Virtual Memory

Arkaprava Basu

Dept. of Computer Science and Automation
Indian Institute of Science
www.csa.iisc.ac.in/~arkapravab/

Acknowledgements

- Some of the slides in the deck were provided by Profs. Luis Ceze (Washington), Nima Horanmand (Stony Brook), Mark Hill, David Wood, Karu Sankaralingam (Wisconsin), Abhishek Bhattacharjee(Yale).

- Development of this course is partially supported by funding from Western Digital corporations.

Memory is virtual !

- Application software sees **virtual** address
  - int * p = malloc(64);

    $\leftarrow$ Virtual address

  - Load, store instructions carries virtual addresses

Memory is virtual !

- Application software sees **virtual** address
  - int * p = malloc(64);

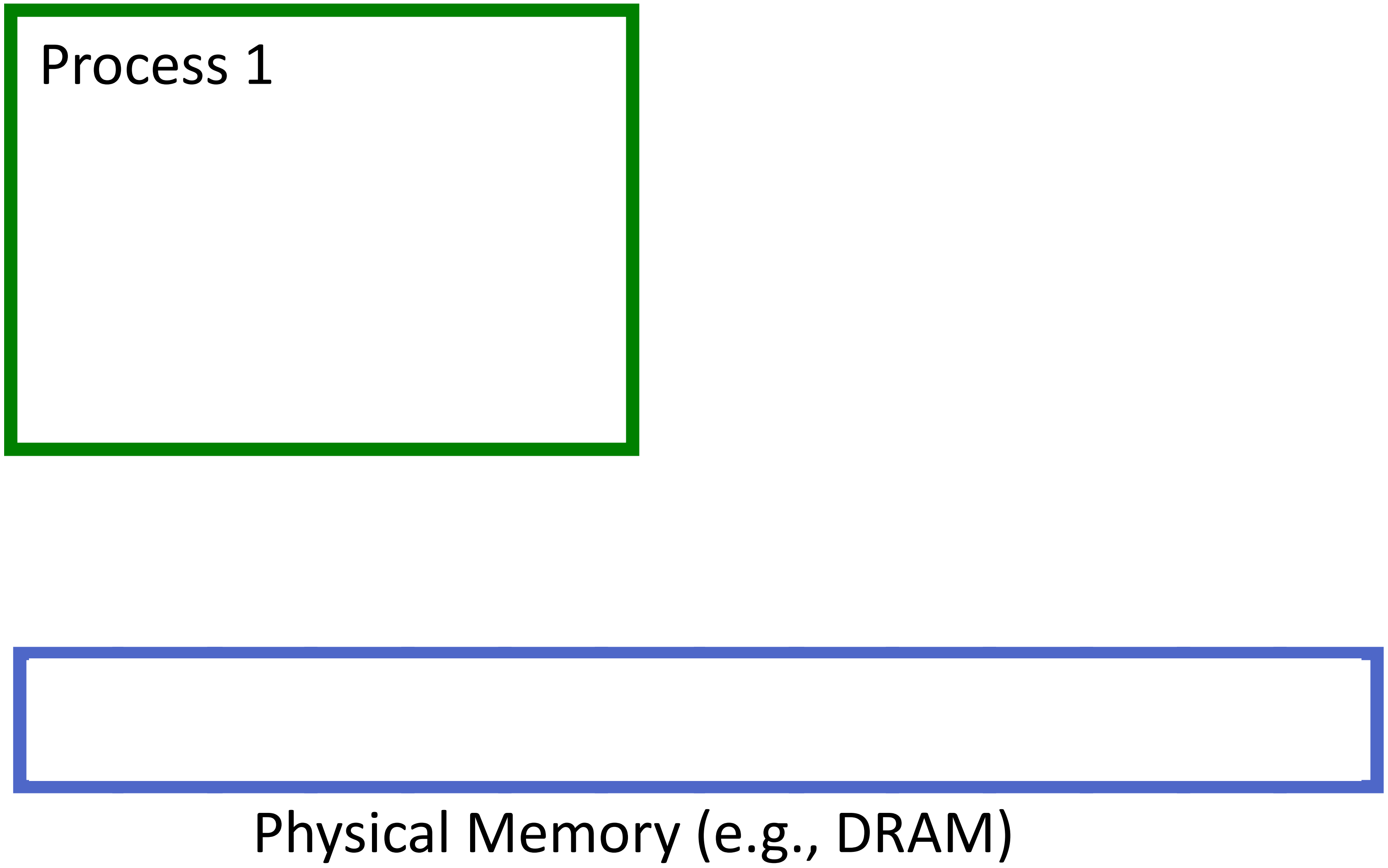
    Virtual address
  - Load, store instructions carries virtual addresses
- Hardware uses (real) **physical** address
  - e.g., to find data, lookup caches etc.
- When an application executes (a.k.a process) virtual addresses are translated to physical addresses, at runtime

Bird’s view of virtual memory

Physical Memory (e.g., DRAM)

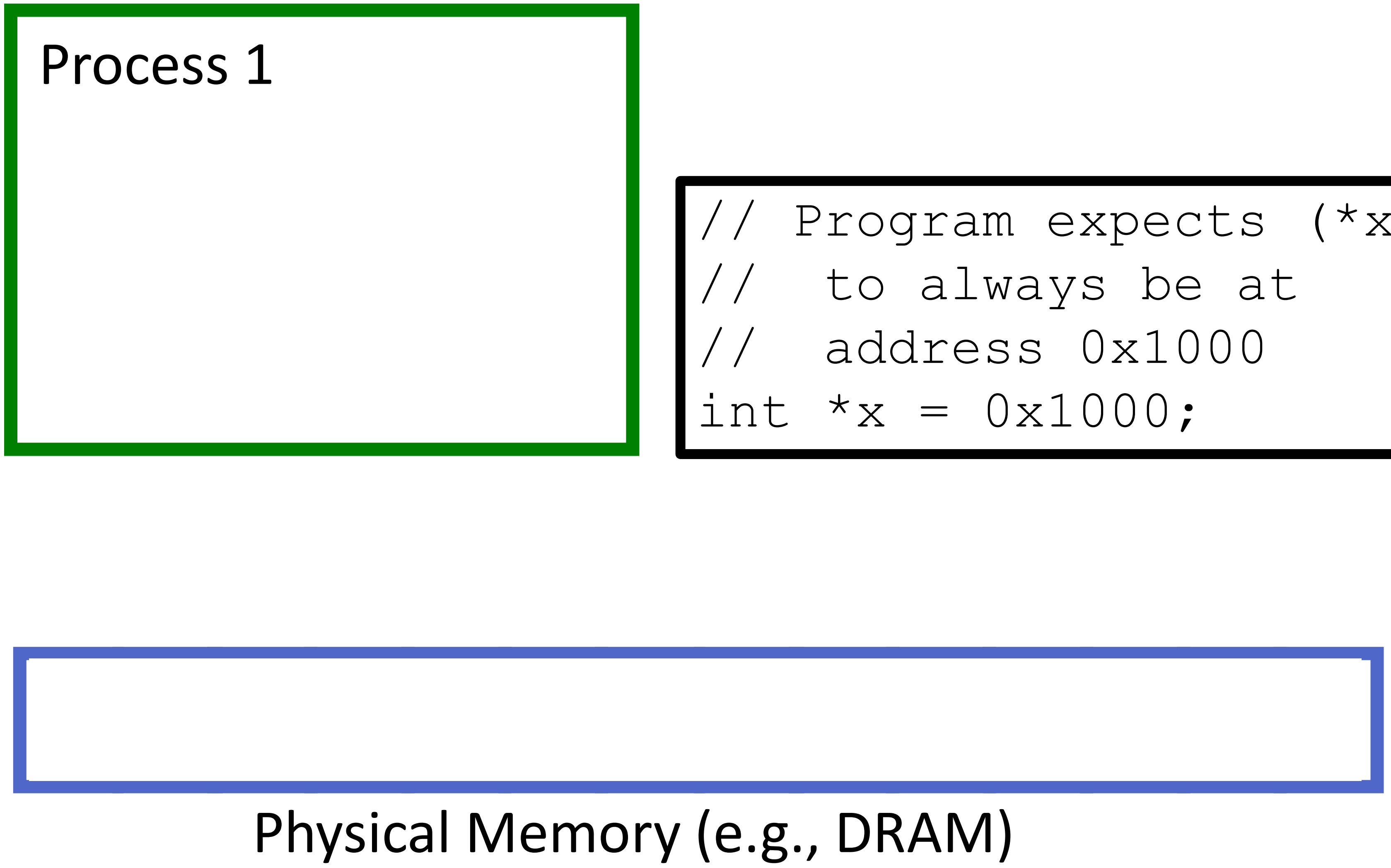
Picture credit: Nima Honarmand, Stony Brook university

Bird's view of virtual memory



Picture credit: Nima Honarmand, Stony Brook university

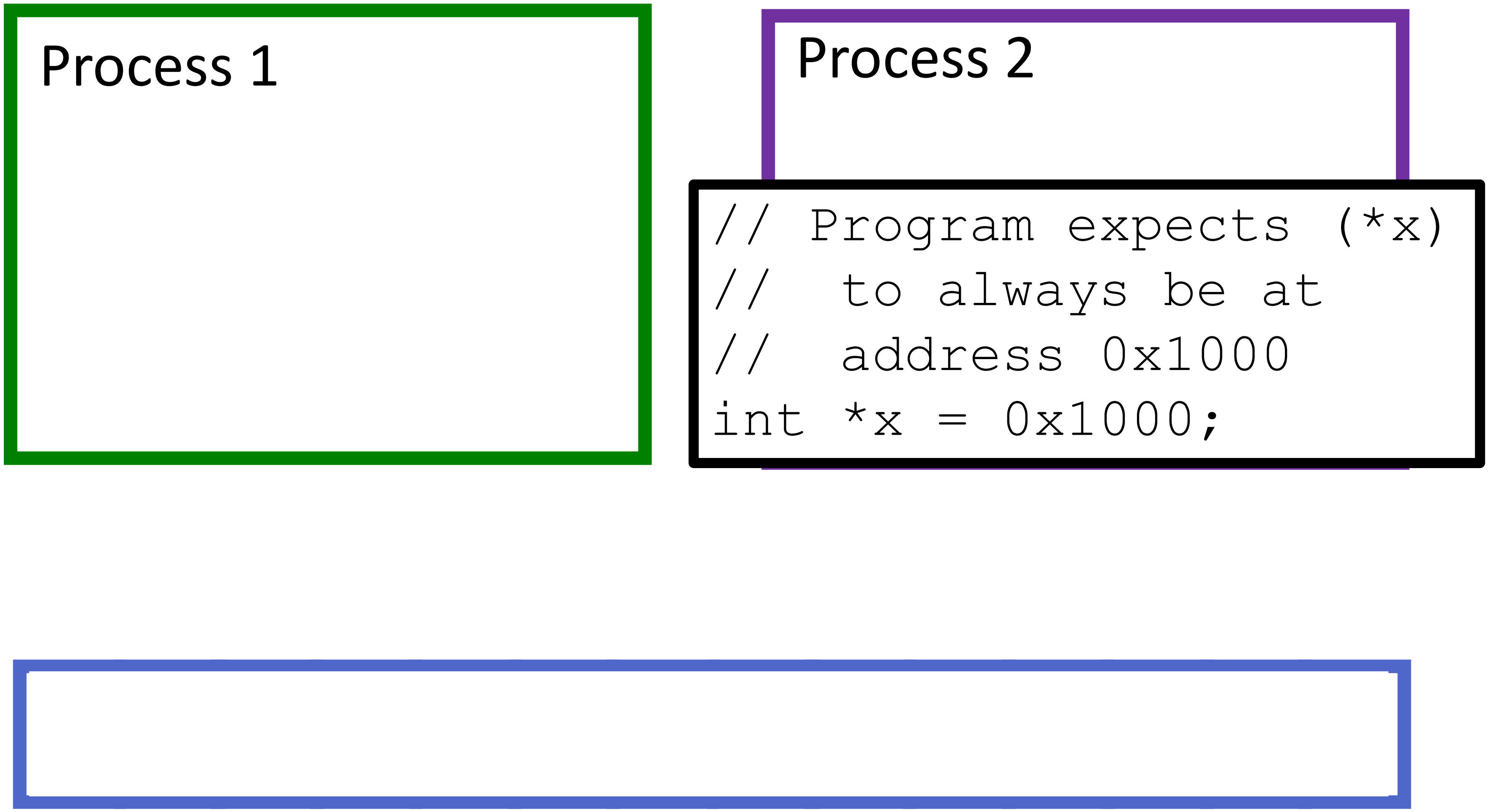
Bird’s view of virtual memory



```
// Program expects (*x)
//  to always be at
//  address 0x1000
int *x = 0x1000;
```

