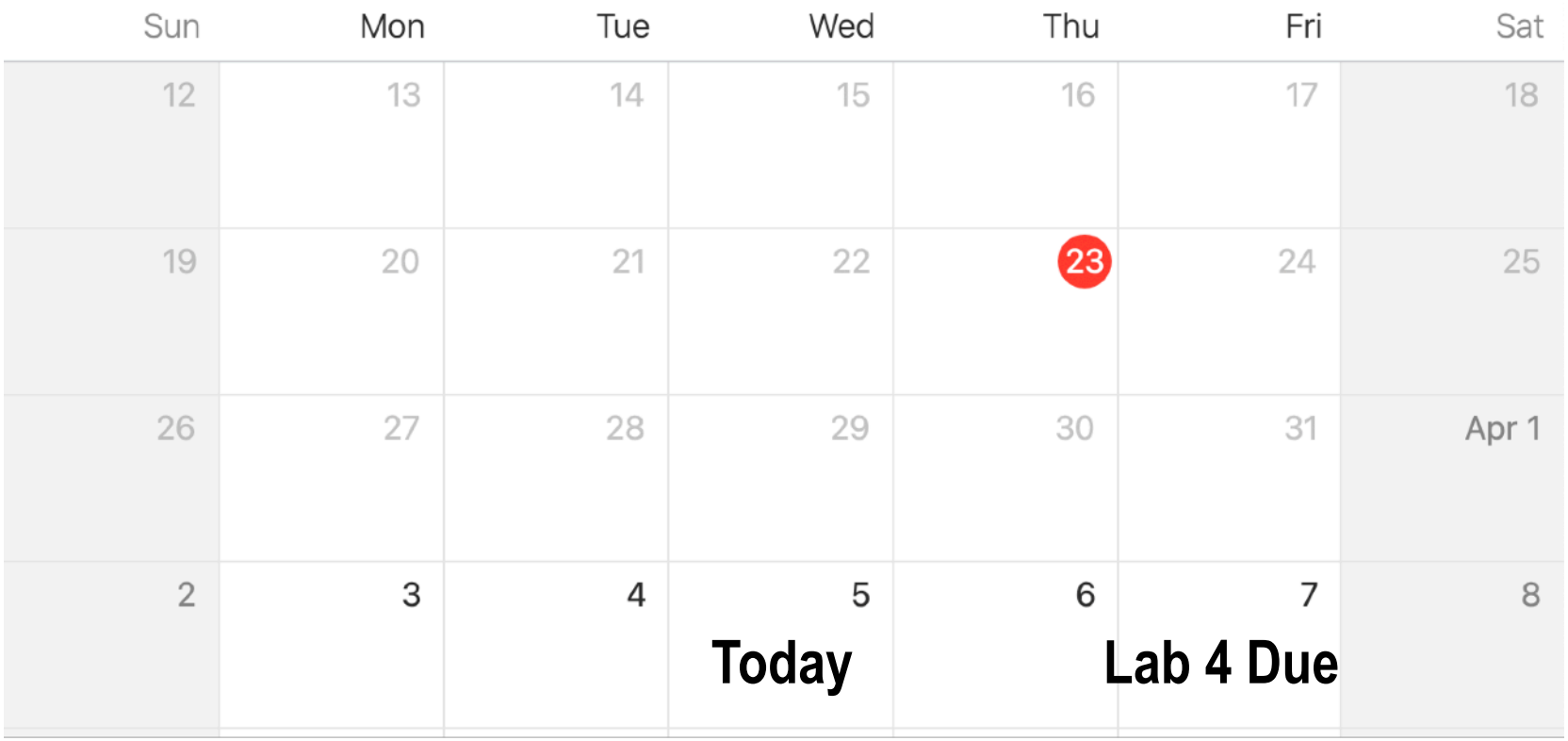
CSC 252: Computer Organization
Spring 2023: Lecture 22

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcements

- Cache problem set: https://www.cs.rochester.edu/courses/252/spring2023/handouts.html Won't be graded.
- Mid-term solution posted: https://www.cs.rochester.edu/courses/252/spring2023/handouts.html



Today

- Three Virtual Memory Optimizations
  - TLB
  - Virtually-indexed, physically-tagged cache
  - Page the page table (a.k.a., multi-level page table)
- Case-study: Intel Core i7/Linux example

Speeding up Address Translation

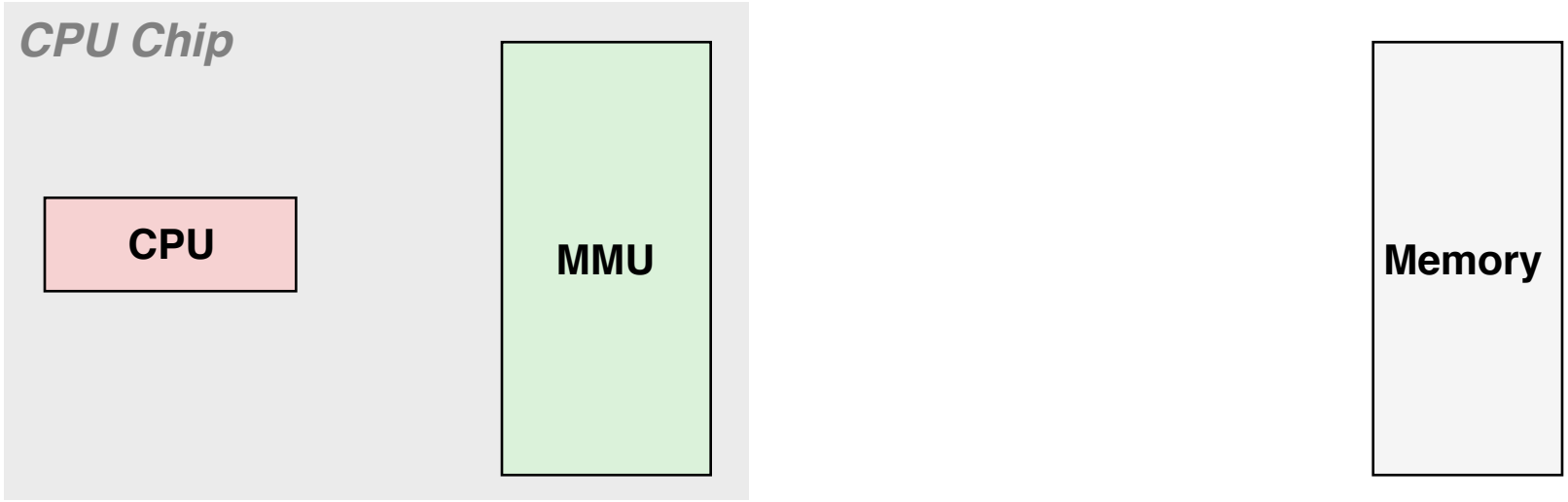
Speeding up Address Translation

- Problem: Every memory load/store requires two memory accesses: one for PTE, another for real
  - The PTE access is kind of an overhead
  - Can we speed it up?

Speeding up Address Translation

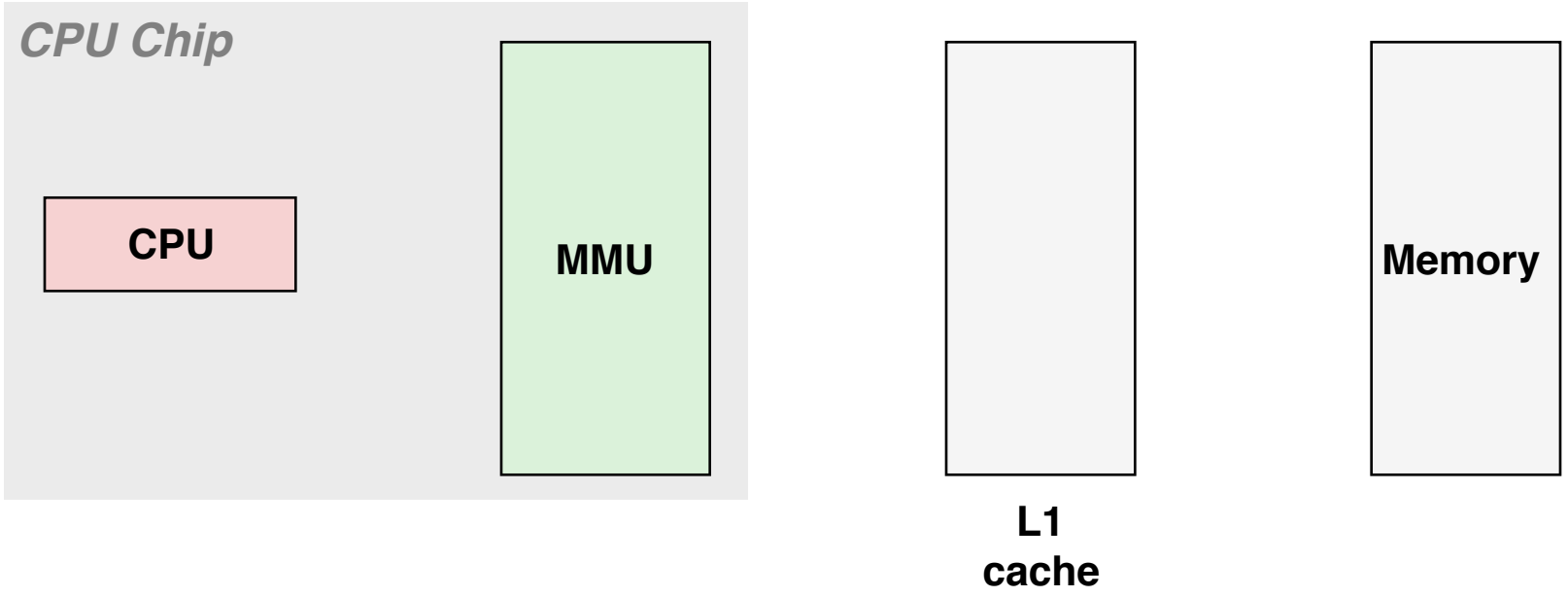
- Problem: Every memory load/store requires two memory accesses: one for PTE, another for real
  - The PTE access is kind of an overhead
  - Can we speed it up?
- Page table entries (PTEs) are already cached in L1 data cache like any other memory data. But:
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay

Recall: Page Table is Cached



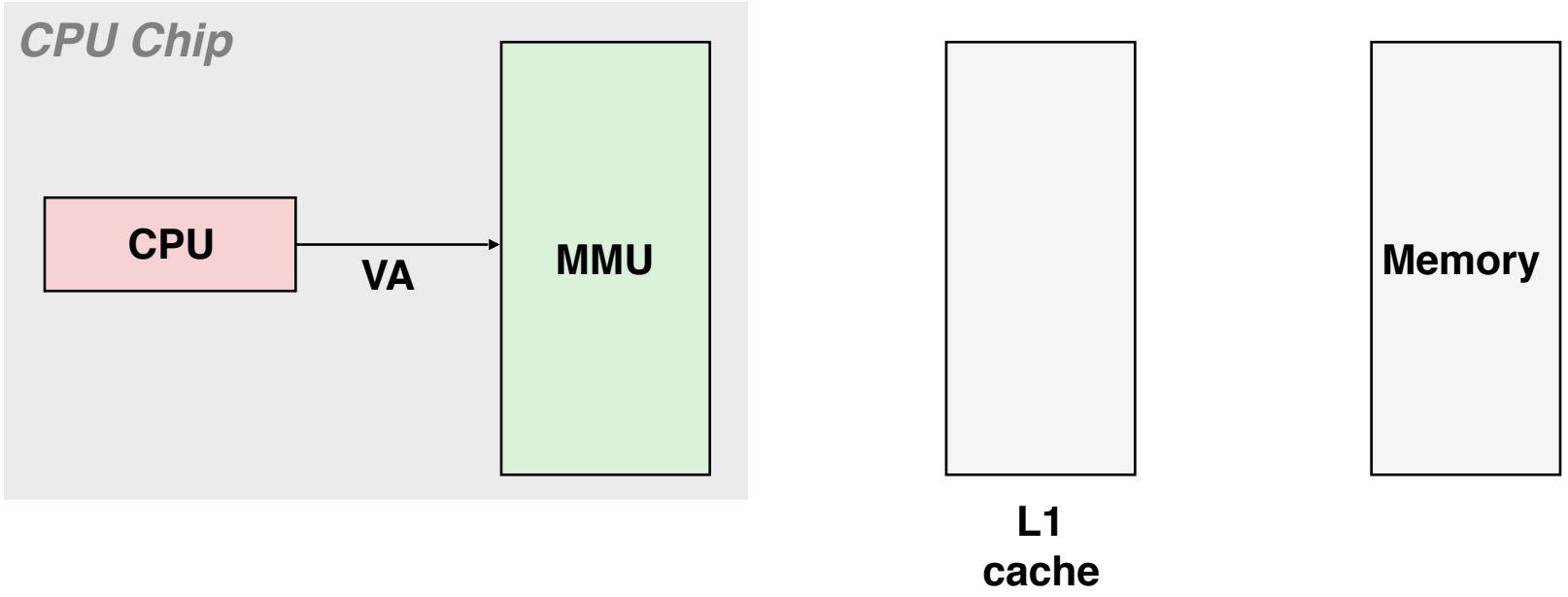
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



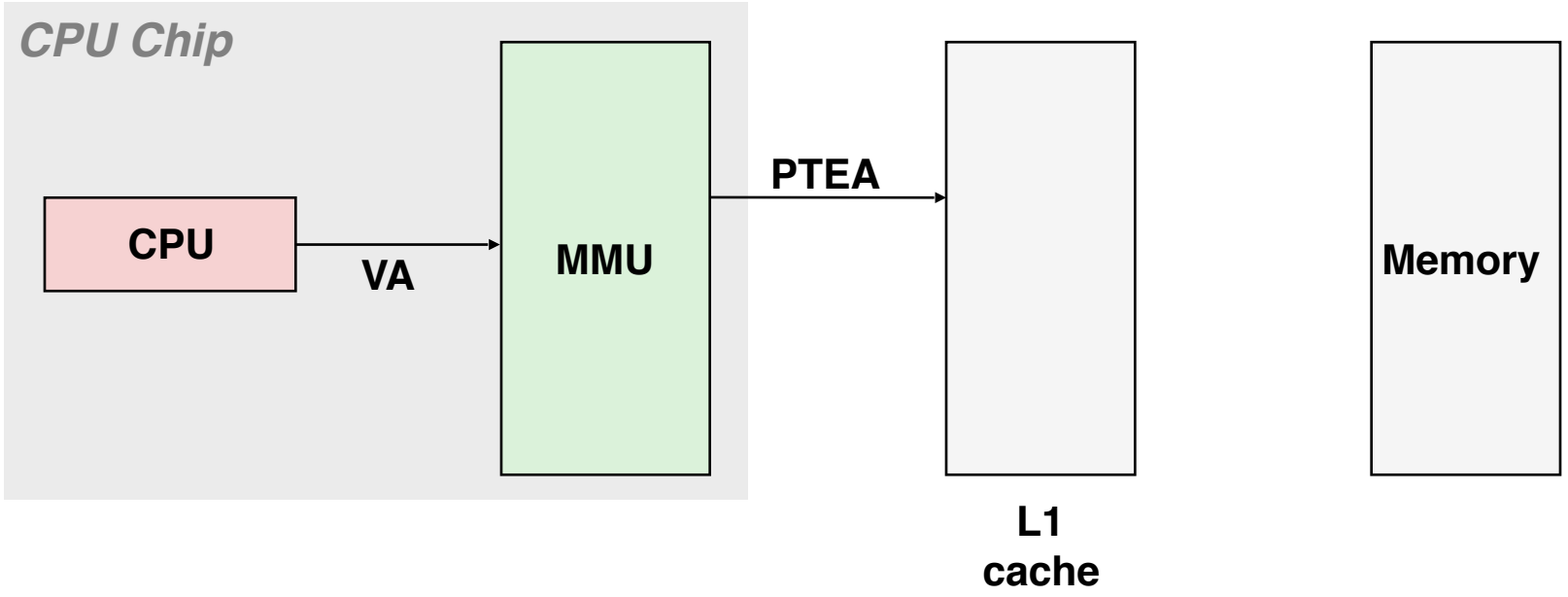
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



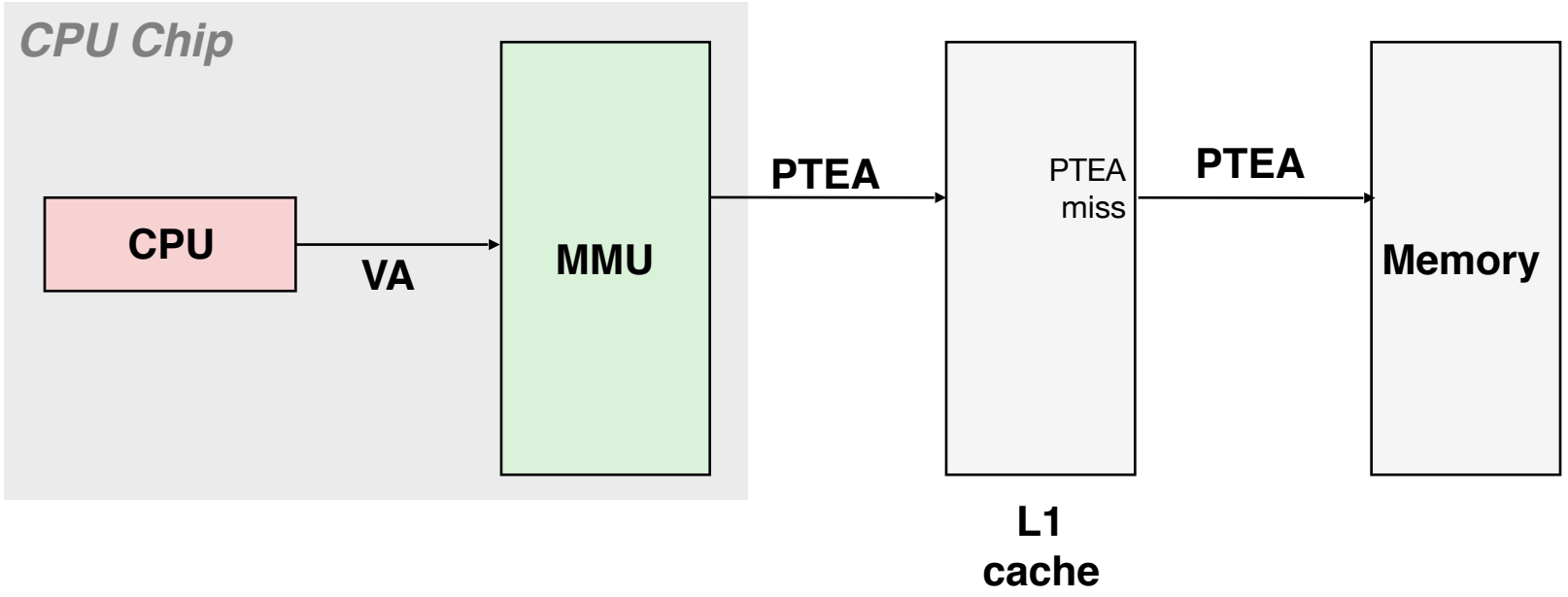
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



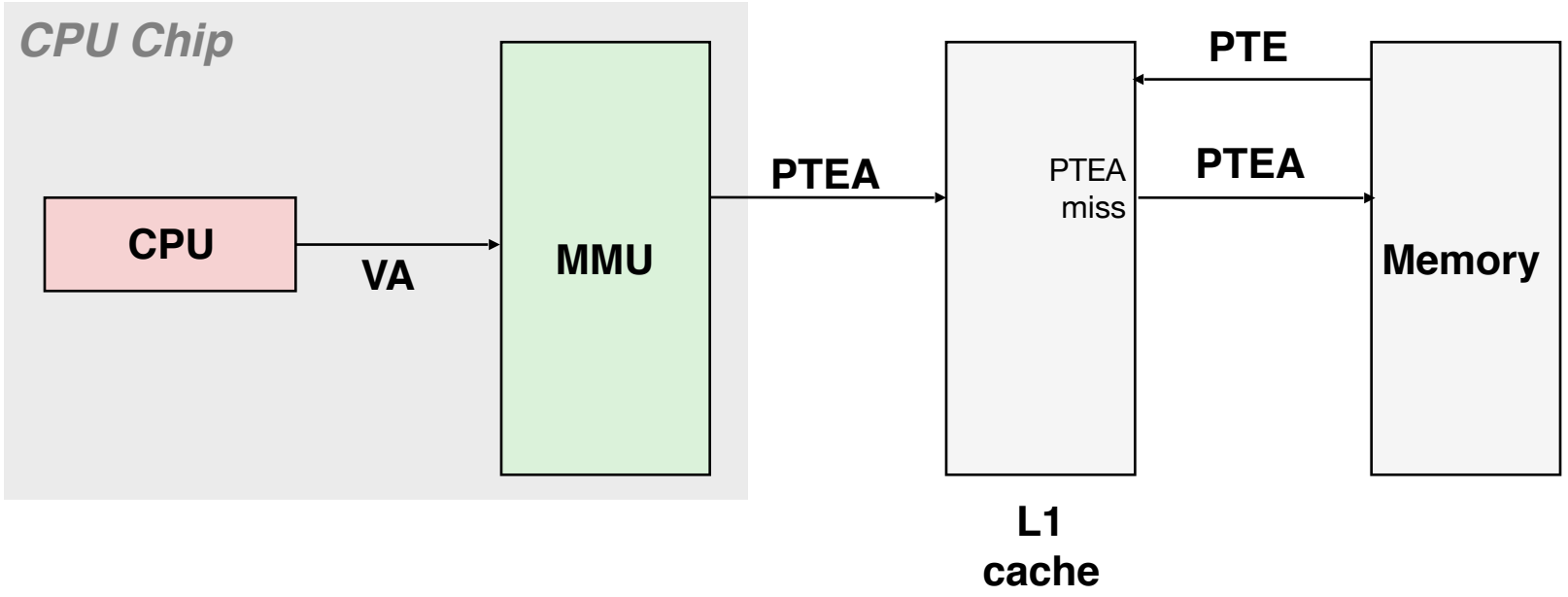
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



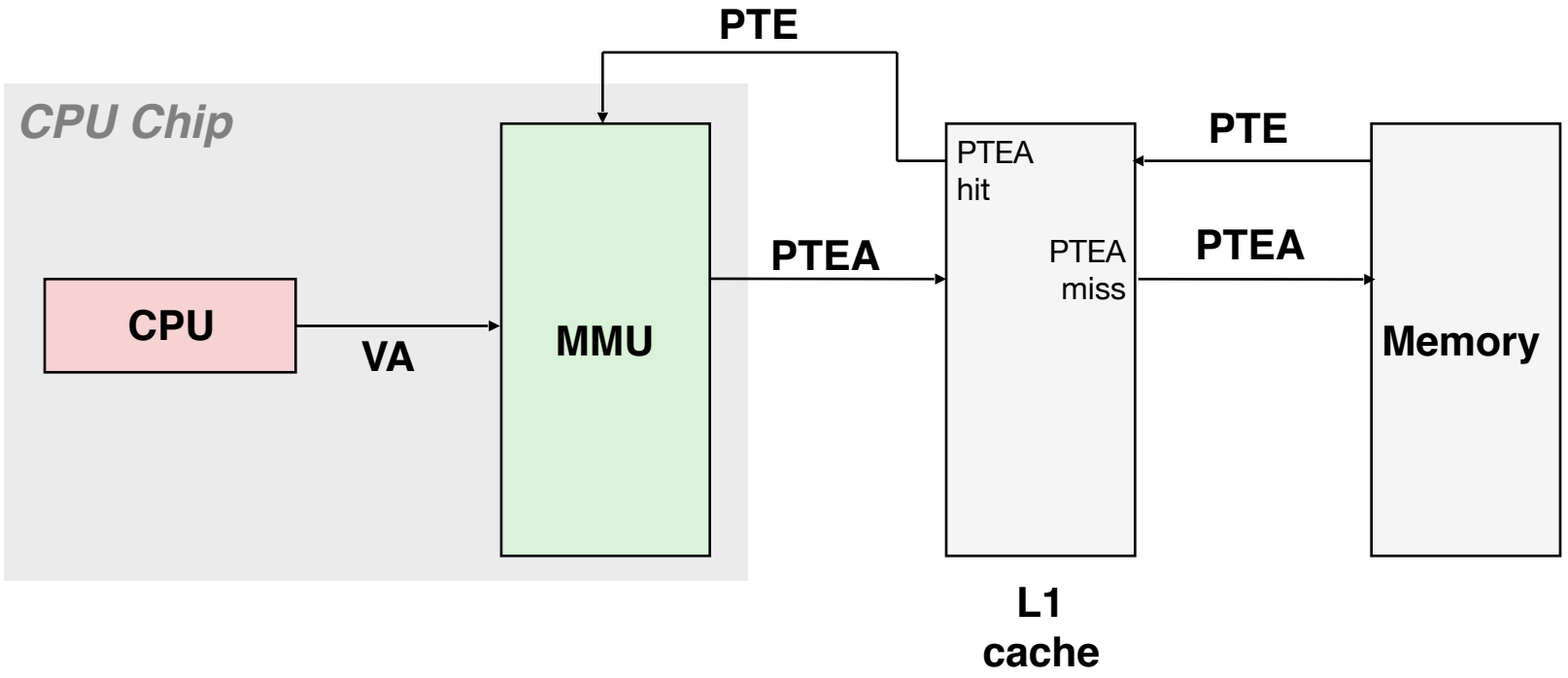
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



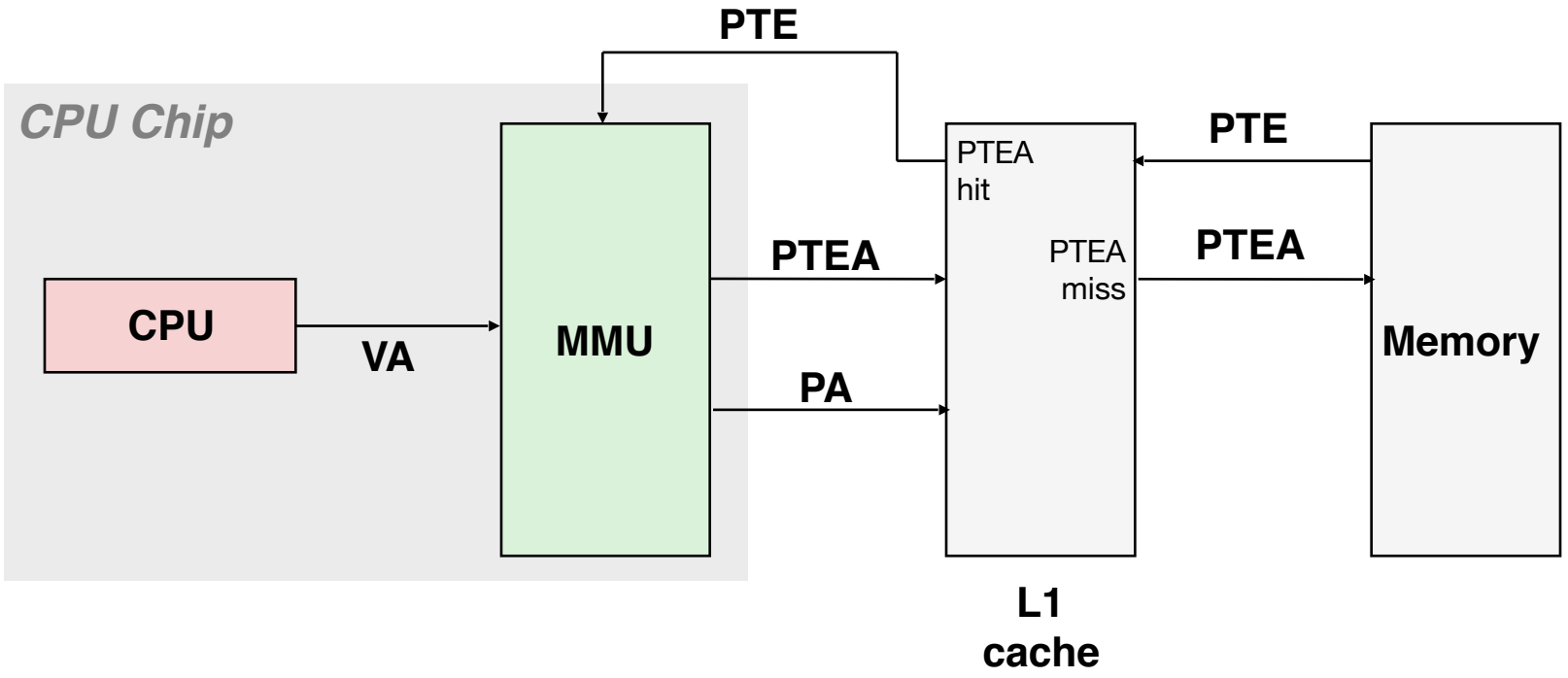
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



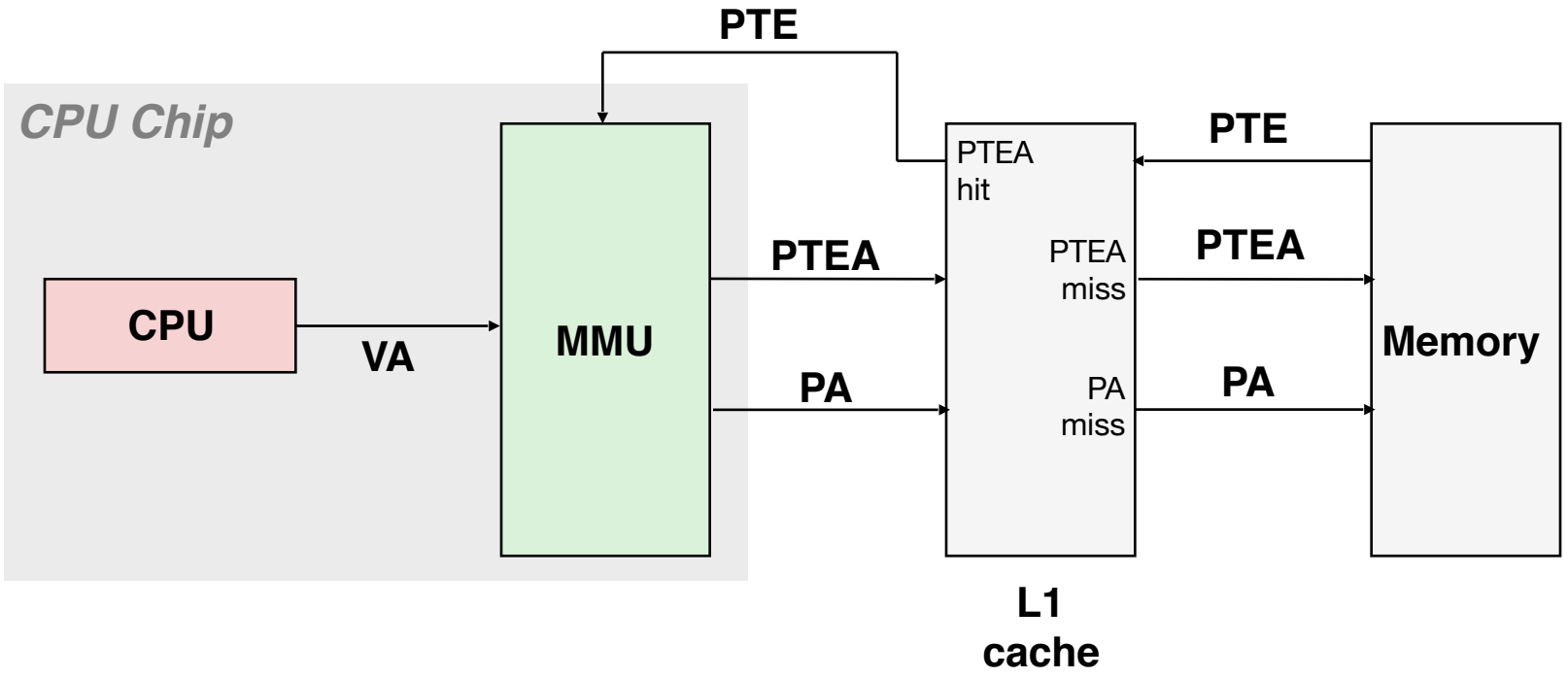
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



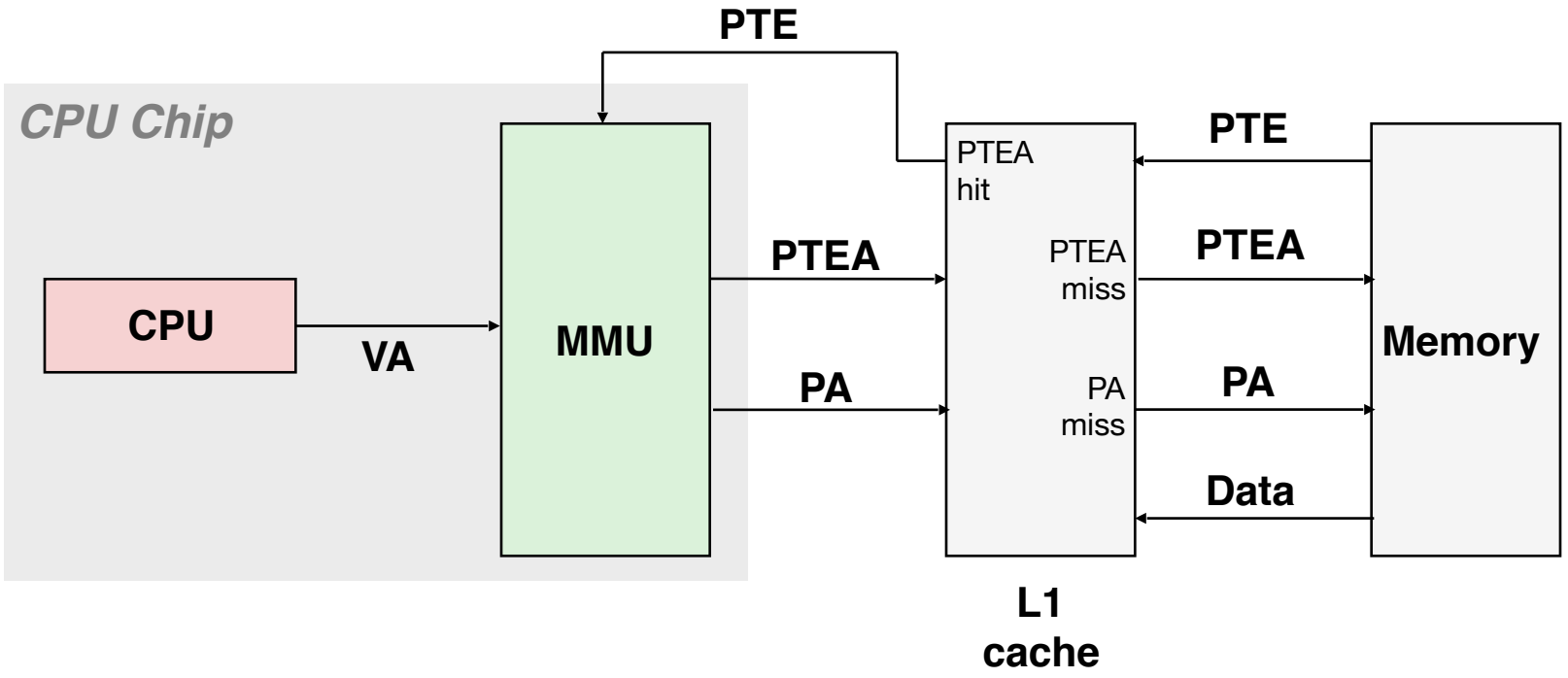
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



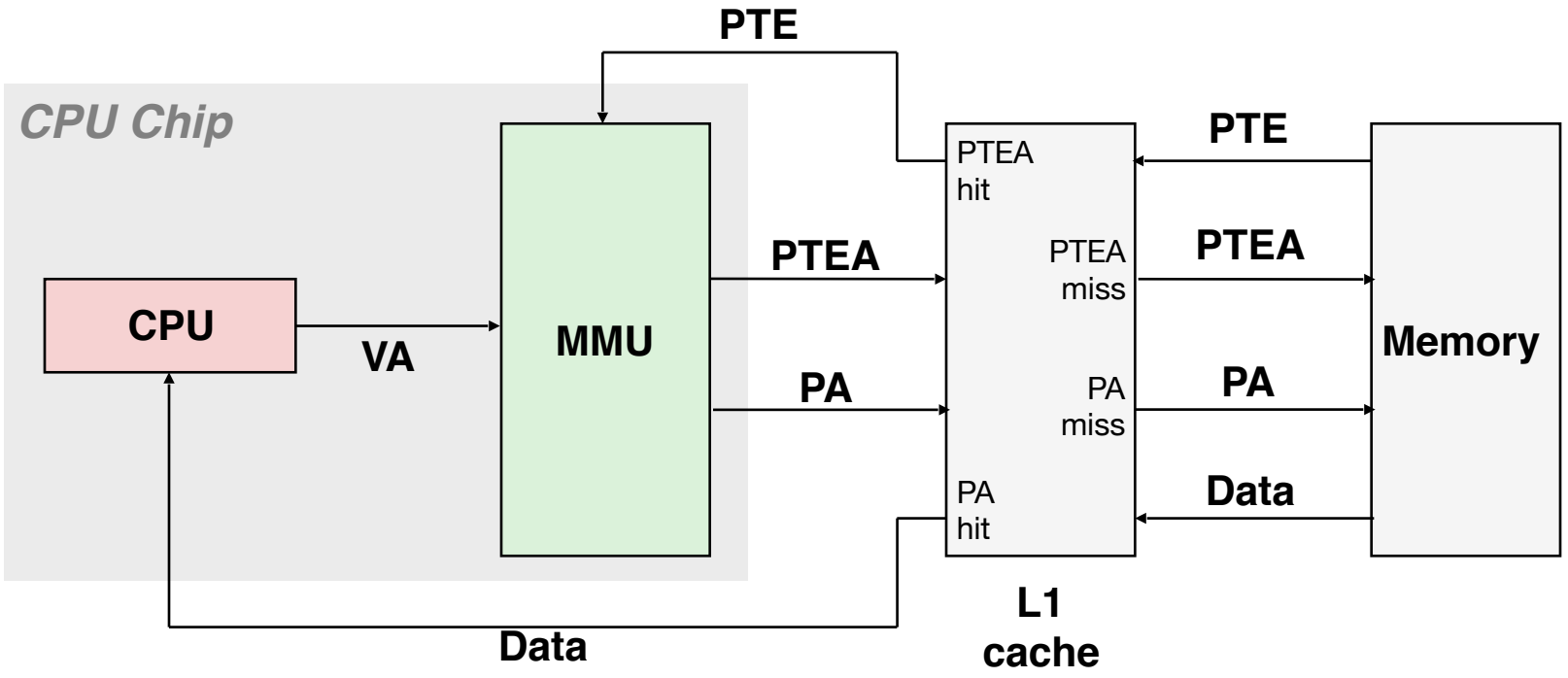
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



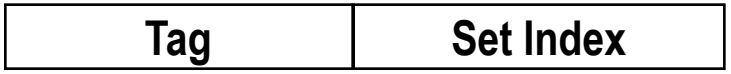
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Speeding up Translation with a TLB

- Solution: *Translation Lookaside Buffer* (TLB)
  - Think of it as a dedicated cache for page table
  - Small set-associative hardware cache in MMU
  - Contains complete page table entries for a small number of pages

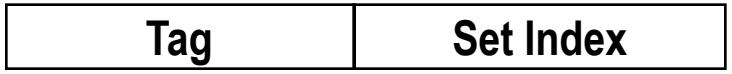
Speeding up Translation with a TLB

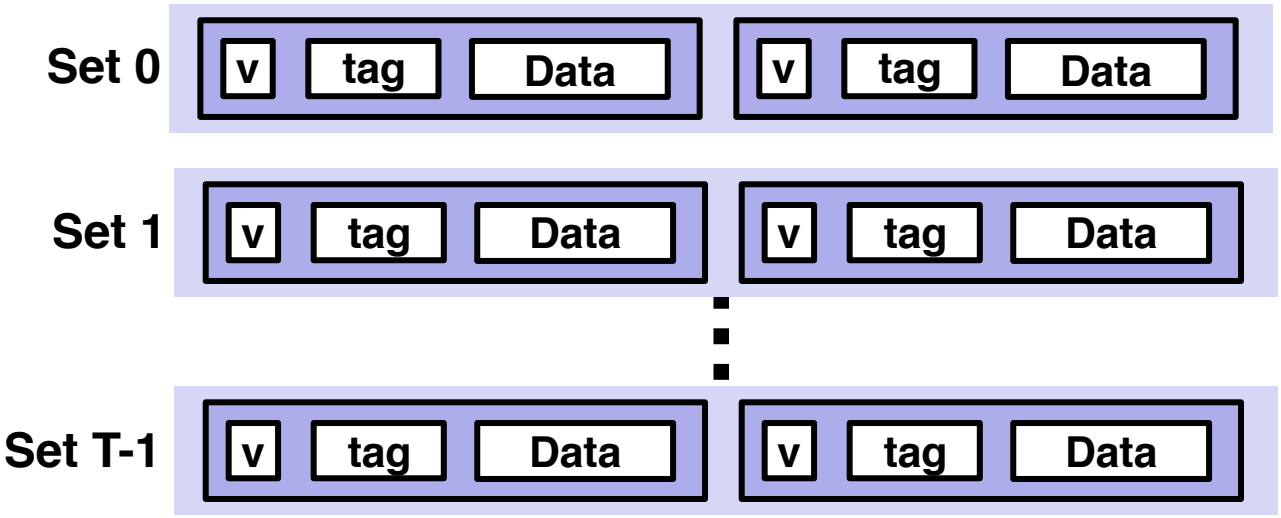
- Solution: *Translation Lookaside Buffer* (TLB)
  - Think of it as a dedicated cache for page table
  - Small set-associative hardware cache in MMU
  - Contains complete page table entries for a small number of pages



