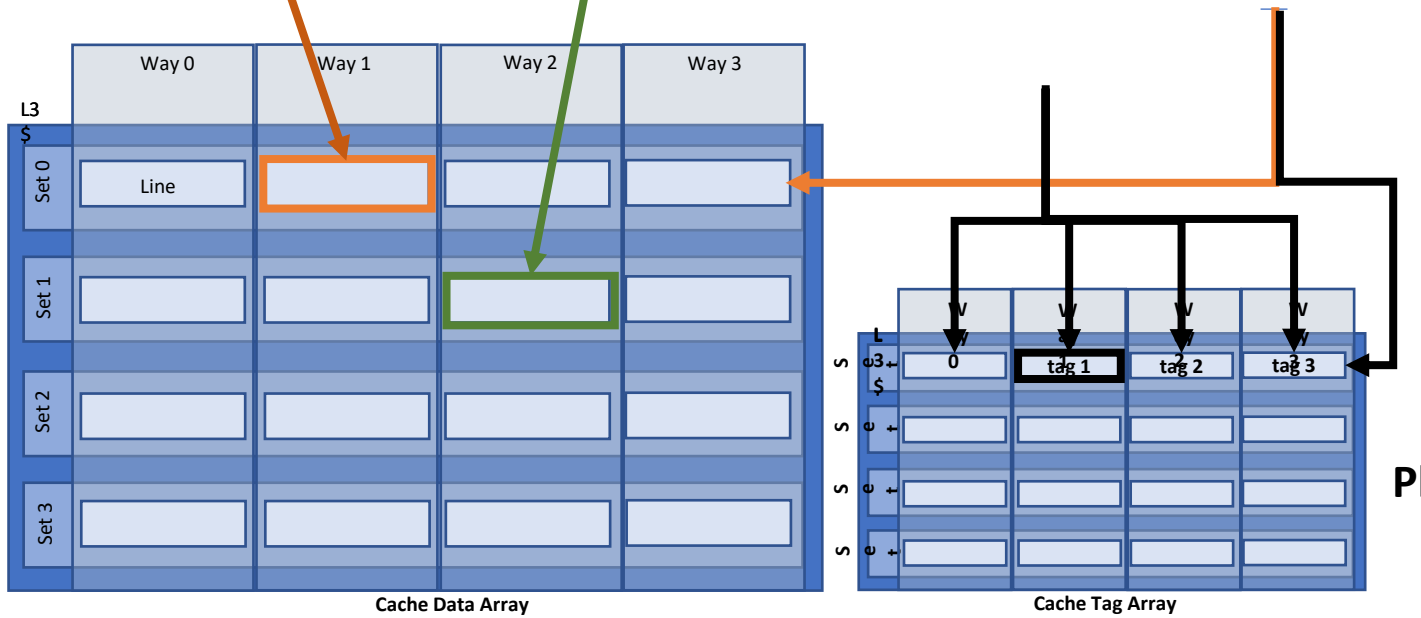


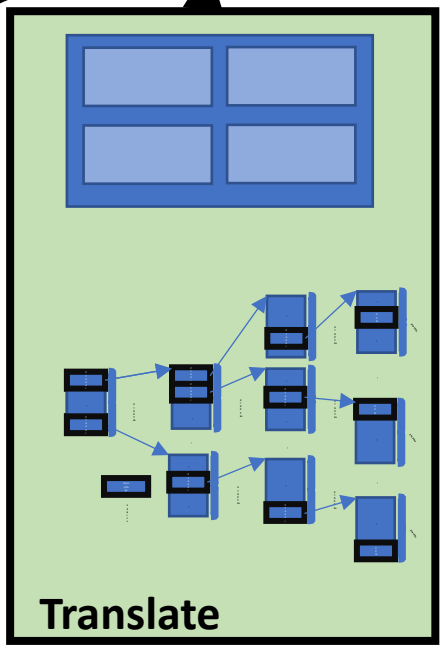
Virtual Caches: The Synonym Problem

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

Process 1:
Virtual Address 1 VPN VPO
0x01111111111111110000111000010010

Process 2:
Virtual Address 1 VPN VPO