Picture credit: Nima Honarmand, Stony Brook university

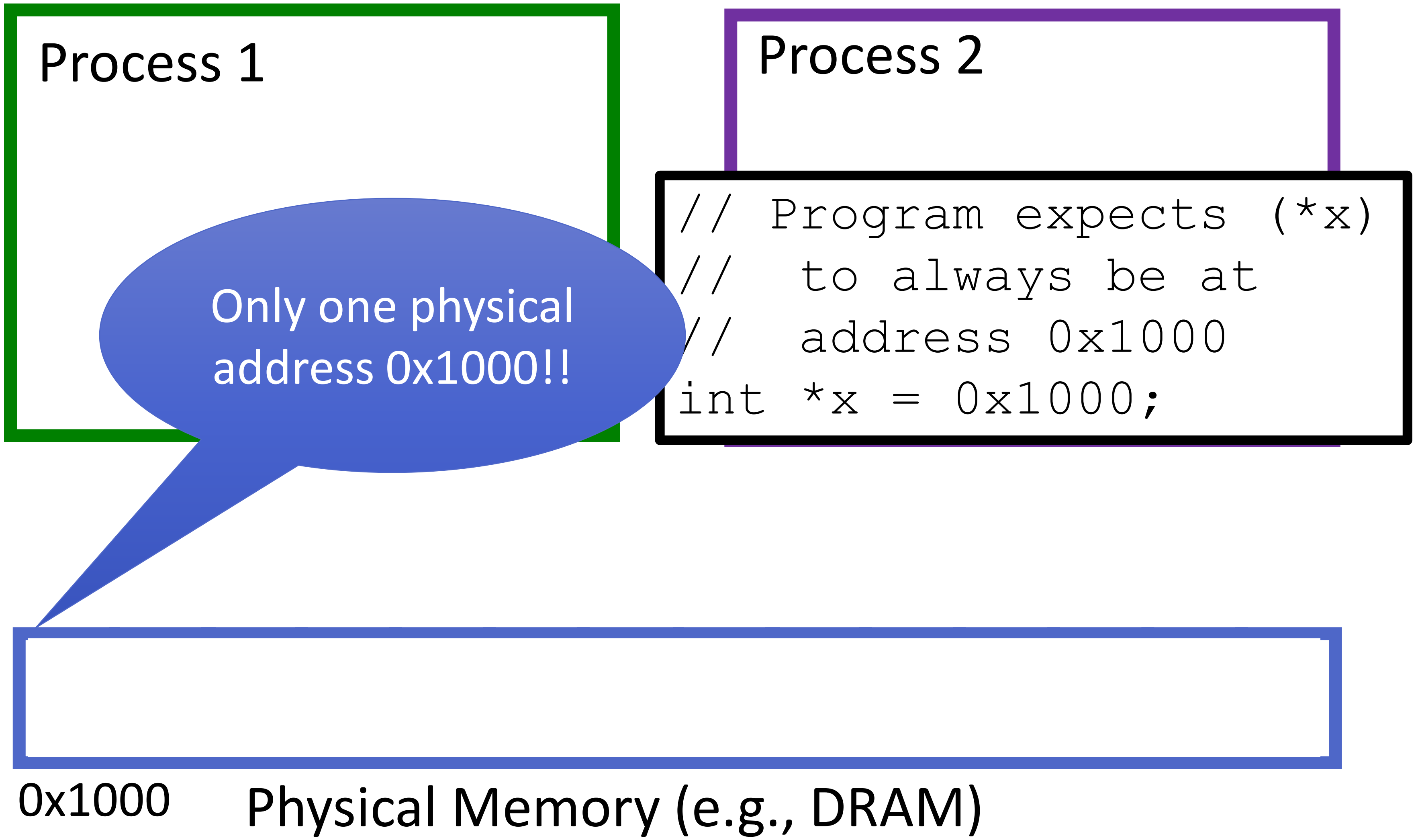
Bird's view of virtual memory



```
// Program expects (*x)
//  to always be at
//  address 0x1000
int *x = 0x1000;
```

Picture credit: Nima Honarmand, Stony Brook university

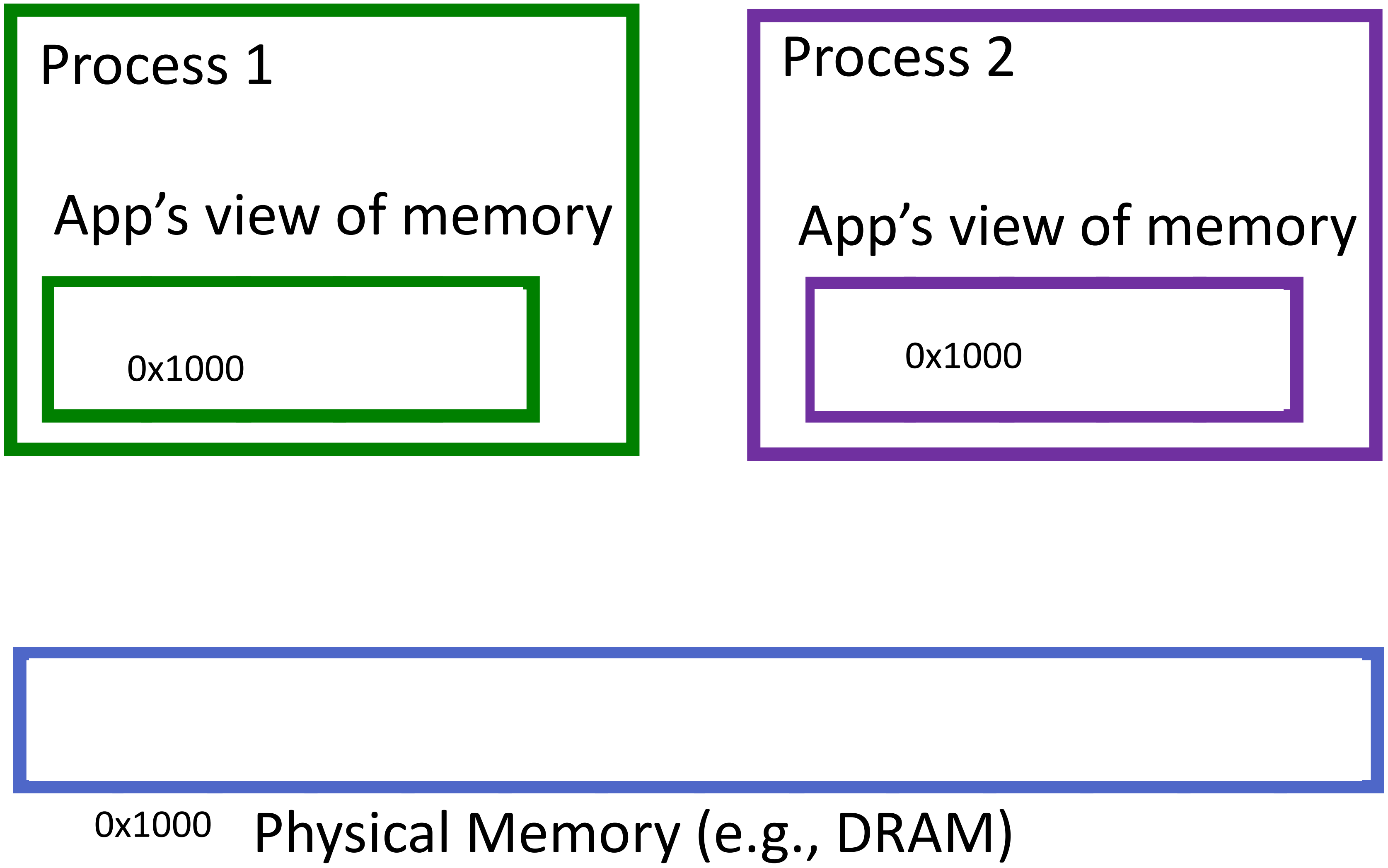
Bird's view of virtual memory


Process 1
Process 2
Only one physical address 0x1000!!
0x1000
Physical Memory (e.g., DRAM)

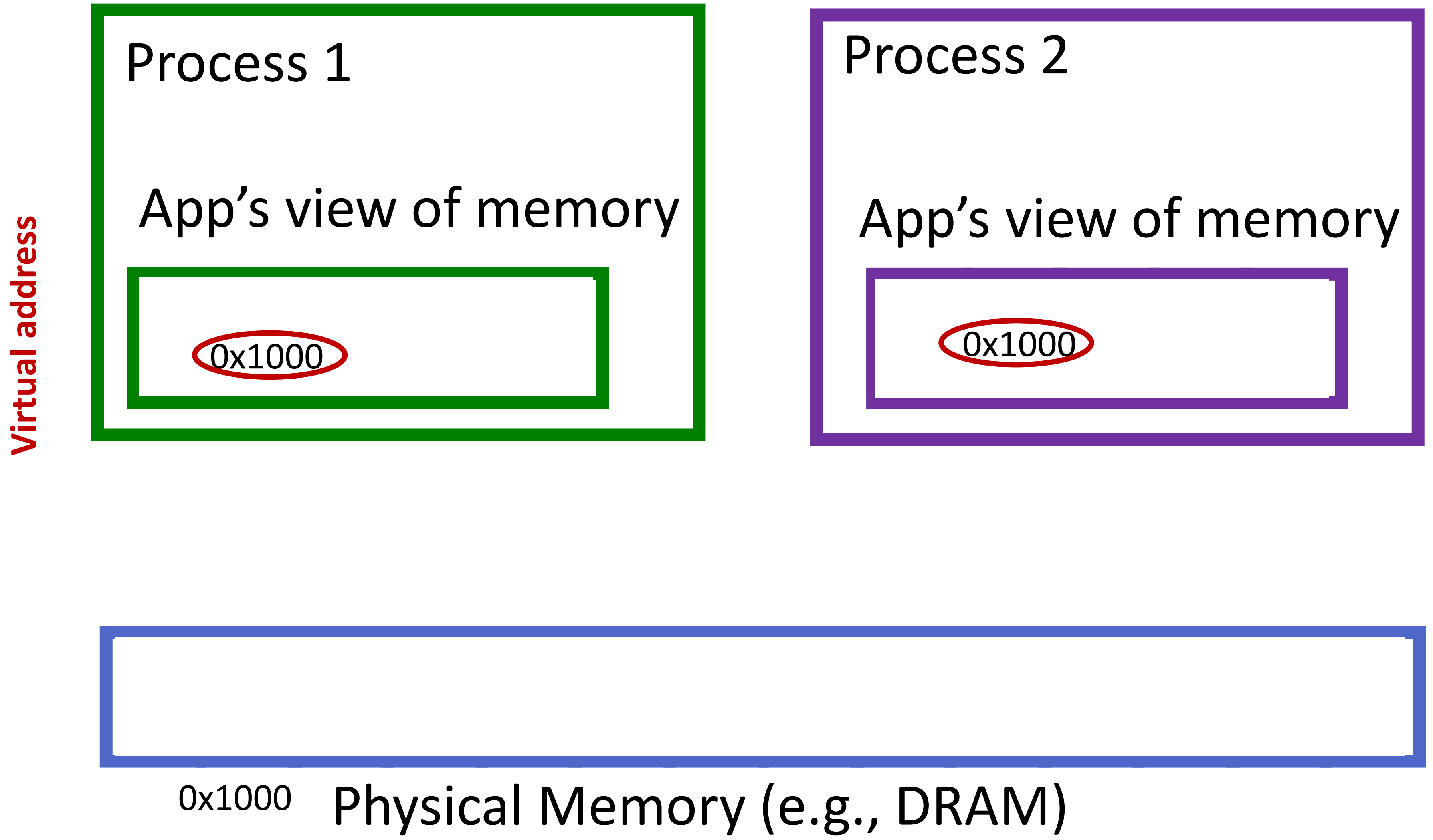
```
// Program expects (*x)
//  to always be at
//  address 0x1000
int *x = 0x1000;
```