Speeding up Translation with a TLB

- Solution: *Translation Lookaside Buffer* (TLB)
  - Think of it as a dedicated cache for page table
  - Small set-associative hardware cache in MMU
  - Contains complete page table entries for a small number of pages

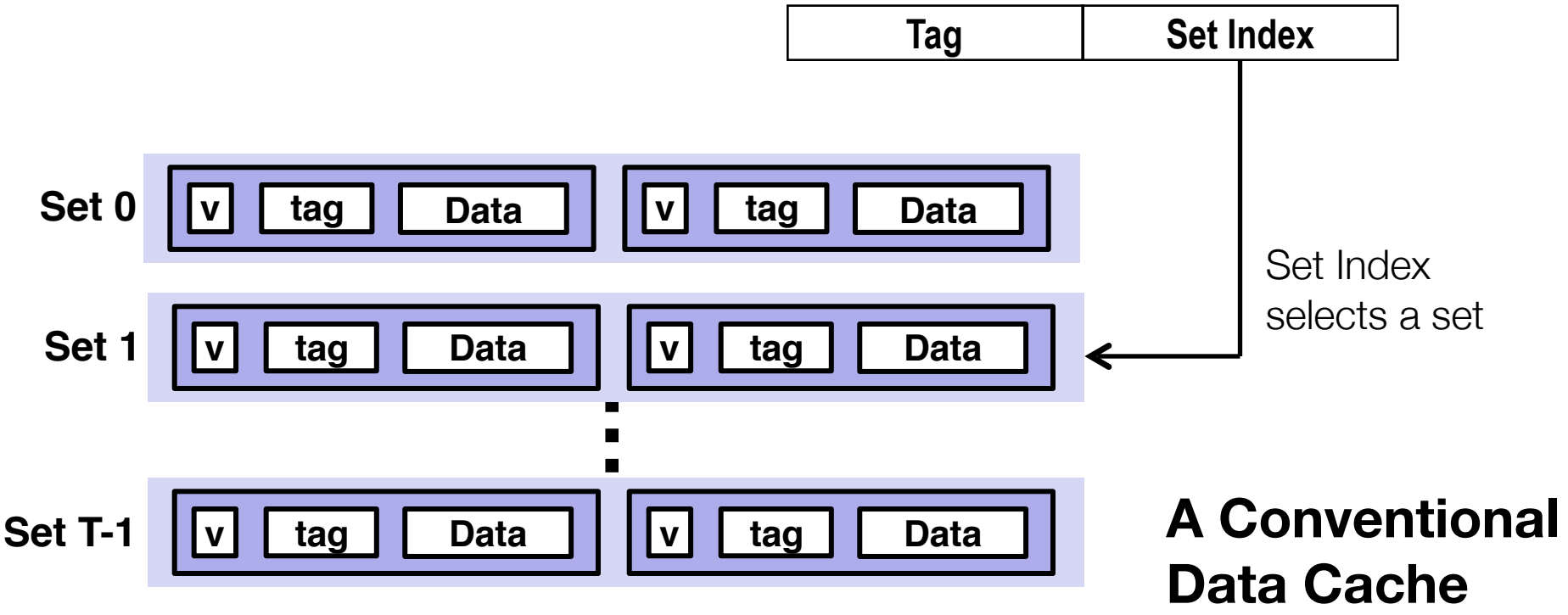




A Conventional Data Cache

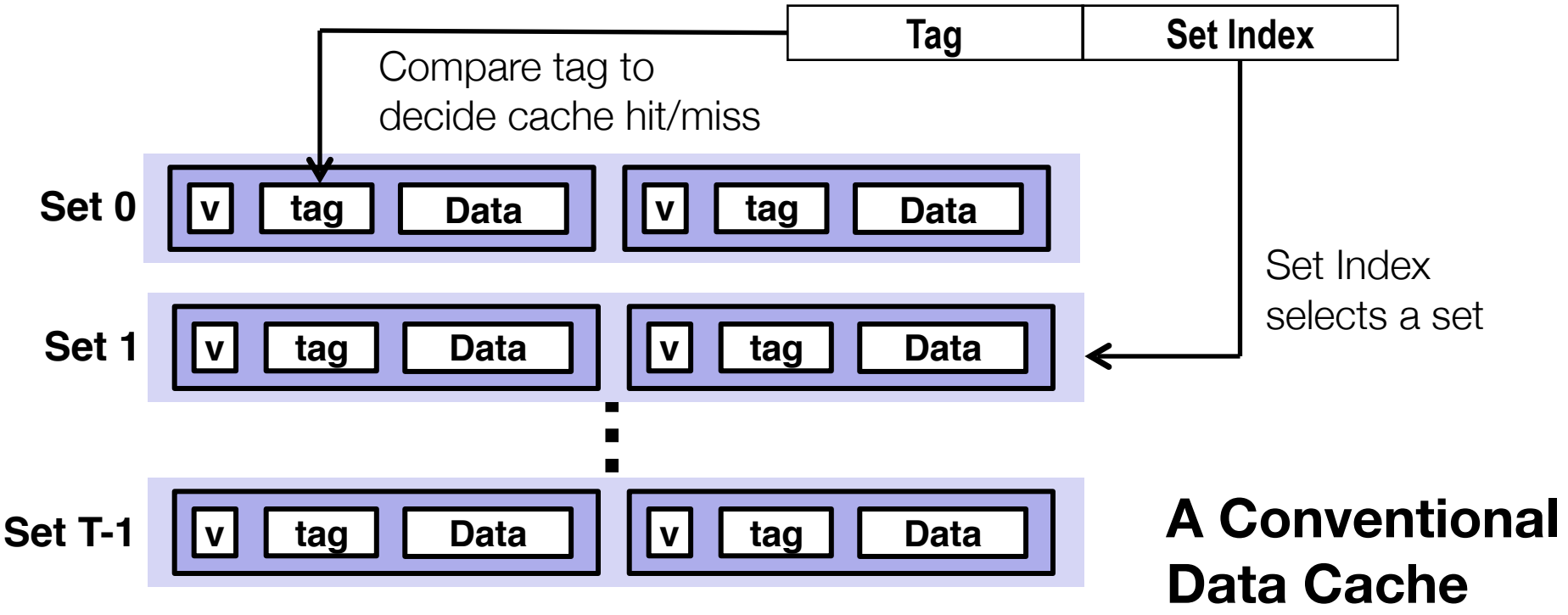
Speeding up Translation with a TLB

- Solution: *Translation Lookaside Buffer* (TLB)
  - Think of it as a dedicated cache for page table
  - Small set-associative hardware cache in MMU
  - Contains complete page table entries for a small number of pages



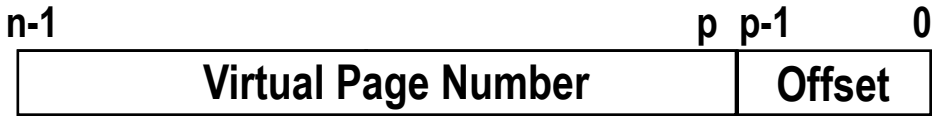
Speeding up Translation with a TLB

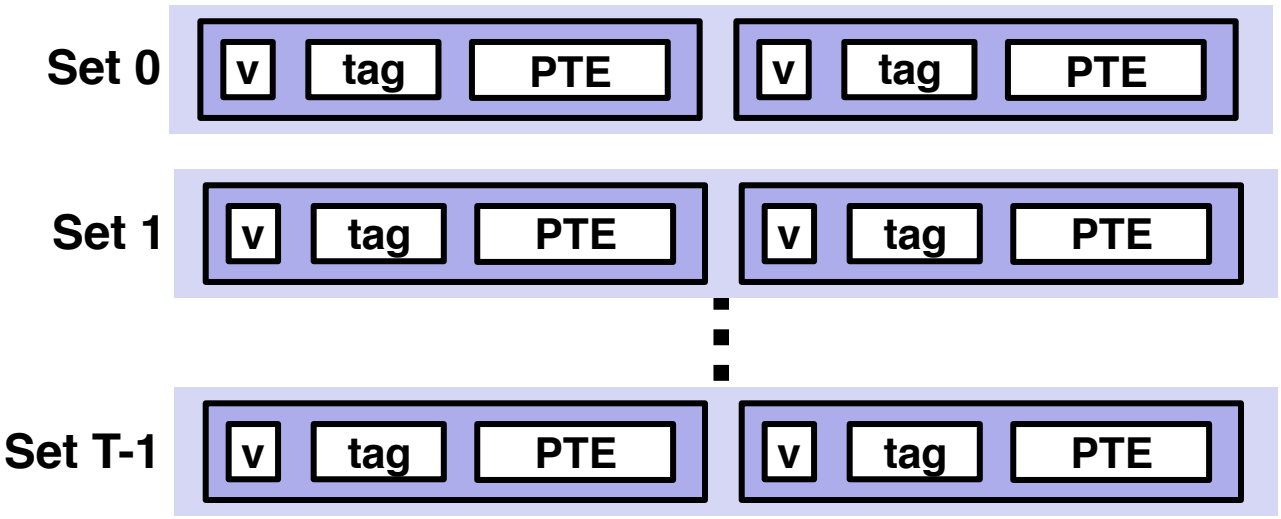
- Solution: *Translation Lookaside Buffer* (TLB)
  - Think of it as a dedicated cache for page table
  - Small set-associative hardware cache in MMU
  - Contains complete page table entries for a small number of pages



Accessing the TLB

- MMU uses the Virtual Page Number portion of the virtual address to access the TLB:

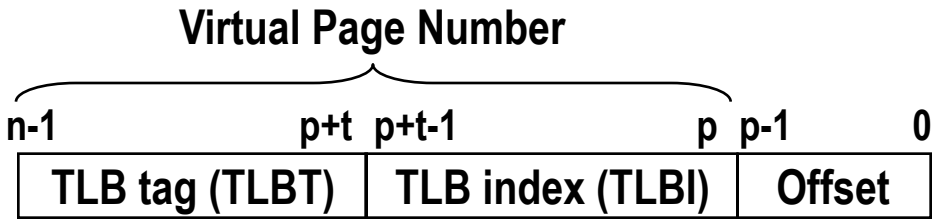


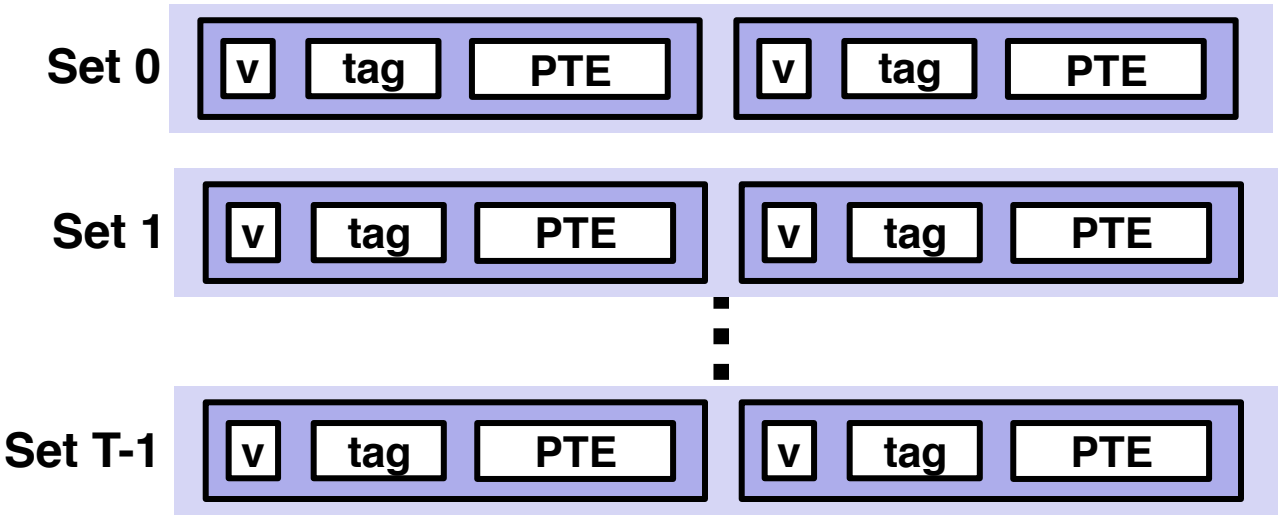


A Page Table Cache

Accessing the TLB

- MMU uses the Virtual Page Number portion of the virtual address to access the TLB:

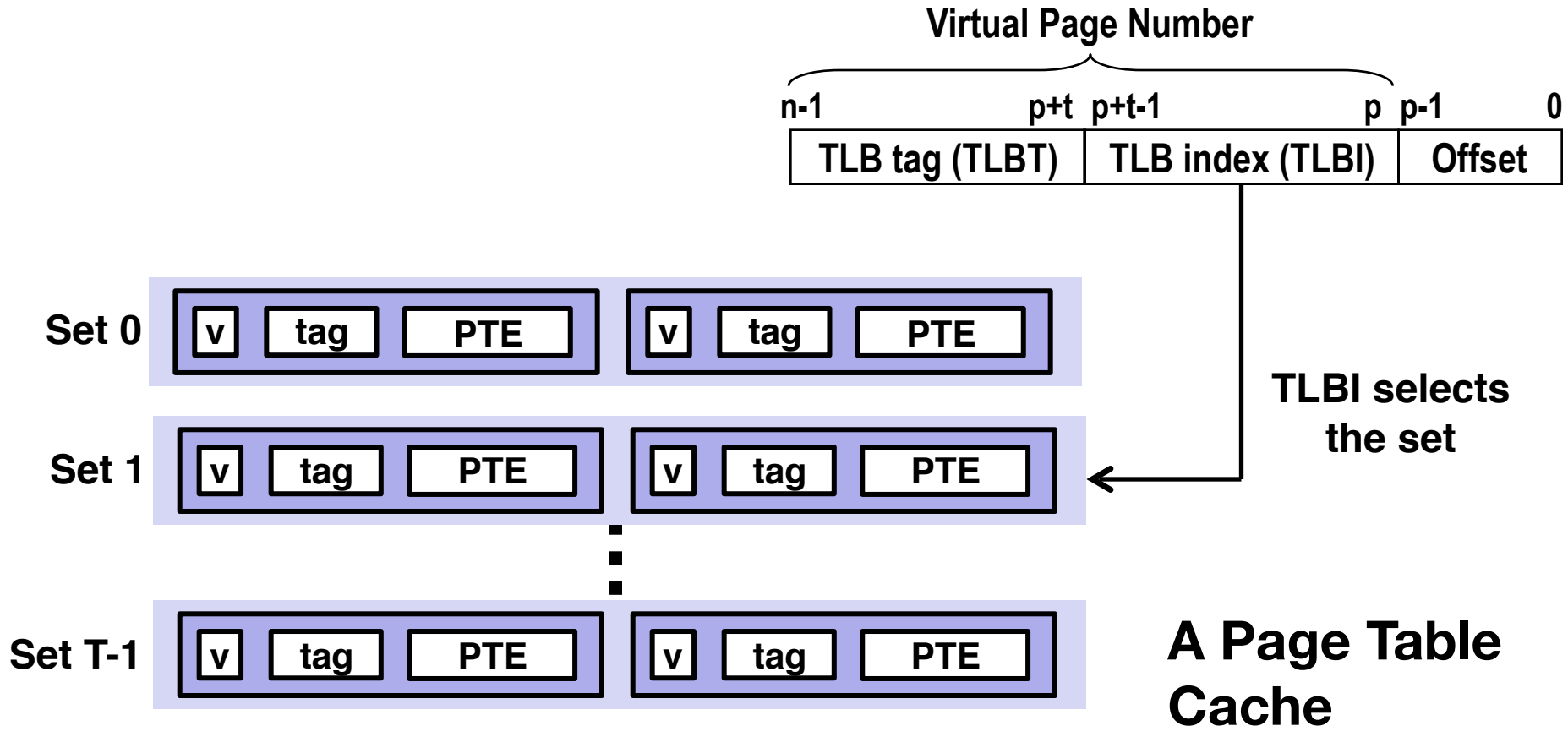




A Page Table Cache

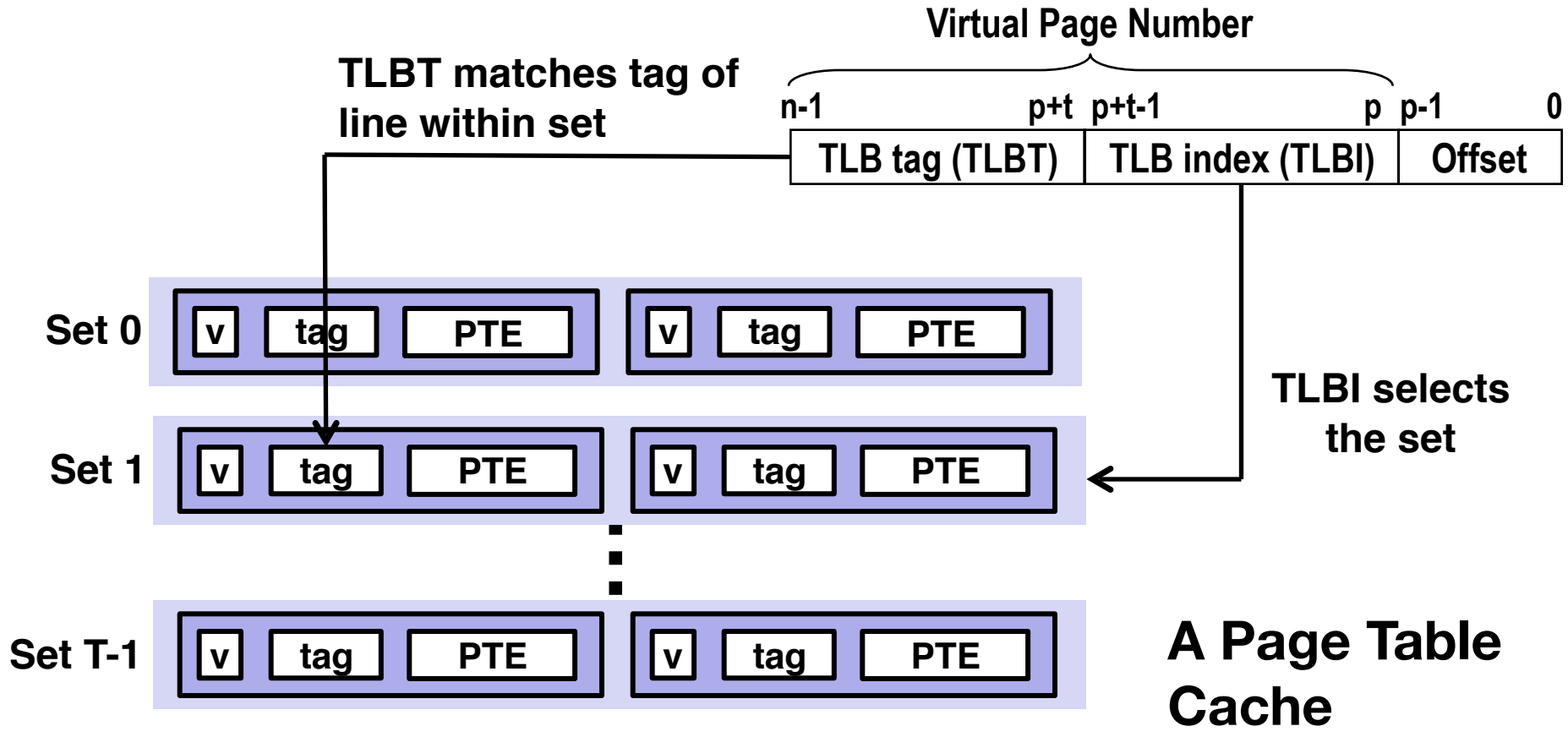
Accessing the TLB

- MMU uses the Virtual Page Number portion of the virtual address to access the TLB:

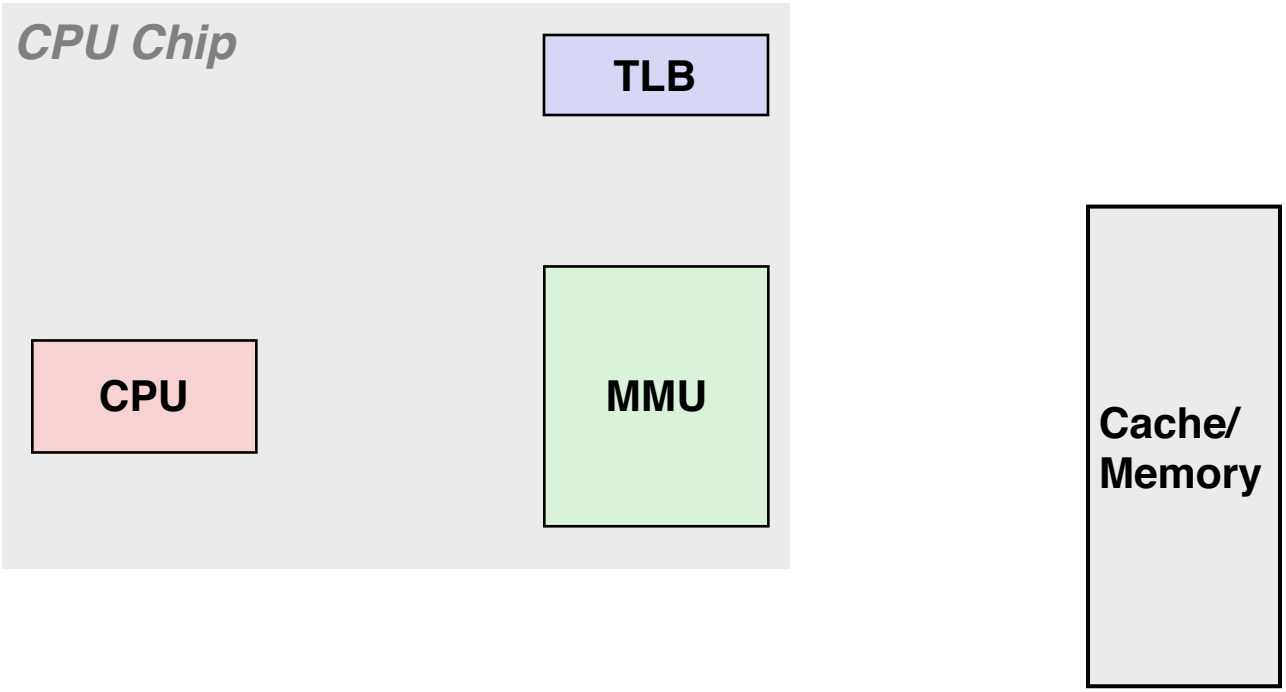


Accessing the TLB

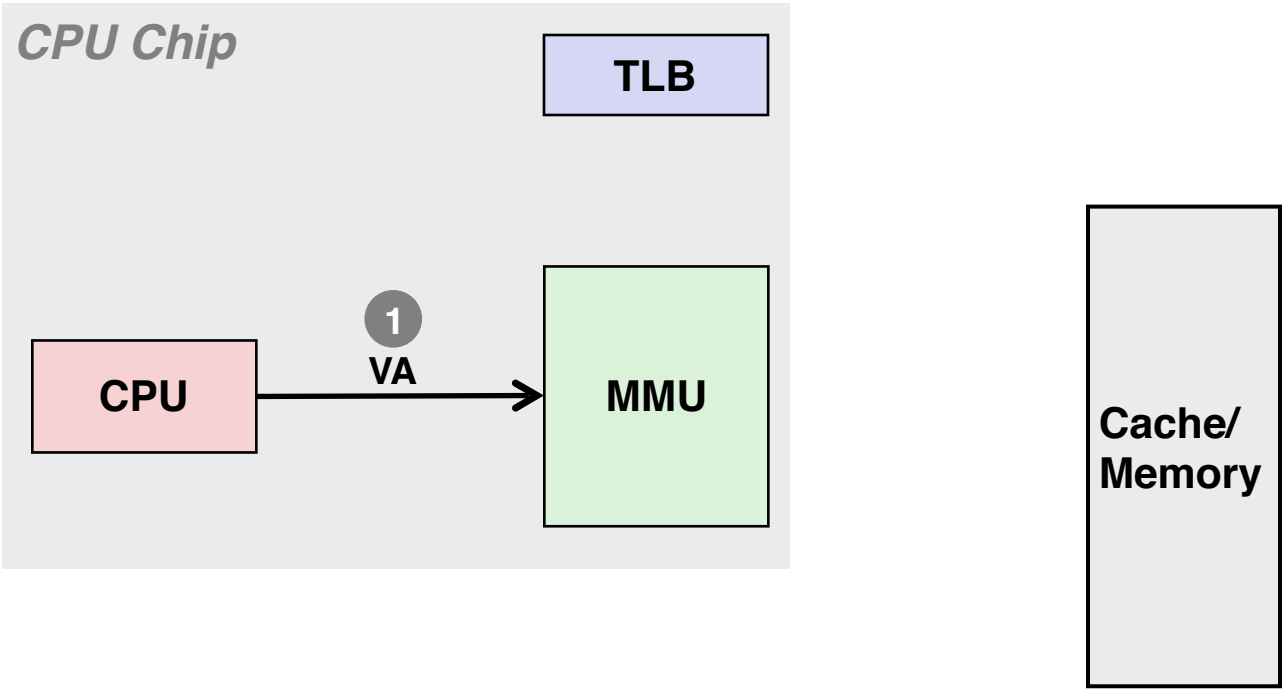
- MMU uses the Virtual Page Number portion of the virtual address to access the TLB:



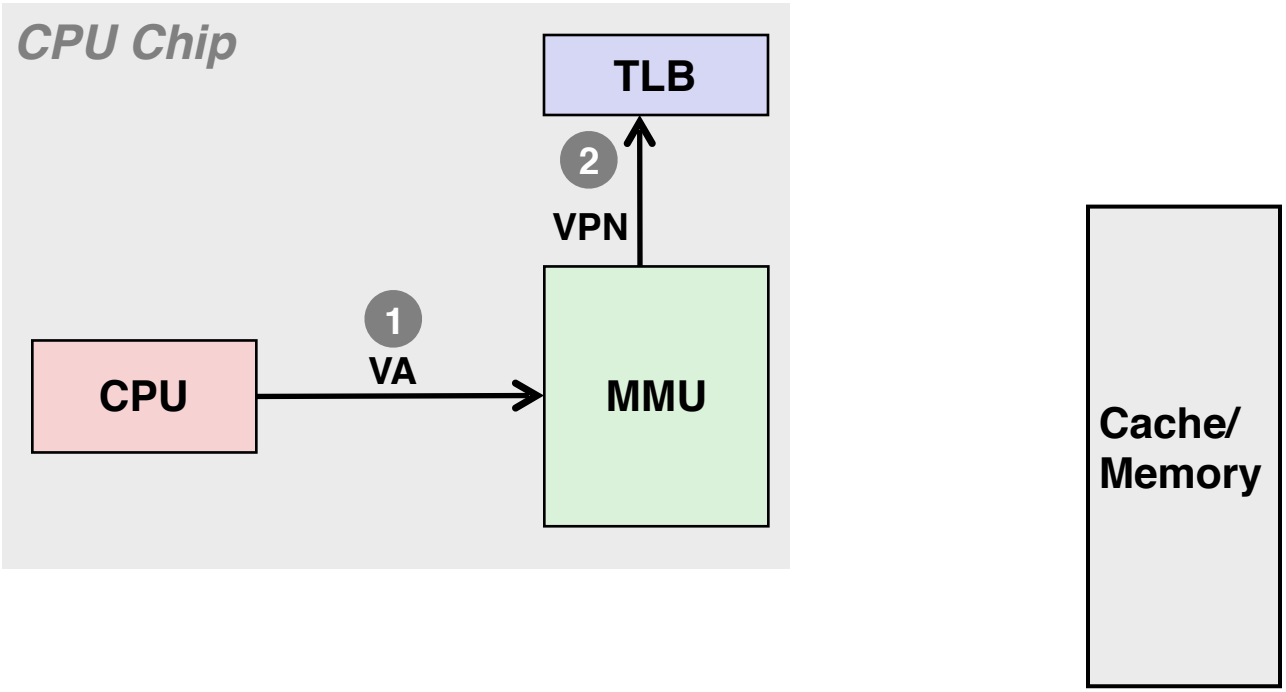
TLB Hit



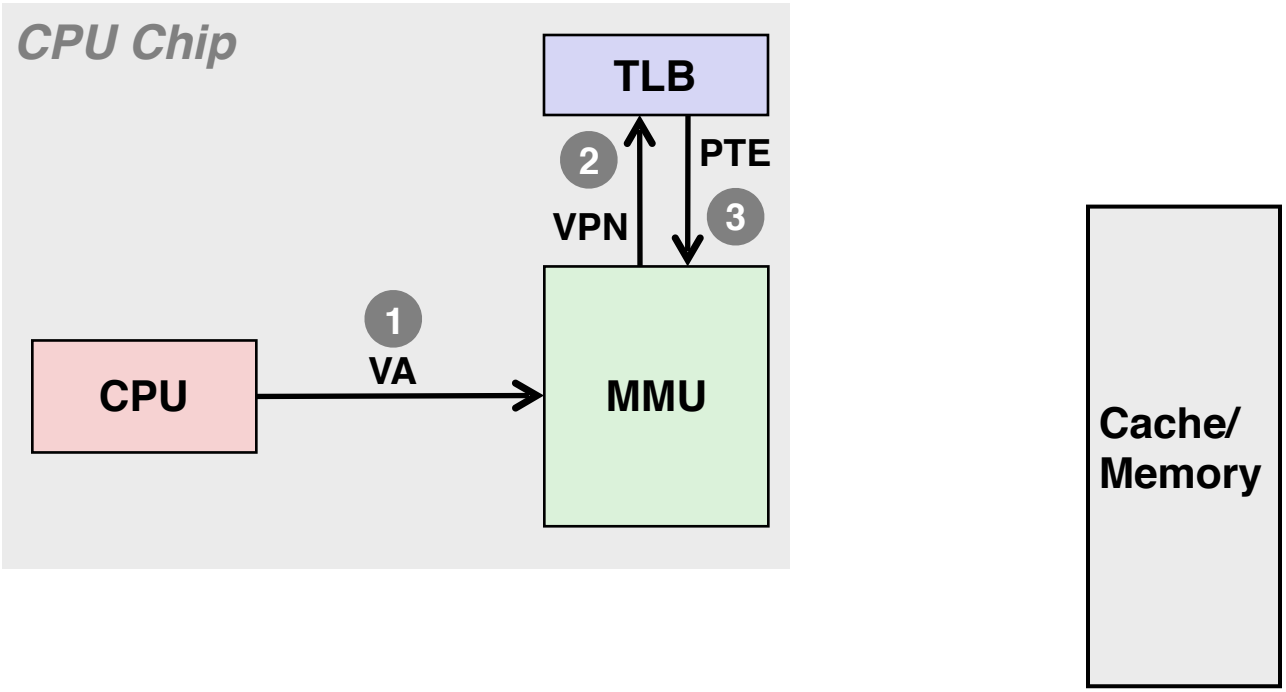
TLB Hit



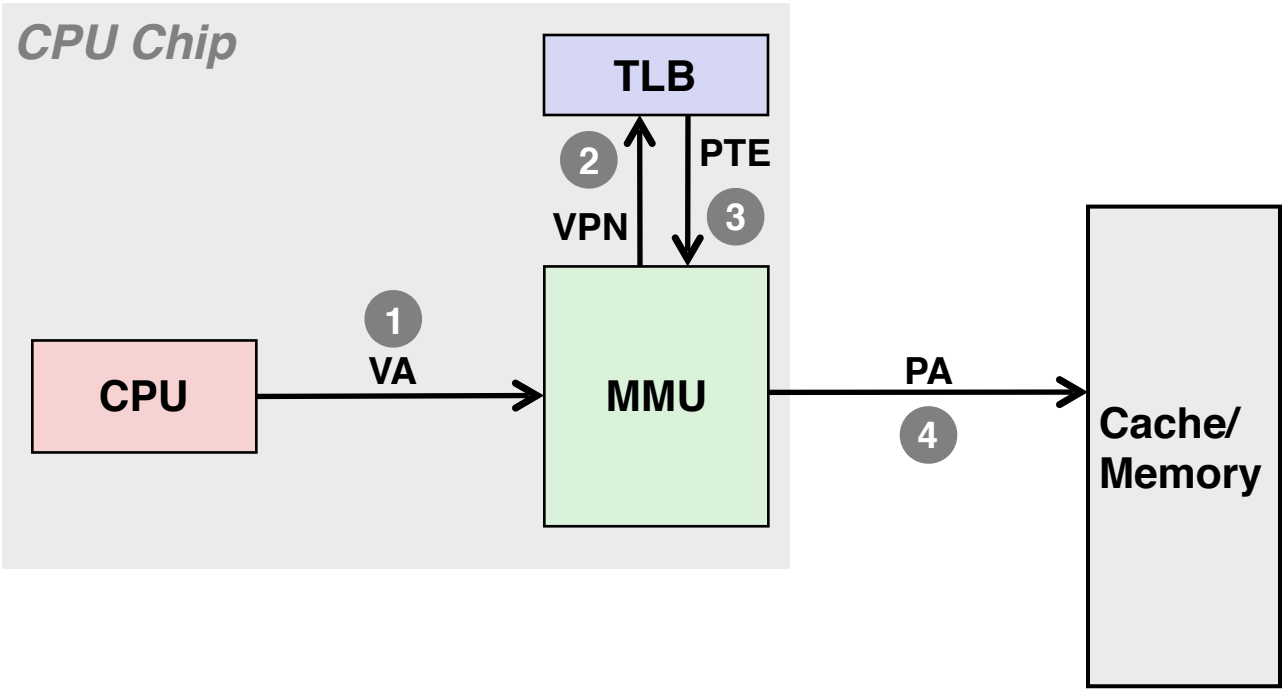
TLB Hit



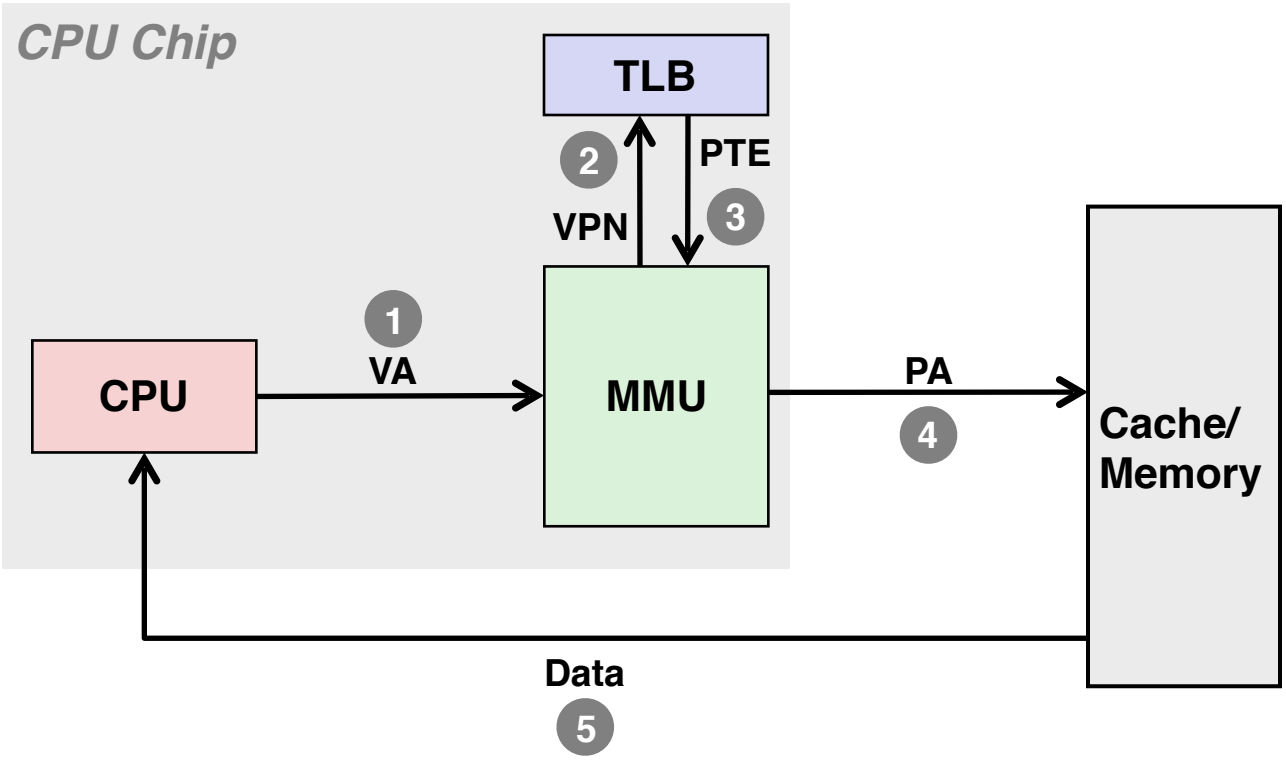
TLB Hit



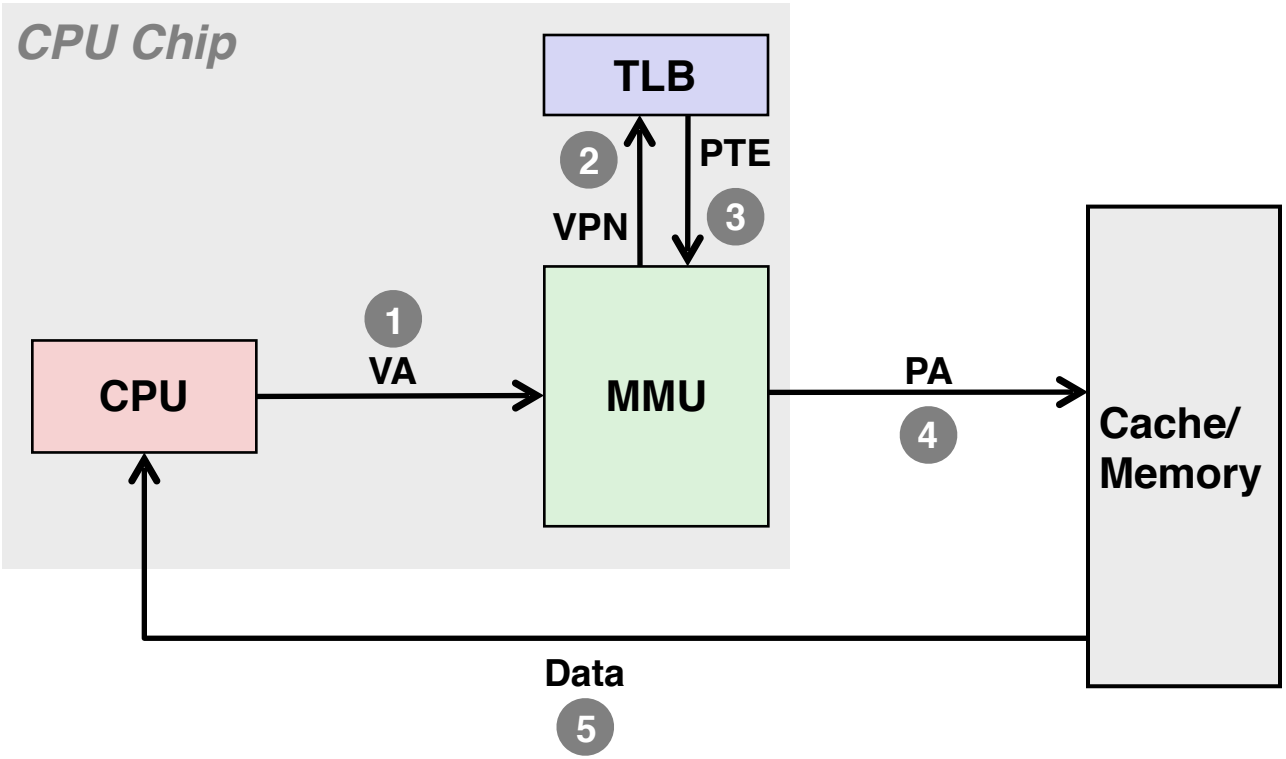
TLB Hit



TLB Hit

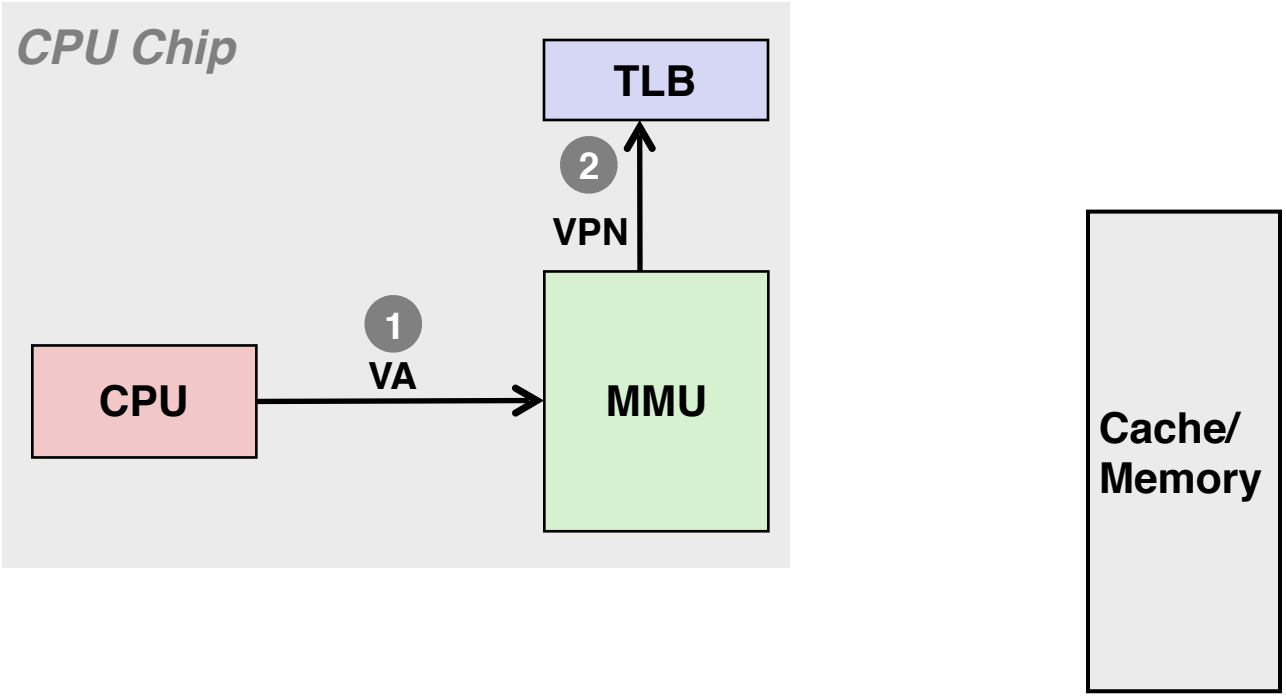


TLB Hit

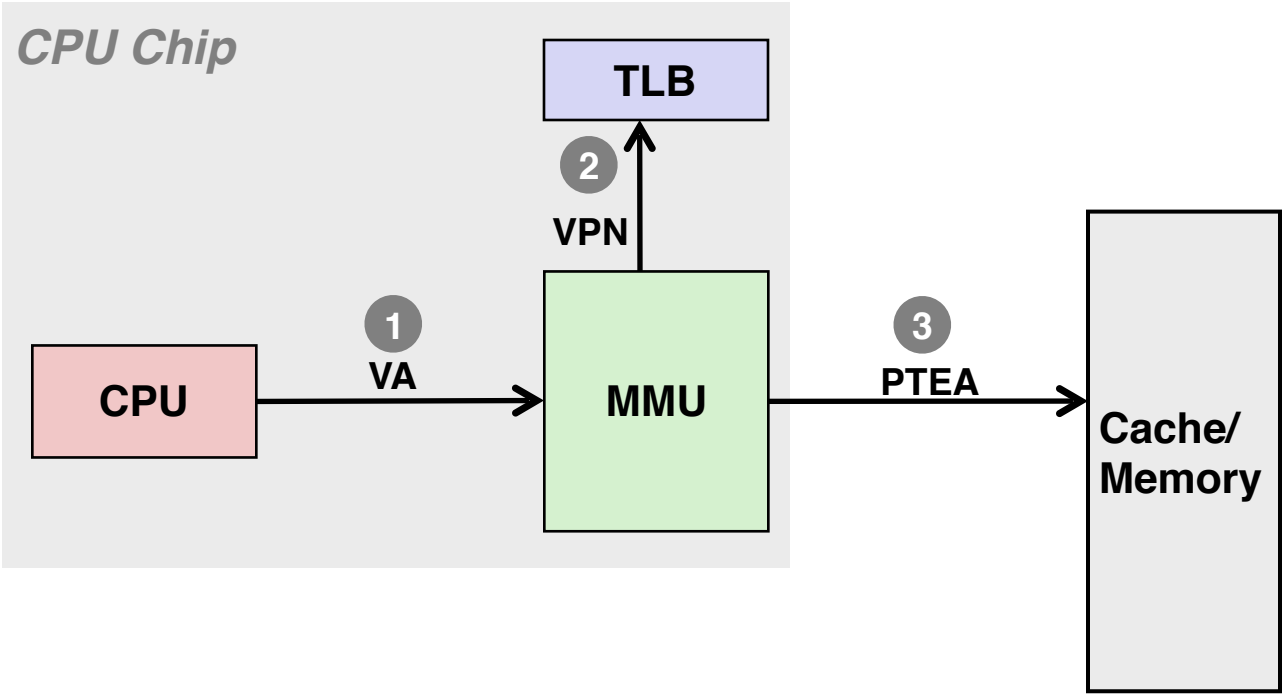


A TLB hit eliminates a memory access

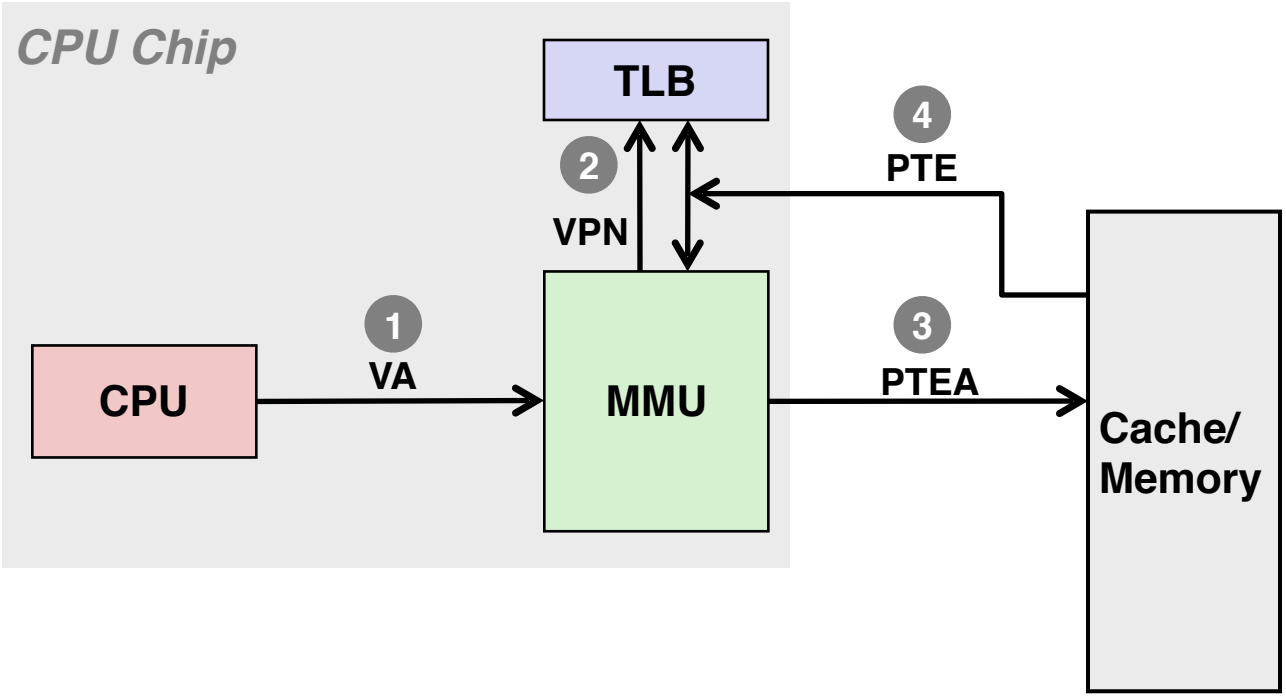
TLB Miss



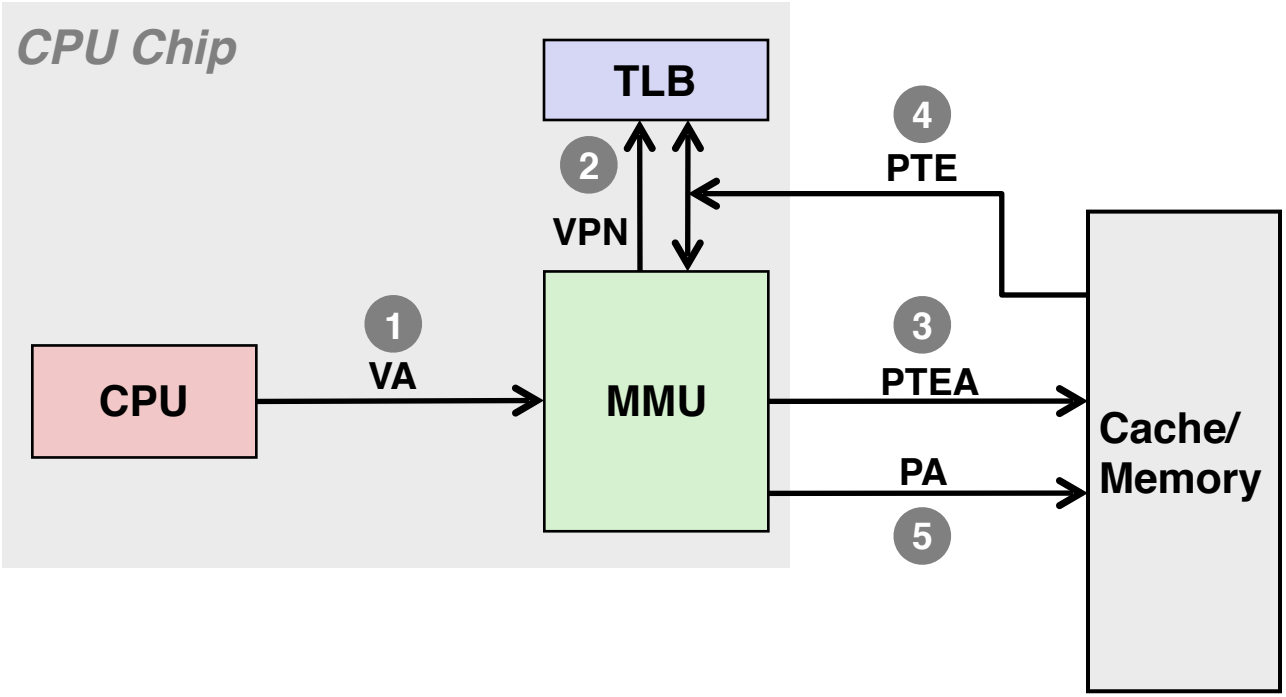
TLB Miss



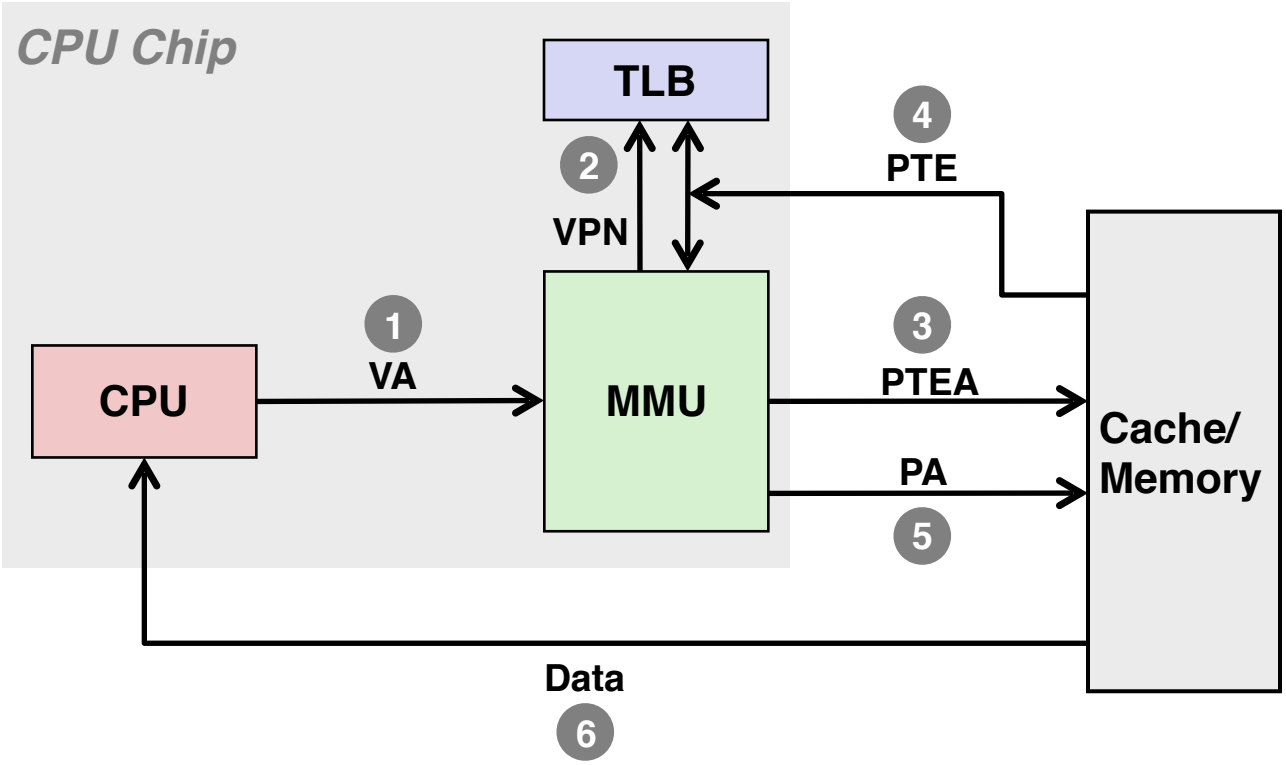
TLB Miss



TLB Miss



TLB Miss

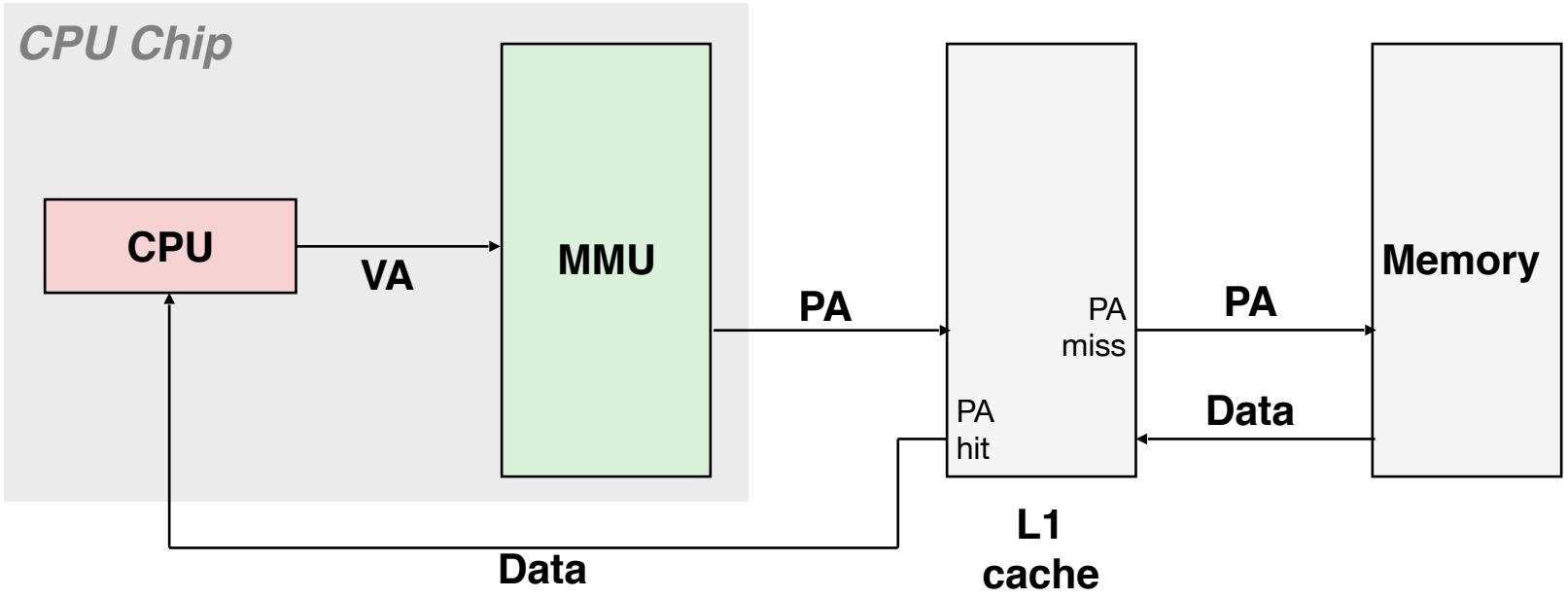

CPU Chip
TLB
4
PTE
2
VPN
1
VA
3
PTEA
CPU
MMU
Cache/
Memory
PA
5
Data
6

Today

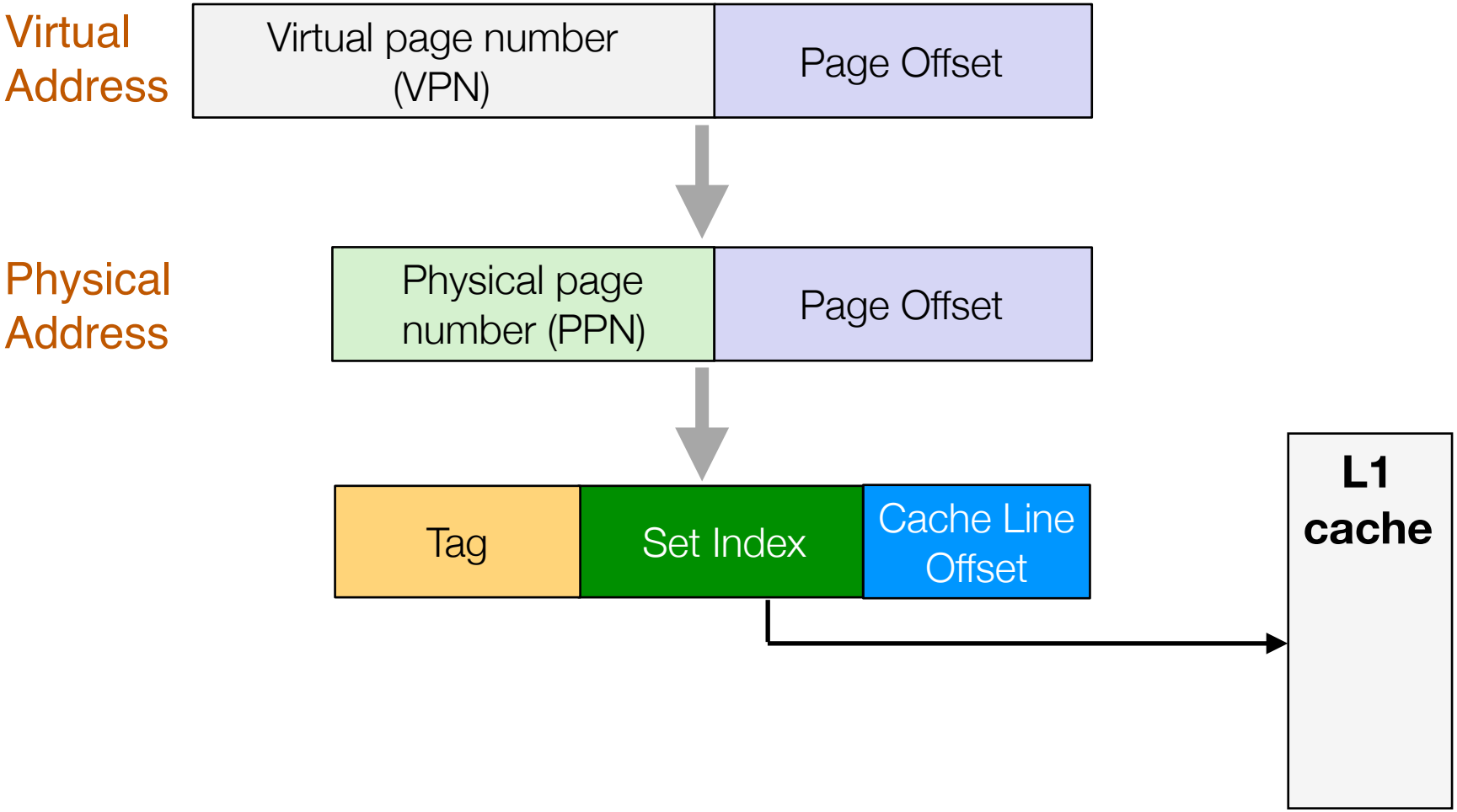
- Three Virtual Memory Optimizations
  - TLB
  - Virtually-indexed, physically-tagged cache
  - Page the page table (a.k.a., multi-level page table)
- Case-study: Intel Core i7/Linux example

Performance Issue in VM

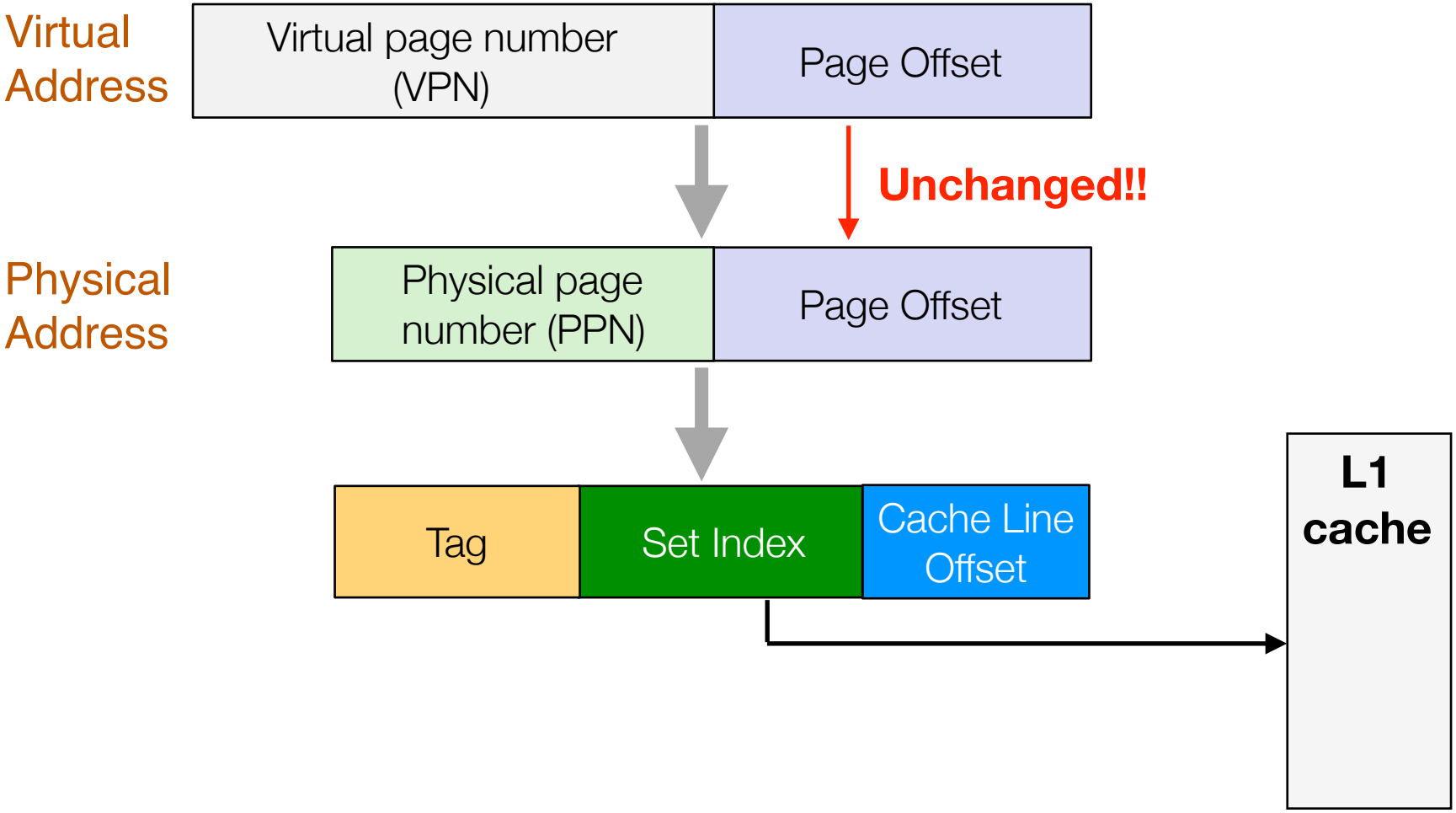
- Address translation and cache accesses are serialized
  - First translate from VA to PA
  - Then use PA to access cache
  - Slow! Can we speed it up?



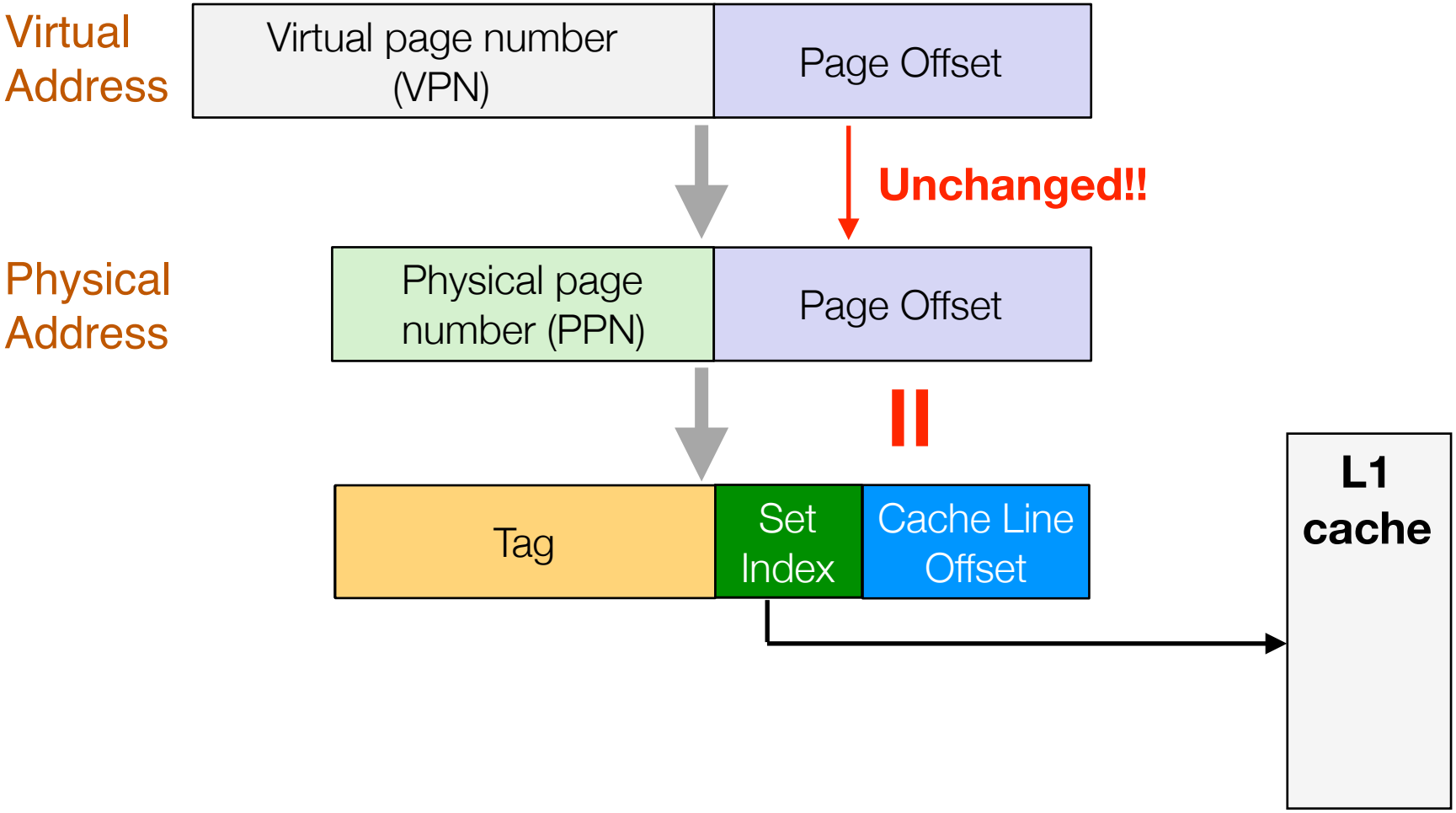
Observe Address Translation



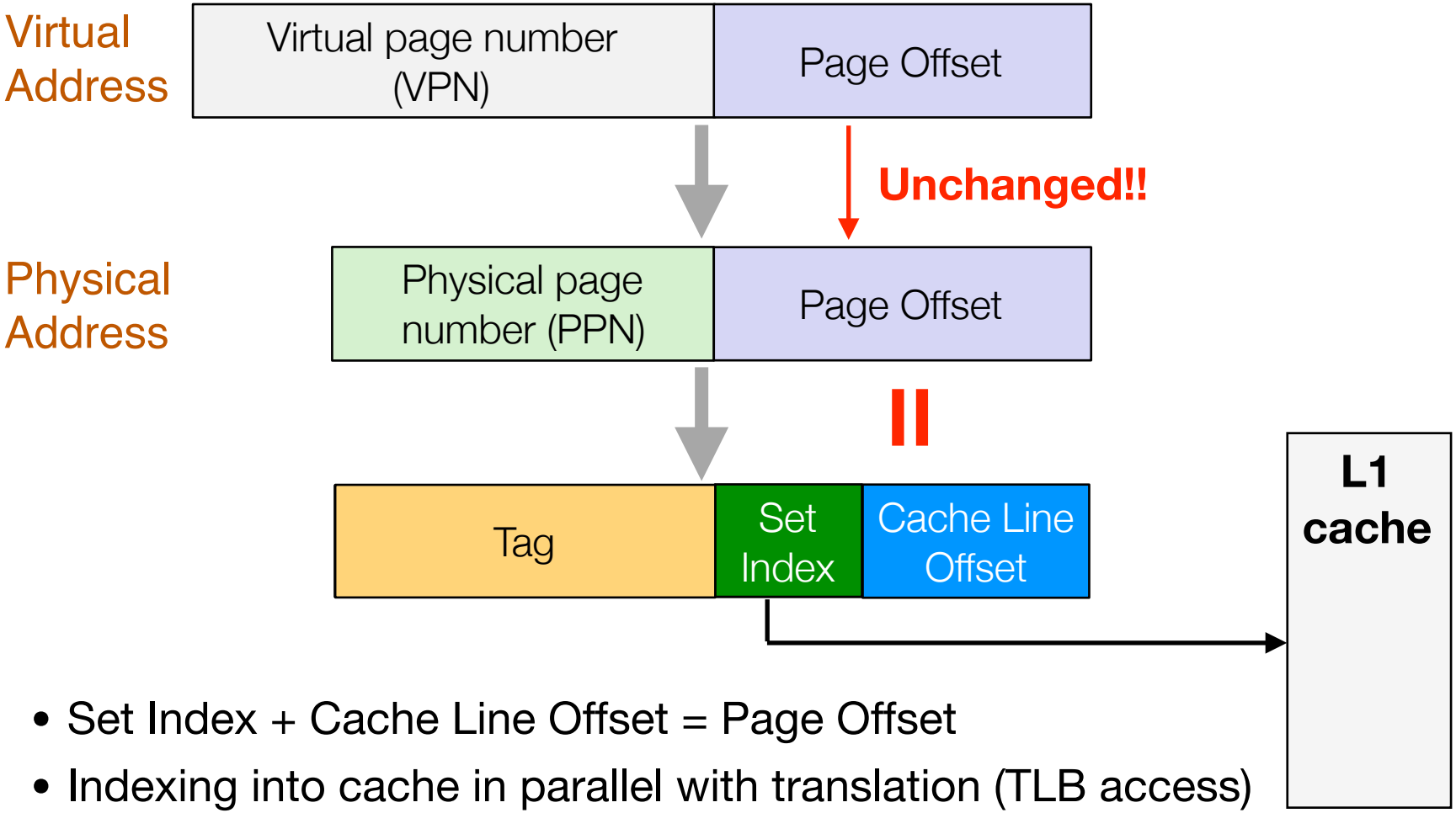
Observe Address Translation



Observe Address Translation

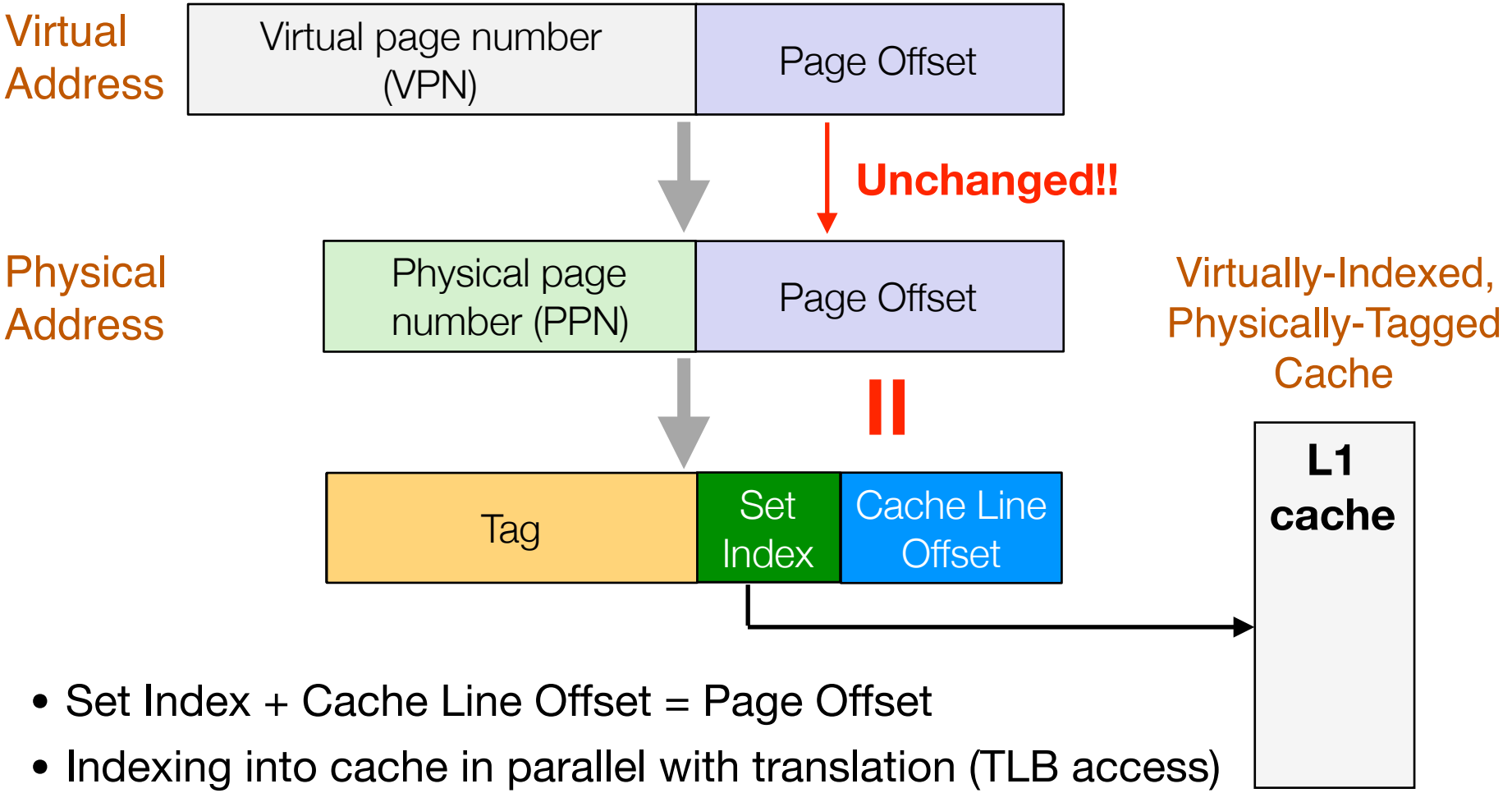


Observe Address Translation



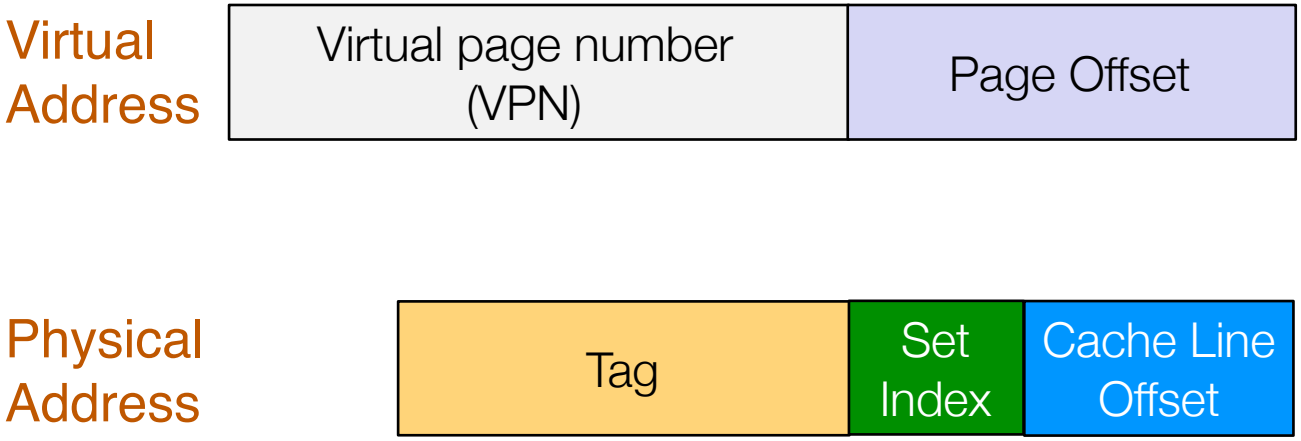
- Set Index + Cache Line Offset = Page Offset
- Indexing into cache in parallel with translation (TLB access)
- If TLB hits, can get the data back in one cycle

Observe Address Translation

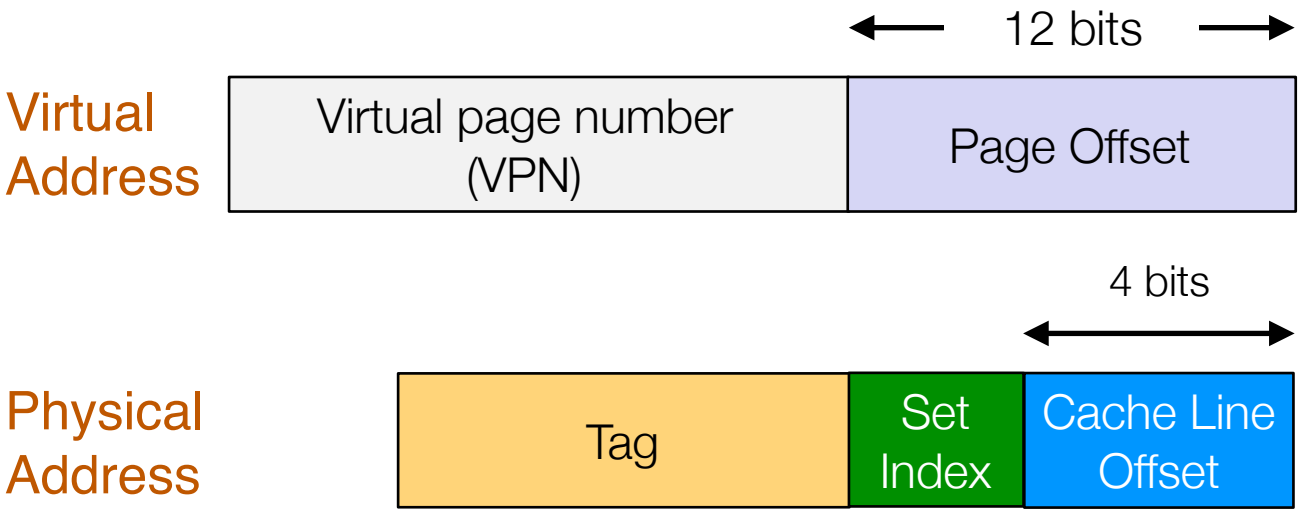


- Set Index + Cache Line Offset = Page Offset
- Indexing into cache in parallel with translation (TLB access)
- If TLB hits, can get the data back in one cycle

Any Implications?