0x01111111111111110000000000110011





Physical Address PPN PPO
0x011000011110111000010011010010011

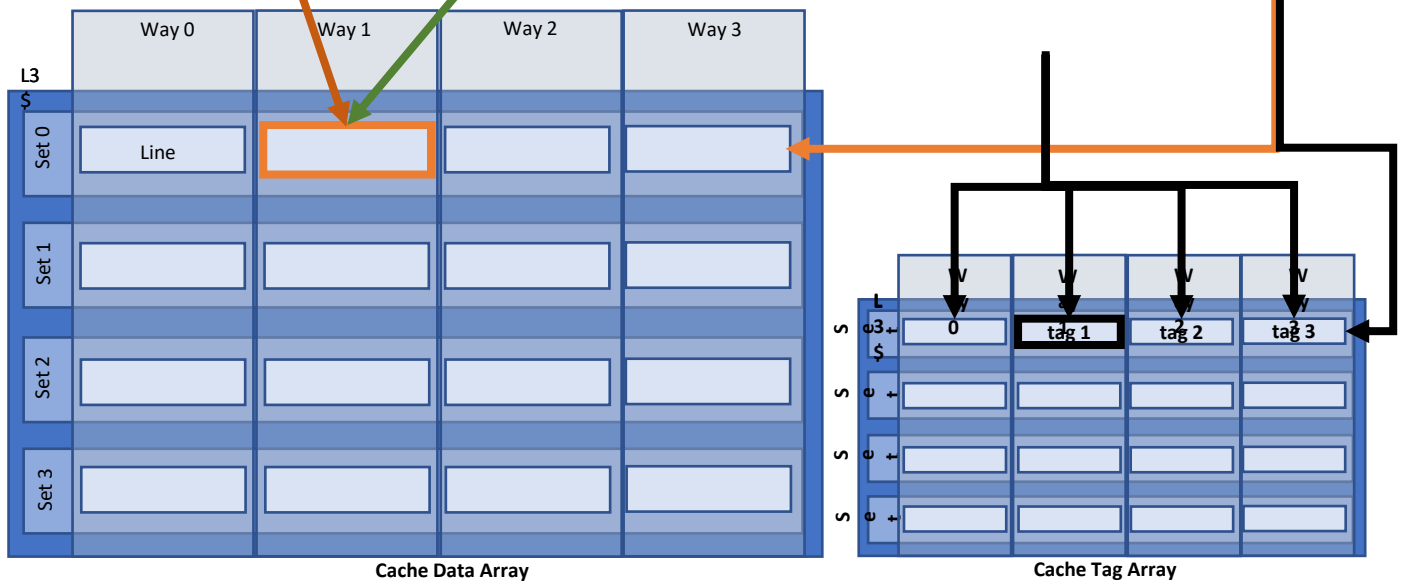
To: lower caches & memory

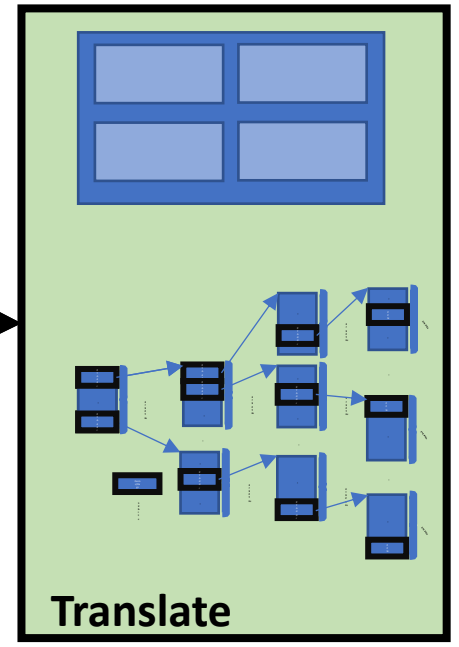
Virtual Caches: The Homonym Problem