Picture credit: Nima Honarmand, Stony Brook university

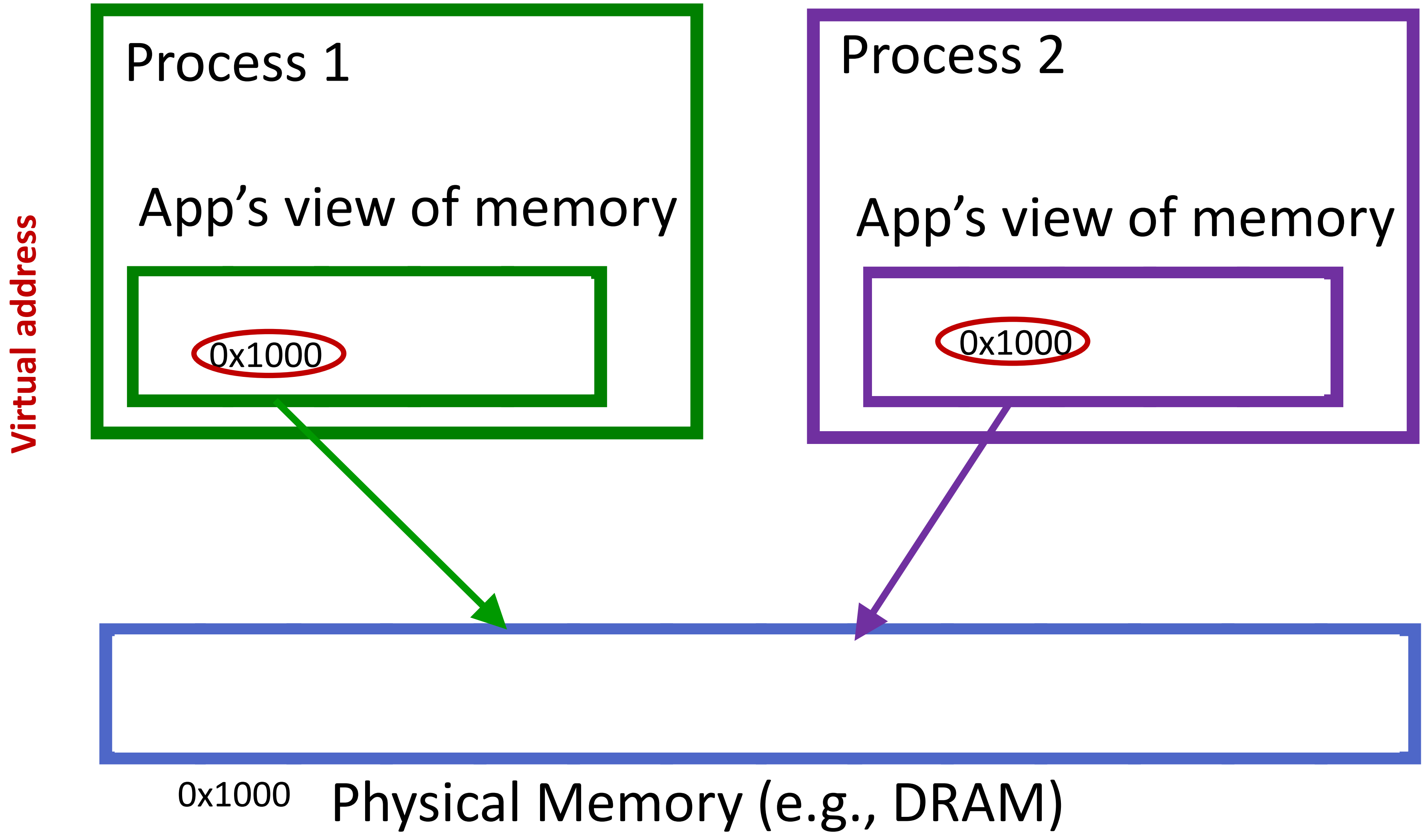
Bird’s view of virtual memory



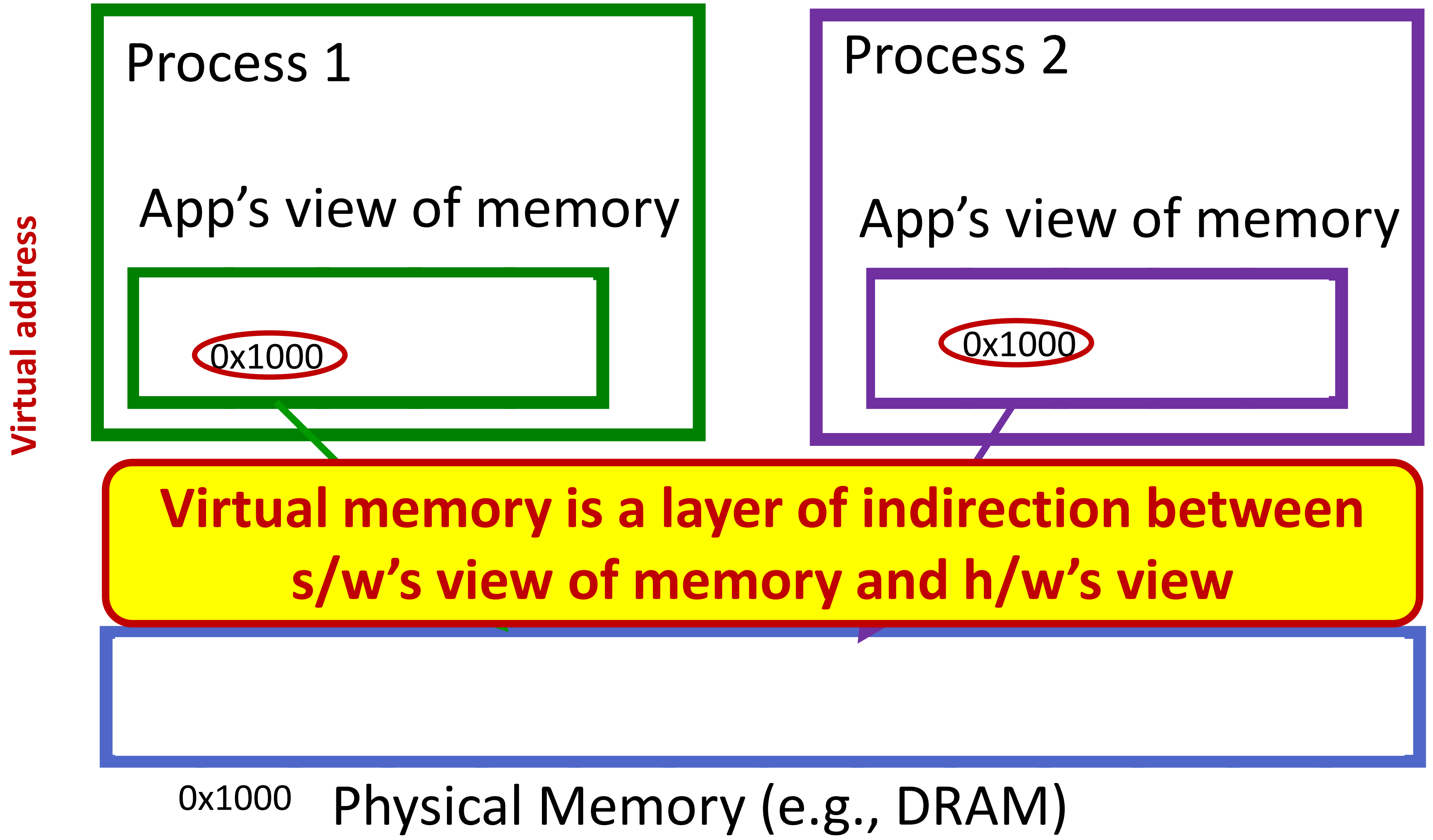
Bird’s view of virtual memory



Bird’s view of virtual memory

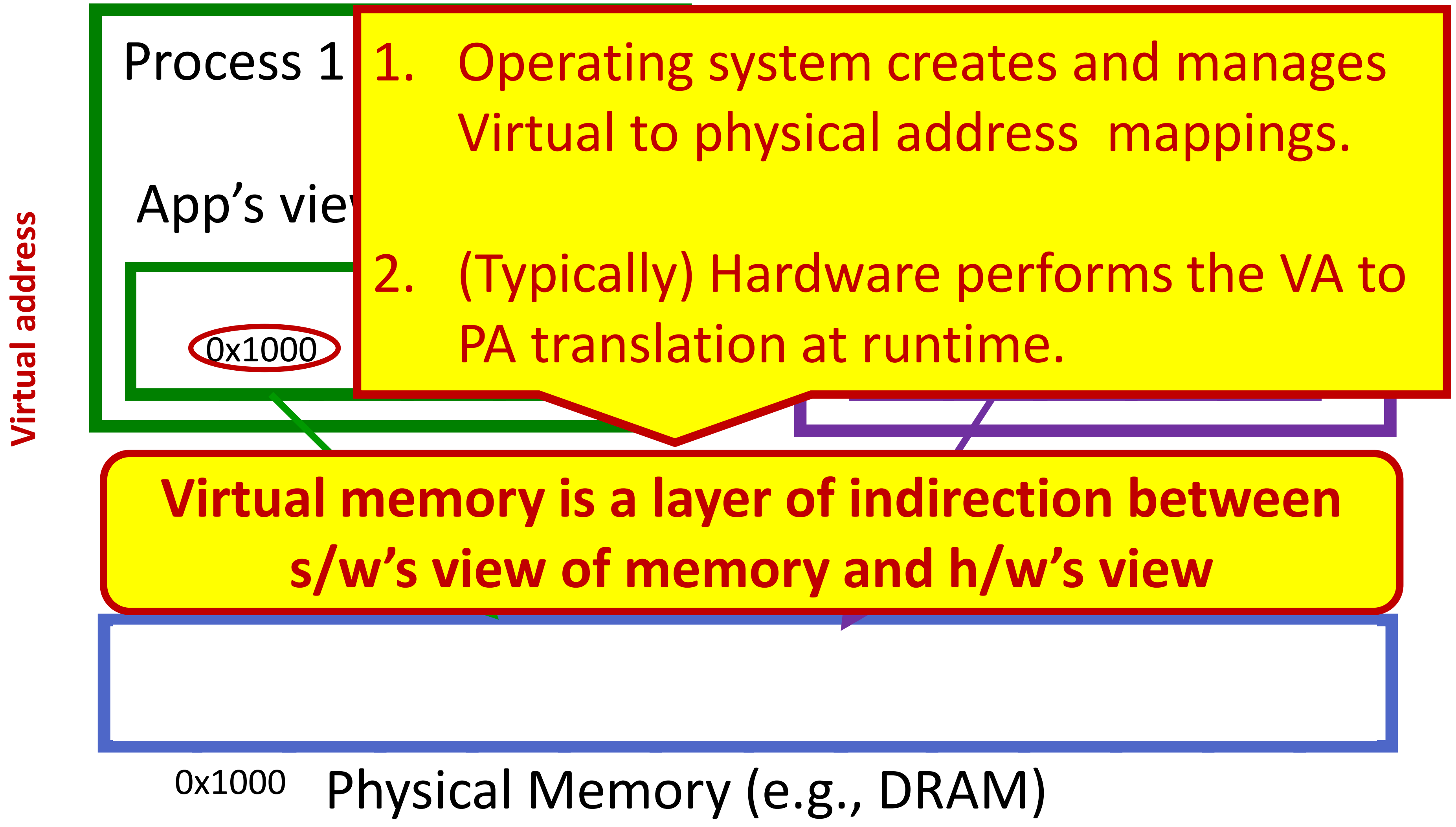


Bird’s view of virtual memory



Virtual memory is a layer of indirection between s/w’s view of memory and h/w’s view