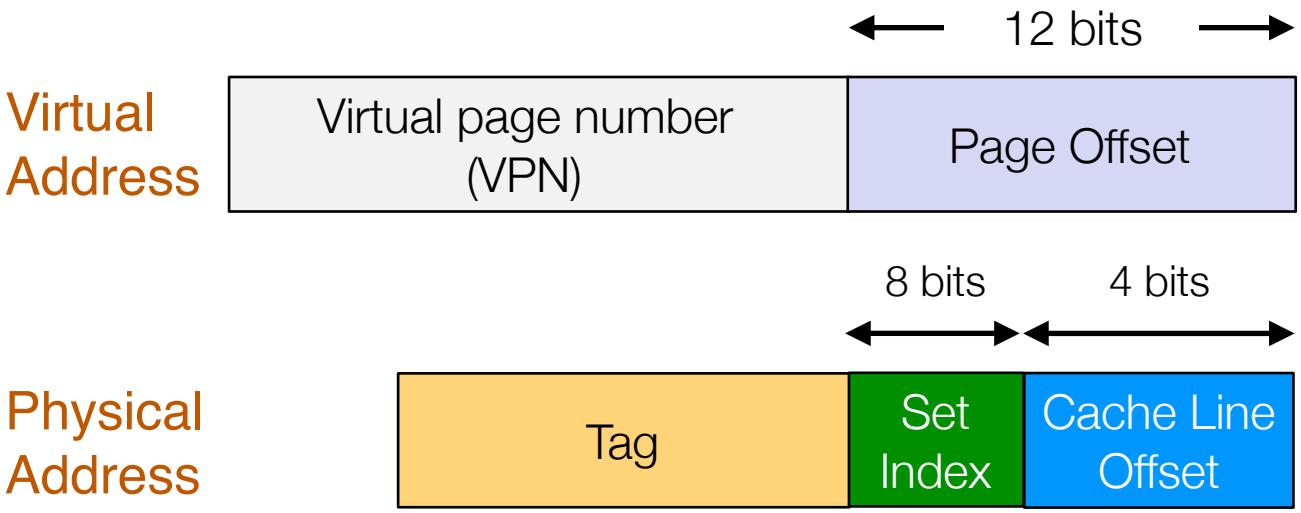


Any Implications?



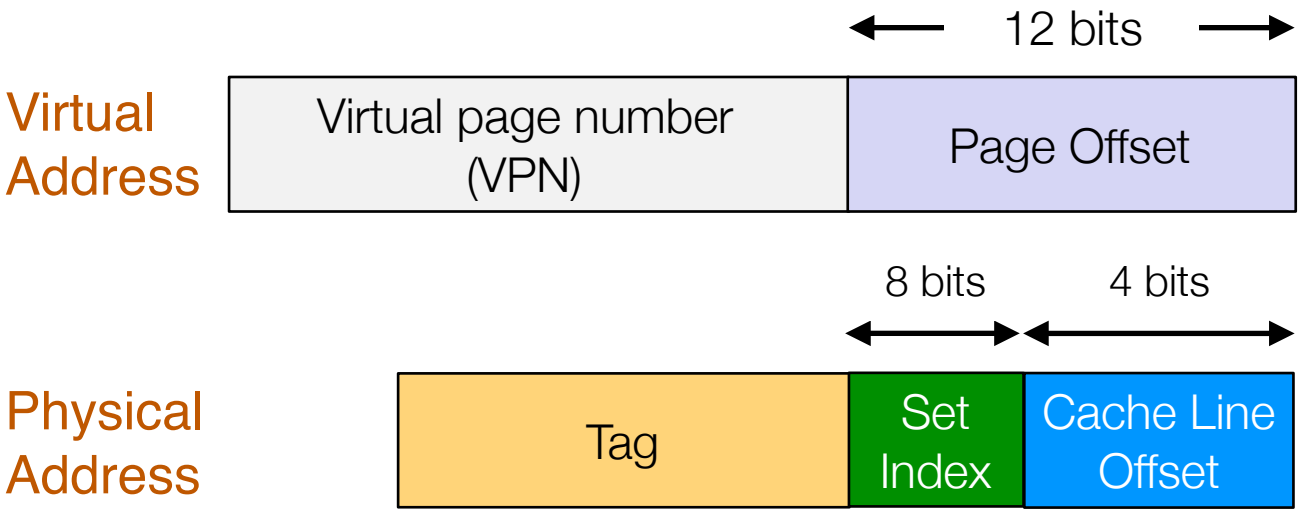
- Assuming 4K page size, cache line size is 16 bytes.

Any Implications?



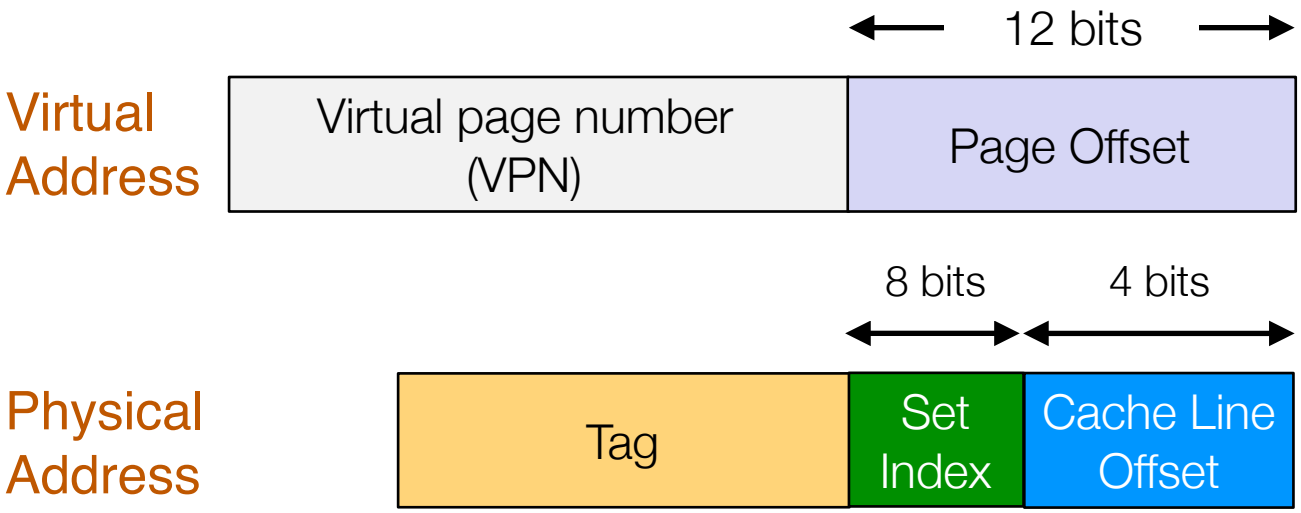
- Assuming 4K page size, cache line size is 16 bytes.
- Set Index = 8 bits. Can only have 256 Sets => Limit cache size

Any Implications?



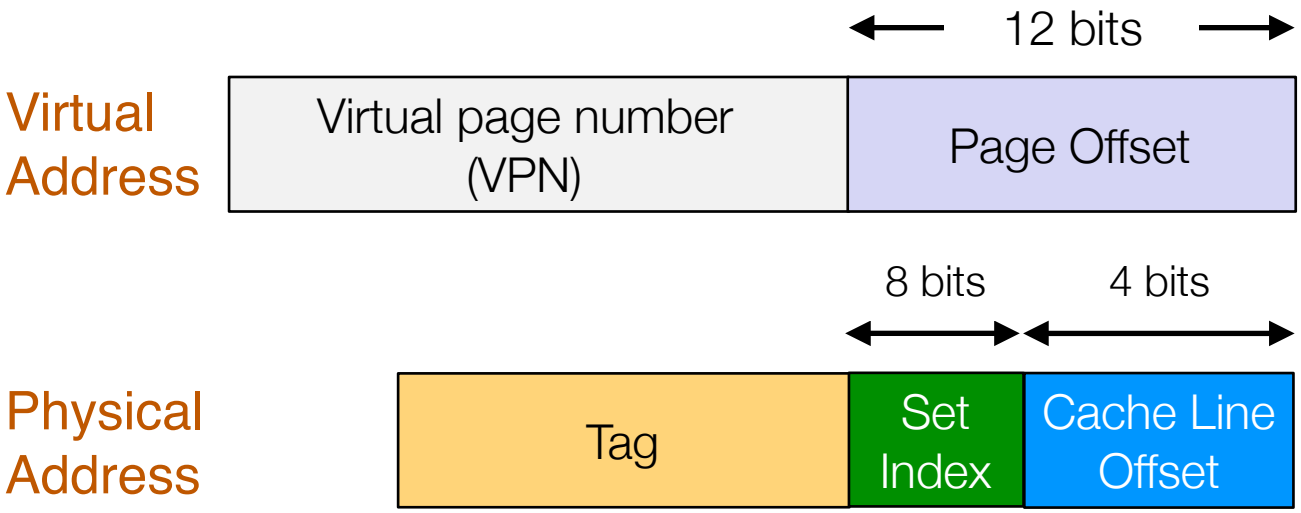
- Assuming 4K page size, cache line size is 16 bytes.
- Set Index = 8 bits. Can only have 256 Sets => Limit cache size
- Increasing cache size then requires increasing associativity

Any Implications?



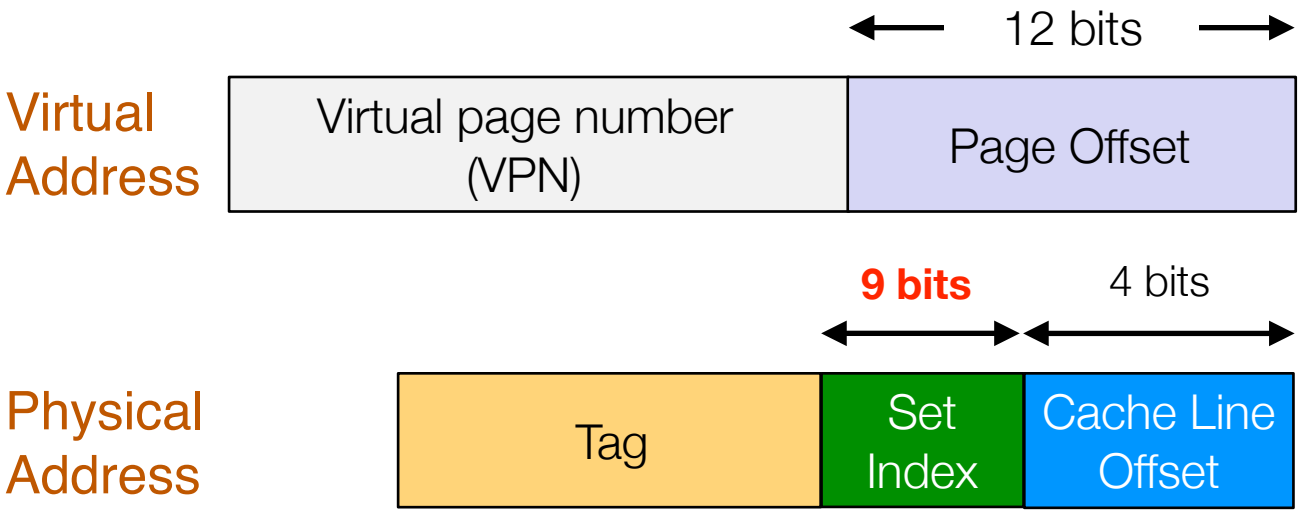
- Assuming 4K page size, cache line size is 16 bytes.
- Set Index = 8 bits. Can only have 256 Sets => Limit cache size
- Increasing cache size then requires increasing associativity
  - Not ideal because that requires comparing more tags

Any Implications?



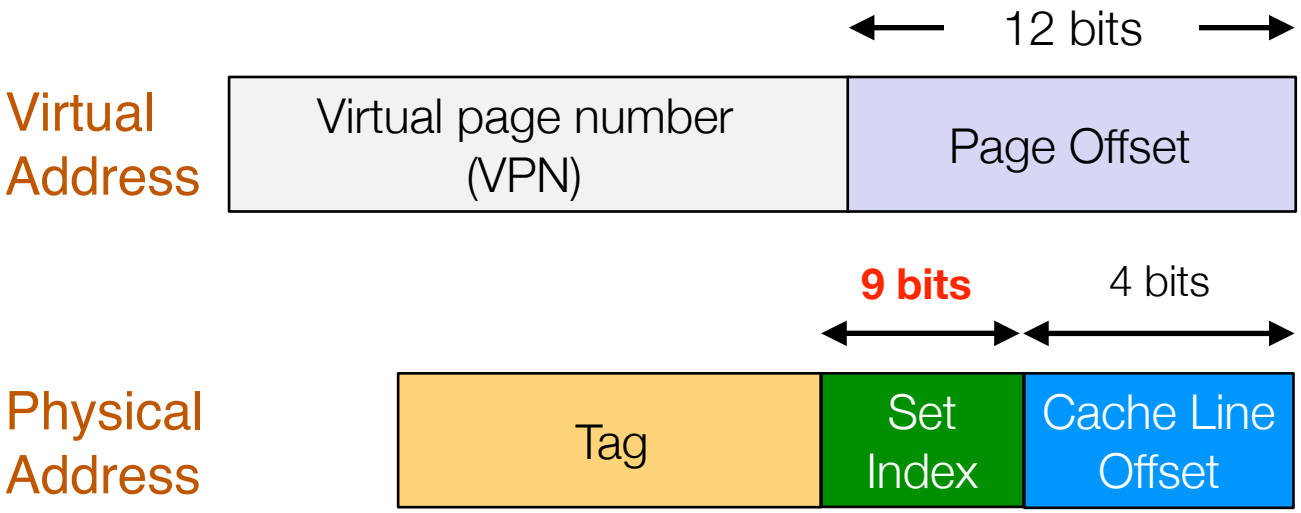
- Assuming 4K page size, cache line size is 16 bytes.
- Set Index = 8 bits. Can only have 256 Sets => Limit cache size
- Increasing cache size then requires increasing associativity
  - Not ideal because that requires comparing more tags
- Solutions?

Any Implications?



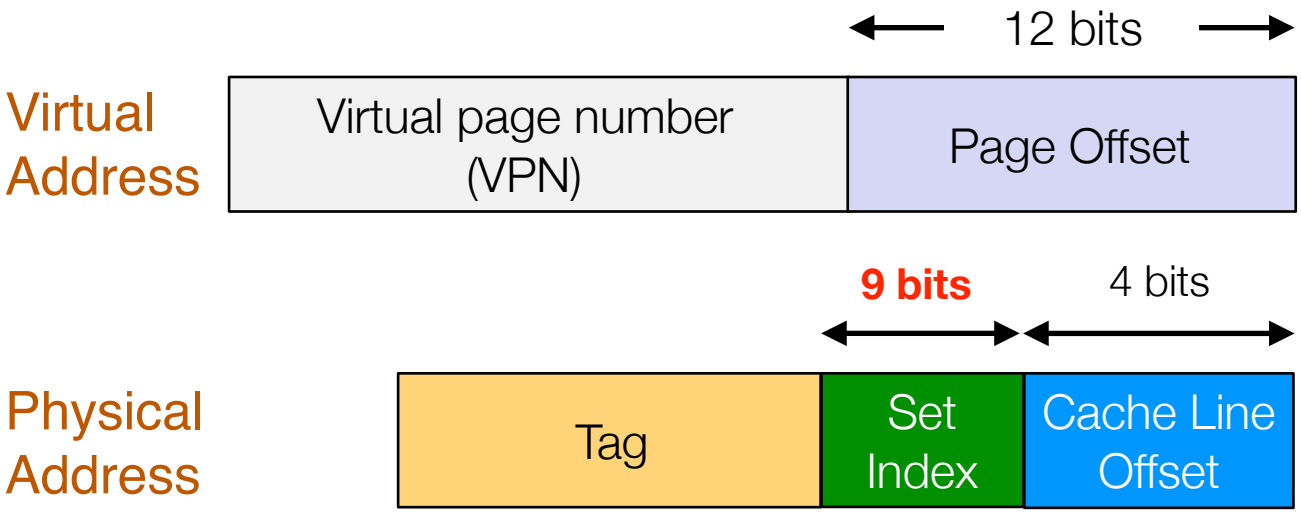
- What if we use 9 bits for Set Index? More Sets now.

Any Implications?



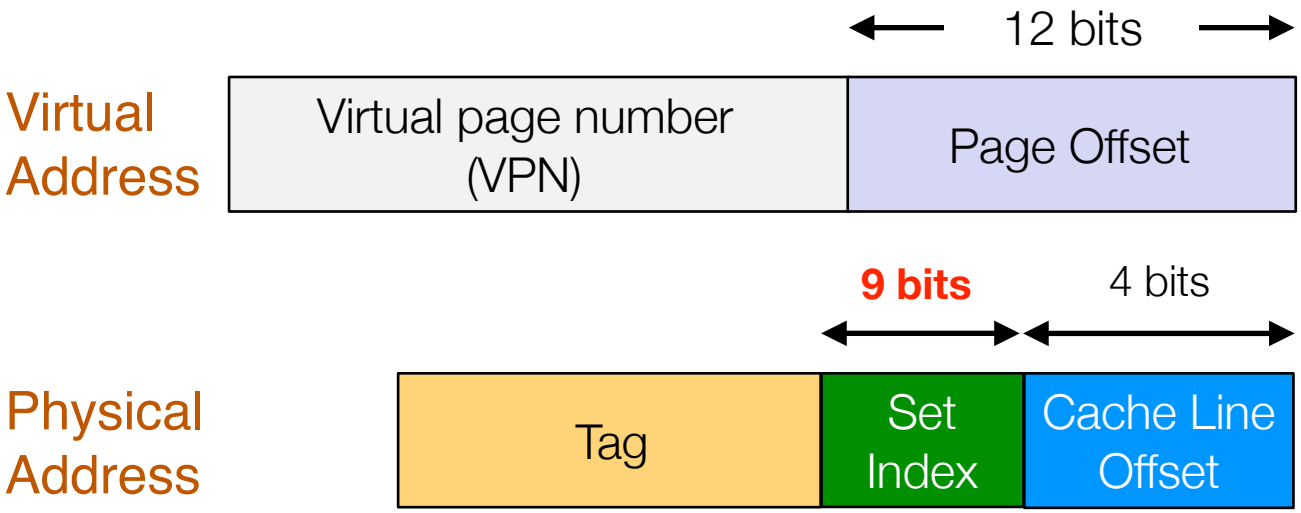
- What if we use 9 bits for Set Index? More Sets now.
- How can this still work?

Any Implications?



- What if we use 9 bits for Set Index? More Sets now.
- How can this still work?
- The least significant bit in VPN and PPN must be the same

Any Implications?



- What if we use 9 bits for Set Index? More Sets now.
- How can this still work?
- The least significant bit in VPN and PPN must be the same
- That is: an even VA must be mapped to an even PA, and an odd VA must be mapped to an odd PA

Today

- Three Virtual Memory Optimizations
  - TLB
  - Virtually-indexed, physically-tagged cache
  - Page the page table (a.k.a., multi-level page table)
- Case-study: Intel Core i7/Linux example

Where Does Page Table Live?

Where Does Page Table Live?

- It needs to be at a specific location where we can find it
  - In main memory, with its start address stored in a special register (PTBR)

Where Does Page Table Live?

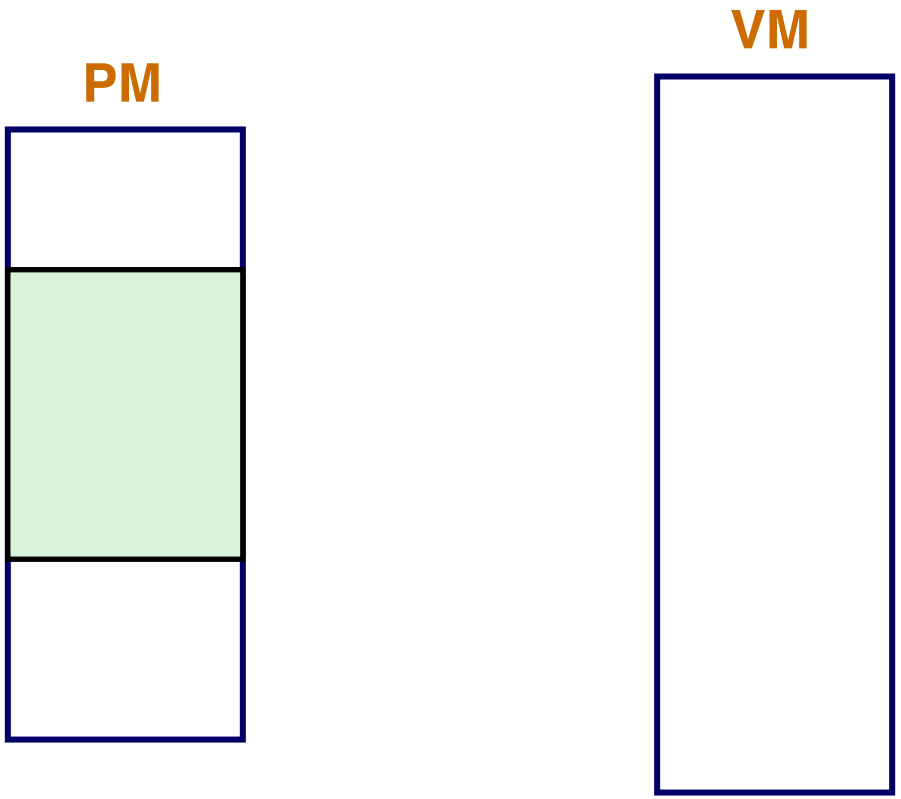
- It needs to be at a specific location where we can find it
  - In main memory, with its start address stored in a special register (PTBR)
- Assume 4KB page, 48-bit virtual memory, each PTE is 8 Bytes
  - 2^{36} PTEs in a page table
  - 512 GB total size per page table??!!

Where Does Page Table Live?

- It needs to be at a specific location where we can find it
  - In main memory, with its start address stored in a special register (PTBR)
- Assume 4KB page, 48-bit virtual memory, each PTE is 8 Bytes
  - 2^{36} PTEs in a page table
  - 512 GB total size per page table??!!
- Problem: Page tables are huge
  - One table per process!
  - Storing them all in main memory wastes space

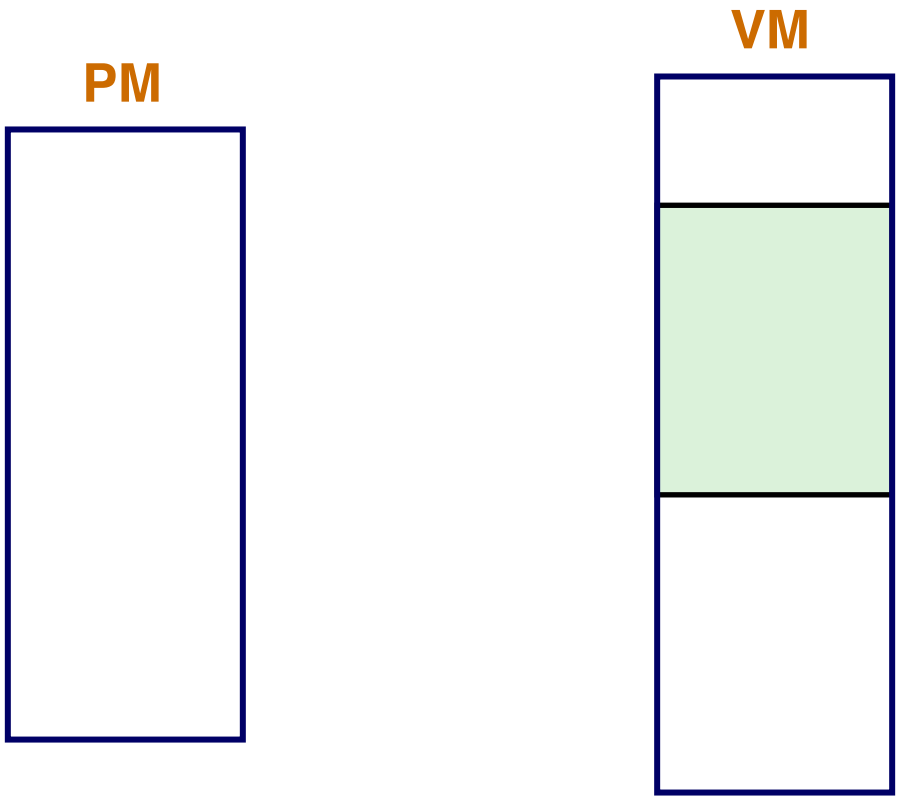
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entires in main memory
- Idea: Put page table in Virtual Memory and swap it just like data



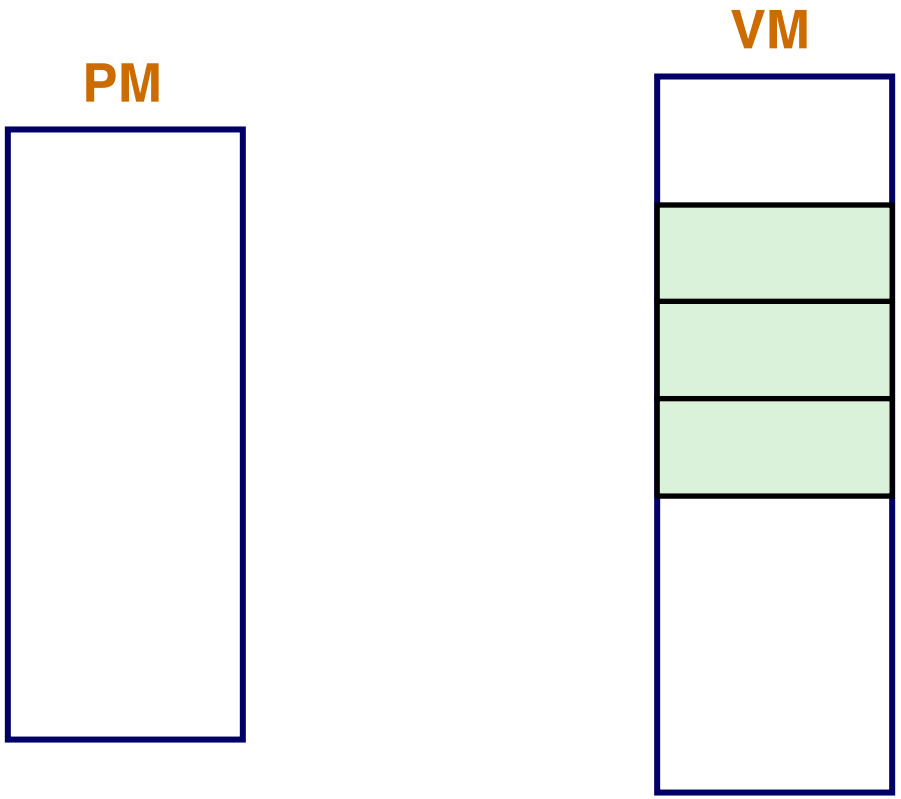
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entires in main memory
- Idea: Put page table in Virtual Memory and swap it just like data



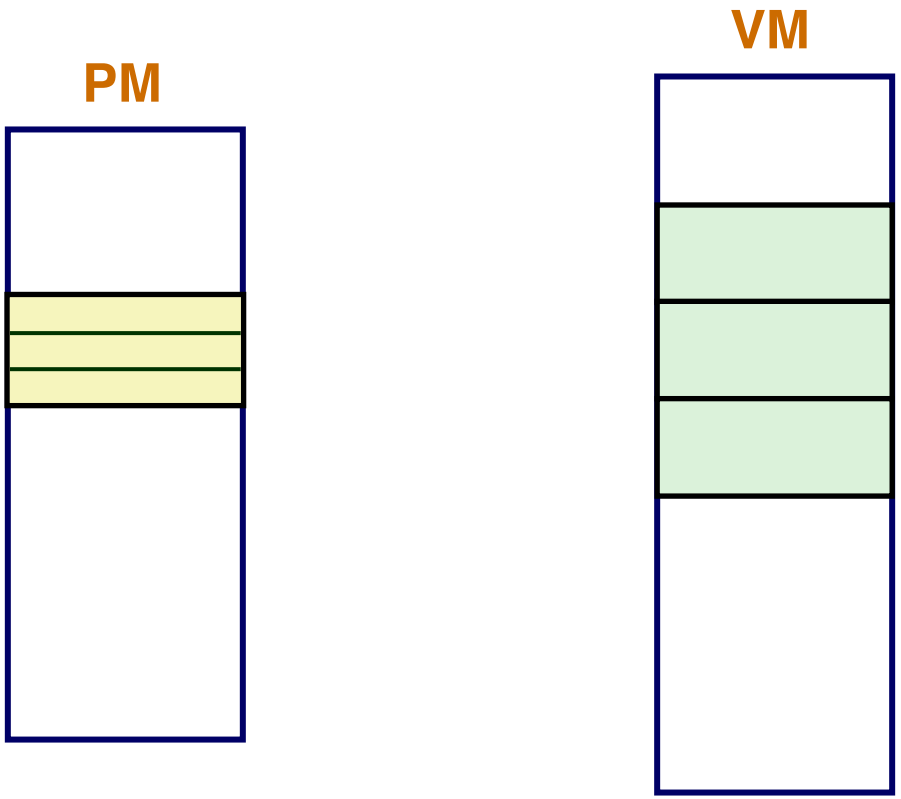
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entires in main memory
- Idea: Put page table in Virtual Memory and swap it just like data



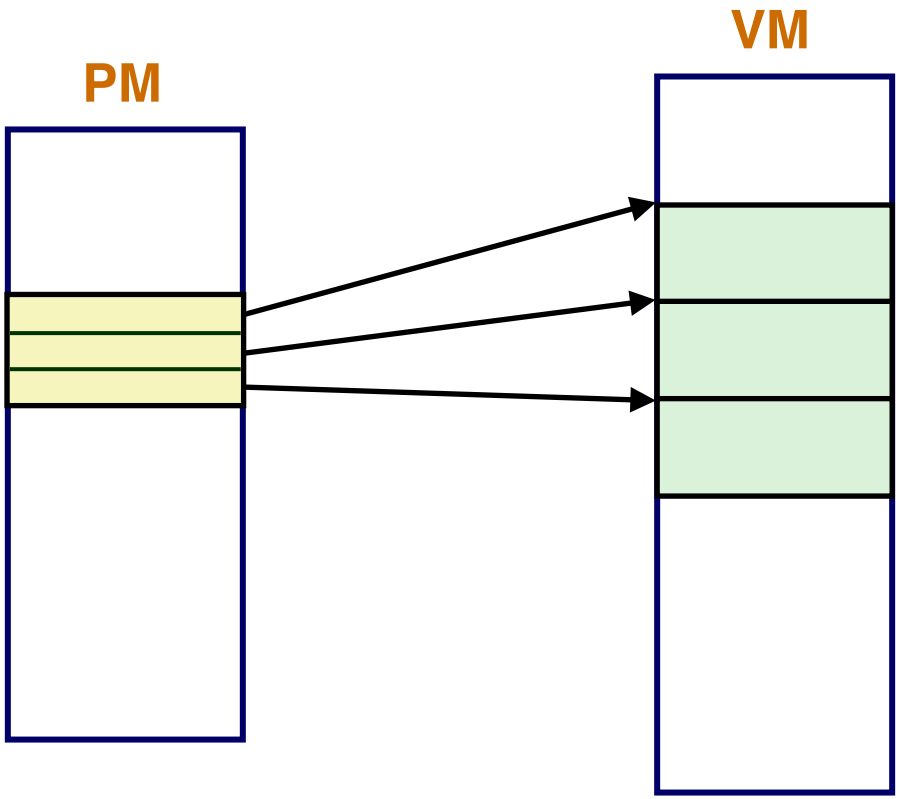
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entires in main memory
- Idea: Put page table in Virtual Memory and swap it just like data



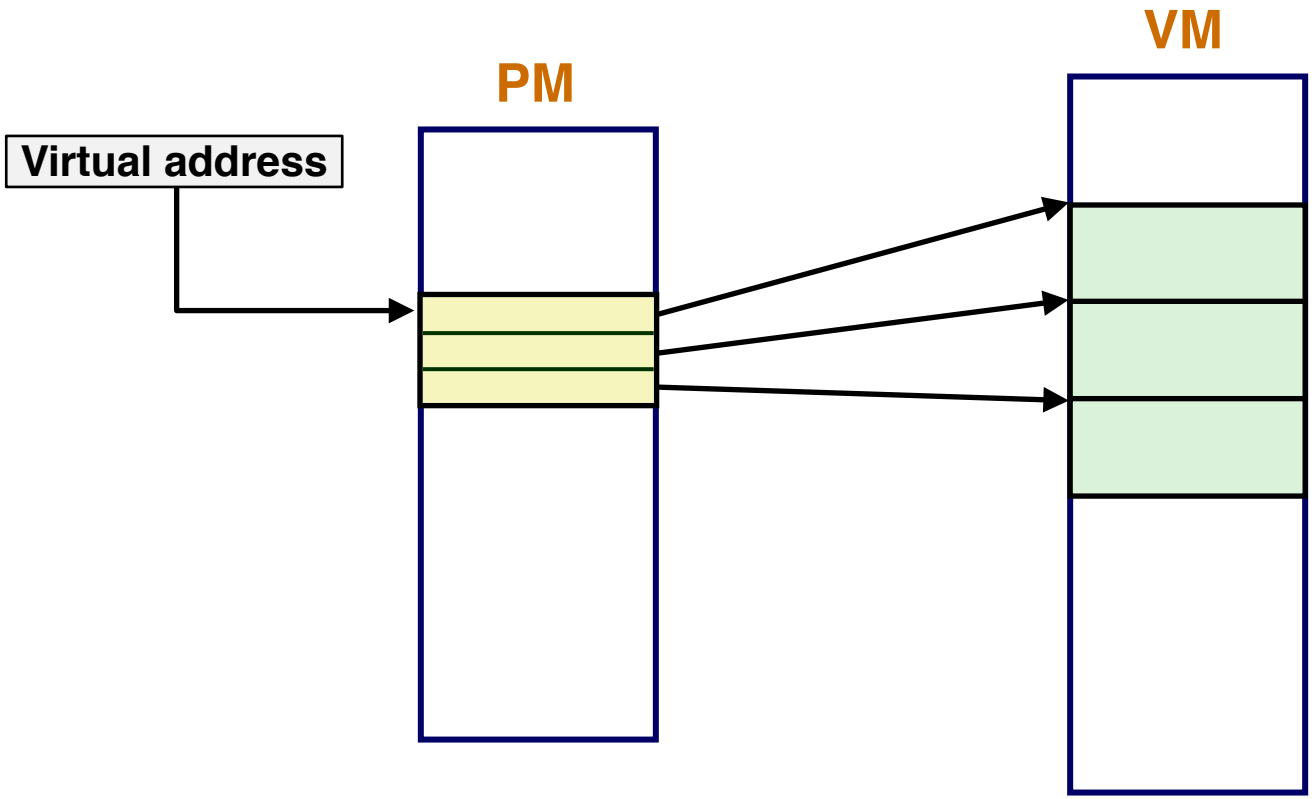
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entires in main memory
- Idea: Put page table in Virtual Memory and swap it just like data



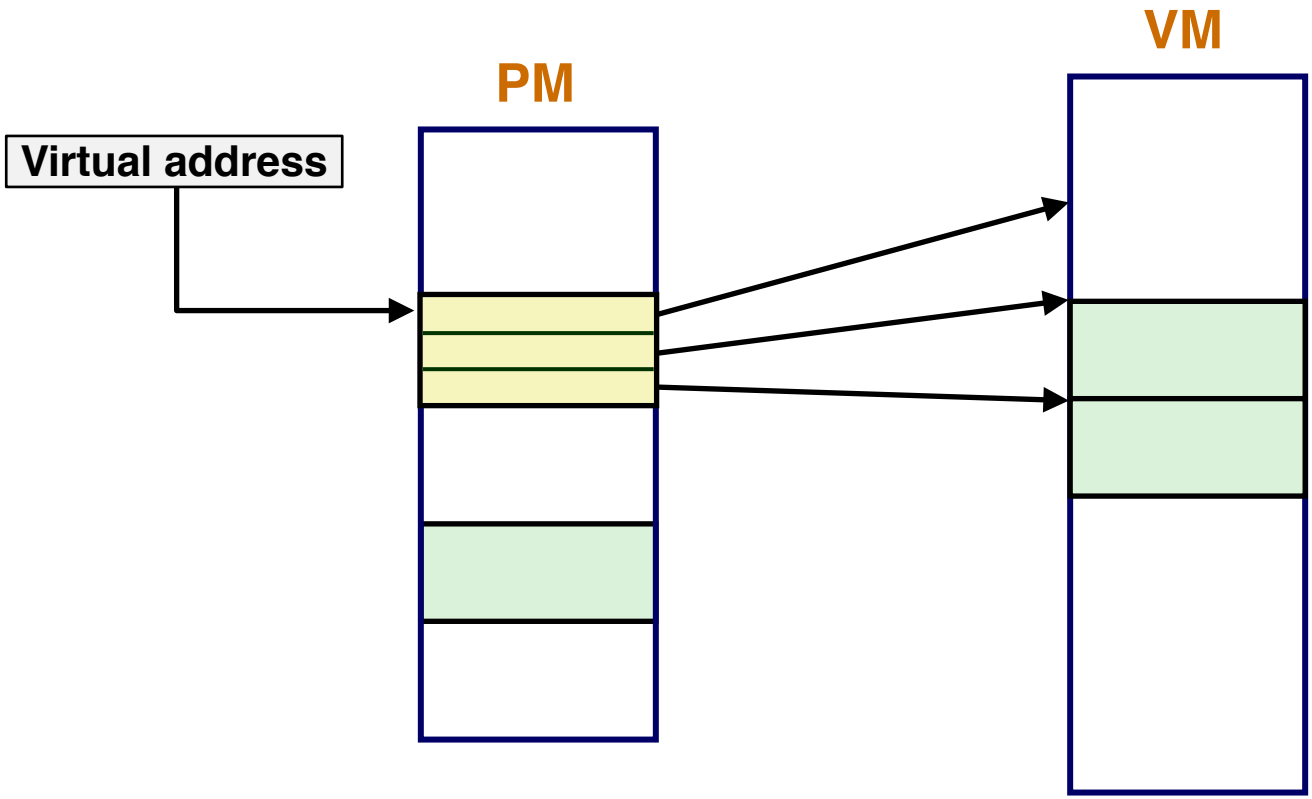
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entires in main memory
- Idea: Put page table in Virtual Memory and swap it just like data



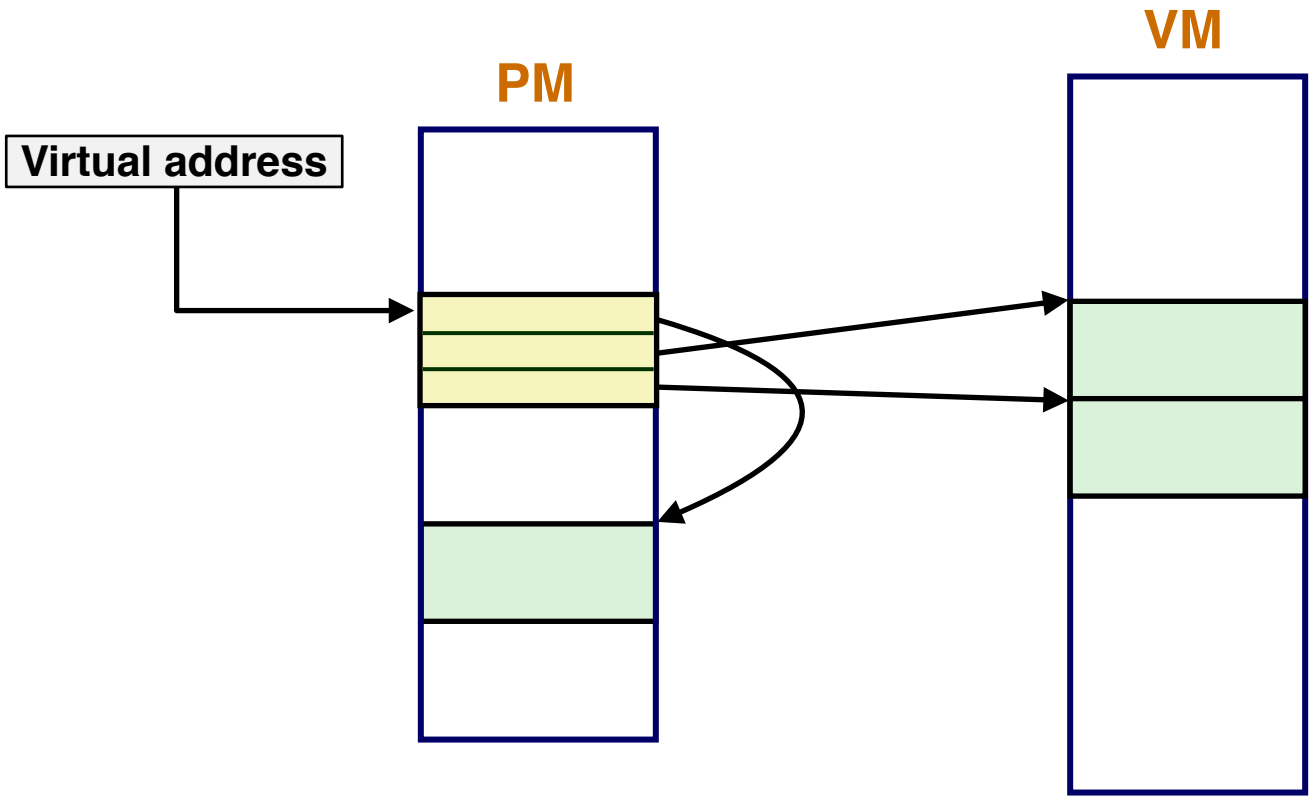
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entires in main memory
- Idea: Put page table in Virtual Memory and swap it just like data



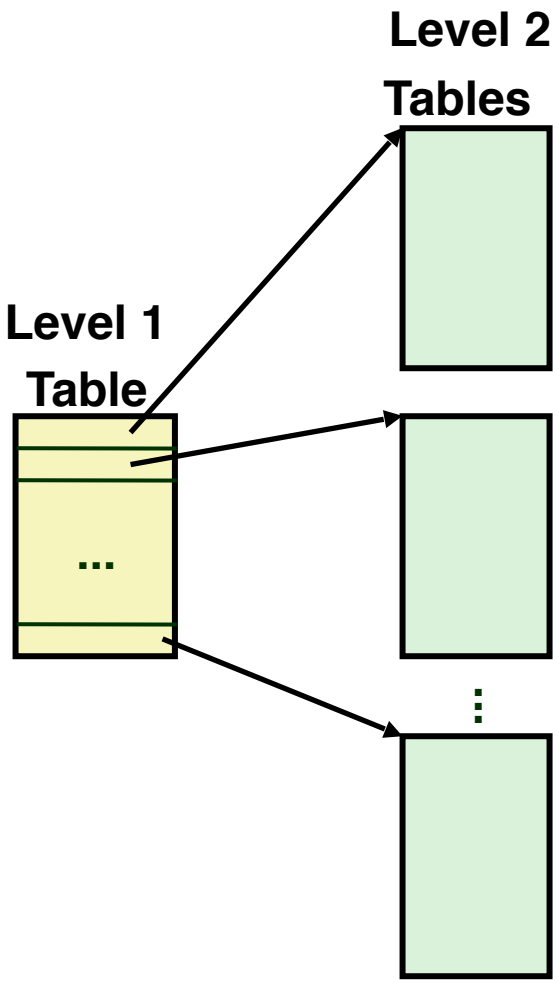
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entires in main memory
- Idea: Put page table in Virtual Memory and swap it just like data

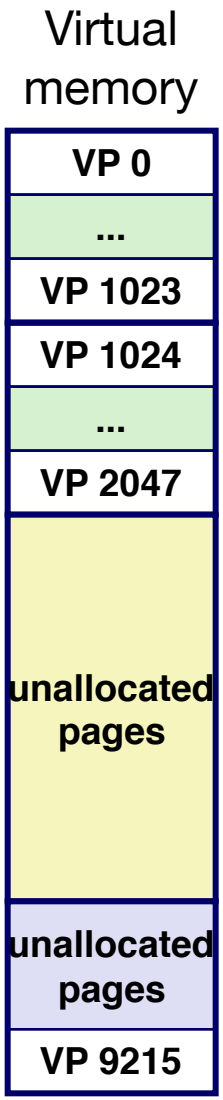


Effectively: A 2-Level Page Table

- Level 1 table:
  - Always in physical memory at a known location.
  - Each L1 PTE points to the start address of a L2 page table.
  - Bring that table to memory on-demand.
- Level 2 table:
  - Each PTE points to an actual data page



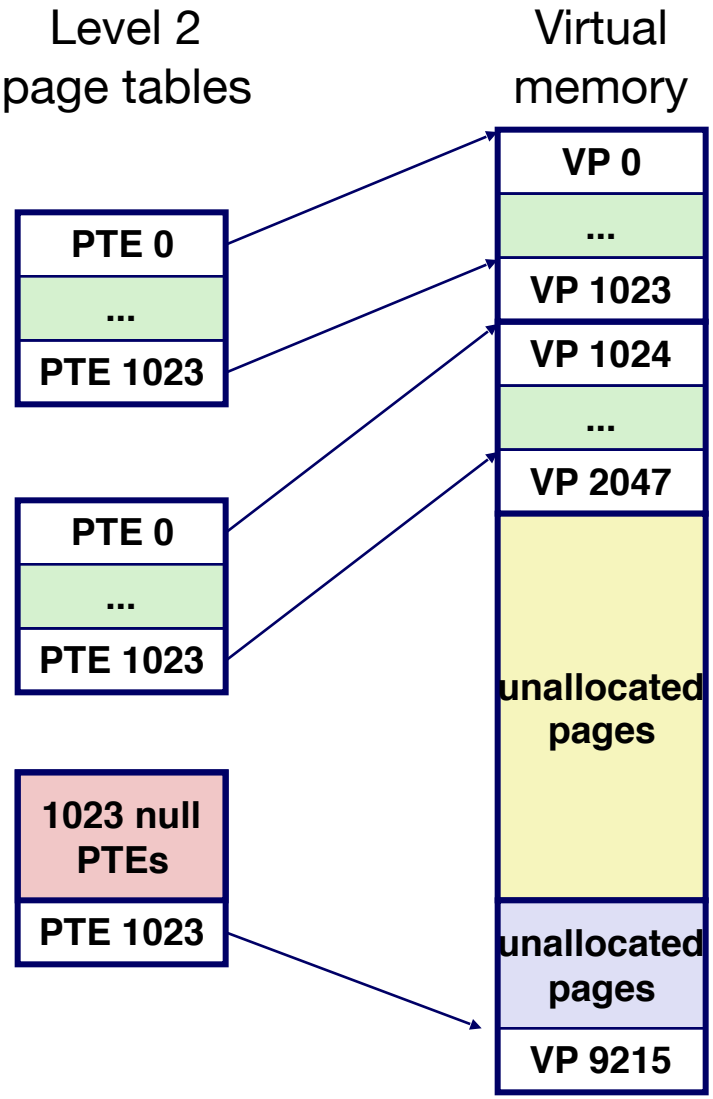
A Two-Level Page Table Hierarchy



32 bit addresses, 4KB pages, 4-byte PTEs

...