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

Process 1:
Virtual Address 1 VPN VPO
0x01111111111111110000011000010011

Process 2:
Virtual Address 2 VPN VPO
0x01111111111111110000000000110011





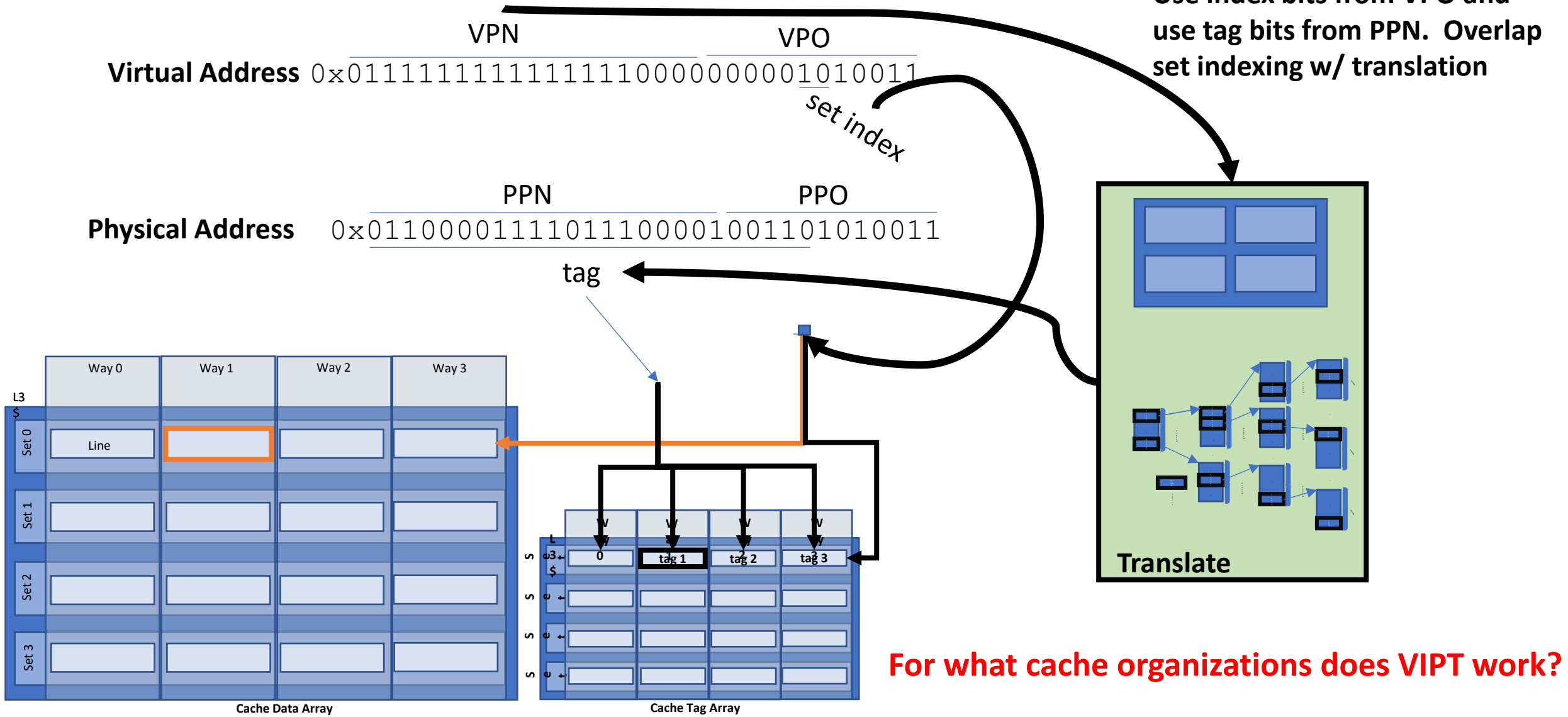
Physical Address 1
0x01100001111011100001001101010011