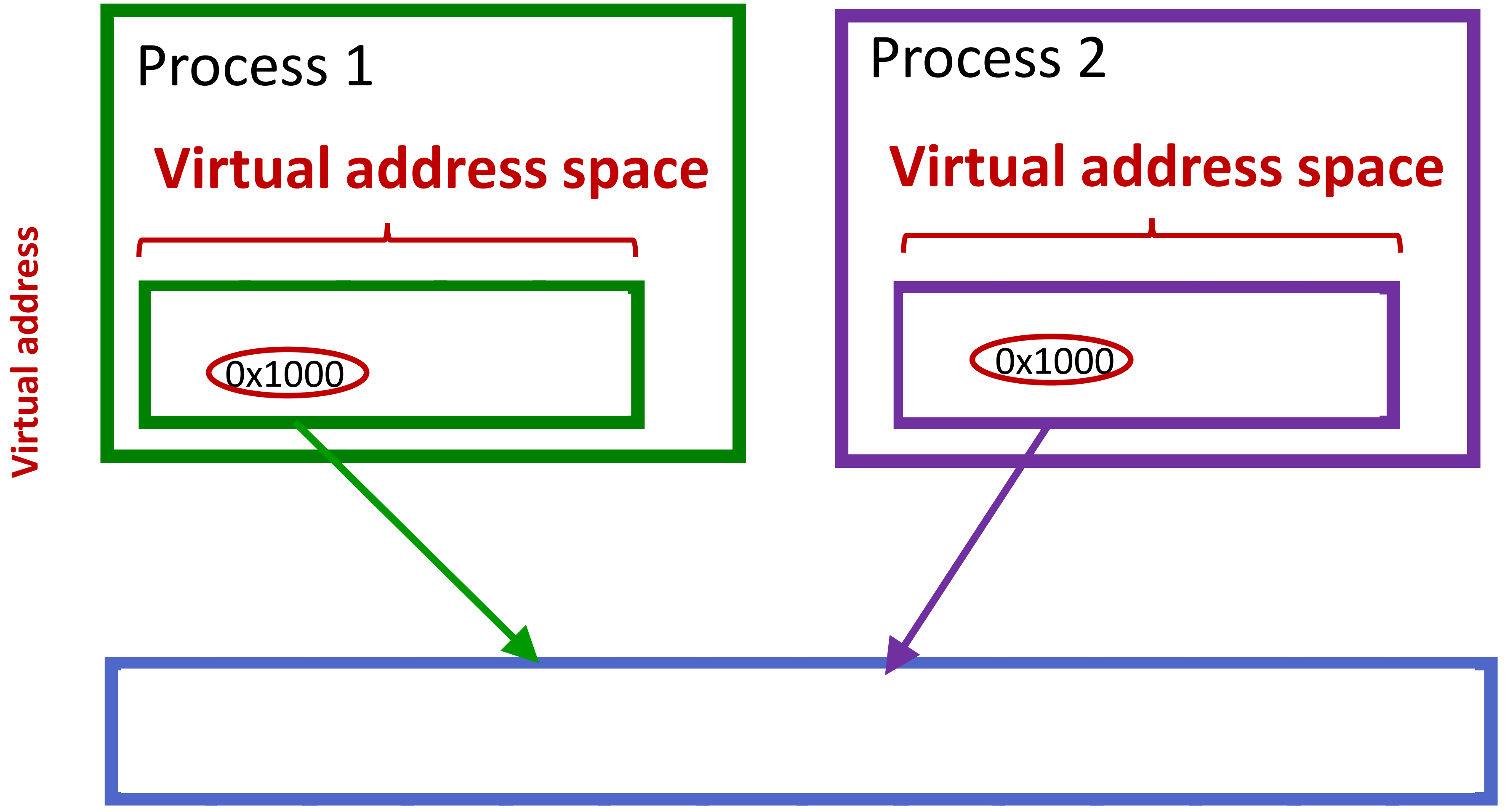
Roles of software and hardware



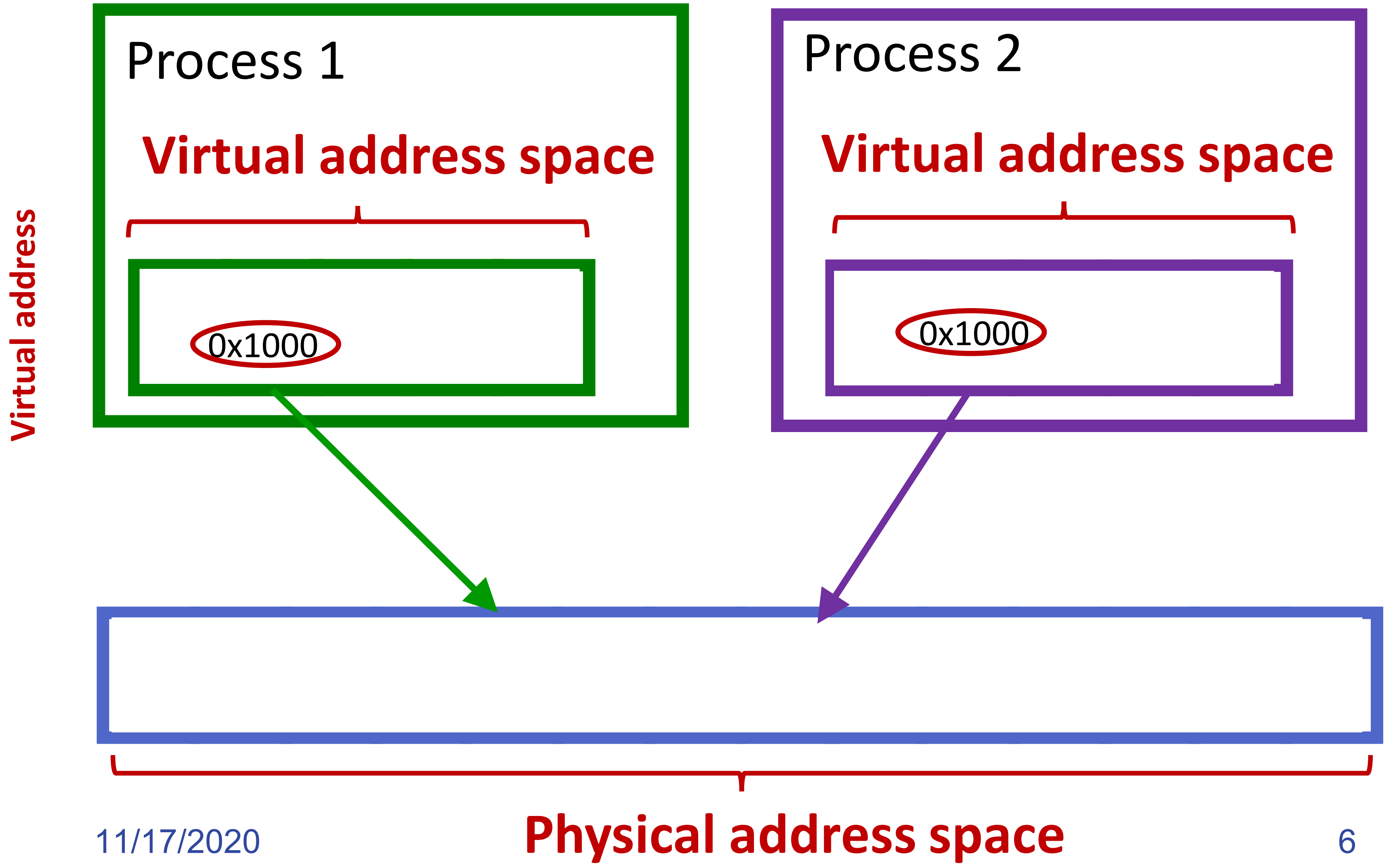
1. Operating system creates and manages Virtual to physical address mappings.
2. (Typically) Hardware performs the VA to PA translation at runtime.

Virtual memory is a layer of indirection between s/w's view of memory and h/w's view

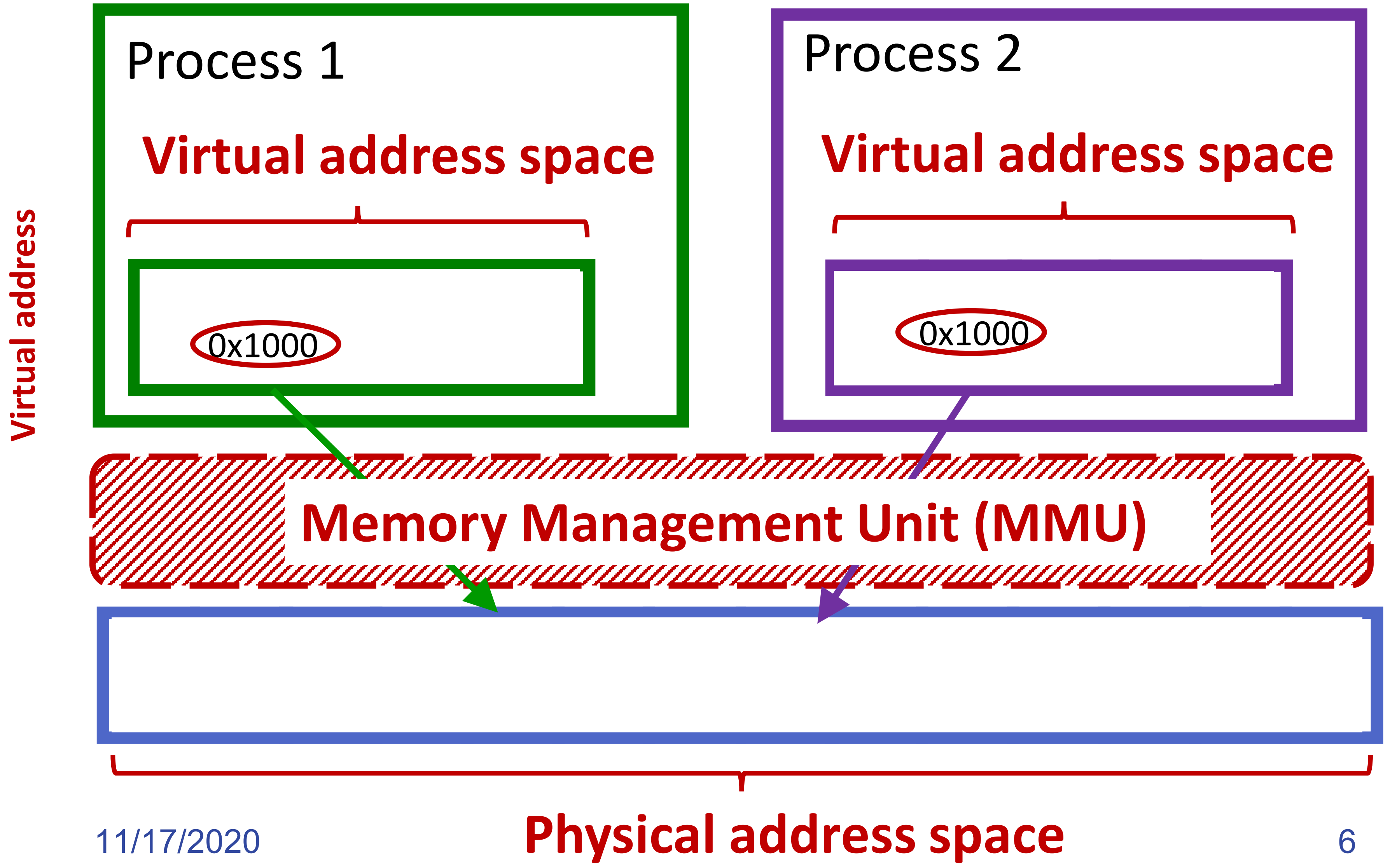
Virtual memory terminology



Virtual memory terminology



Virtual memory terminology



Why virtual memory?