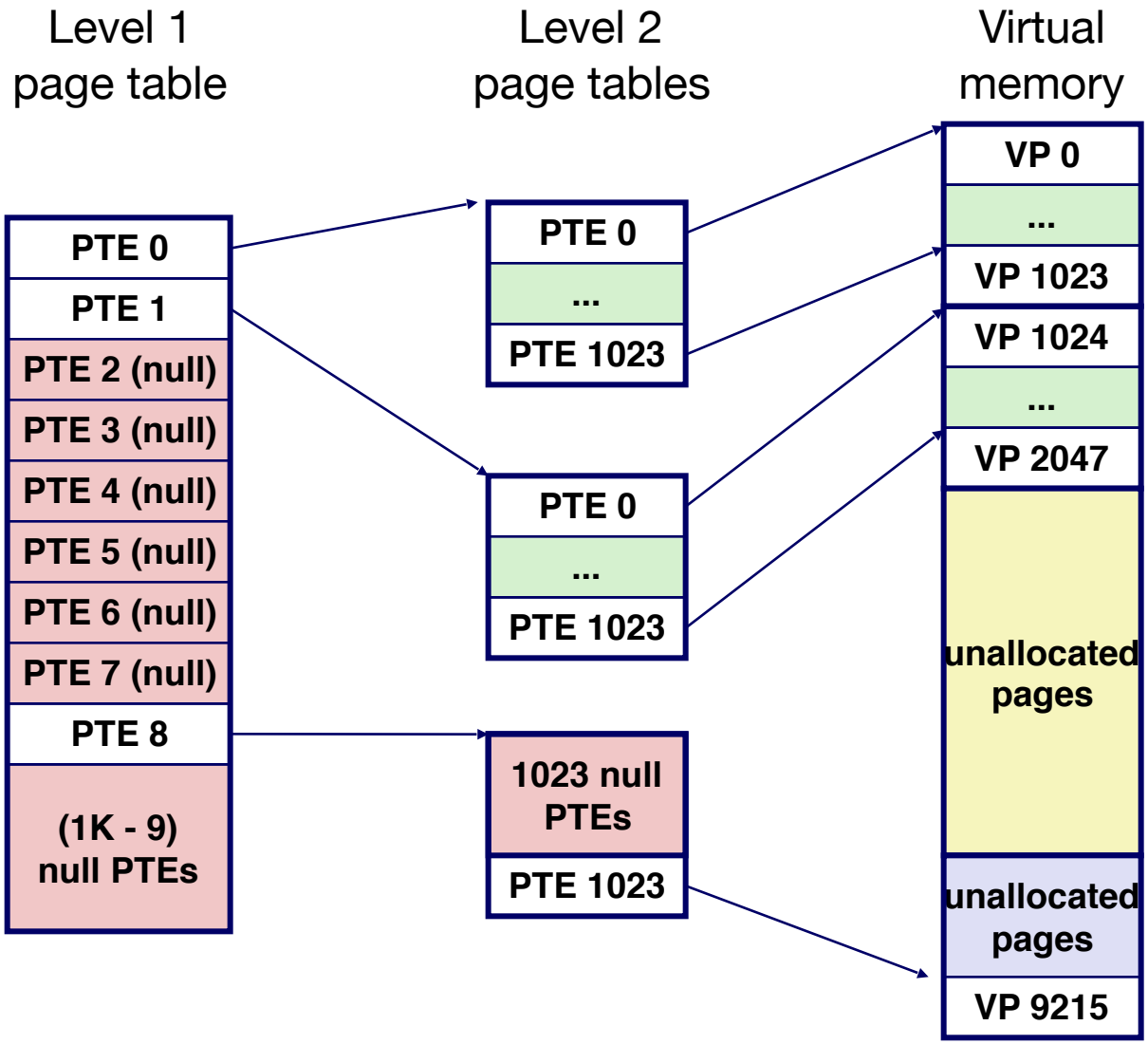
A Two-Level Page Table Hierarchy



32 bit addresses, 4KB pages, 4-byte PTEs

...

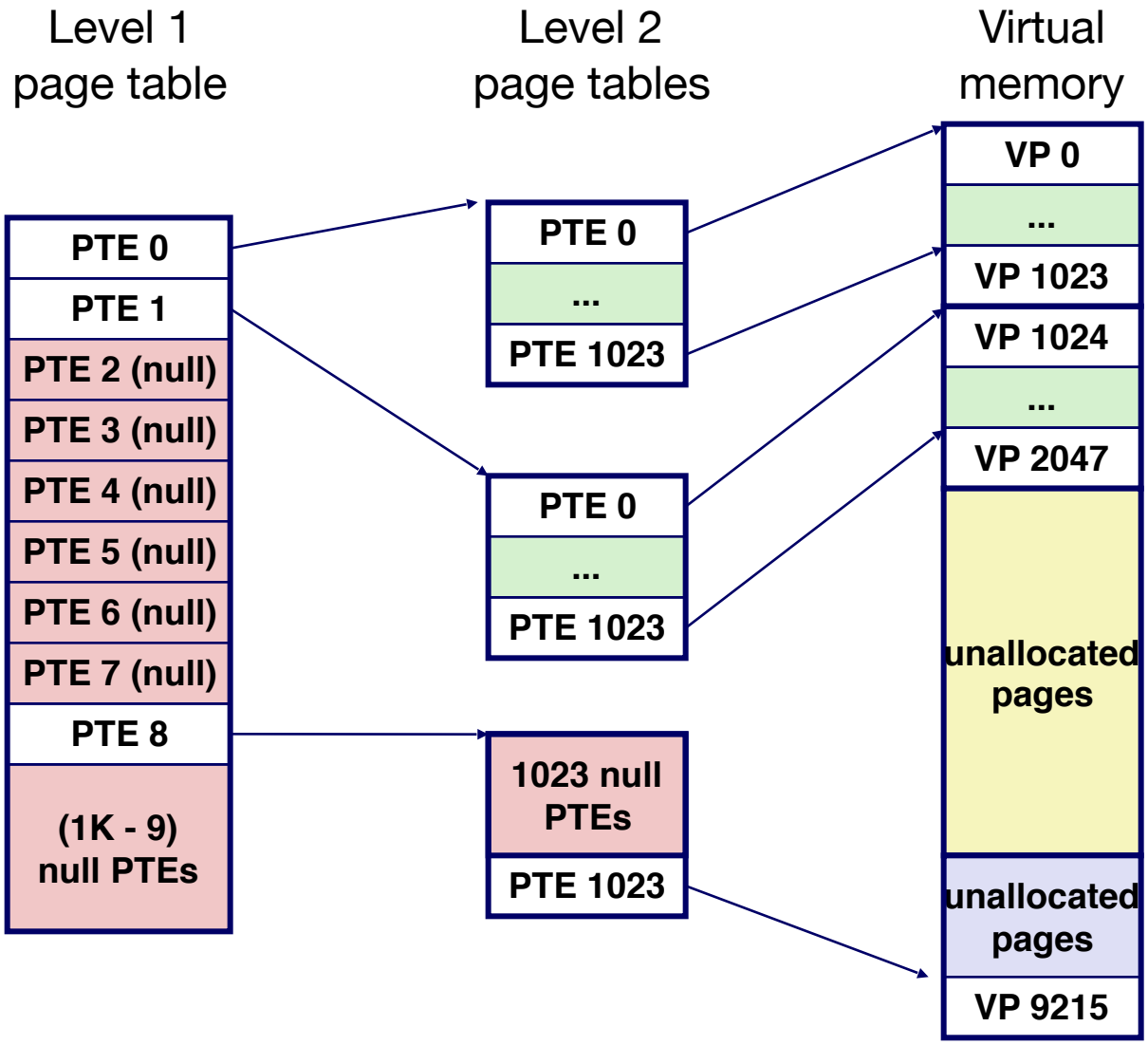
A Two-Level Page Table Hierarchy



32 bit addresses, 4KB pages, 4-byte PTEs

...

A Two-Level Page Table Hierarchy

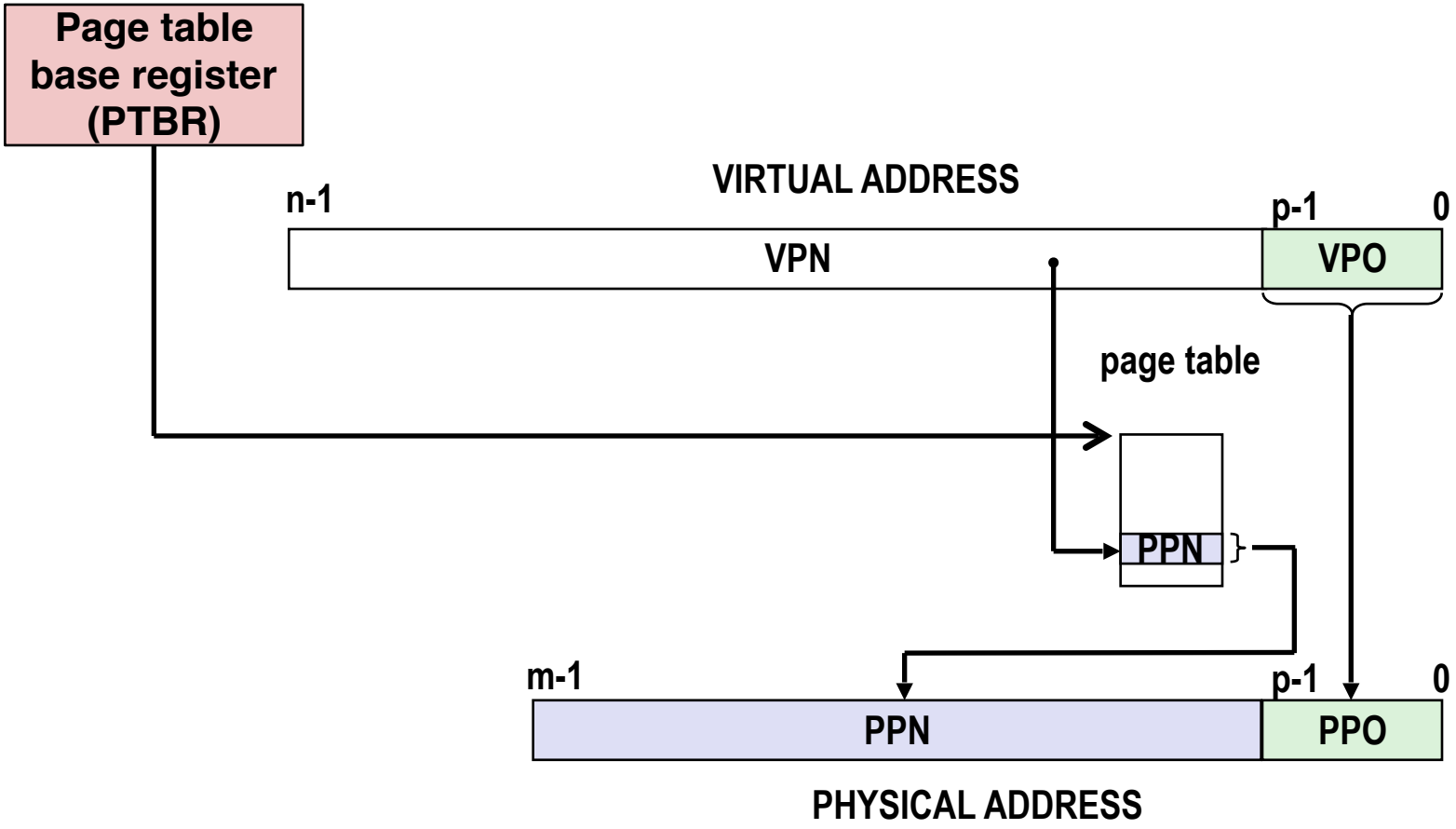


- Level 2 page table size:
  - $2^{32} / 2^{12} * 4 = 4$ MB
- Level 1 page table size:
  - $(2^{32} / 2^{12} * 4) / 2^{12} * 4 = 4$ KB

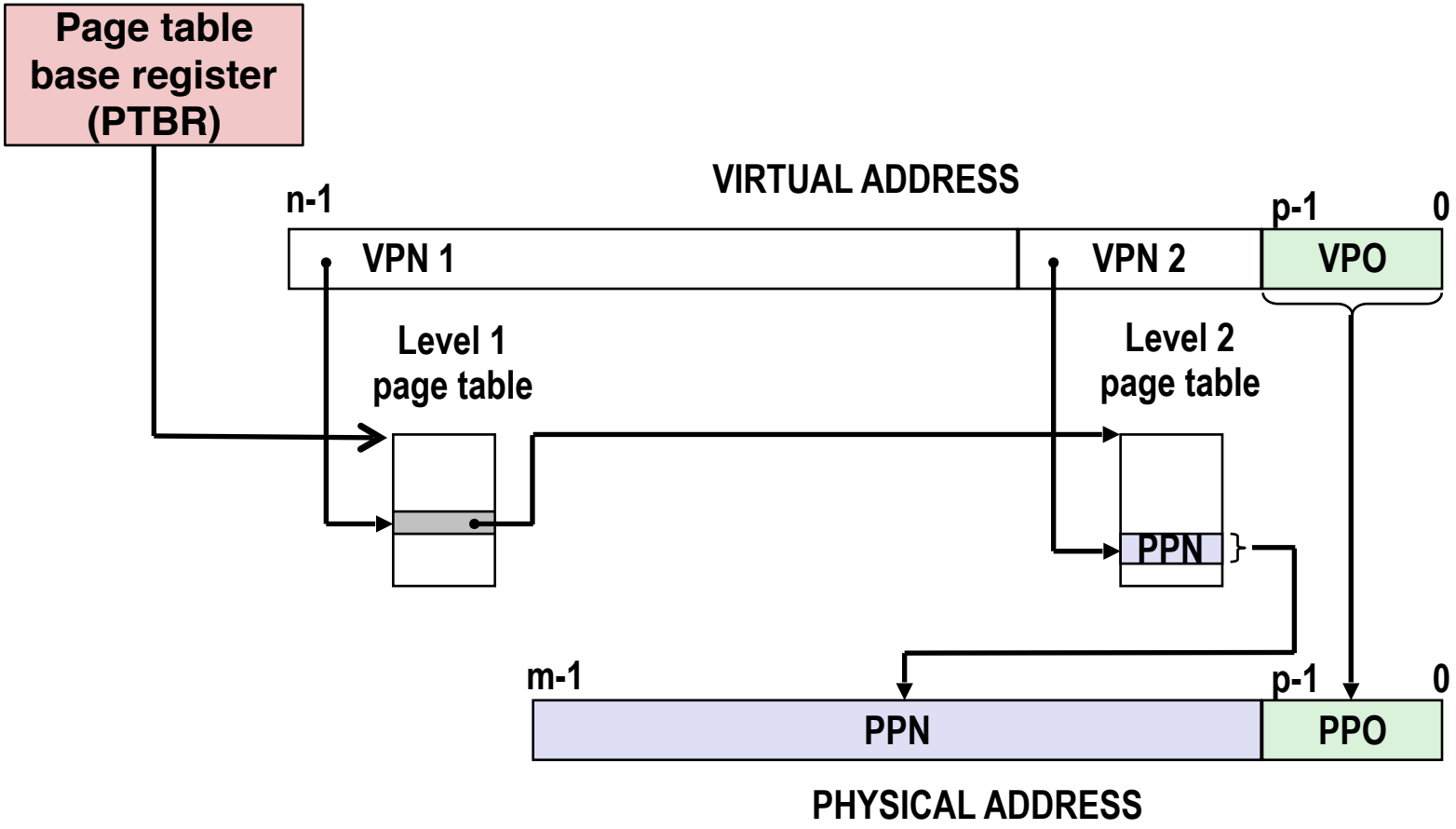
32 bit addresses, 4KB pages, 4-byte PTEs

...

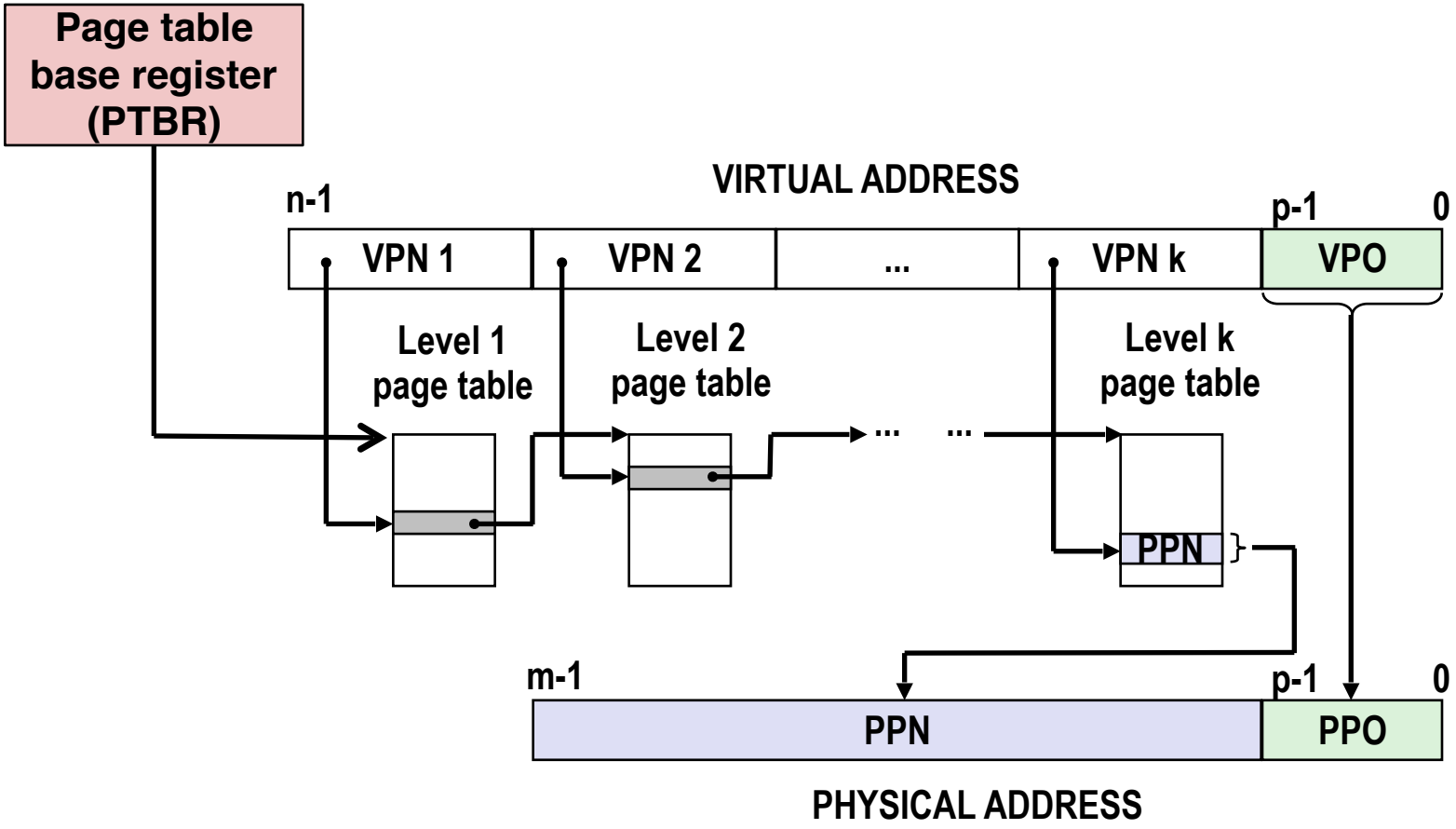
How to Access a 2-Level Page Table?



How to Access a 2-Level Page Table?



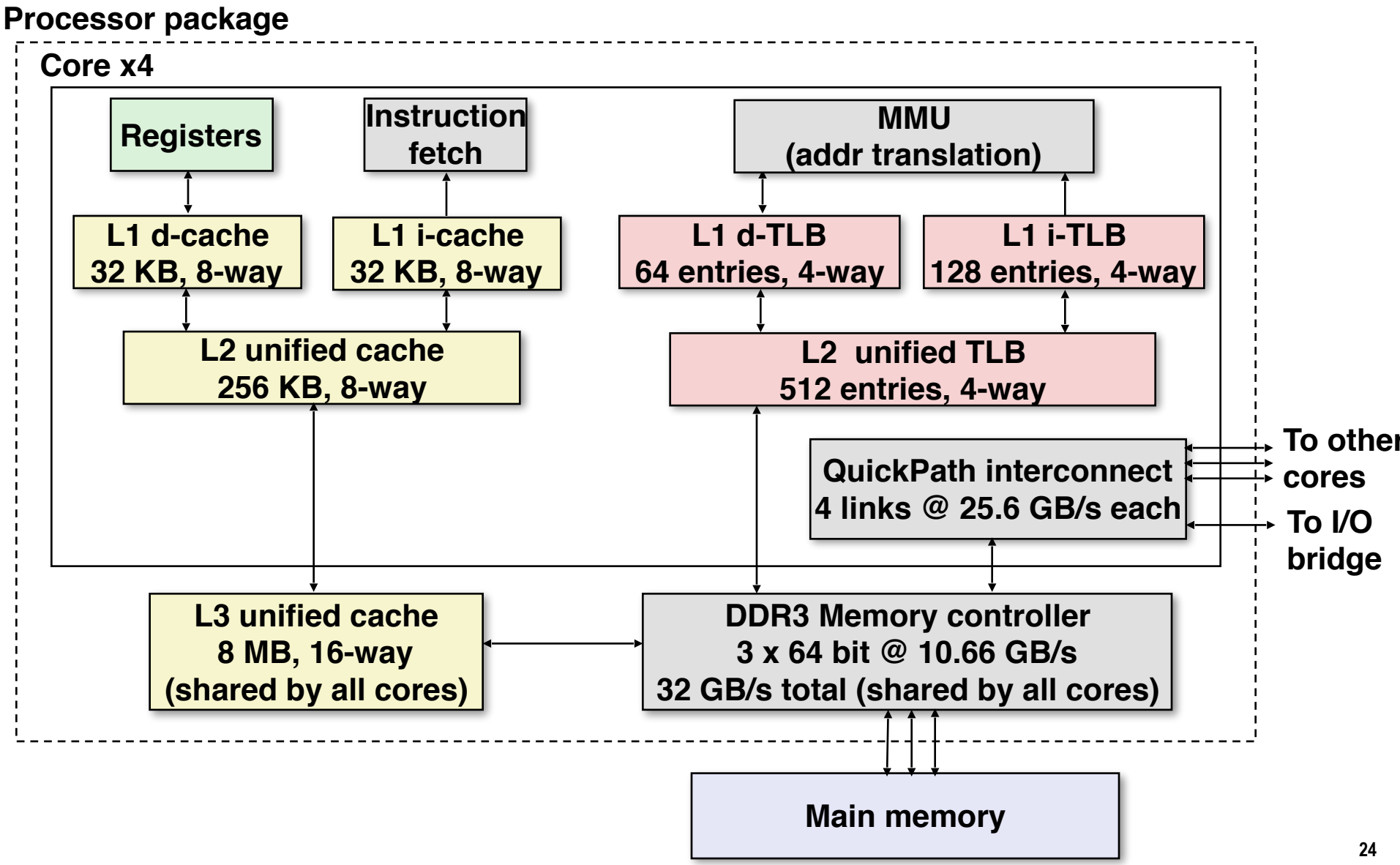
Translating with a k-level Page Table



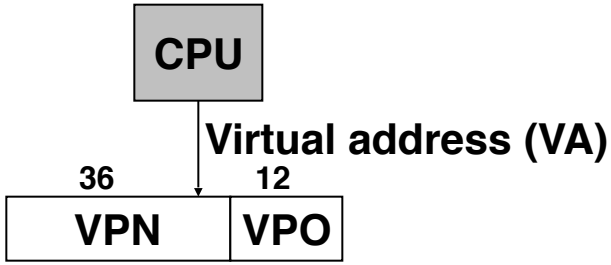
Today

- Three Virtual Memory Optimizations
  - TLB
  - Virtually-indexed, physically-tagged cache
  - Page the page table (a.k.a., multi-level page table)
- Case-study: Intel Core i7/Linux example

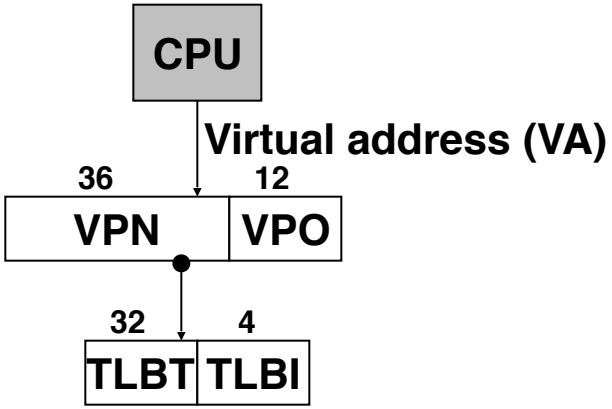
Intel Core i7 Memory System



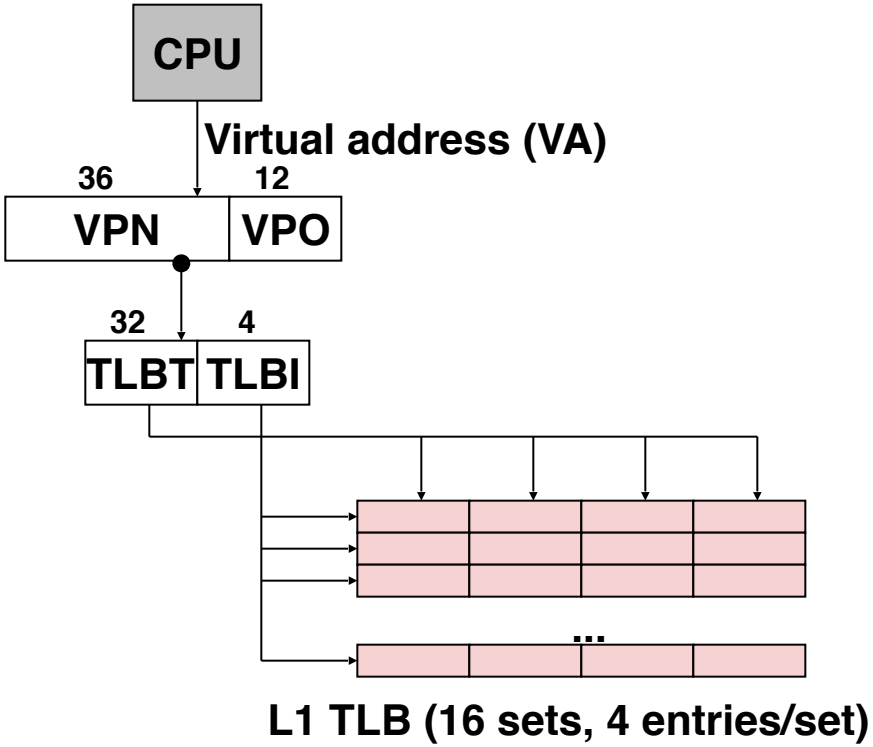
End-to-End Core i7 Address Translation



End-to-End Core i7 Address Translation

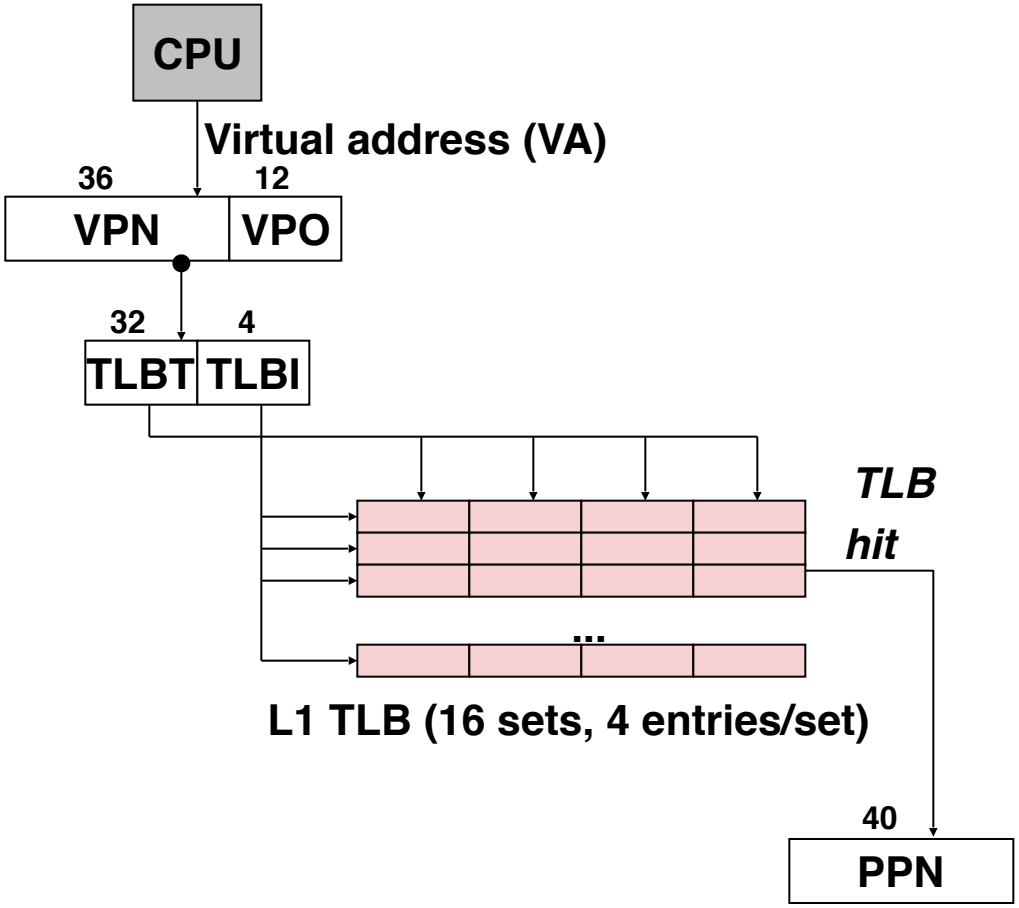


End-to-End Core i7 Address Translation

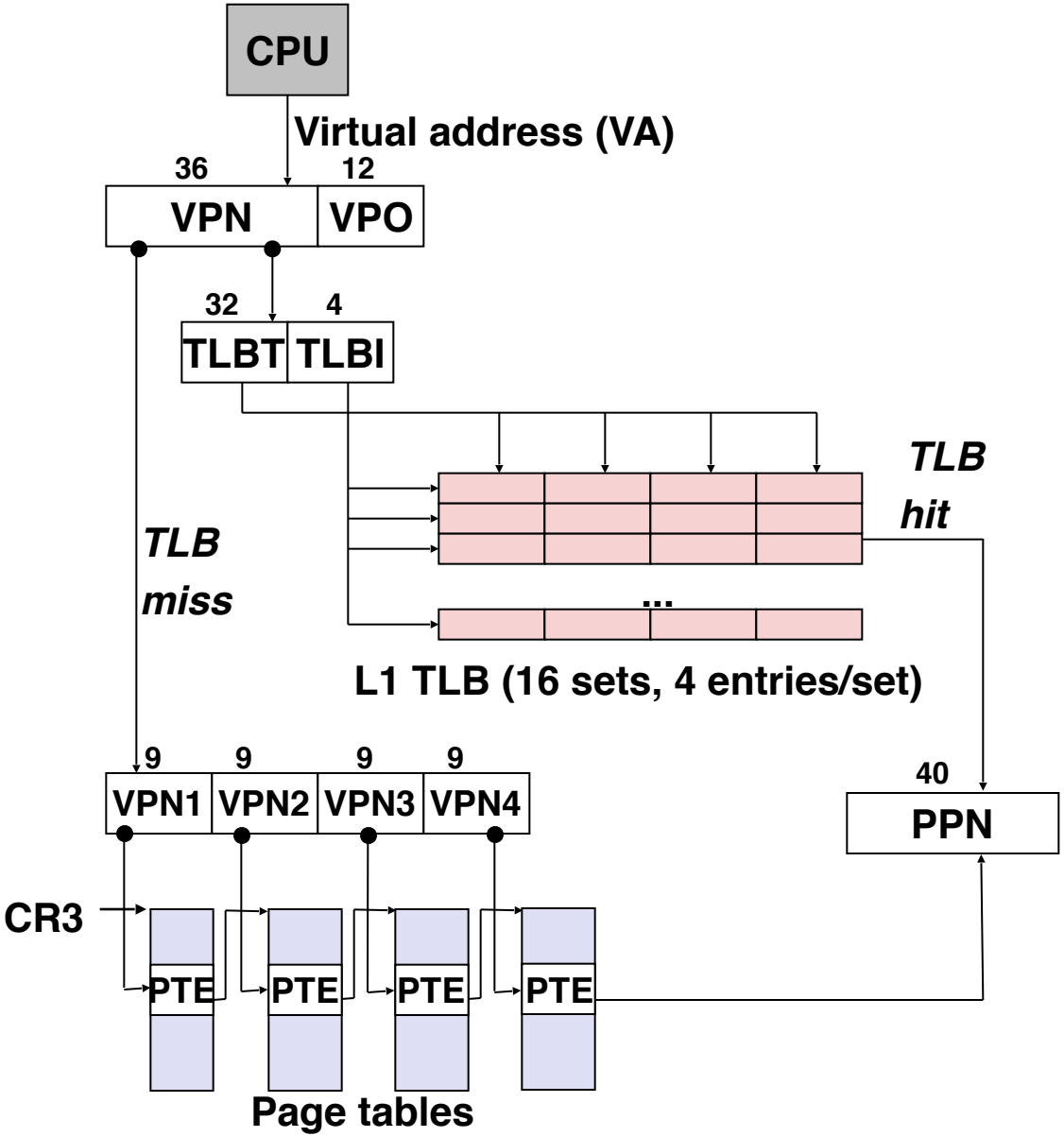


L1 TLB (16 sets, 4 entries/set)