Physical Address 2
0x01000000111110000010011010010011

To: lower caches & memory

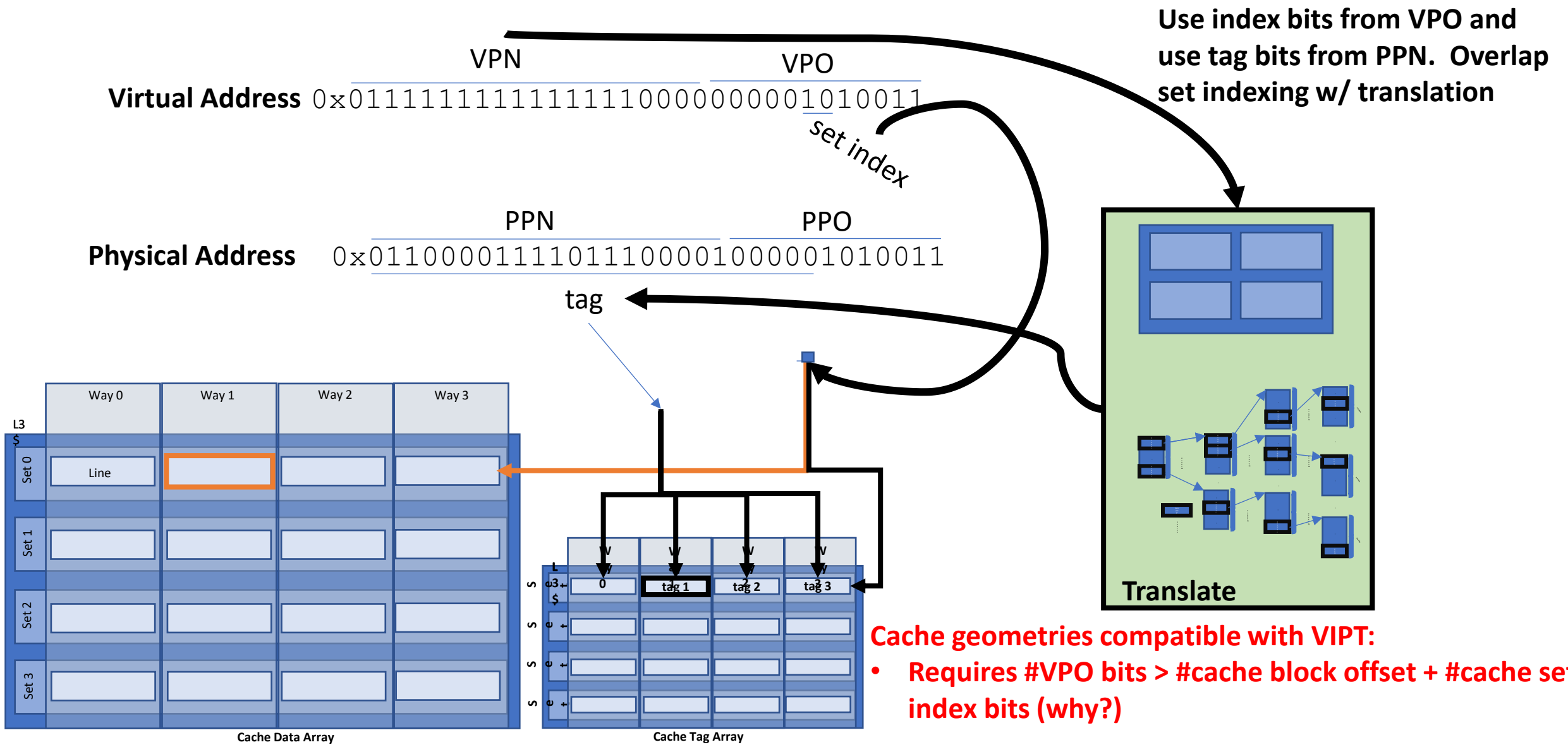
Virtually Indexed, Physically Tagged Caches

Use index bits from VPO and use tag bits from PPN. Overlap set indexing w/ translation



For what cache organizations does VIPT work?

Virtually Indexed, Physically Tagged Caches



Cache geometries compatible with VIPT:

- **Requires #VPO bits > #cache block offset + #cache set index bits (why?)**

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