Why virtual memory?

- **Relocation**: Ease of programming by providing application contiguous view of memory
  - But actual physical memory can be scattered

Why virtual memory?

- **Relocation**: Ease of programming by providing application contiguous view of memory
  - But actual physical memory can be scattered
- **Resource management:** Application writer can assume there is enough memory
  - But, system can manage physical memory across concurrent processes
- **Isolation:** Bug in one process should not corrupt memory of another under multi-programming
  - Provide each process with separate virtual address space
- **Protection:** Enforce rules on what memory a process can or cannot access

Why virtual memory?

Why virtual memory?

- **Sharing/communication**: Inter-process communication and sharing

Why virtual memory?

- **Sharing/communication**: Inter-process communication and sharing
  - Same physical address can be mapped onto two process's virtual address space with different virtual address

Why virtual memory?

- **Sharing/communication**: Inter-process communication and sharing
  - Same physical address can be mapped onto two process’s virtual address space with different virtual address
    - Could be used for inter-process-communication

Why virtual memory?

- **Sharing/communication**: Inter-process communication and sharing
  - Same physical address can be mapped onto two process's virtual address space with different virtual address
    - Could be used for inter-process-communication
    - Very useful for code/library sharing

Why virtual memory?

- **Sharing/communication**: Inter-process communication and sharing
  - Same physical address can be mapped onto two process’s virtual address space with different virtual address
    - Could be used for inter-process-communication
    - Very useful for code/library sharing
- **Illusion of larger memory:** Allows memory overcommit by providing illusion of memory larger than actually available

Why virtual memory?

- **Sharing/communication**: Inter-process communication and sharing
  - Same physical address can be mapped onto two process's virtual address space with different virtual address
    - Could be used for inter-process-communication
    - Very useful for code/library sharing
- **Illusion of larger memory:** Allows memory overcommit by providing illusion of memory larger than actually available
  - Can write program without worrying about amount of free memory in a system where the program will run

Agenda

What is virtual memory?

Hardware implementations of virtual memory

Software management of virtual memory

Research opportunity in virtual memory

Agenda

What is virtual memory?

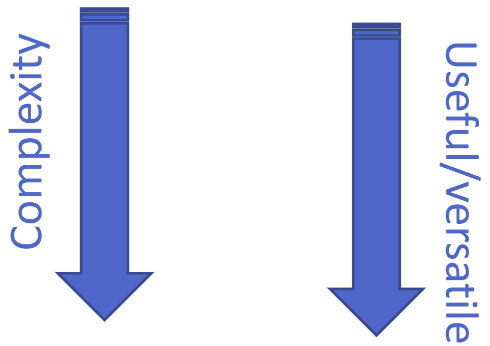
Hardware implementations of virtual memory

Software management of virtual memory

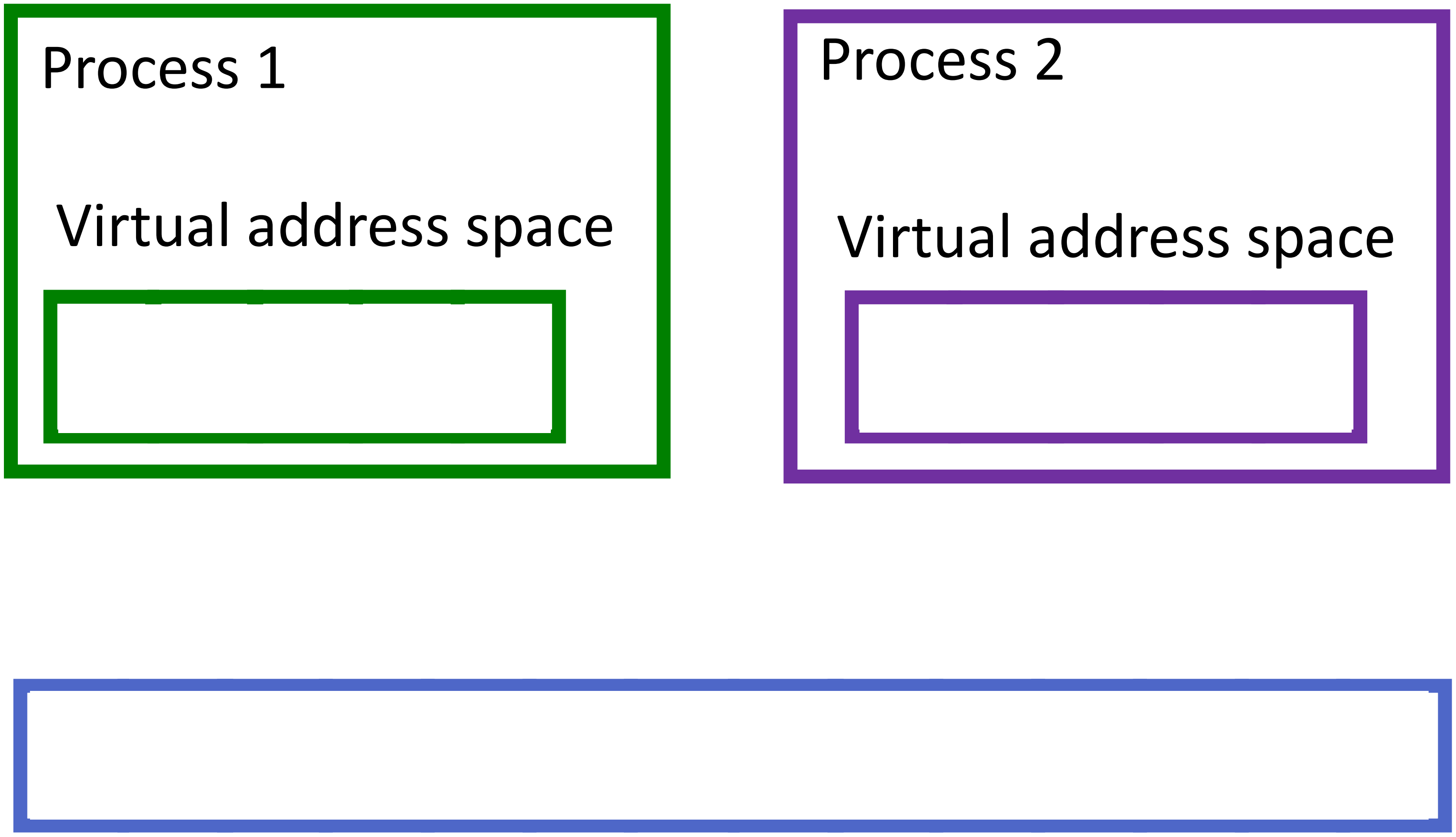
Research opportunity in virtual memory

Implementing virtual memory

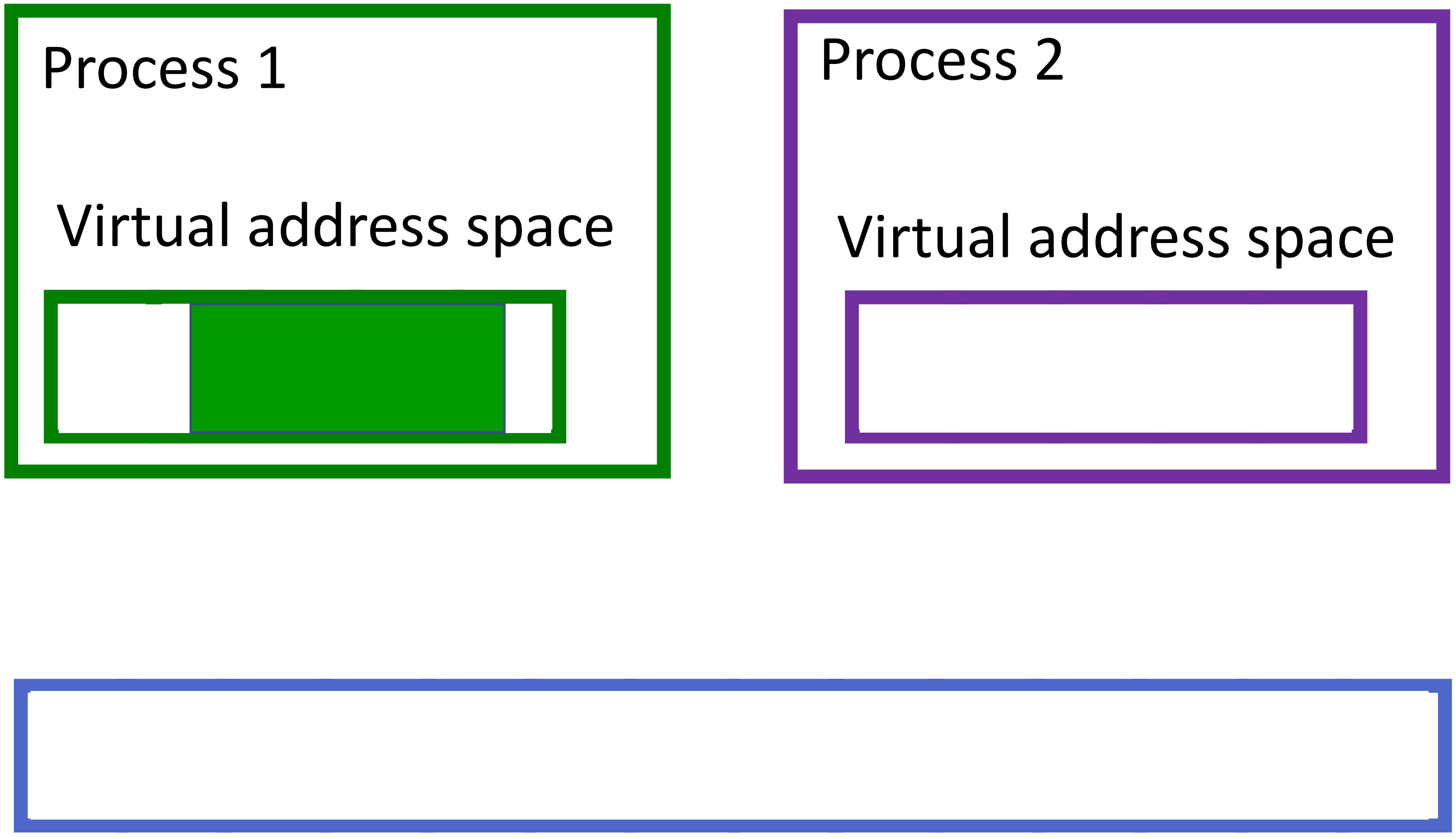
- Many ways to implement virtual memory
  - Base and bound register
  - Segmentation
  - Paging