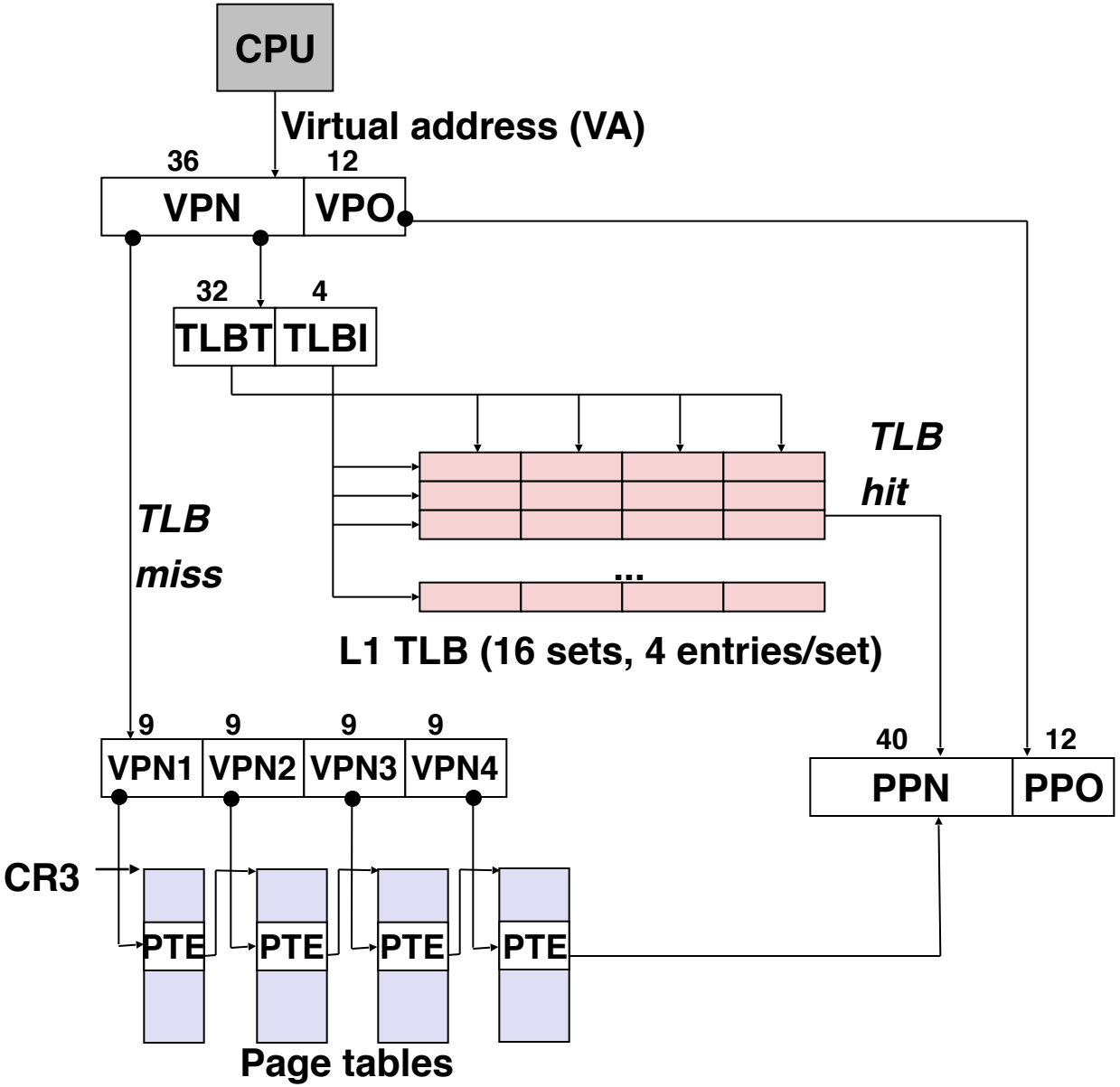
End-to-End Core i7 Address Translation



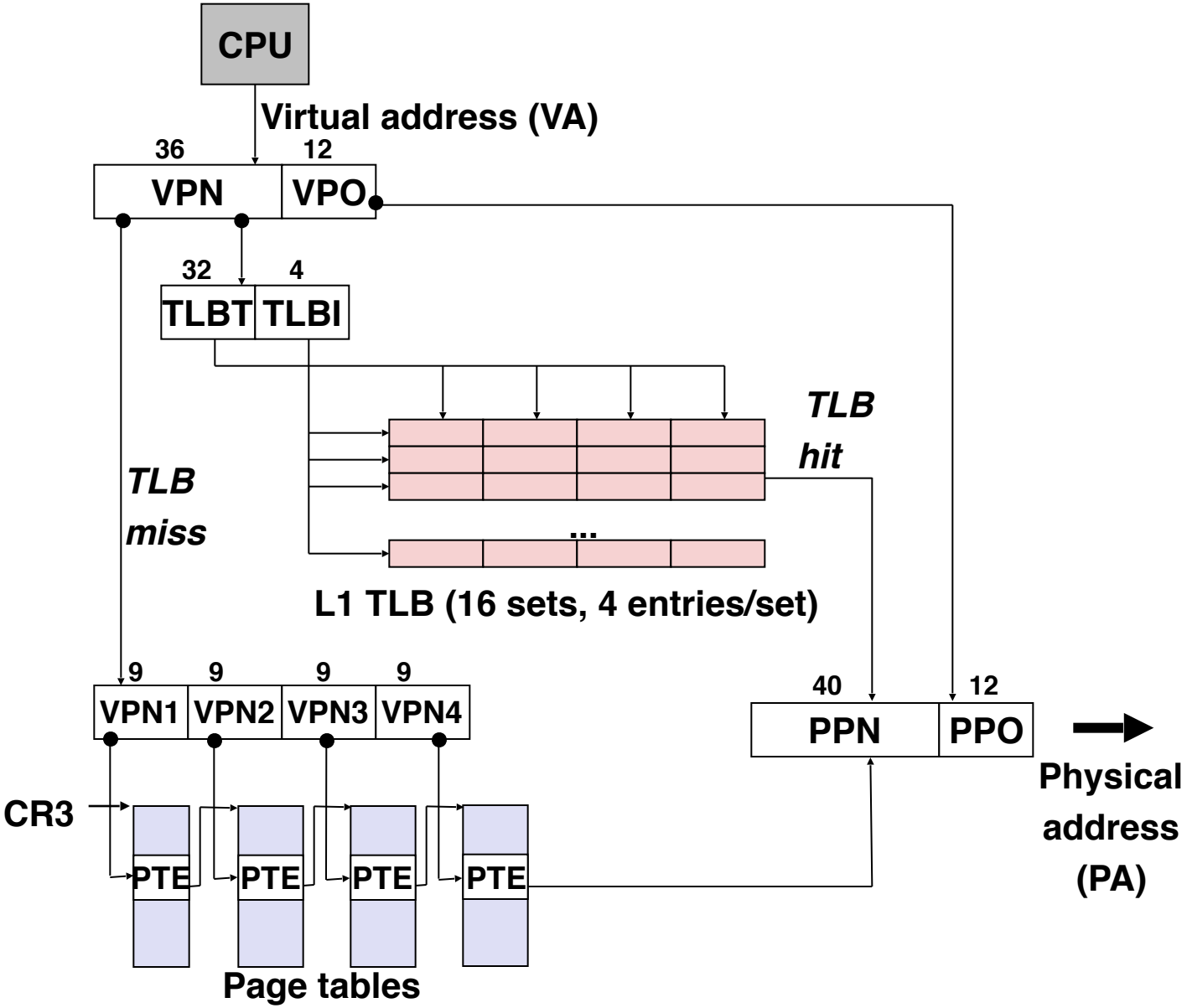
End-to-End Core i7 Address Translation



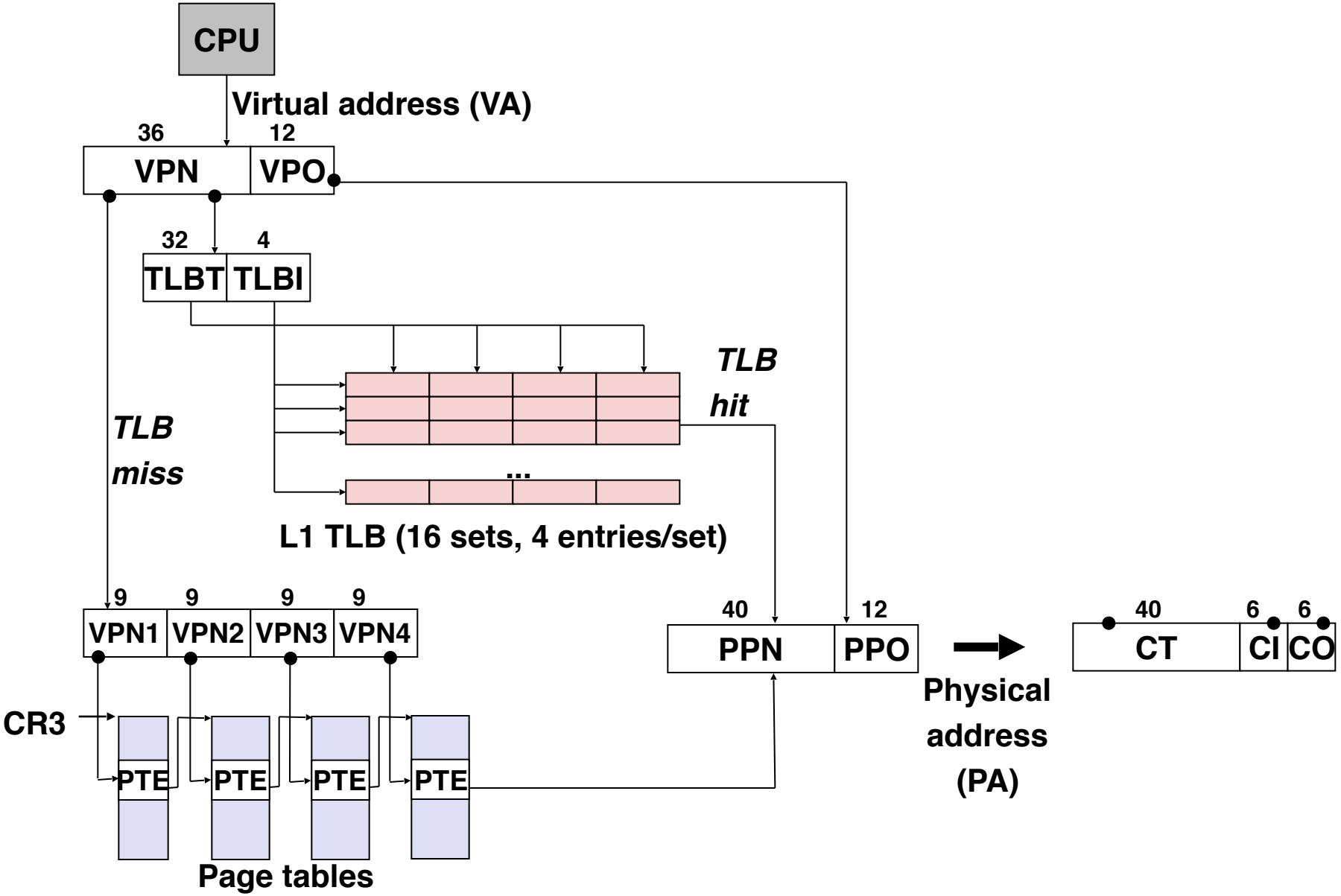
End-to-End Core i7 Address Translation



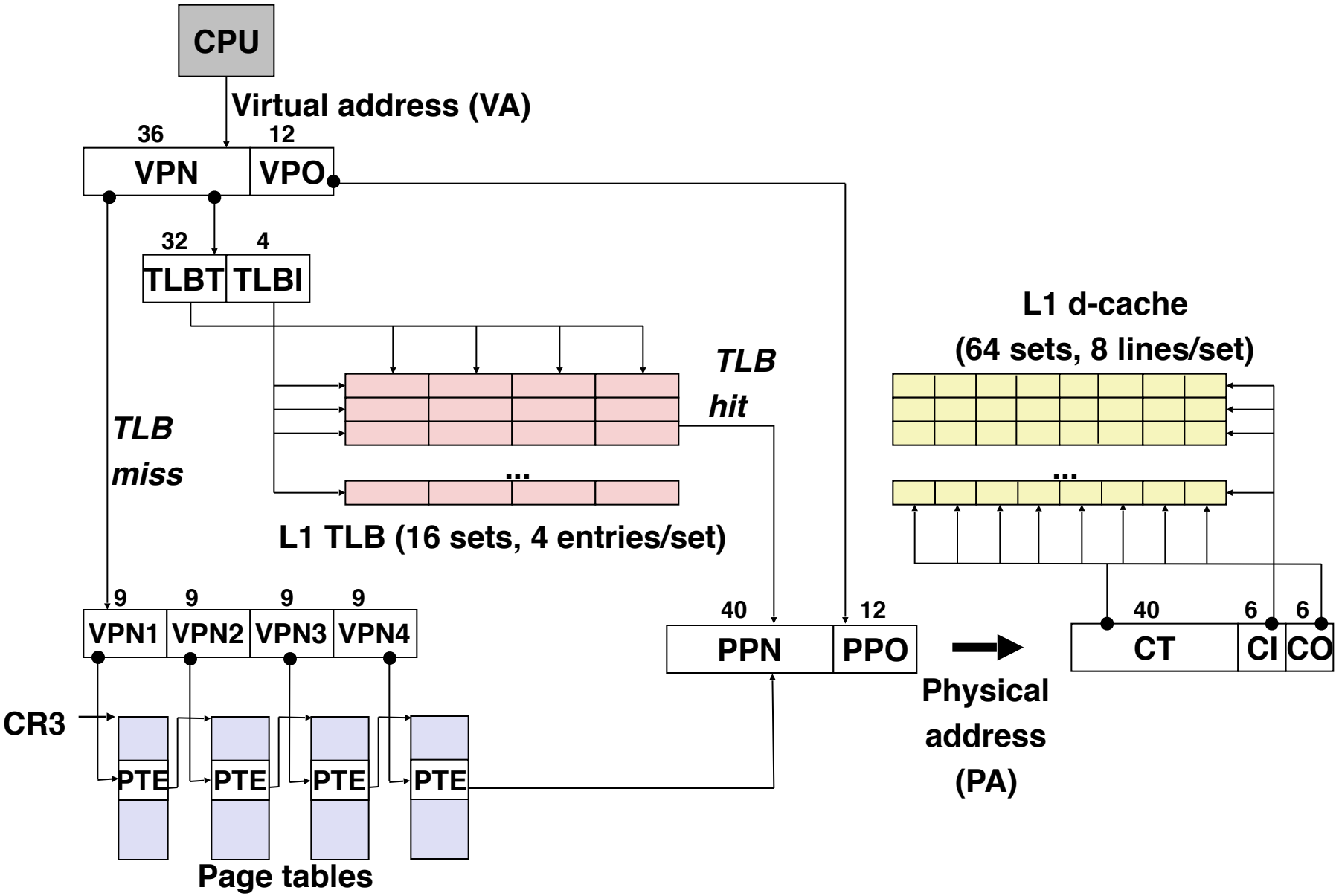
End-to-End Core i7 Address Translation



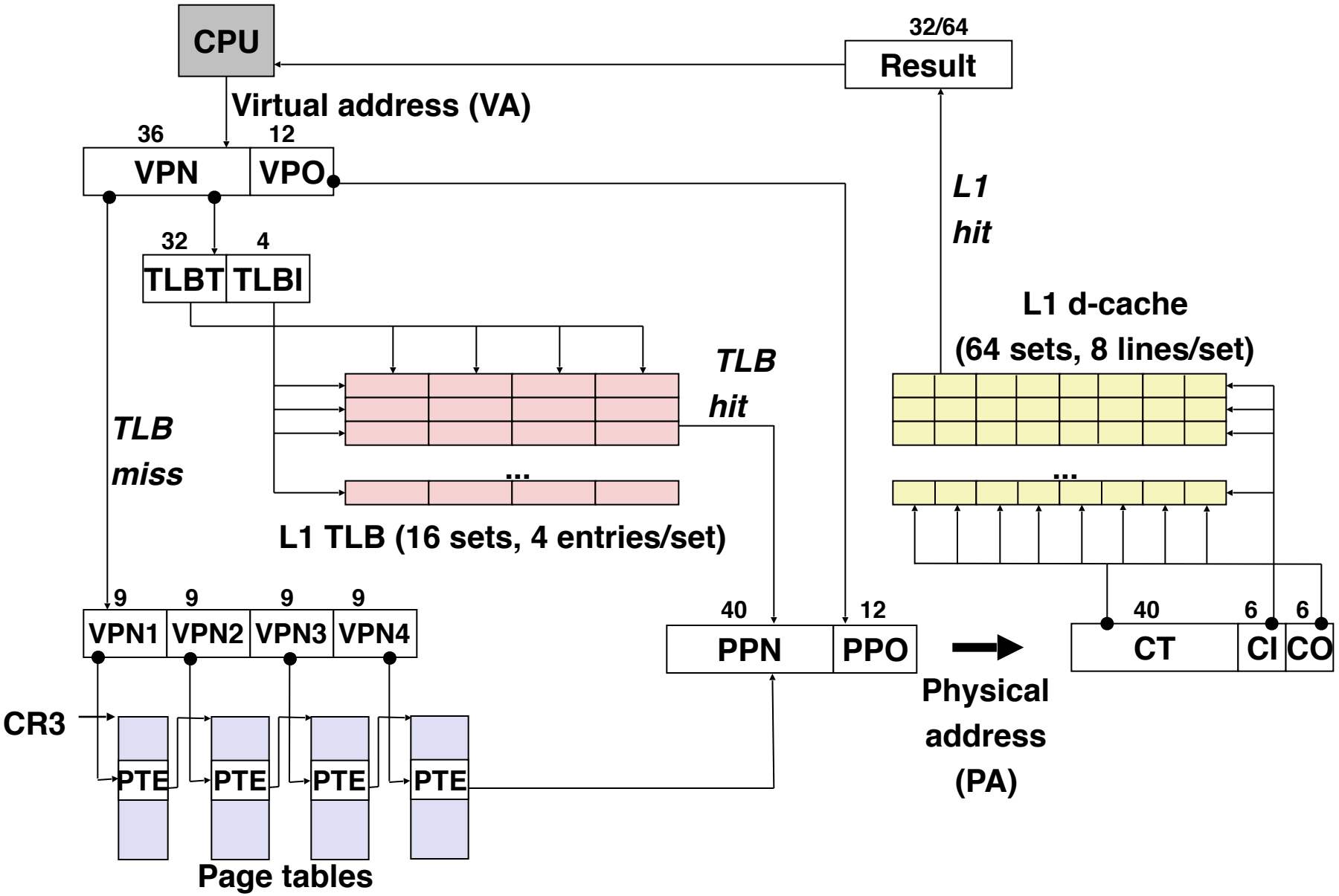
End-to-End Core i7 Address Translation



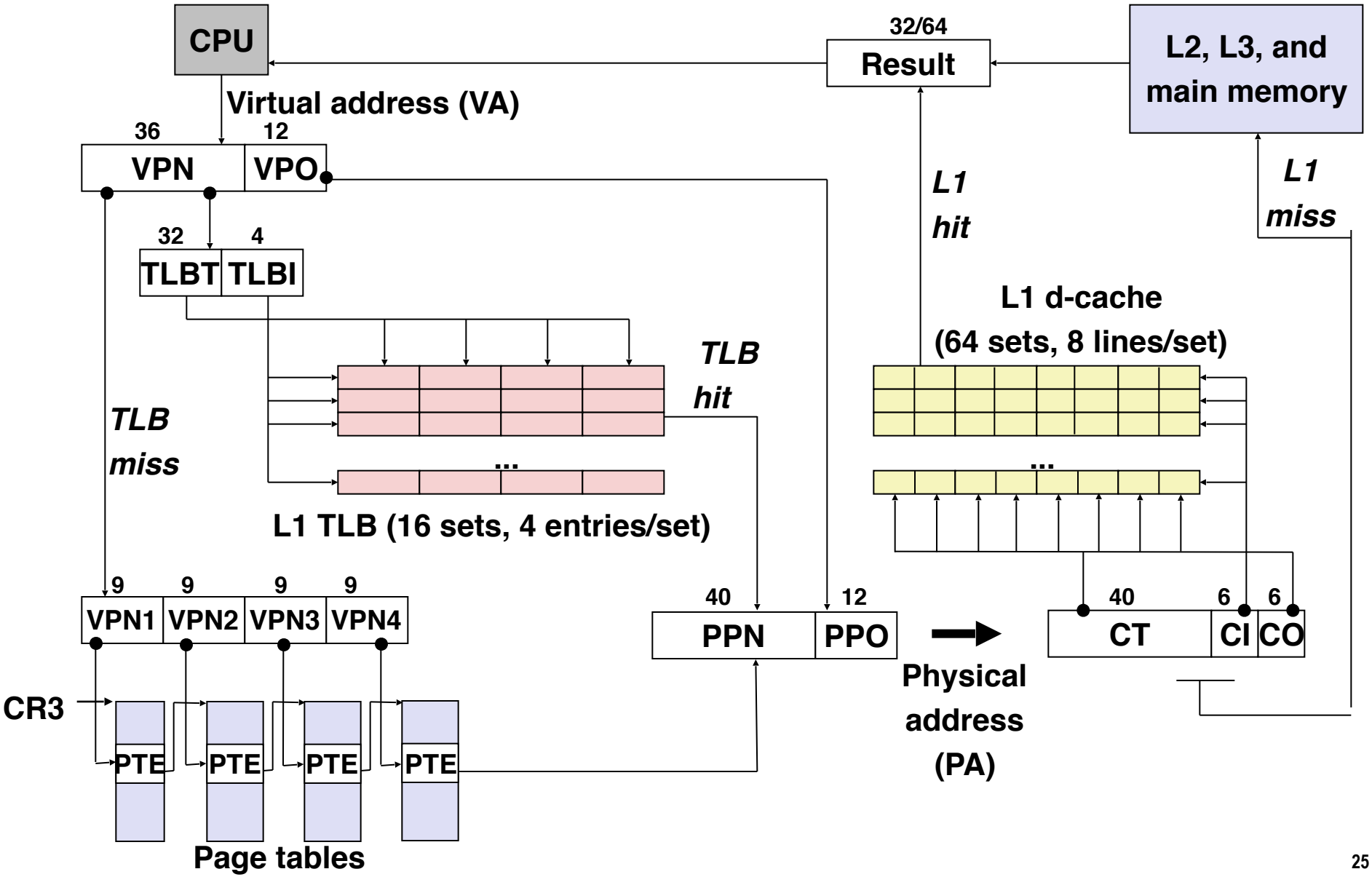
End-to-End Core i7 Address Translation



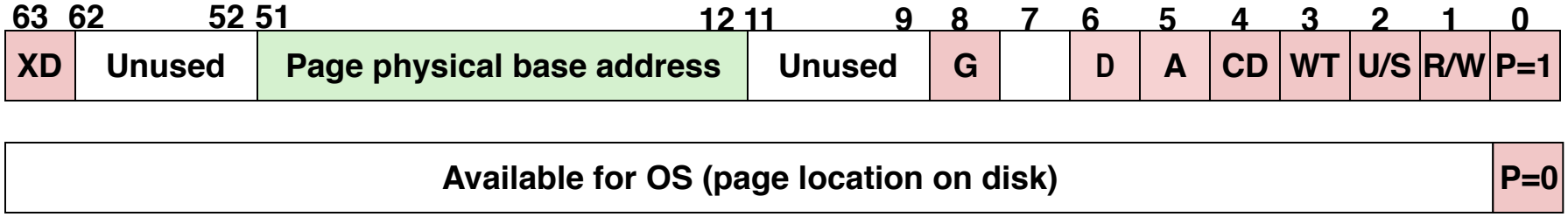
End-to-End Core i7 Address Translation



End-to-End Core i7 Address Translation



Core i7 Level 4 Page Table Entries



Each entry references a 4K child page. Significant fields:

P: Child page is present in memory (1) or not (0)

R/W: Read-only or read-write access permission for child page

U/S: User or supervisor mode access

WT: Write-through or write-back cache policy for this page

A: Reference bit (set by MMU on reads and writes, cleared by software)

D: Dirty bit (set by MMU on writes, cleared by software)

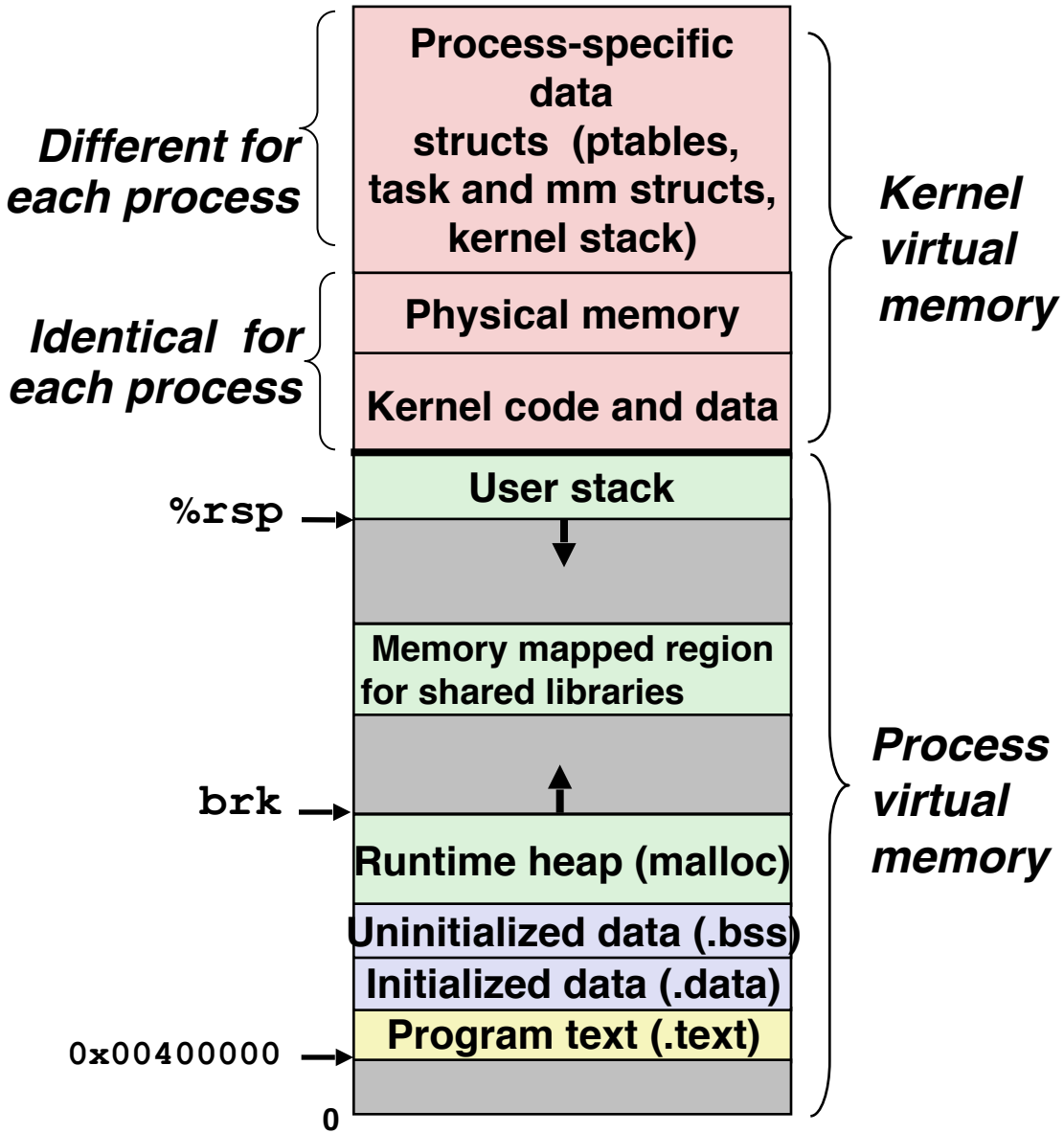
Page physical base address: 40 most significant bits of physical page address (forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.

Today

- Memory mapping
- Dynamic memory allocation

Virtual Address Space of a Linux Process

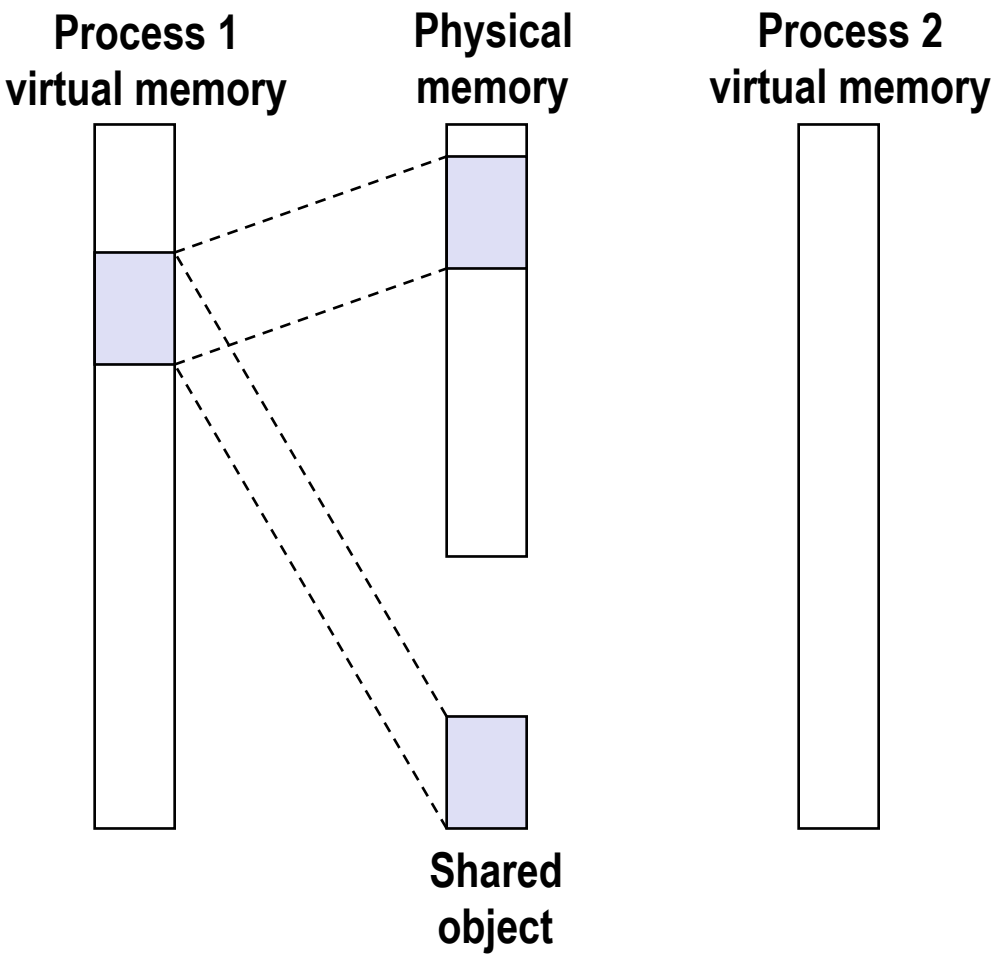


Memory Mapping For Sharing

- Multiple processes often share data
  - Different processes that run the same code (e.g., shell)
  - Different processes linked to the same standard libraries
  - Different processes share the same file
- It is wasteful to create exact copies of the share object
- Memory mapping allow us to easily share objects
  - Different VM pages point to the same physical page/object

Sharing Revisited: Shared Objects

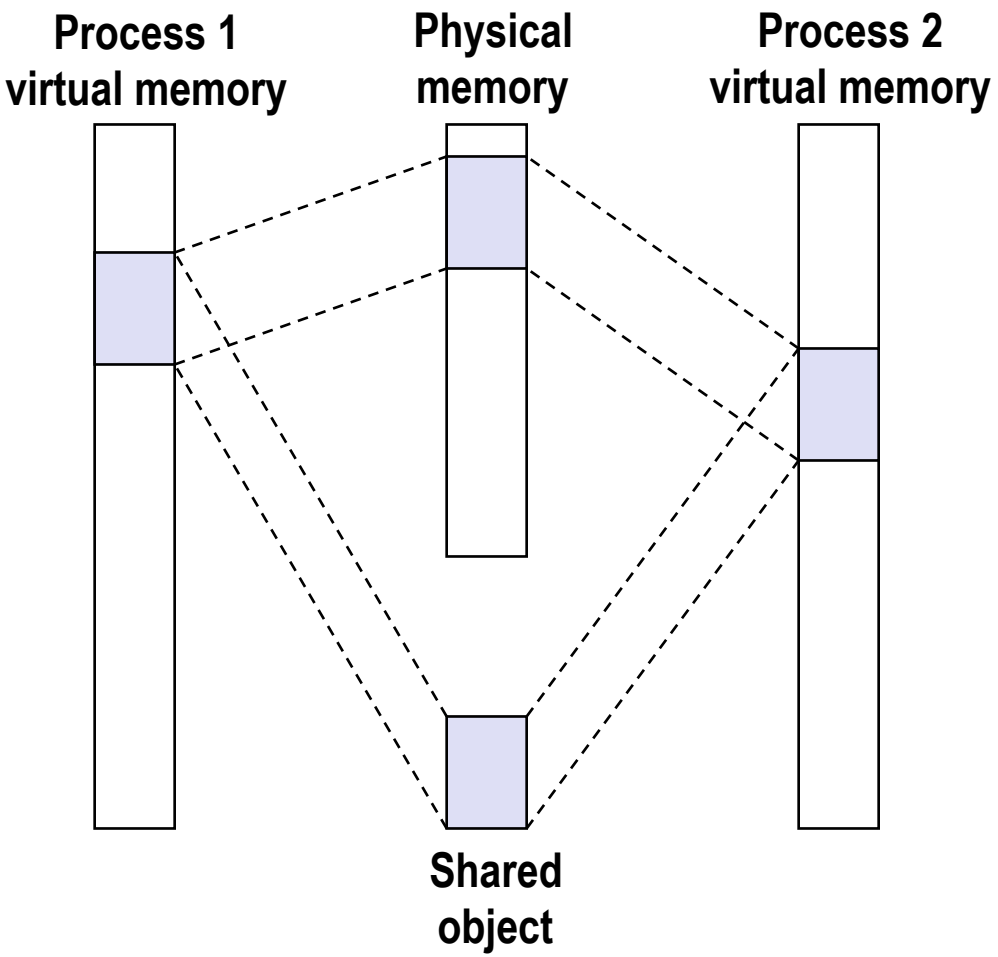
- Process 1 maps the shared object.



- The kernel remembers that the object (backed by a unique file) is mapped by Proc. 1 to some physical pages.

Sharing Revisited: Shared Objects

- Process 2 maps the shared object.



- The kernel remembers that the object (backed by a unique file) is mapped by Proc. 1 to some physical pages.
- Now when Proc. 2 wants to access the same object, the kernel can simply point the PTEs of Proc. 2 to the already-mapped physical pages.

The Problem…

- What if Proc. 1 now wants to modify the shared object, but doesn’t want the modification to be visible to Proc. 2

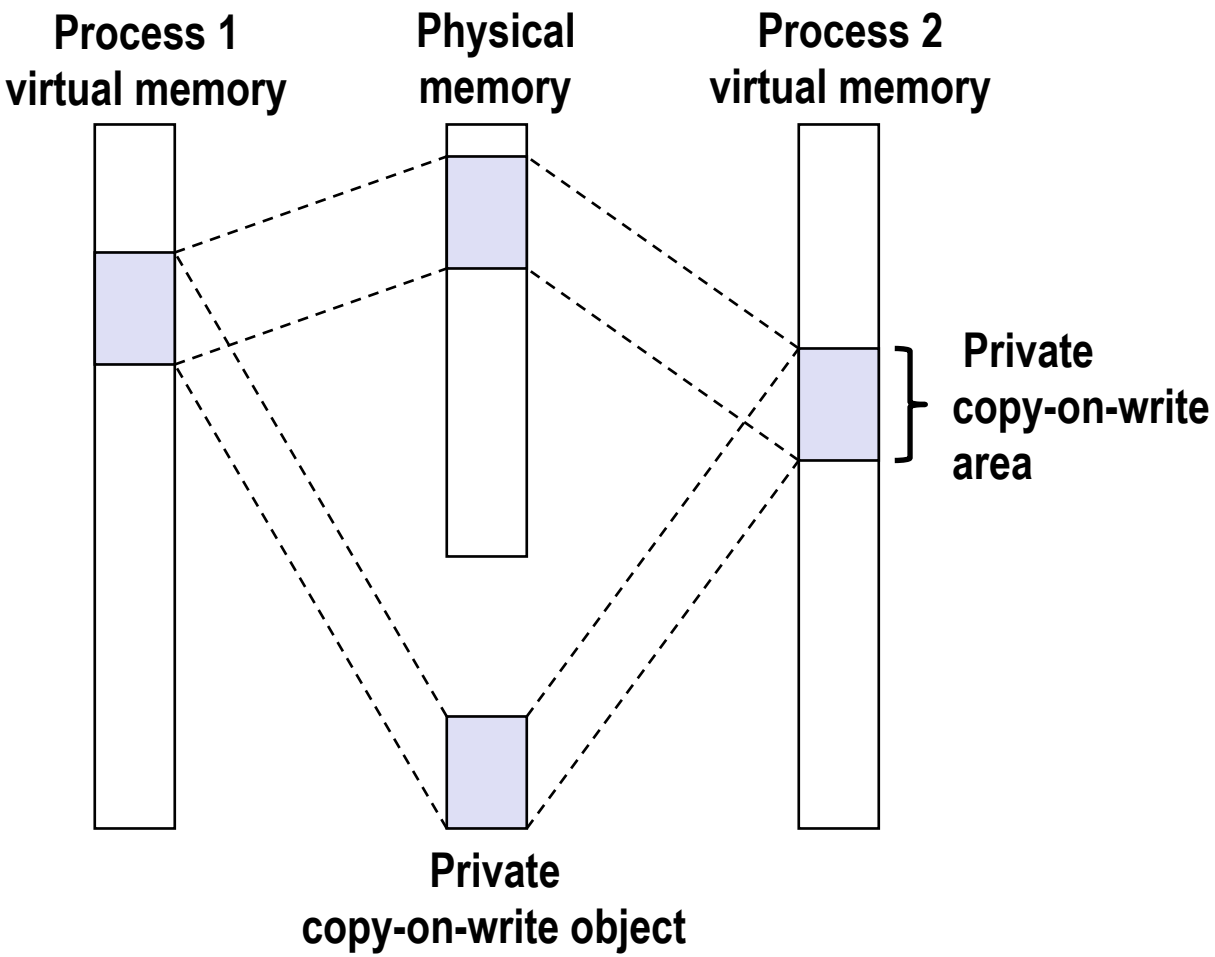
The Problem…

- What if Proc. 1 now wants to modify the shared object, but doesn’t want the modification to be visible to Proc. 2
- Simplest solution: always create duplicate copies of shared objects at the cost of wasting space. Not ideal.

The Problem…

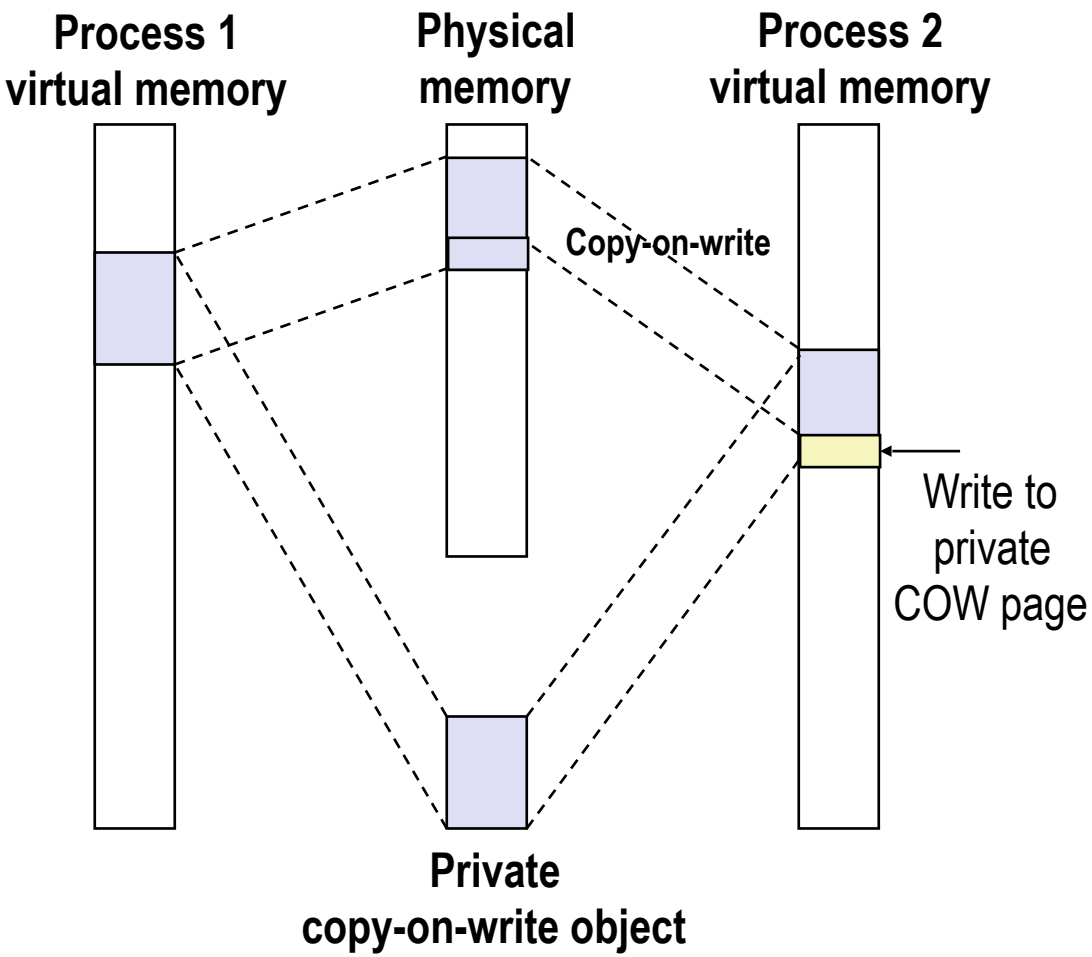
- What if Proc. 1 now wants to modify the shared object, but doesn't want the modification to be visible to Proc. 2
- Simplest solution: always create duplicate copies of shared objects at the cost of wasting space. Not ideal.
- Idea: Copy-on-write (COW)
  - First pretend that both processes will share the objects without modifying them. If modification happens, create separate copies.