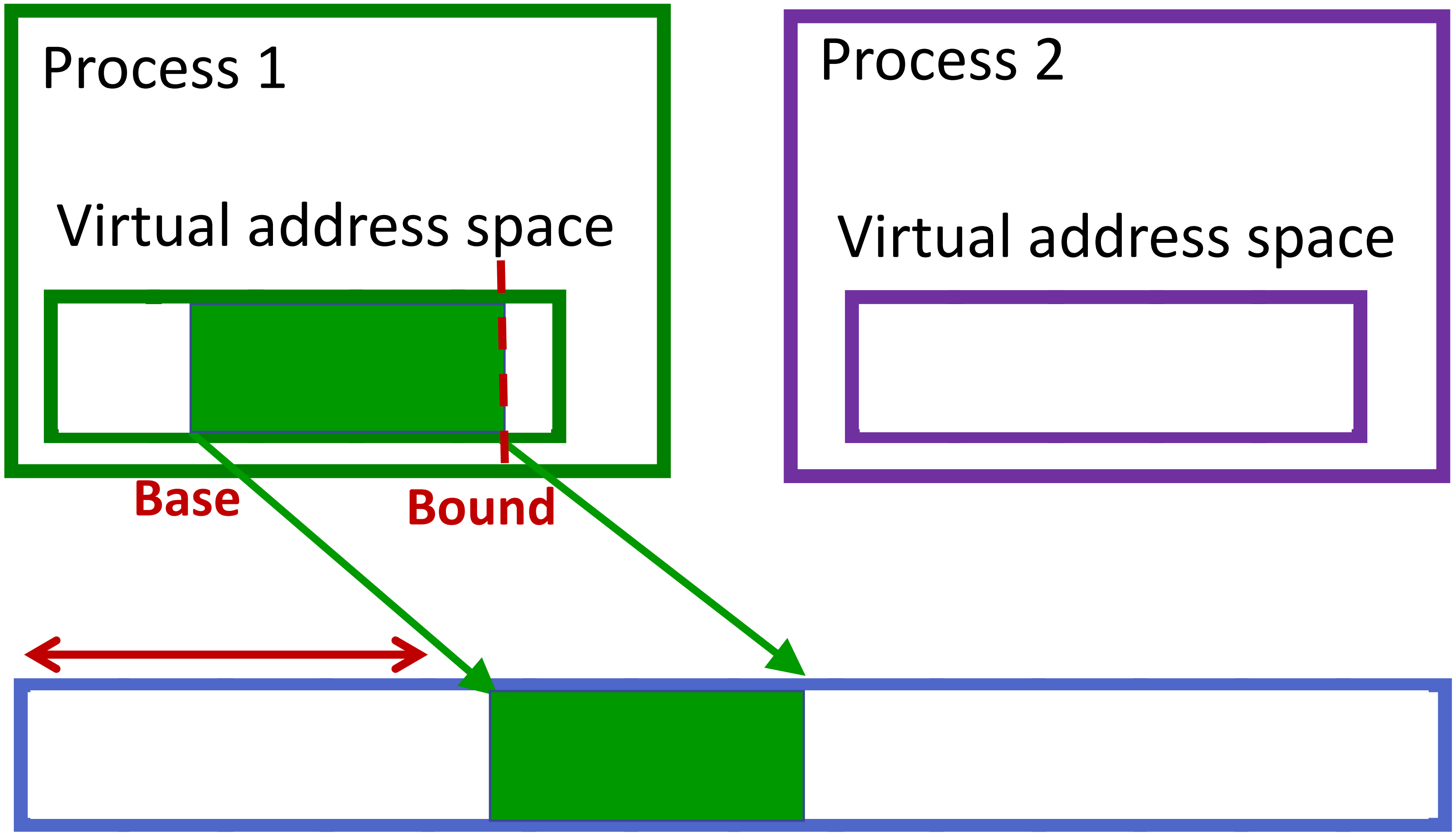
Base and bound registers



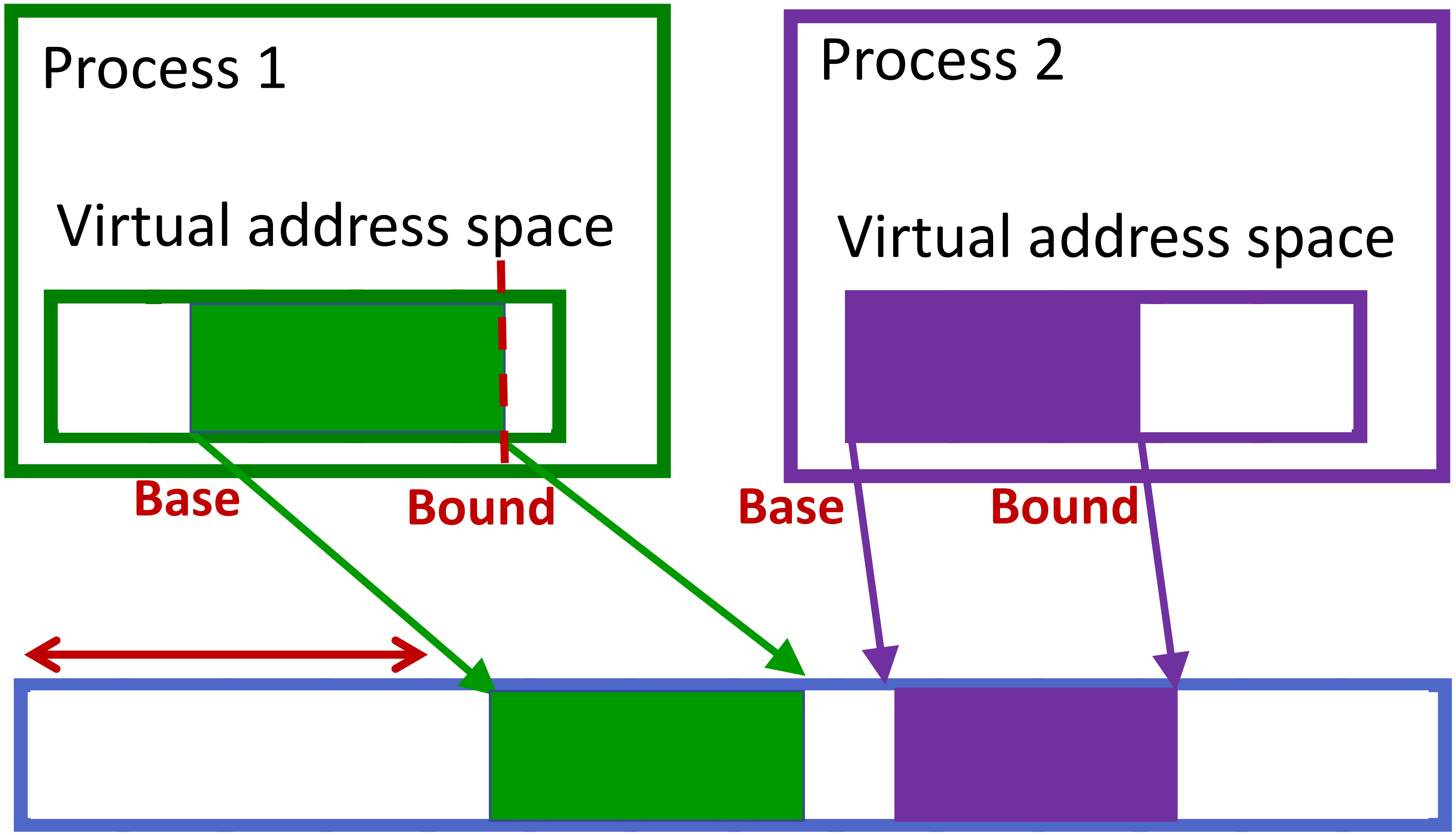
Base and bound registers



Base and bound registers



Base and bound registers



Base and bound registers

- How it works?
  - Each CPU core has base and bound registers
  - Each process has own *values* for base and bound
  - Base and bound values are assigned by OS
  - On a context switch, a process’s base and bound registers are saved/restored by the OS

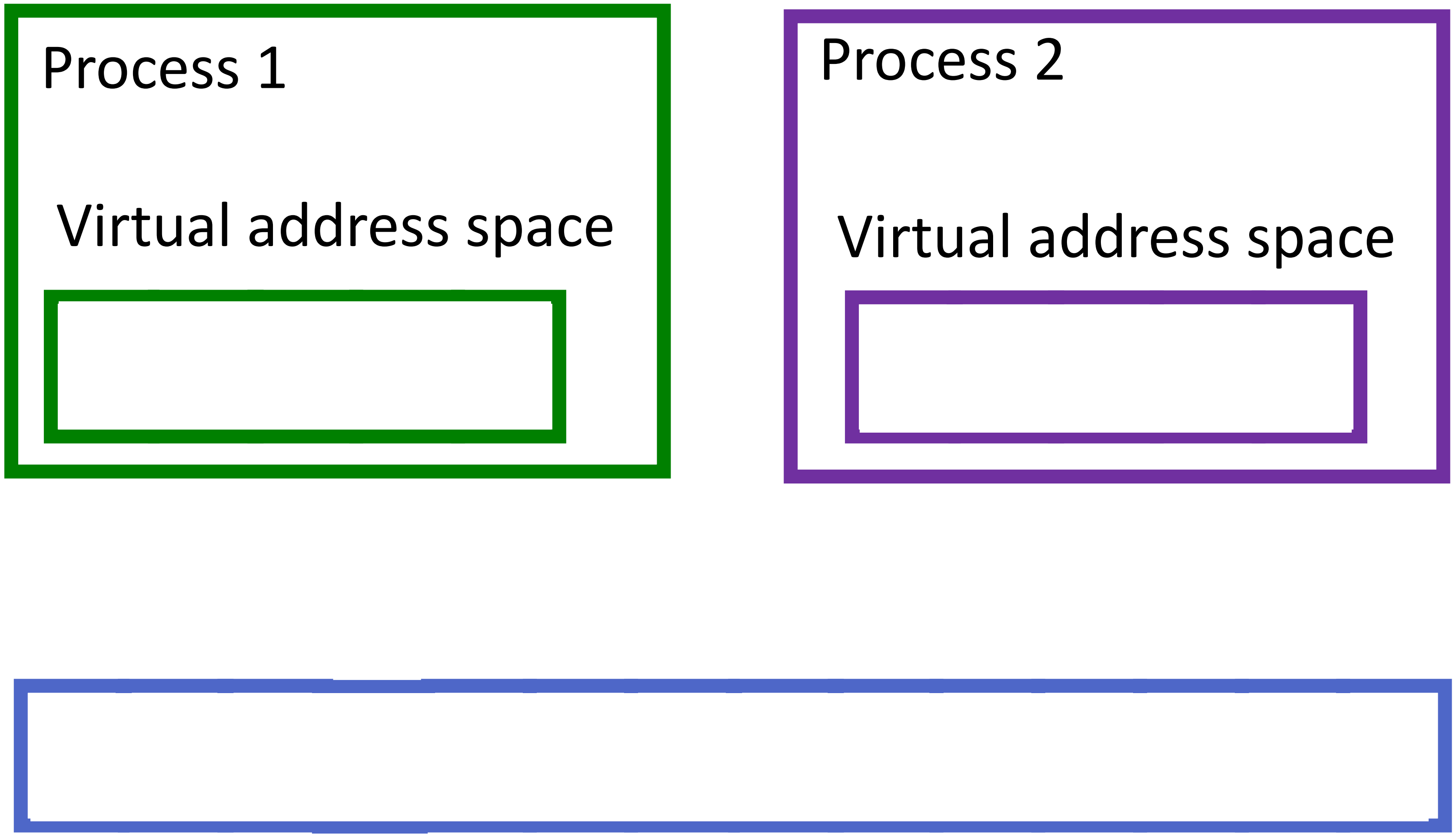
Base and bound registers

- How it works?
  - Each CPU core has base and bound registers
  - Each process has own *values* for base and bound
  - Base and bound values are assigned by OS
  - On a context switch, a process's base and bound registers are saved/restored by the OS
- Example: Cray-1 (very old system)
- Advantage
  - Simple

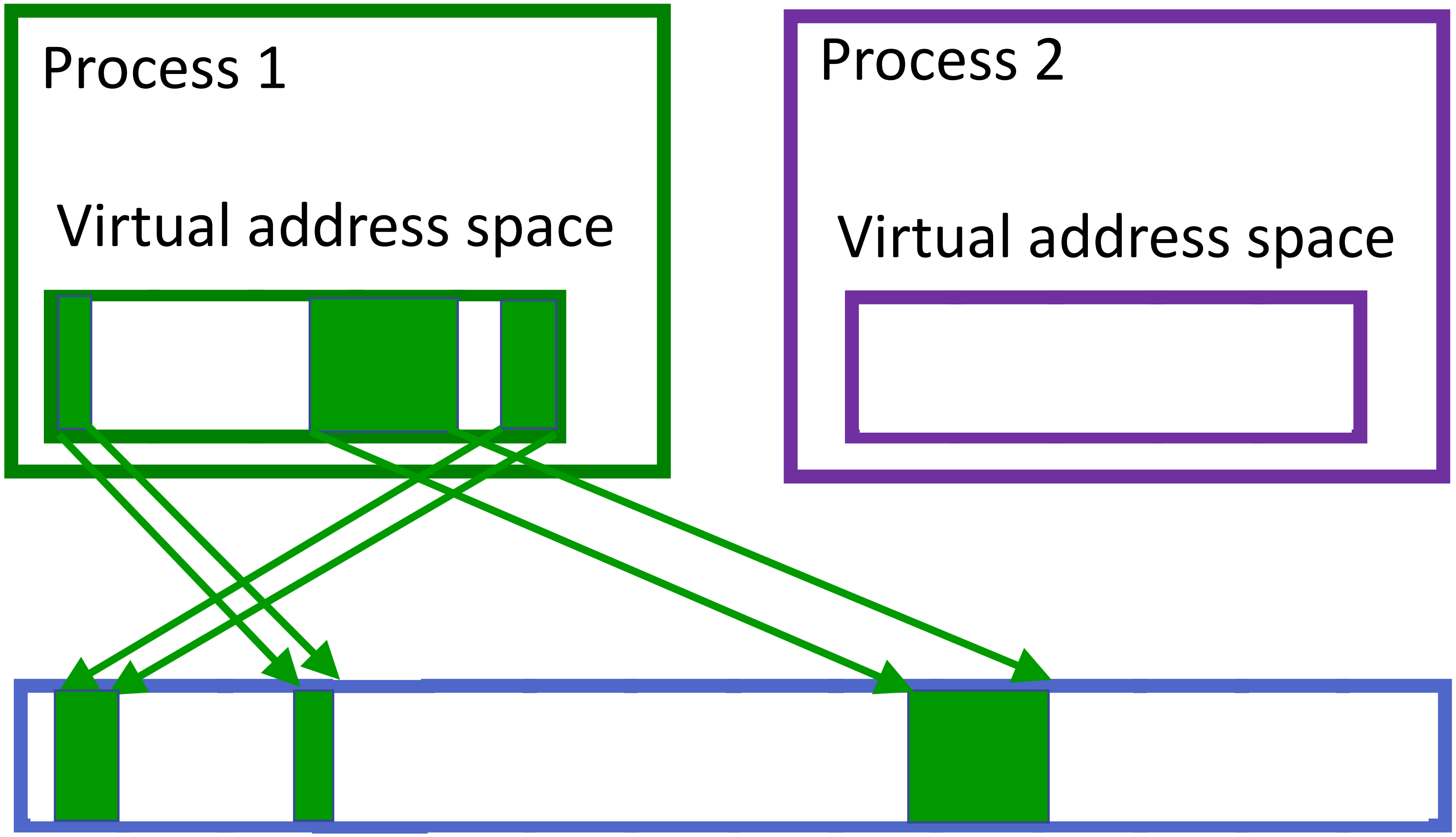
Base and bound registers

- How it works?
  - Each CPU core has base and bound registers
  - Each process has own *values* for base and bound
  - Base and bound values are assigned by OS
  - On a context switch, a process's base and bound registers are saved/restored by the OS
- Example: Cray-1 (very old system)
- Advantage
  - Simple
- Disadvantage
  - Needs contiguous physical memory
  - Deallocation of memory is possible only on the edge
  - Cannot allocate more memory if no free memory at the edge

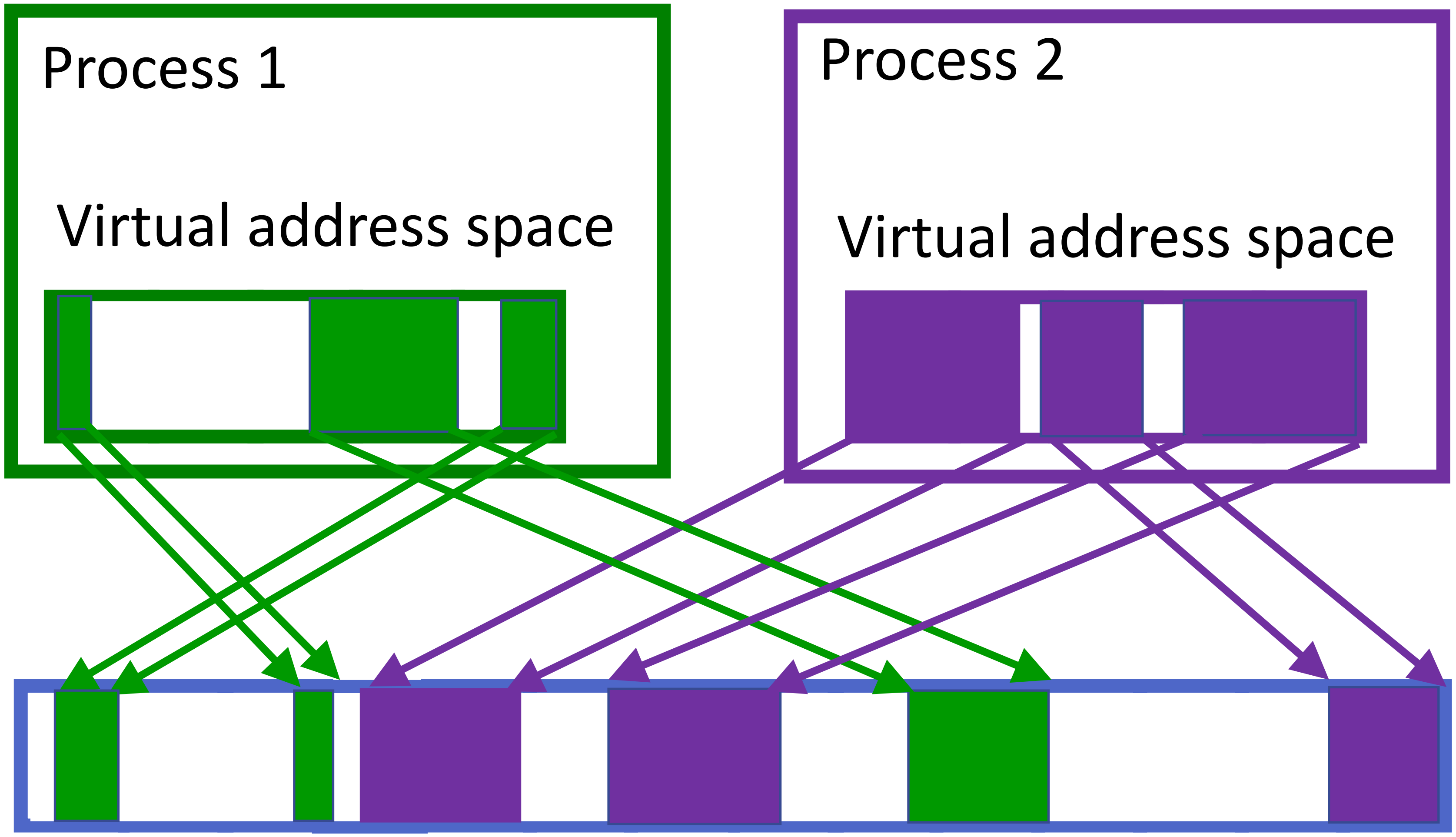
Segmentation



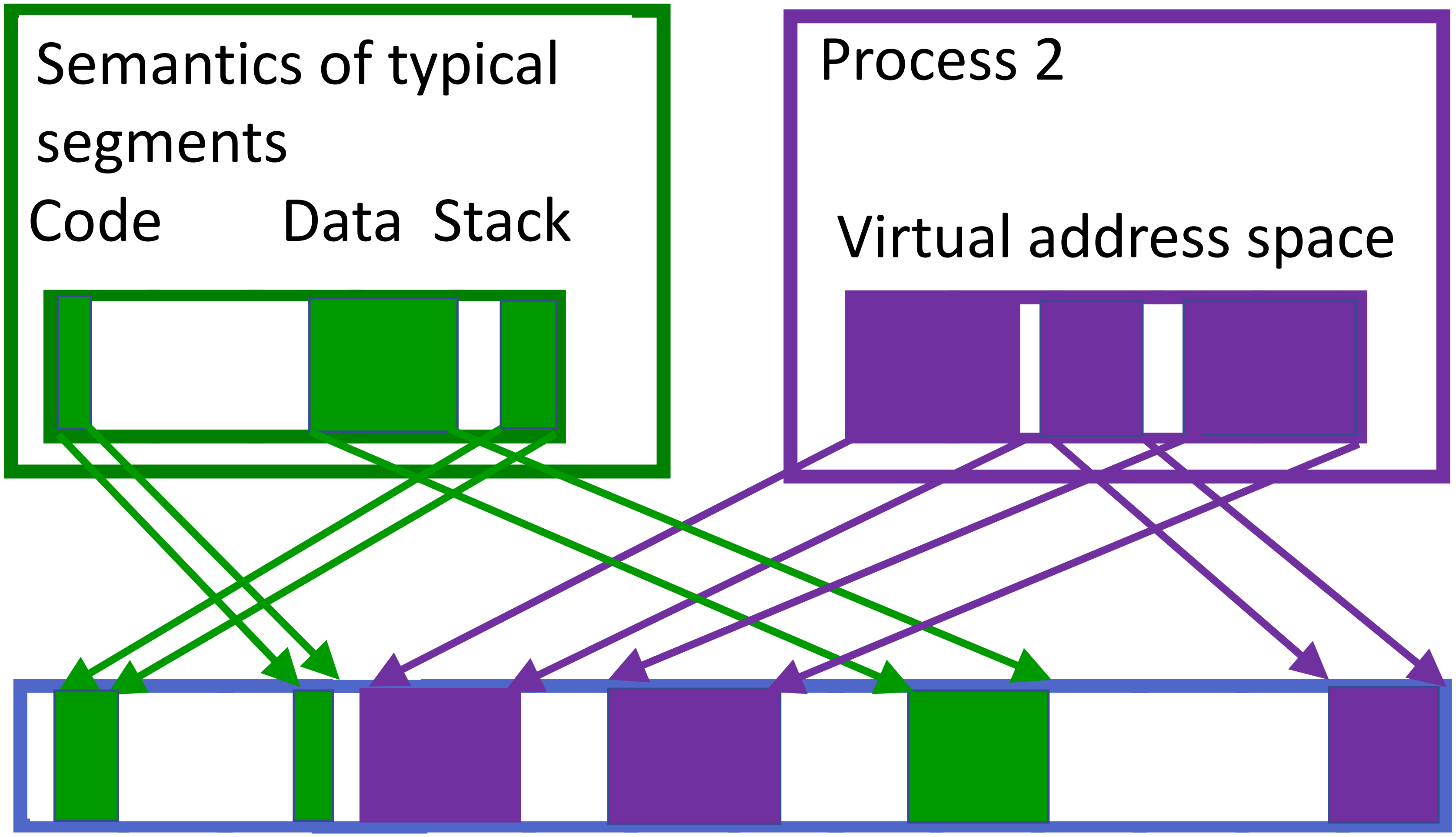
Segmentation



Segmentation



Segmentation



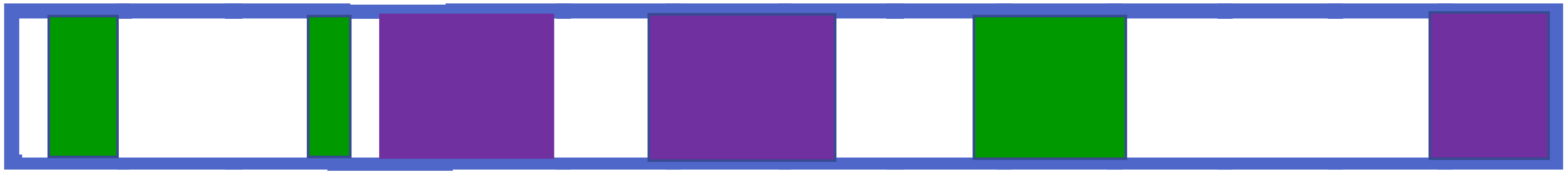
Segmentation: How it works?