Private Copy-on-write (COW) Objects



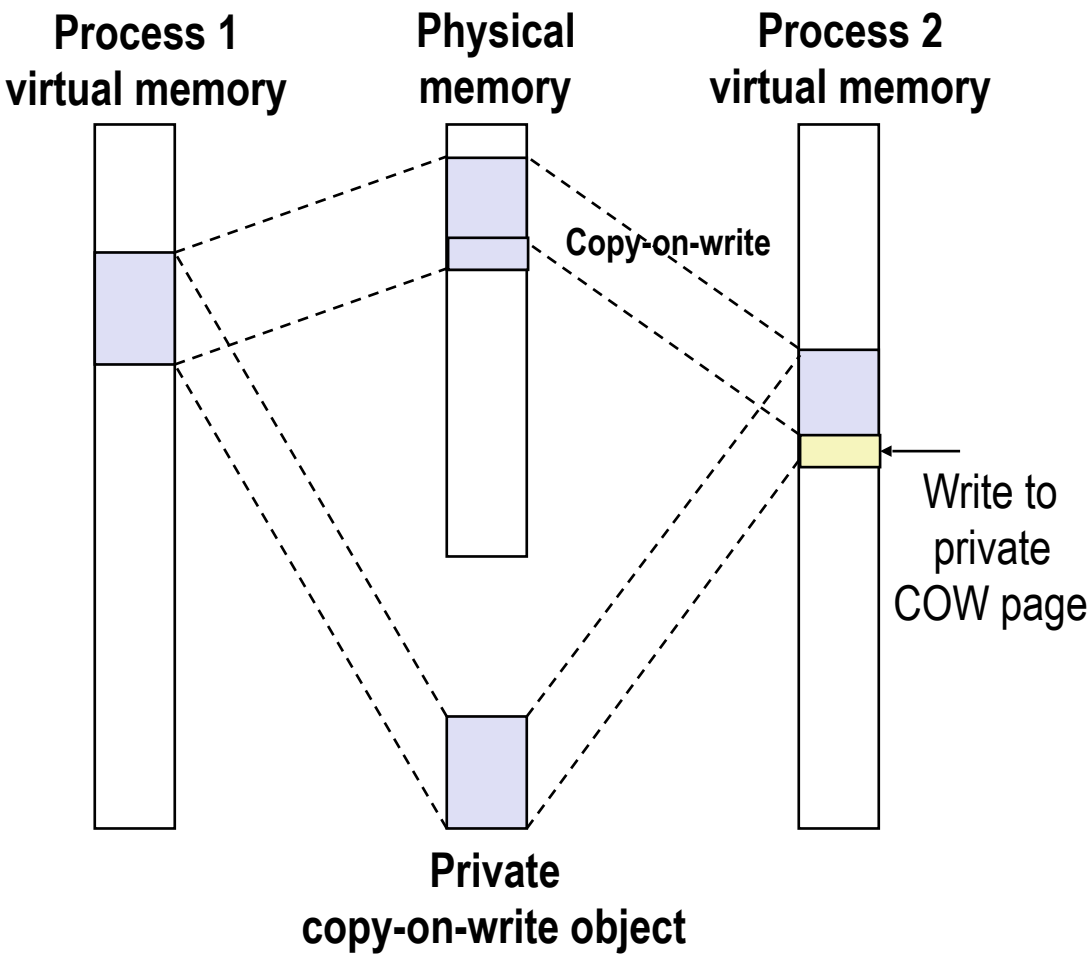
- Two processes mapping a *private copy-on-write (COW)* object.
- Area flagged as private copy-on-write (COW)
- PTEs in private areas are flagged as read-only

Private Copy-on-write (COW) Objects



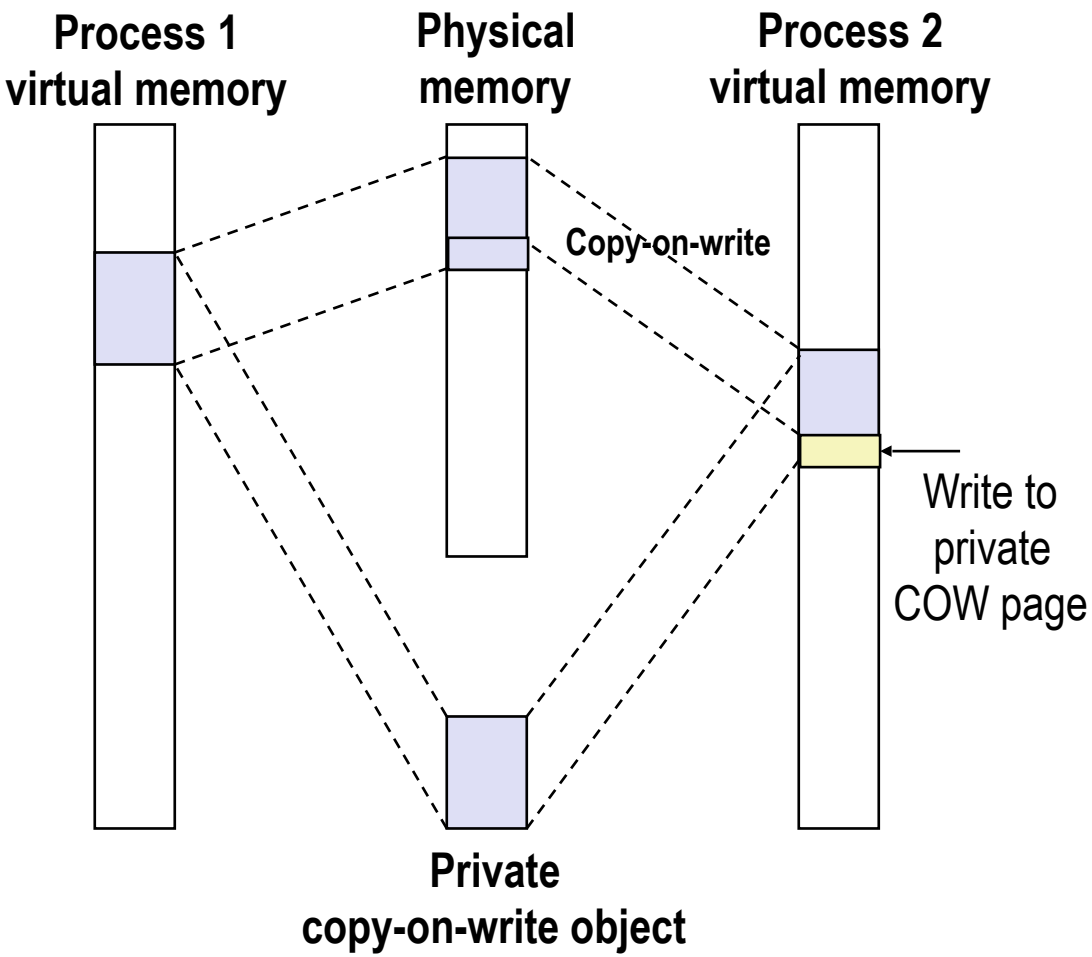
- Instruction writing to private page triggers page (protection) fault.

Private Copy-on-write (COW) Objects



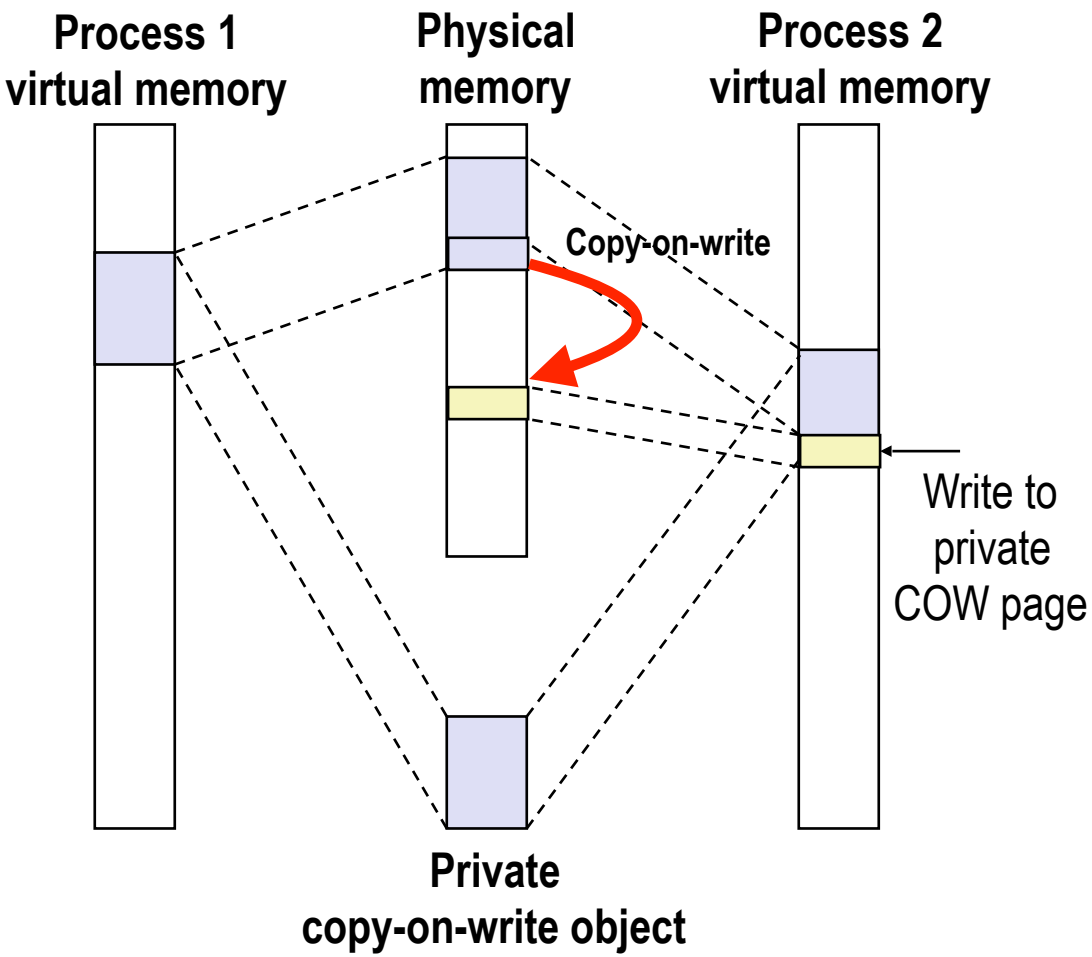
- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it's a COW object

Private Copy-on-write (COW) Objects



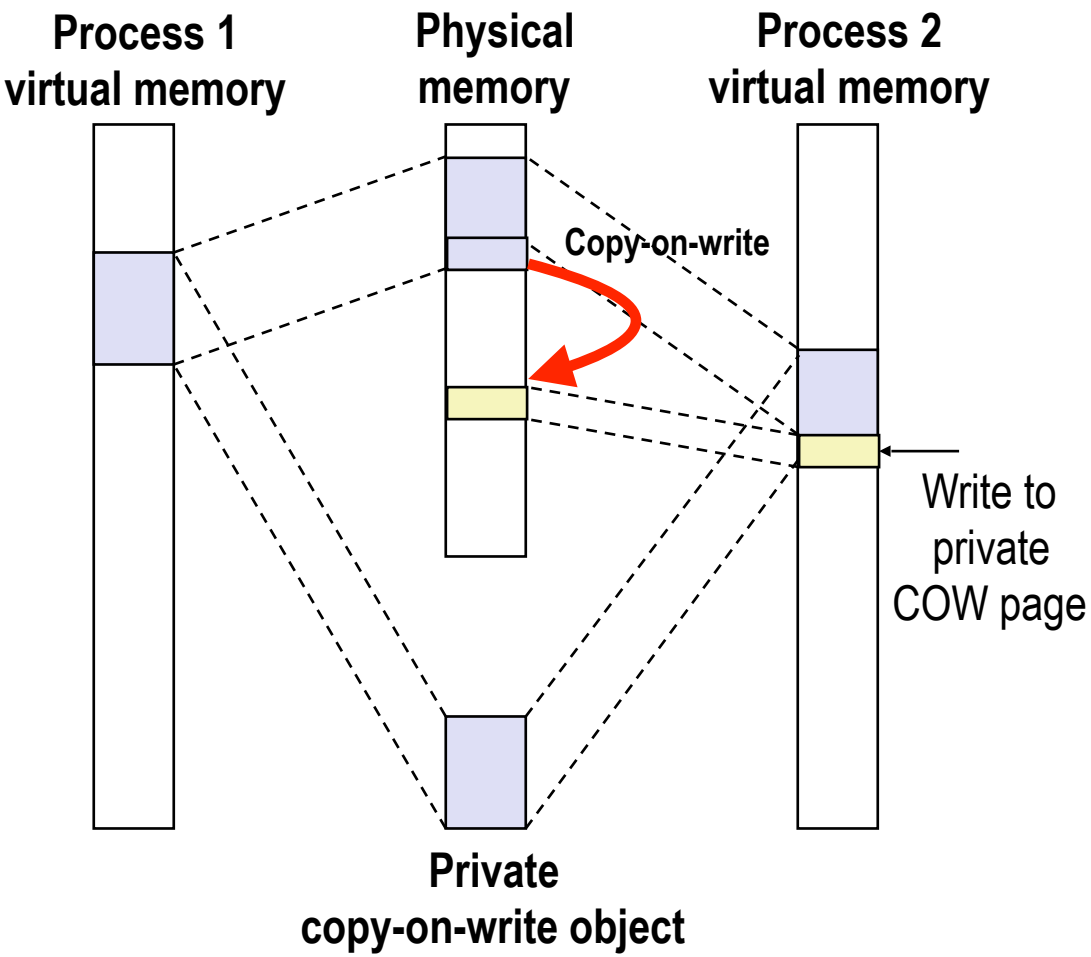
- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it's a COW object
- Handler then creates new R/W page.

Private Copy-on-write (COW) Objects



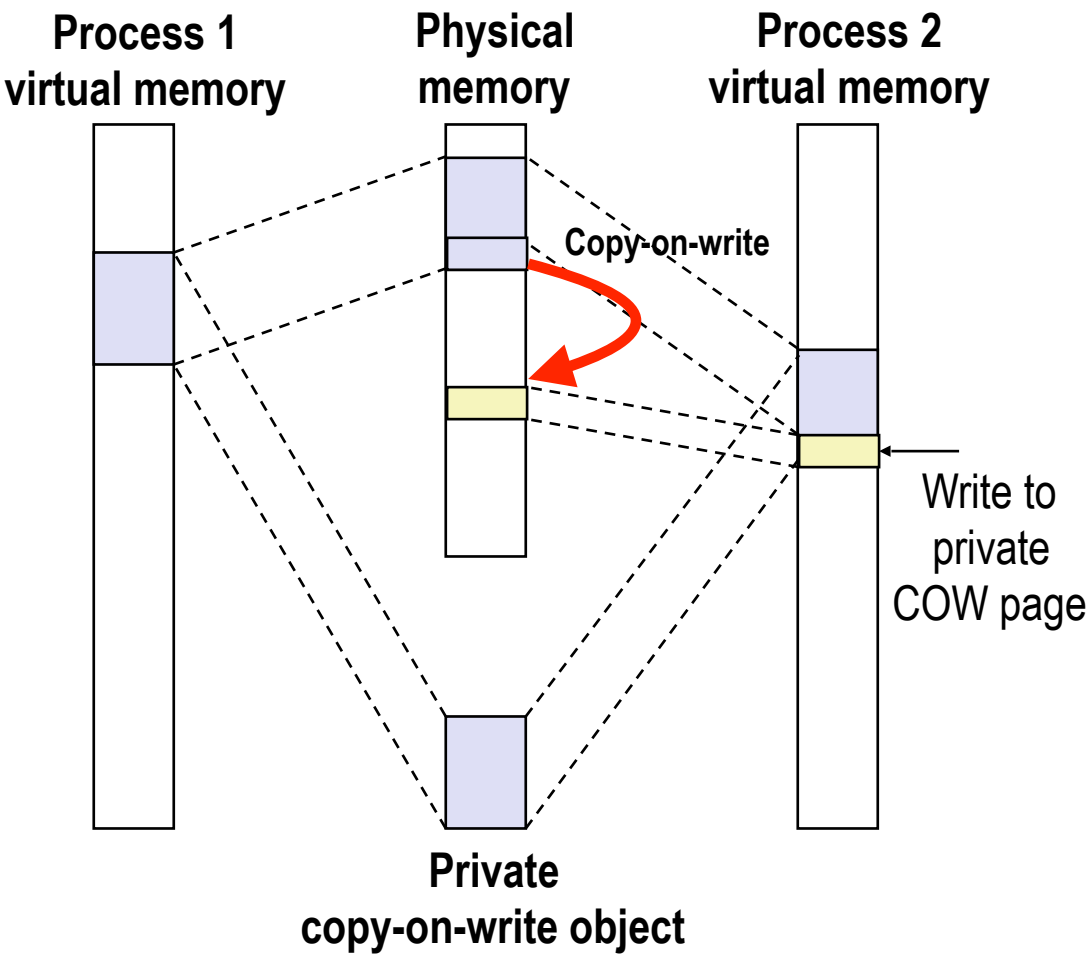
- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it's a COW object
- Handler then creates new R/W page.

Private Copy-on-write (COW) Objects



- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it's a COW object
- Handler then creates new R/W page.
- Instruction restarts upon handler return.

Private Copy-on-write (COW) Objects



- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it's a COW object
- Handler then creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

User-Level Memory Mapping

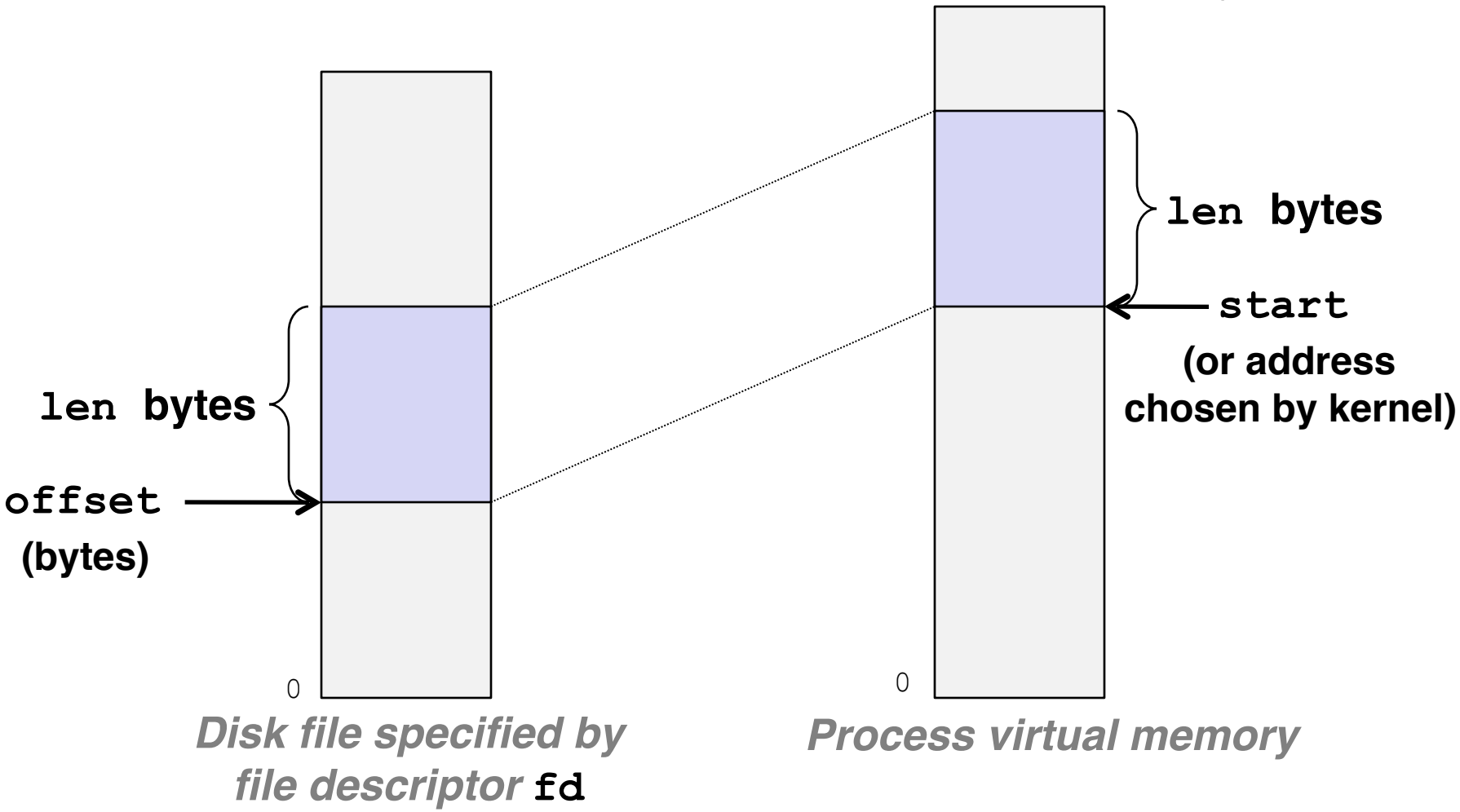
```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- Map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start`
  - `start`: may be NULL for “pick an address”
  - `prot`: PROT_READ, PROT_WRITE, ...
  - `flags`: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...

- Return a pointer to start of mapped area (may not be `start`)

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```



Example: Using `mmap` to Copy Files

- Copying a file to stdout without transferring data to user space
  - i.e., no file data is copied to user stack

```c
#include "csapp.h"

void mmapcopy(int fd, int size)
{

    /* Ptr to memory mapped area */
    char *bufp;

    bufp = mmap(NULL, size,
                PROT_READ,
                MAP_PRIVATE,
                fd, 0);
    Write(1, bufp, size);
    return;
}
```

mmapcopy.c

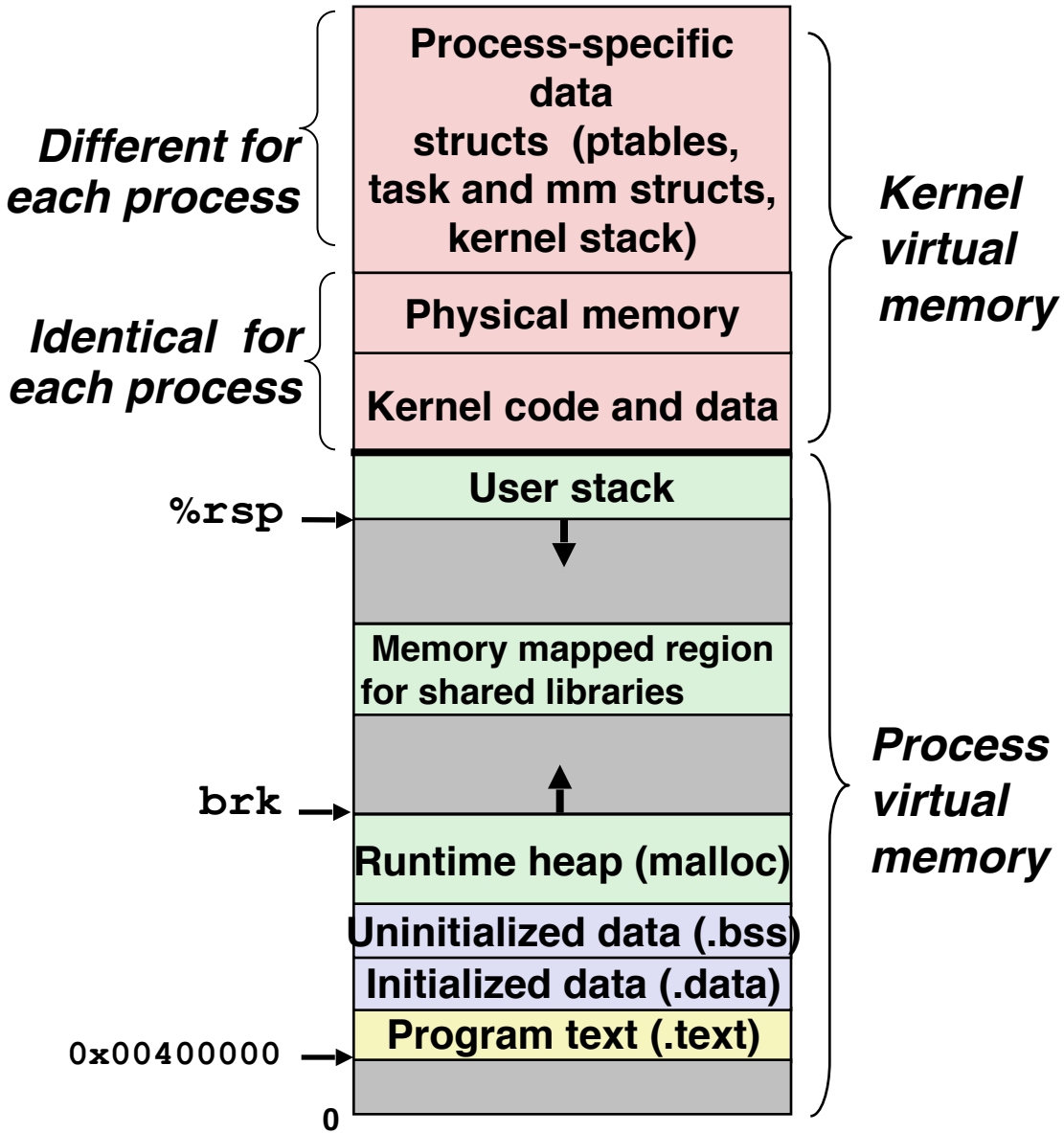
```c
/* mmapcopy driver */
int main(int argc, char **argv)
{
    struct stat stat;
    int fd;

    /* Check for required cmd line arg */
    if (argc != 2) {
        printf("usage: %s <filename>\n",
                argv[0]);
        exit(0);
    }

    /* Copy input file to stdout */
    fd = Open(argv[1], O_RDONLY, 0);
    Fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
    exit(0);
}
```

mmapcopy.c

Today

- Memory mapping
- Dynamic memory allocation

Virtual Address Space of a Linux Process

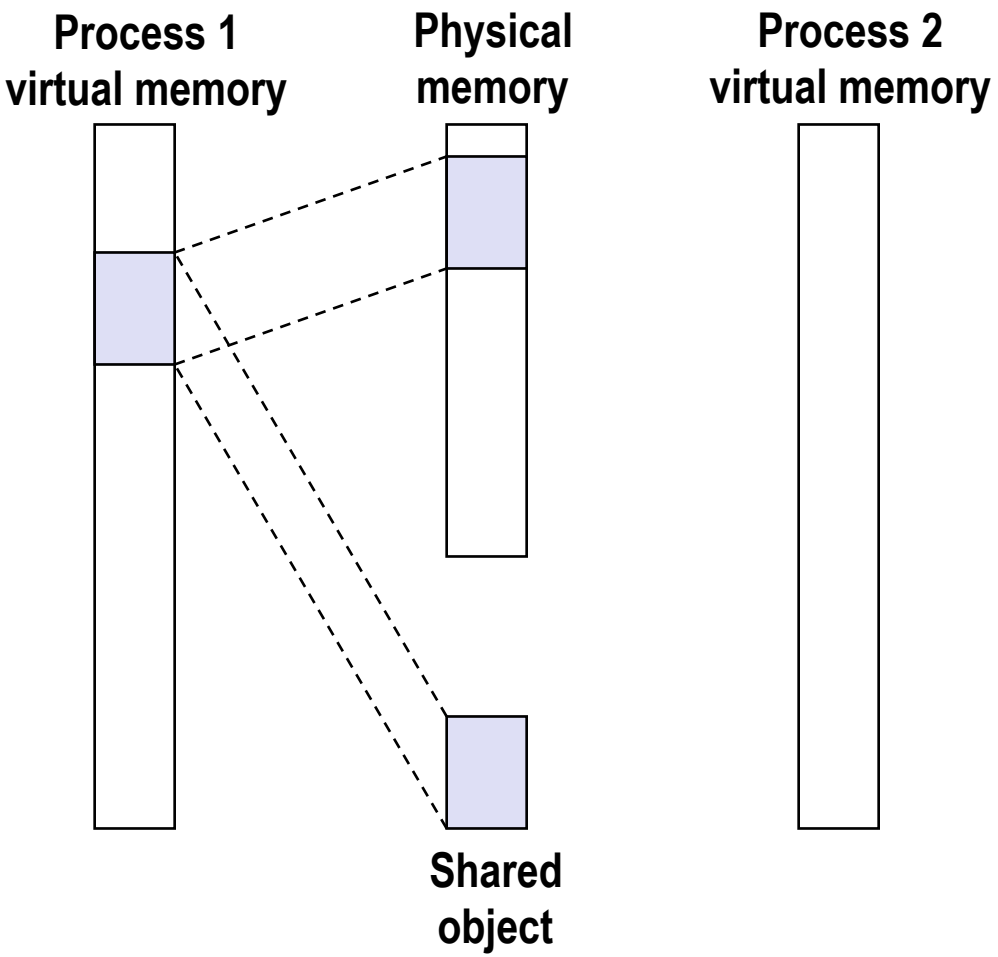


Memory Mapping For Sharing

- Multiple processes often share data
  - Different processes that run the same code (e.g., shell)
  - Different processes linked to the same standard libraries
  - Different processes share the same file
- It is wasteful to create exact copies of the share object
- Memory mapping allow us to easily share objects
  - Different VM pages point to the same physical page/object

Sharing Revisited: Shared Objects

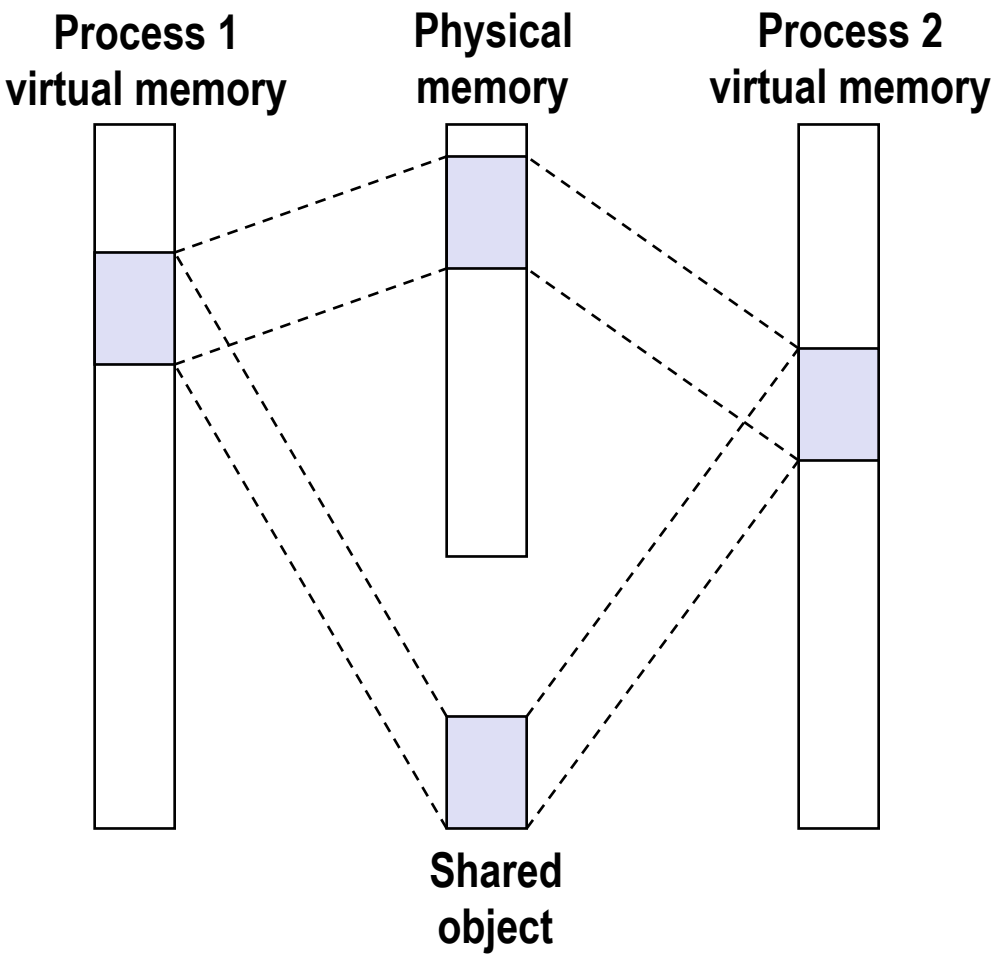
- Process 1 maps the shared object.



- The kernel remembers that the object (backed by a unique file) is mapped by Proc. 1 to some physical pages.

Sharing Revisited: Shared Objects

- Process 2 maps the shared object.



- The kernel remembers that the object (backed by a unique file) is mapped by Proc. 1 to some physical pages.
- Now when Proc. 2 wants to access the same object, the kernel can simply point the PTEs of Proc. 2 to the already-mapped physical pages.

The Problem…

- What if Proc. 1 now wants to modify the shared object, but doesn’t want the modification to be visible to Proc. 2

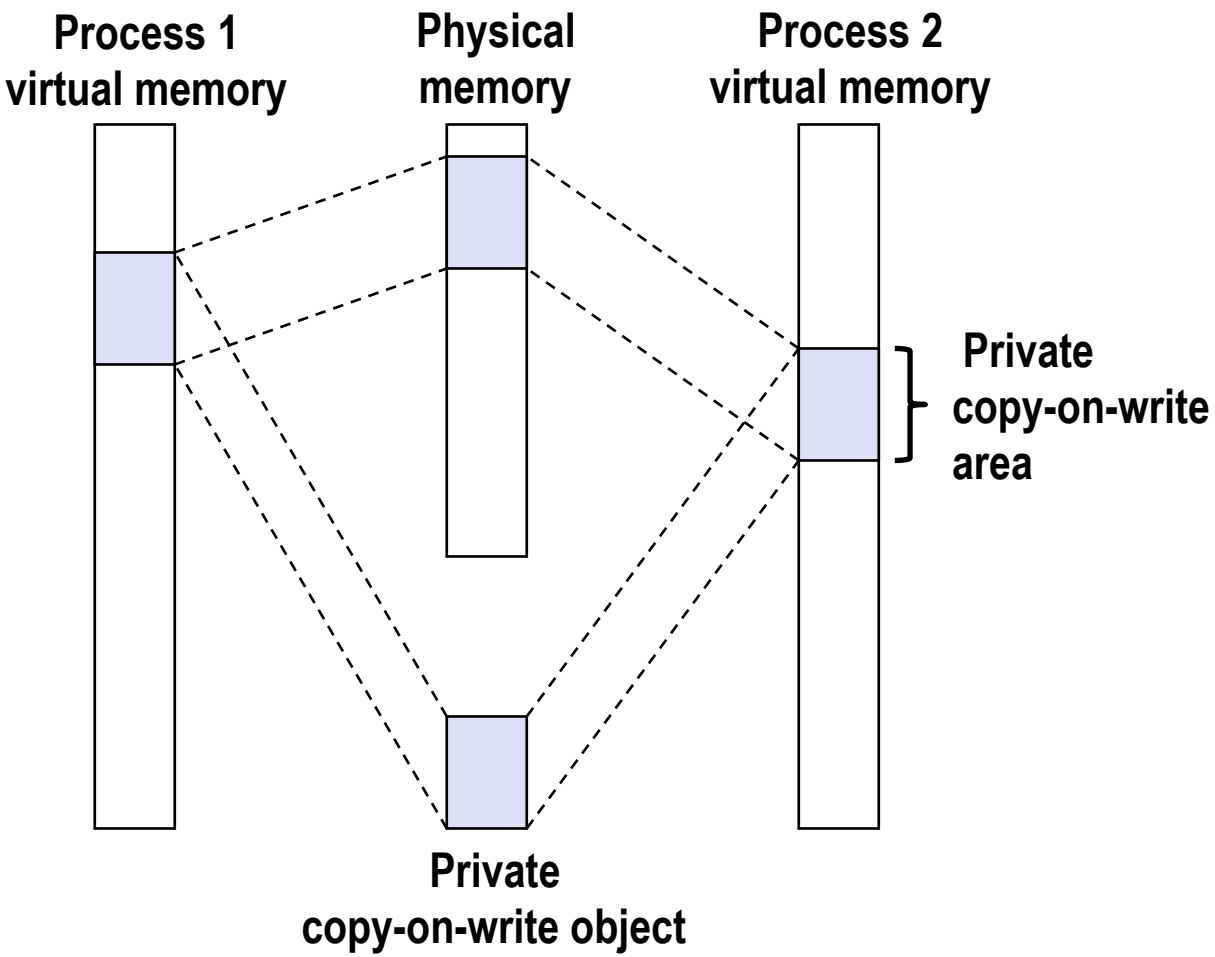
The Problem…

- What if Proc. 1 now wants to modify the shared object, but doesn’t want the modification to be visible to Proc. 2
- Simplest solution: always create duplicate copies of shared objects at the cost of wasting space. Not ideal.

The Problem…

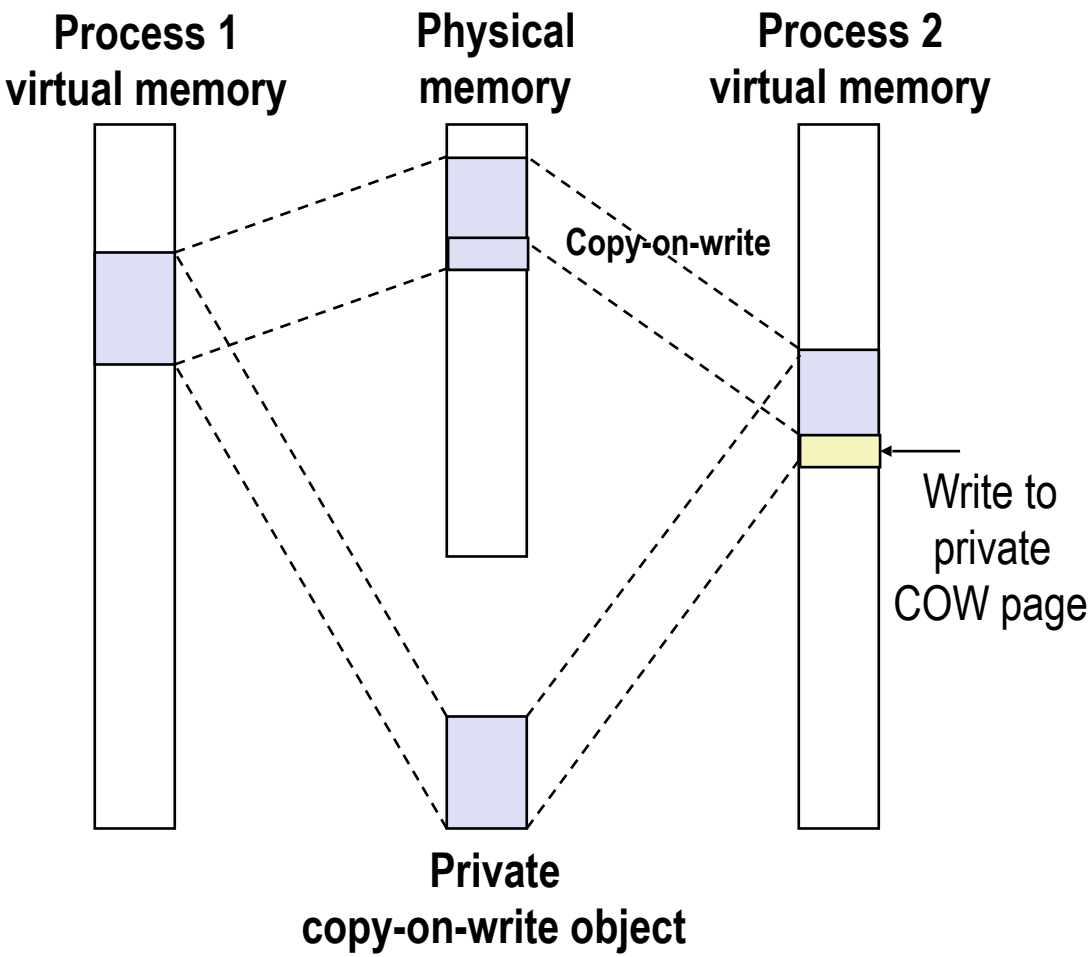
- What if Proc. 1 now wants to modify the shared object, but doesn't want the modification to be visible to Proc. 2
- Simplest solution: always create duplicate copies of shared objects at the cost of wasting space. Not ideal.
- Idea: Copy-on-write (COW)
  - First pretend that both processes will share the objects without modifying them. If modification happens, create separate copies.

Private Copy-on-write (COW) Objects



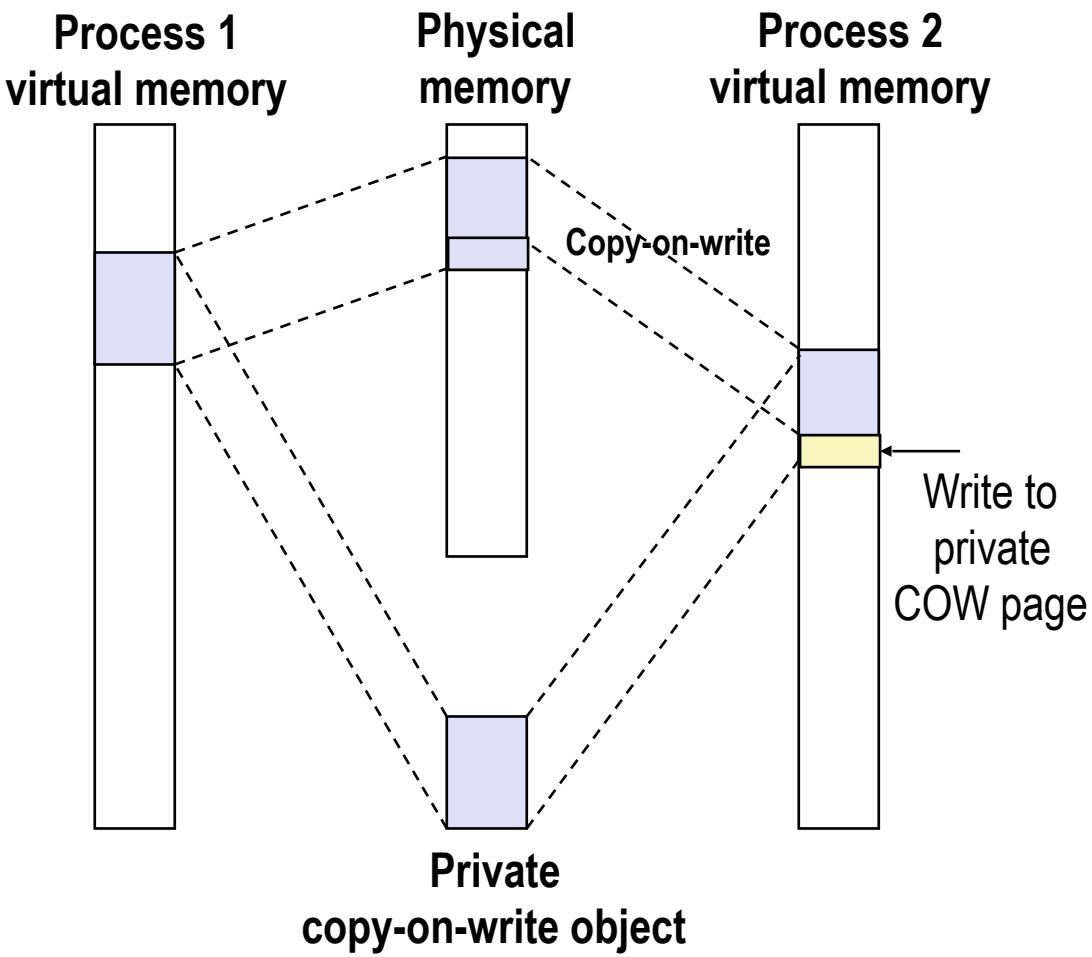
- Two processes mapping a *private copy-on-write (COW)* object.
- Area flagged as private copy-on-write (COW)
- PTEs in private areas are flagged as read-only

Private Copy-on-write (COW) Objects



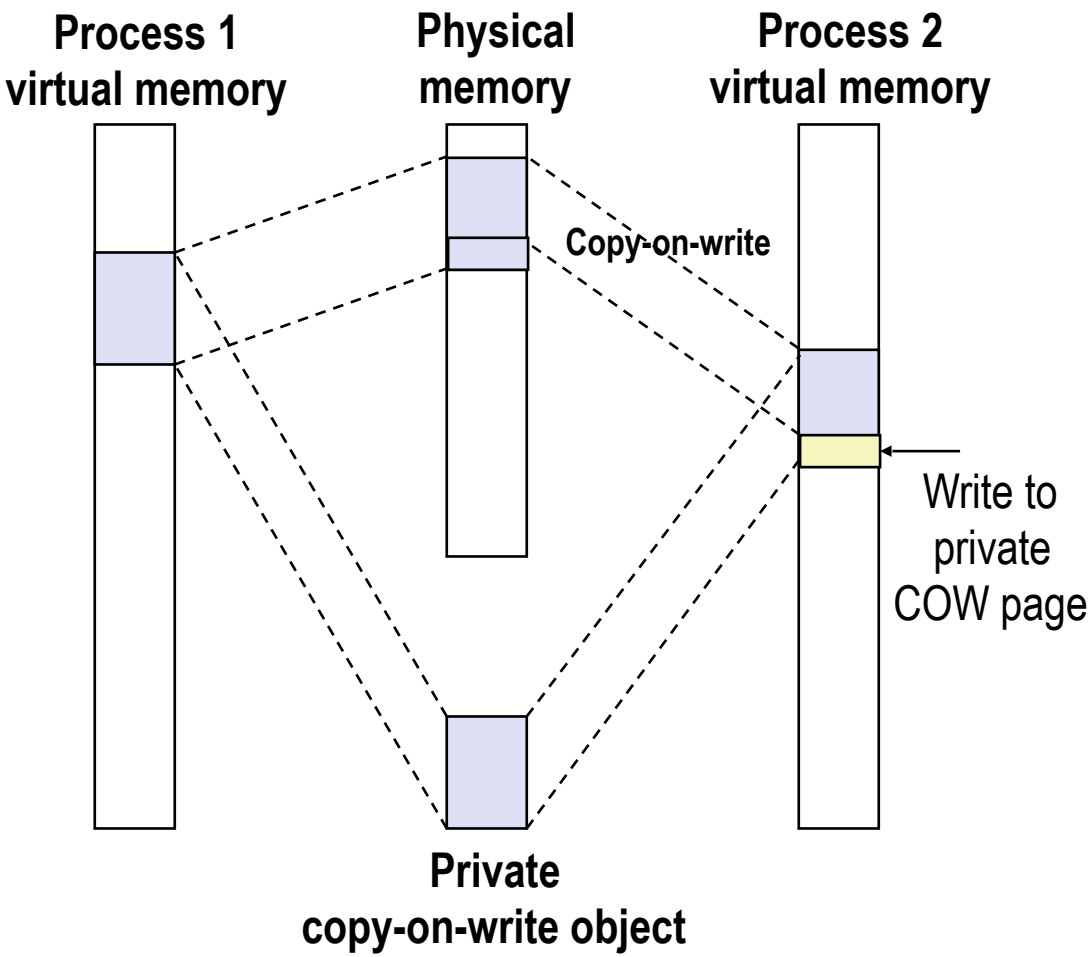
- Instruction writing to private page triggers page (protection) fault.

Private Copy-on-write (COW) Objects



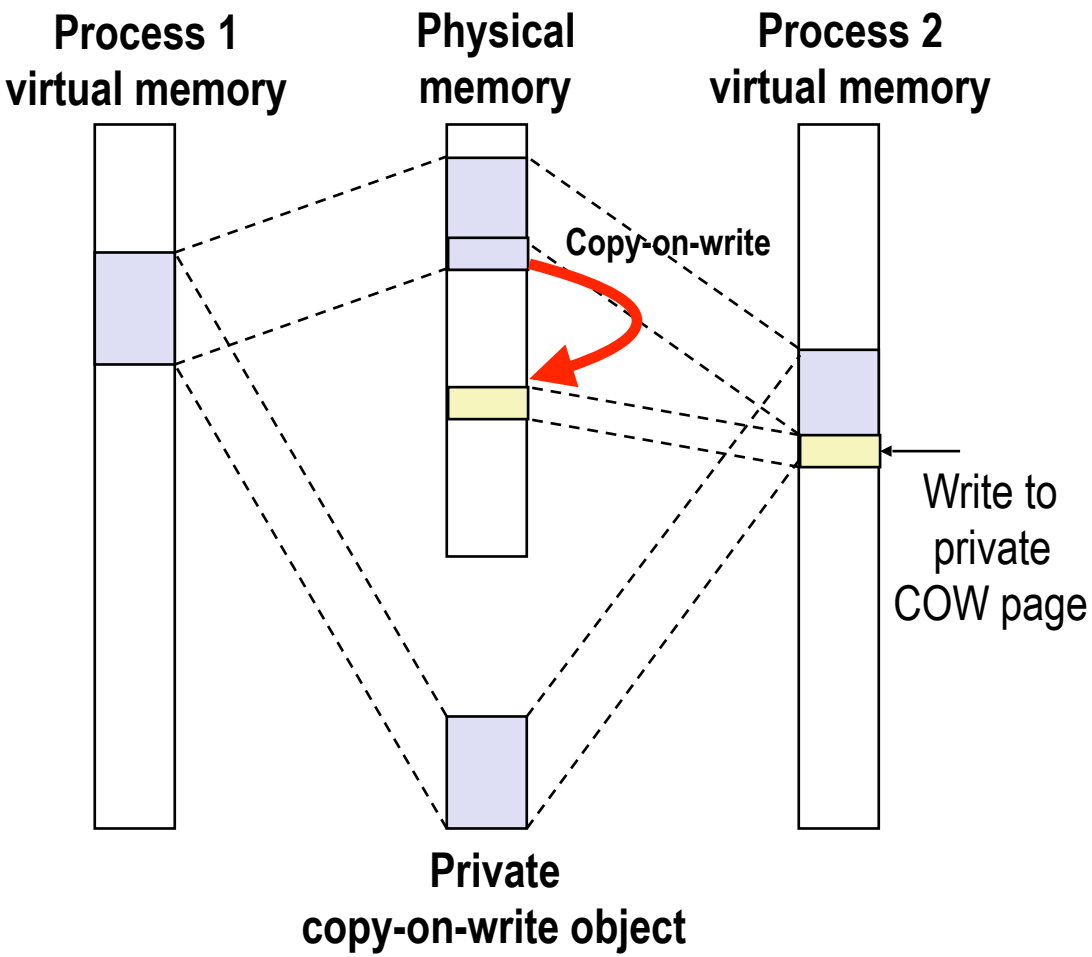
- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it's a COW object

Private Copy-on-write (COW) Objects



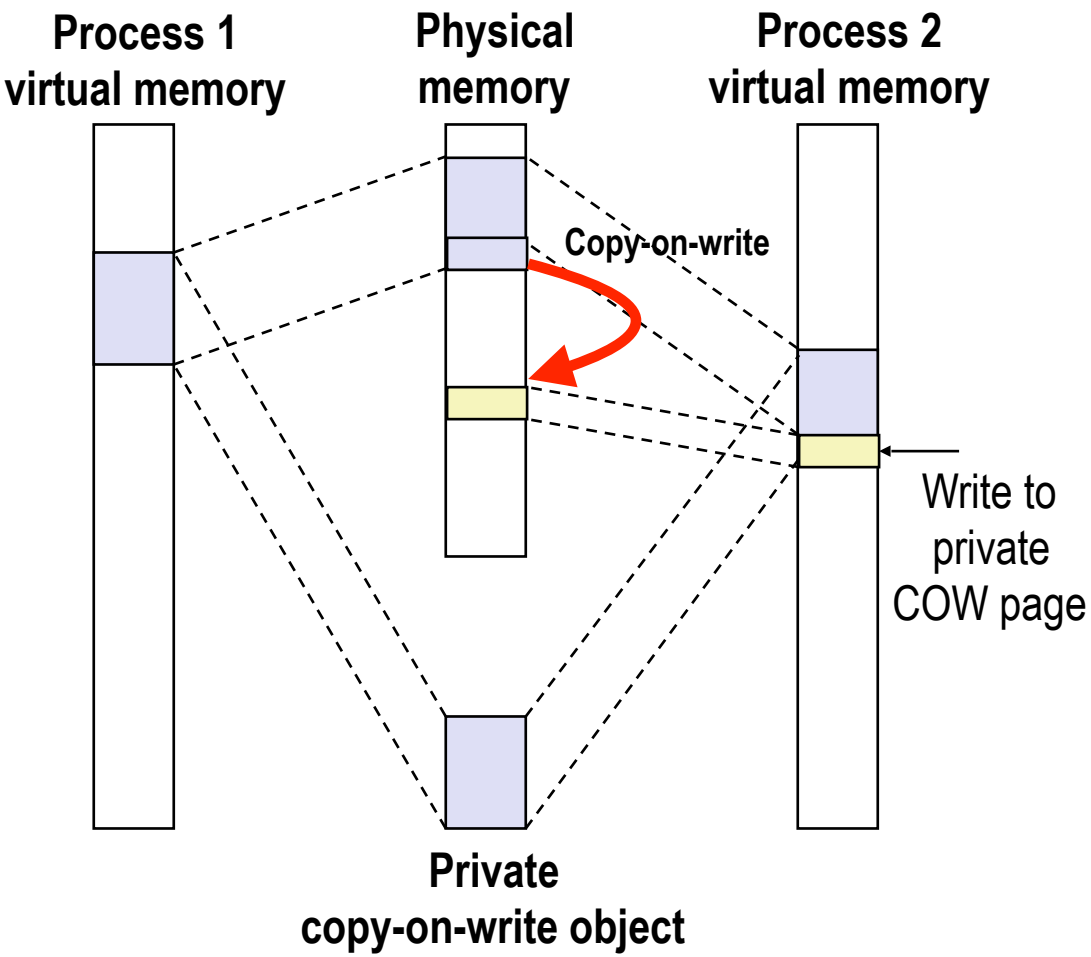
- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it's a COW object
- Handler then creates new R/W page.

Private Copy-on-write (COW) Objects



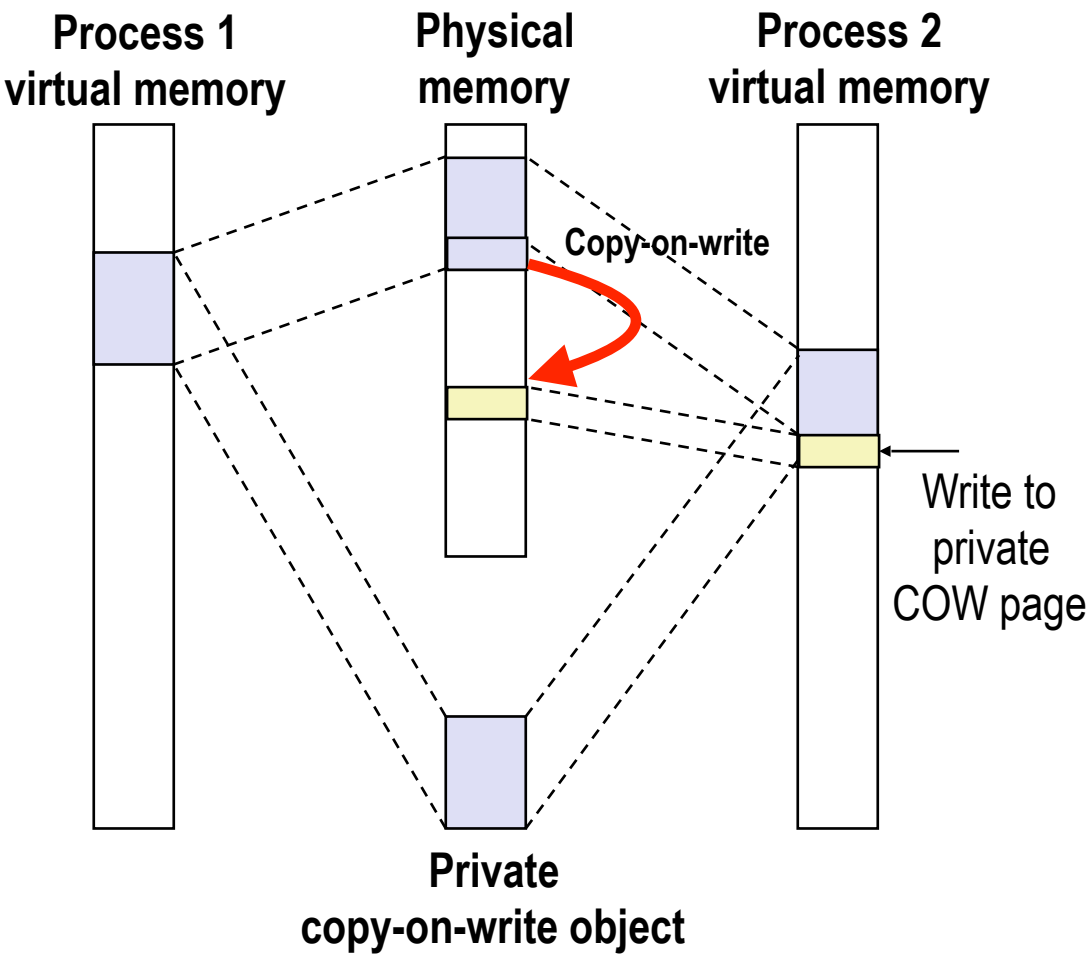
- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it's a COW object
- Handler then creates new R/W page.

Private Copy-on-write (COW) Objects



- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it's a COW object
- Handler then creates new R/W page.
- Instruction restarts upon handler return.

Private Copy-on-write (COW) Objects



- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it's a COW object
- Handler then creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

User-Level Memory Mapping

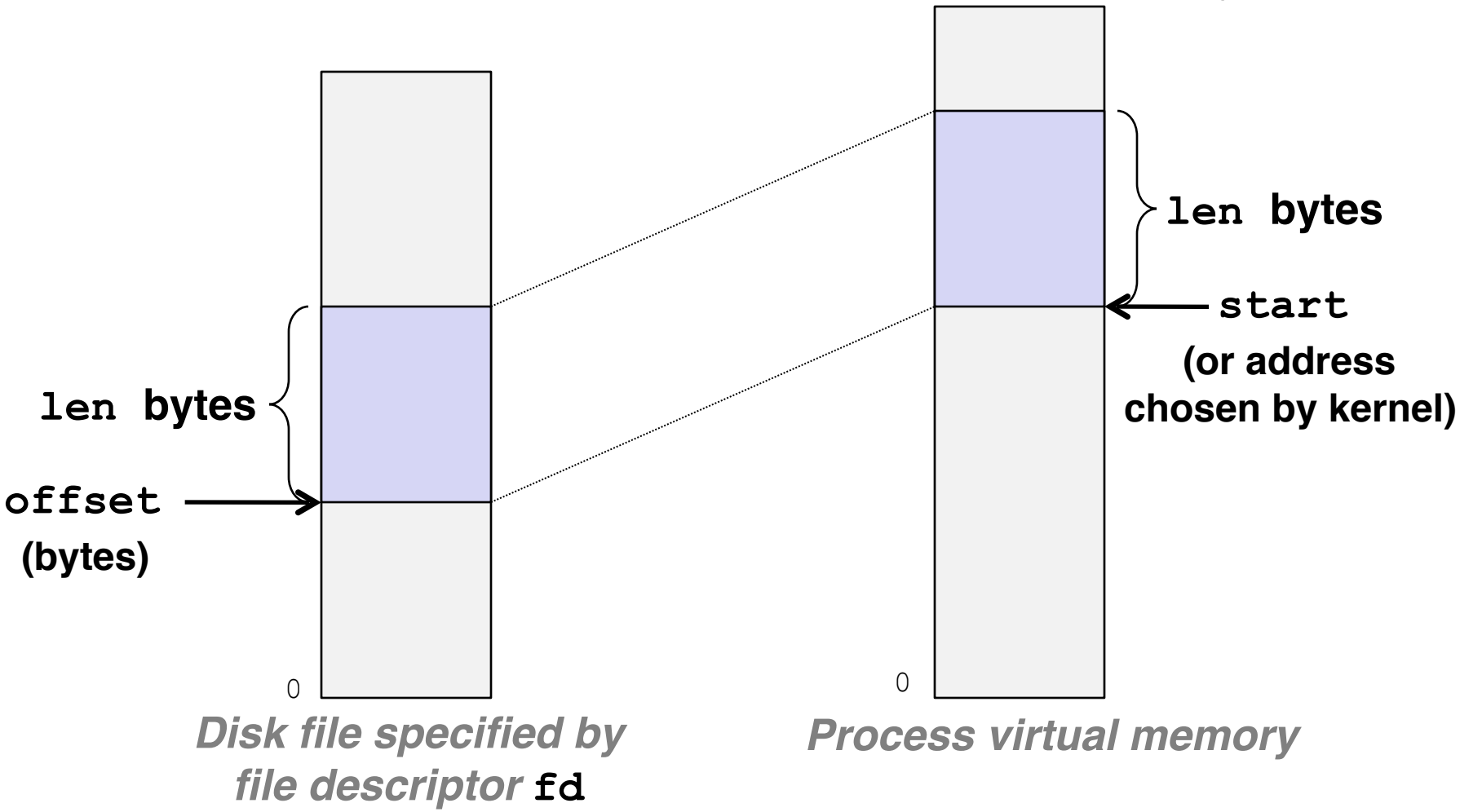
```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- Map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start`
  - `start`: may be NULL for “pick an address”
  - `prot`: PROT_READ, PROT_WRITE, ...
  - `flags`: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...

- Return a pointer to start of mapped area (may not be `start`)

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```



Example: Using `mmap` to Copy Files

- Copying a file to stdout without transferring data to user space
  - i.e., no file data is copied to user stack

```c
#include "csapp.h"

void mmapcopy(int fd, int size)
{

    /* Ptr to memory mapped area */
    char *bufp;

    bufp = mmap(NULL, size,
                PROT_READ,
                MAP_PRIVATE,
                fd, 0);
    Write(1, bufp, size);
    return;
}
```

mmapcopy.c

```c
/* mmapcopy driver */
int main(int argc, char **argv)
{
    struct stat stat;
    int fd;

    /* Check for required cmd line arg */
    if (argc != 2) {
        printf("usage: %s <filename>\n",
                argv[0]);
        exit(0);
    }

    /* Copy input file to stdout */
    fd = Open(argv[1], O_RDONLY, 0);
    Fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
    exit(0);
}
```

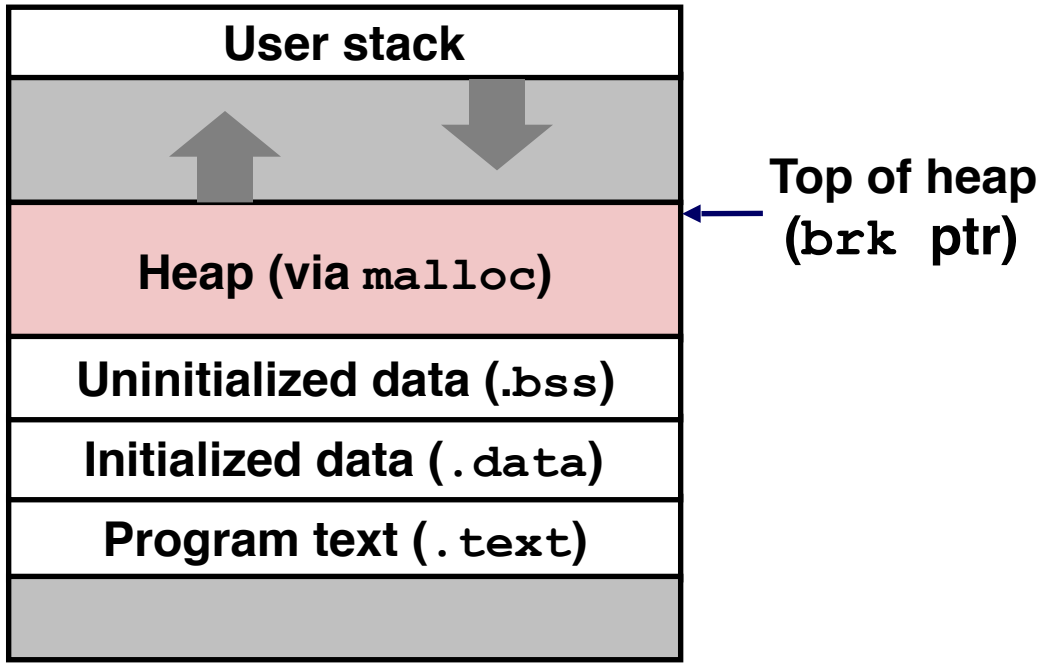
mmapcopy.c

Today

- Memory mapping
- Dynamic memory allocation
  - Basic concepts
  - Implicit free lists

Dynamic Memory Allocation

- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire VM at run time.
- Dynamic memory allocators manage an area of process virtual memory known as the *heap*.



The `malloc/free` Functions

```
#include <stdlib.h>

void *malloc(size_t size)
```

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

The `malloc/free` Functions

```
#include <stdlib.h>
```

```
void *malloc(size_t size)
```

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

```
void free(void *p)
```

- Returns the block pointed at by **`p`** to pool of available memory
- `p` must come from a previous call to `malloc` or `realloc`

The `malloc/free` Functions

```
#include <stdlib.h>
```

```
void *malloc(size_t size)
```

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

```
void free(void *p)
```

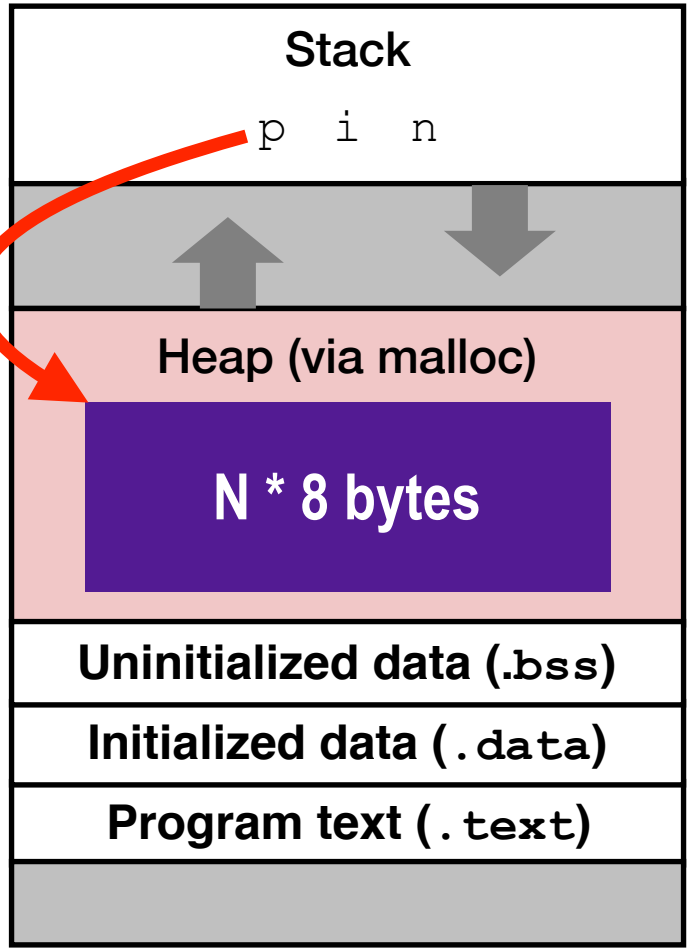
- Returns the block pointed at by **`p`** to pool of available memory
- `p` must come from a previous call to `malloc` or `realloc`

Other functions

- `calloc`: Version of `malloc` that initializes allocated block to zero.
- `realloc`: Changes the size of a previously allocated block.
- `sbrk`: Used internally by allocators to grow or shrink the heap

`malloc` Example

```c
#include <stdio.h>
#include <stdlib.h>

void foo(int n) {
    int i, *p;

    /* Allocate a block of n ints */
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i=0; i<n; i++)
        p[i] = i;


    /* Return allocated block to the heap */
    free(p);
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

Why Do We Need Dynamic Allocation?

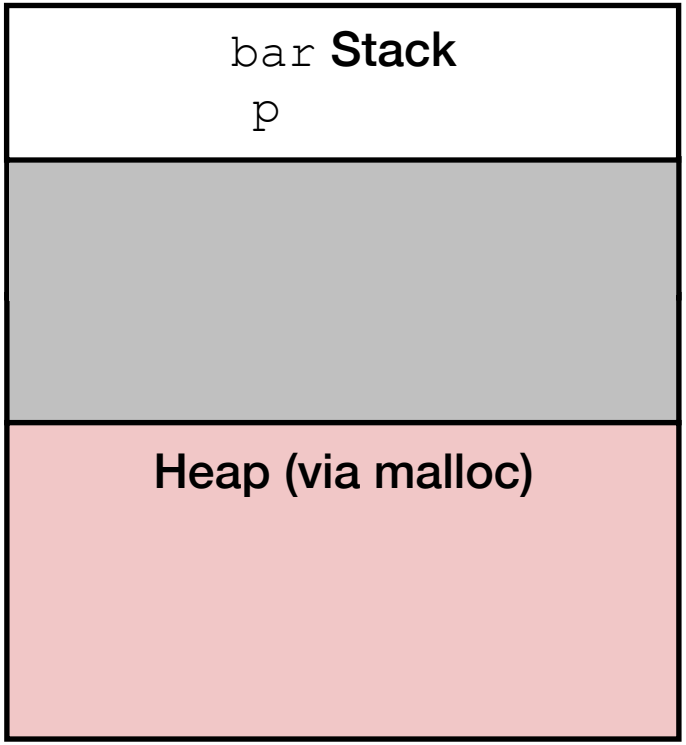
- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

```
int* foo(int n) {
  int i, *p;

  p = (int *) malloc(n * sizeof(int));
  if (p == NULL) exit(0);

  for (i=0; i<n; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo(5);

  printf("%d\n", p[0]);
}
```

Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

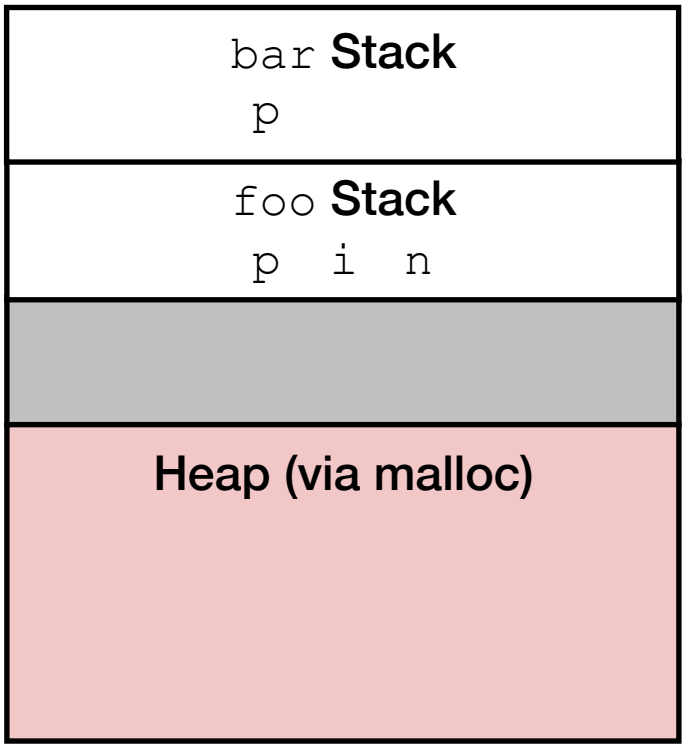
```
int* foo(int n) {
  int i, *p;

  p = (int *) malloc(n * sizeof(int));
  if (p == NULL) exit(0);

  for (i=0; i<n; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo(5);

  printf("%d\n", p[0]);
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

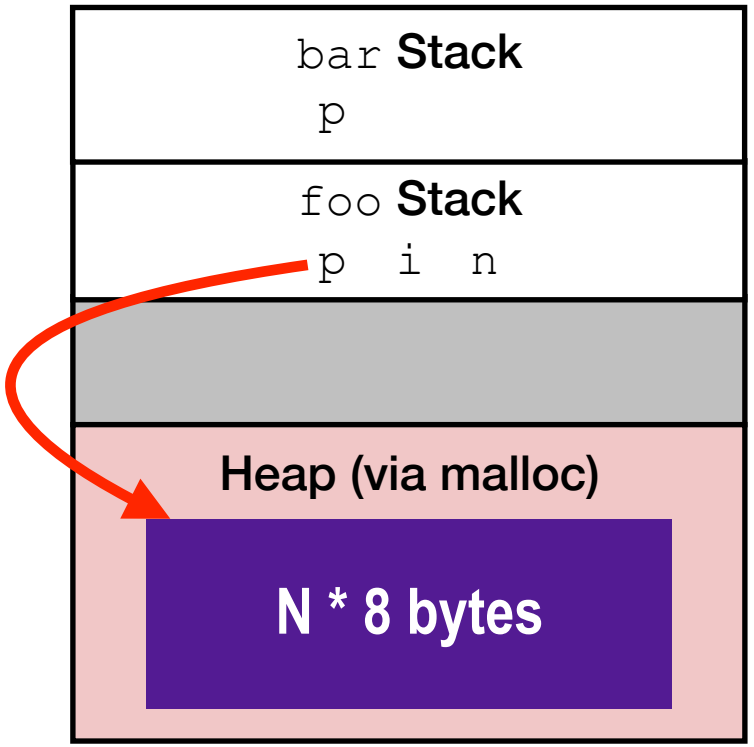
```
int* foo(int n) {
  int i, *p;

  p = (int *) malloc(n * sizeof(int));
  if (p == NULL) exit(0);

  for (i=0; i<n; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo(5);

  printf("%d\n", p[0]);
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

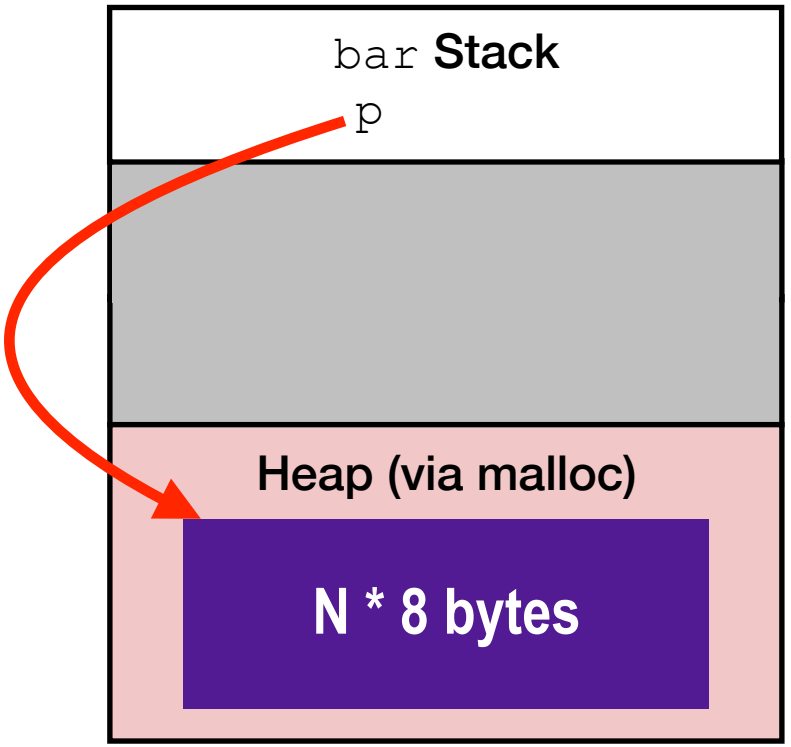
```
int* foo(int n) {
  int i, *p;

  p = (int *) malloc(n * sizeof(int));
  if (p == NULL) exit(0);

  for (i=0; i<n; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo(5);

  printf("%d\n", p[0]);
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

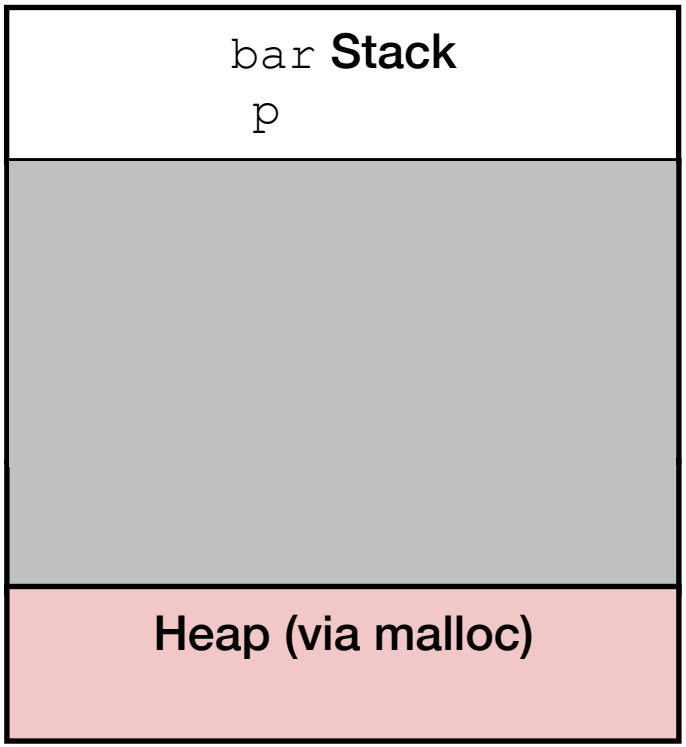
```
int* foo(int n) {
  int i, *p;

  p = (int *) malloc(n * sizeof(int));
  if (p == NULL) exit(0);

  for (i=0; i<n; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo(5);

  printf("%d\n", p[0]);
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

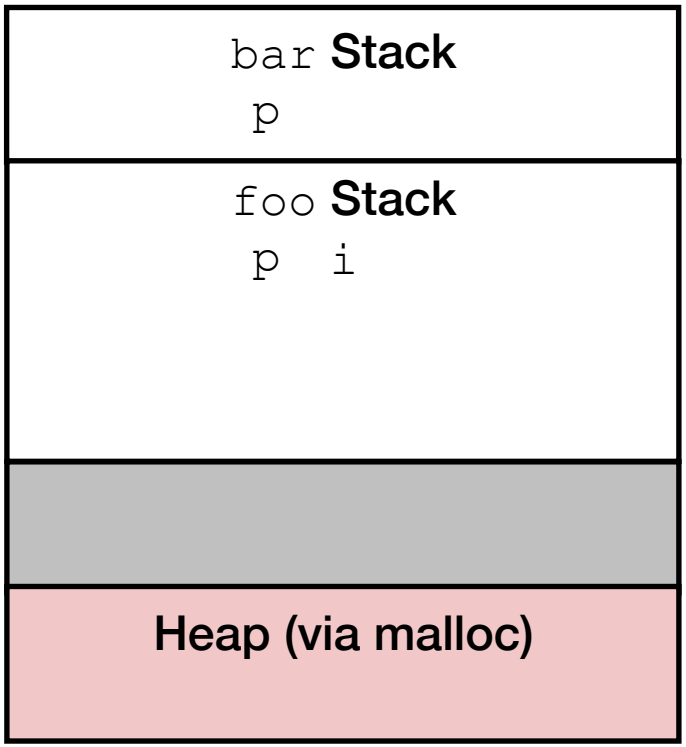
```
int* foo() {
  int i;
  int p[5];

  for (i=0; i<5; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo();

  printf("%d\n", p[0]);
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

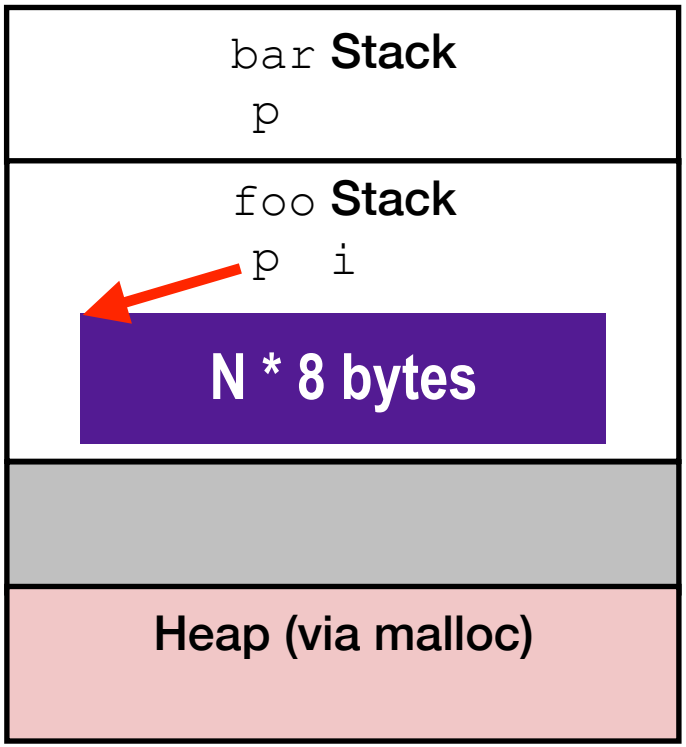
```
int* foo() {
  int i;
  int p[5];

  for (i=0; i<5; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo();

  printf("%d\n", p[0]);
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

```
int* foo() {
  int i;
  int p[5];

  for (i=0; i<5; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo();

  printf("%d\n", p[0]);
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

```
int* foo() {
  int i;
  int p[5];

  for (i=0; i<5; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo();

  printf("%d\n", p[0]);
}
```