Virtual address

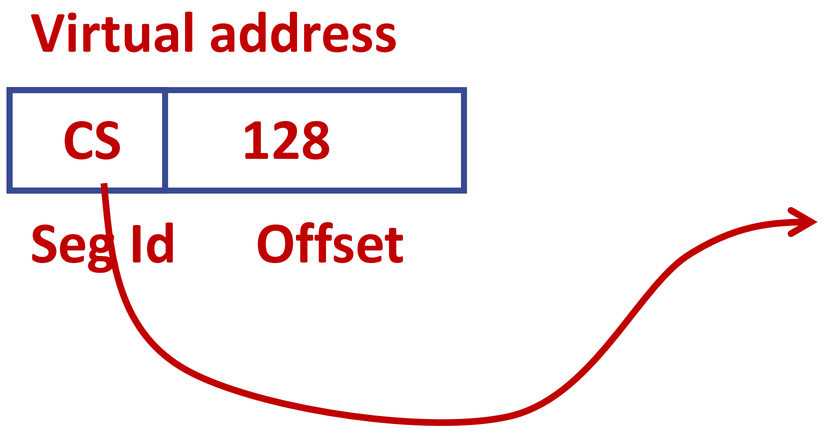
CS	128
Seg Id	Offset

	Prot	Base	Length
SS	R/W	0x0100	1024
CS	R	0x30000	1024
DS	R/W	0x40100	16384

Segment table

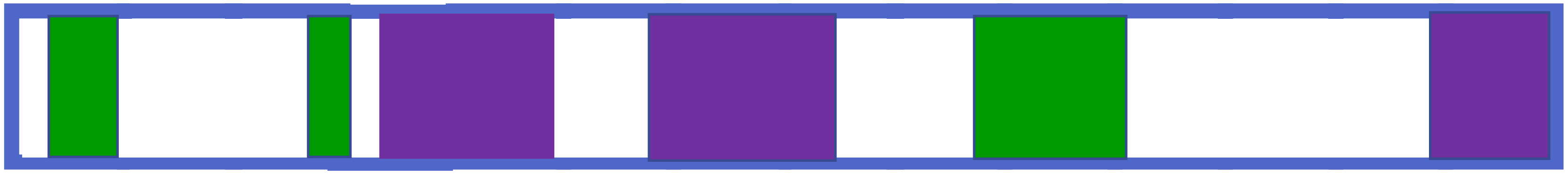


Segmentation: How it works?



	Prot	Base	Length
SS	R/W	0x0100	1024
CS	R	0x30000	1024
DS	R/W	0x40100	16384

Segment table



Segmentation: How it works?

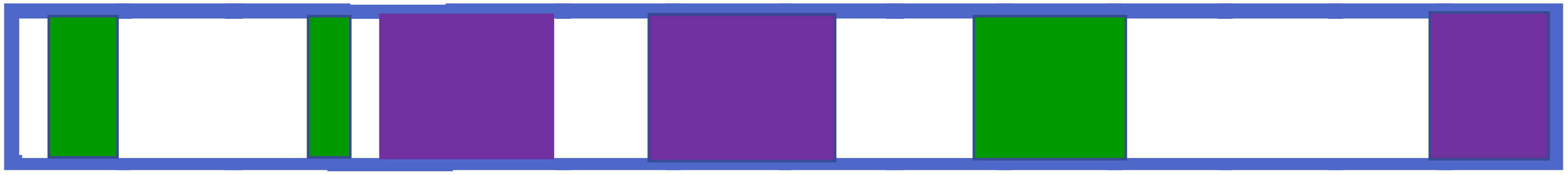
Virtual address

CS	128
Seg Id	Offset

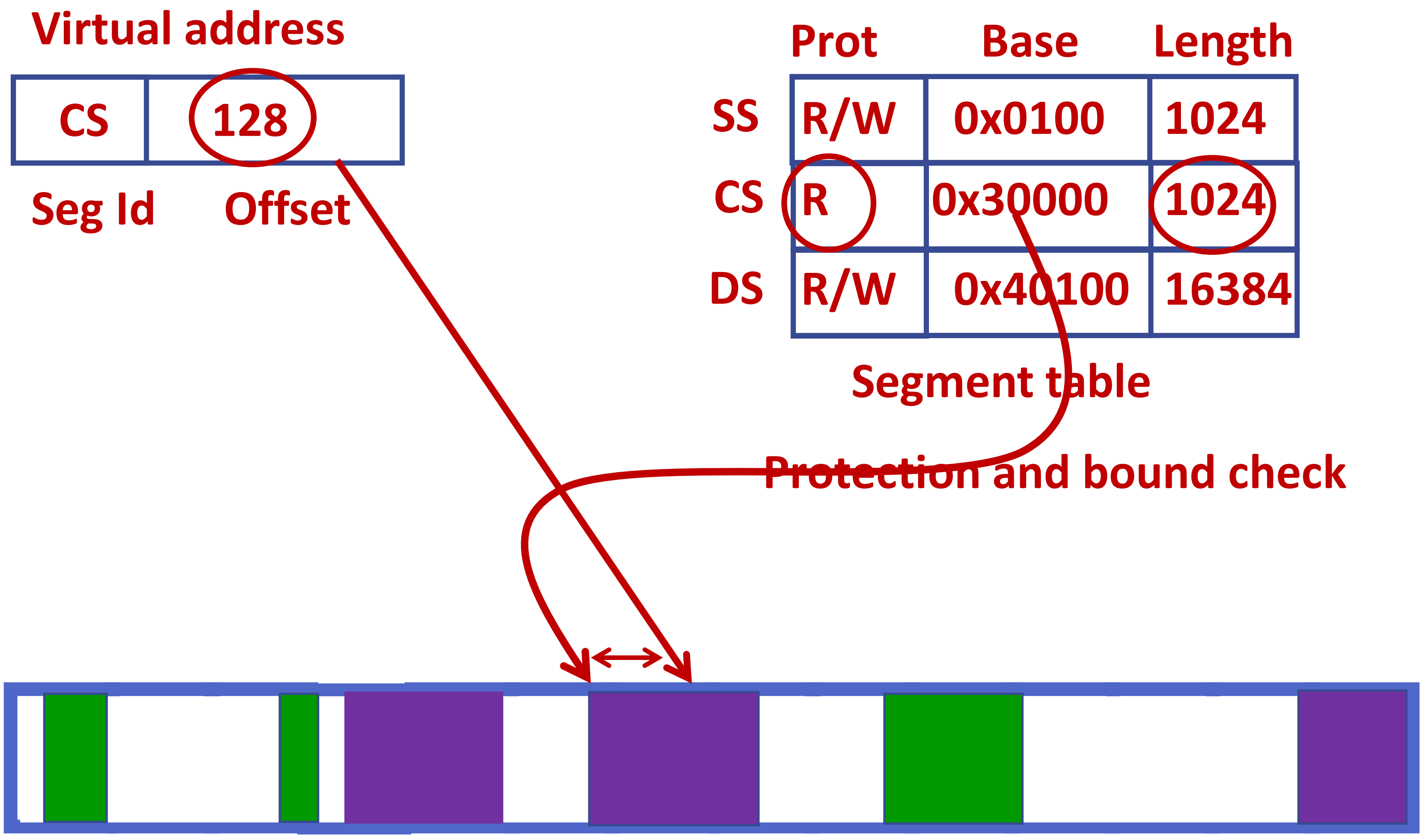
	Prot	Base	Length
SS	R/W	0x0100	1024
CS	R	0x30000	1024
DS	R/W	0x40100	16384

Segment table

Protection and bound check



Segmentation: How it works?



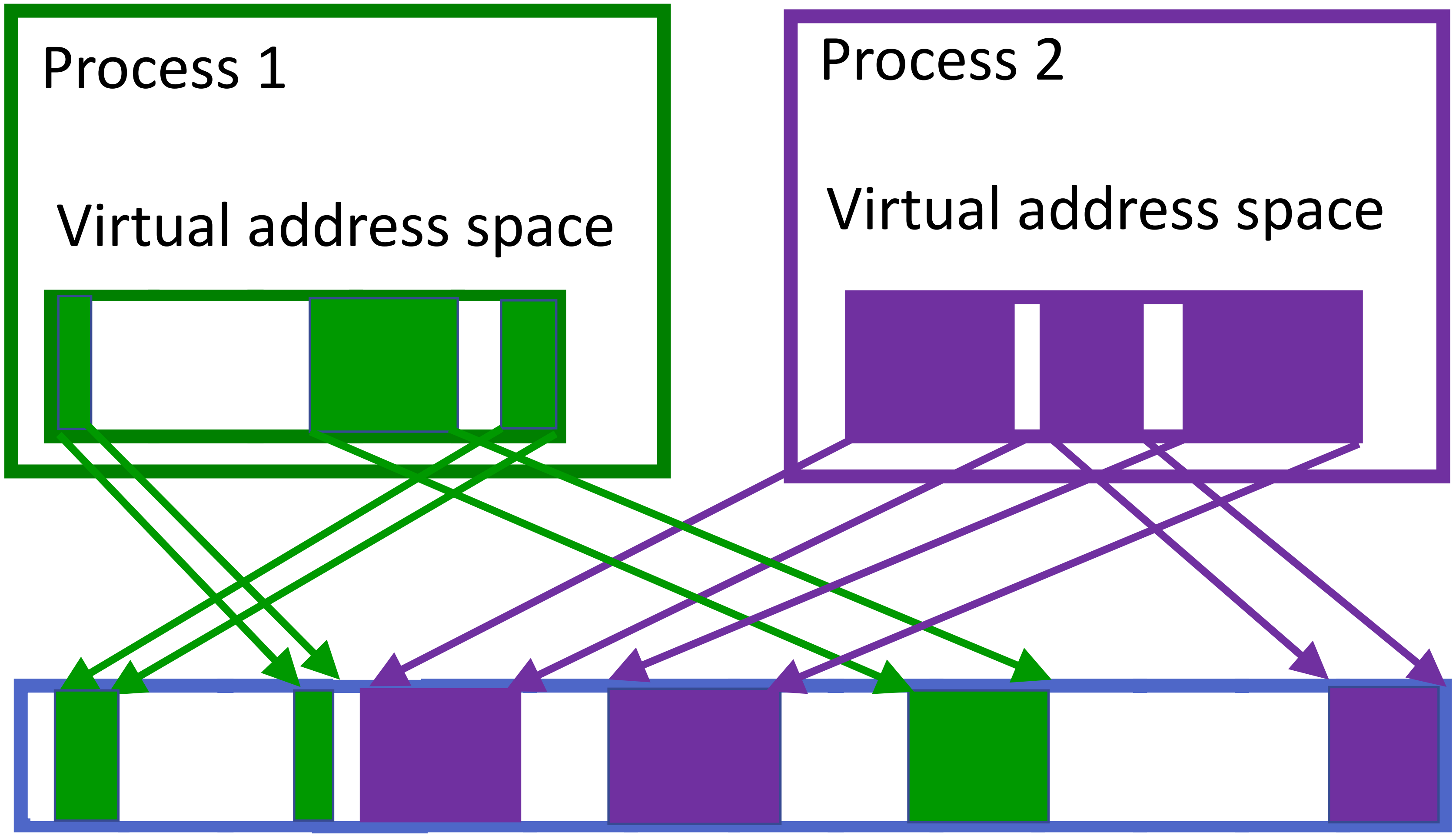
	Prot	Base	Length
SS	R/W	0x0100	1024
CS	R	0x30000	1024
DS	R/W	0x40100	16384

Segmentation: How it works?

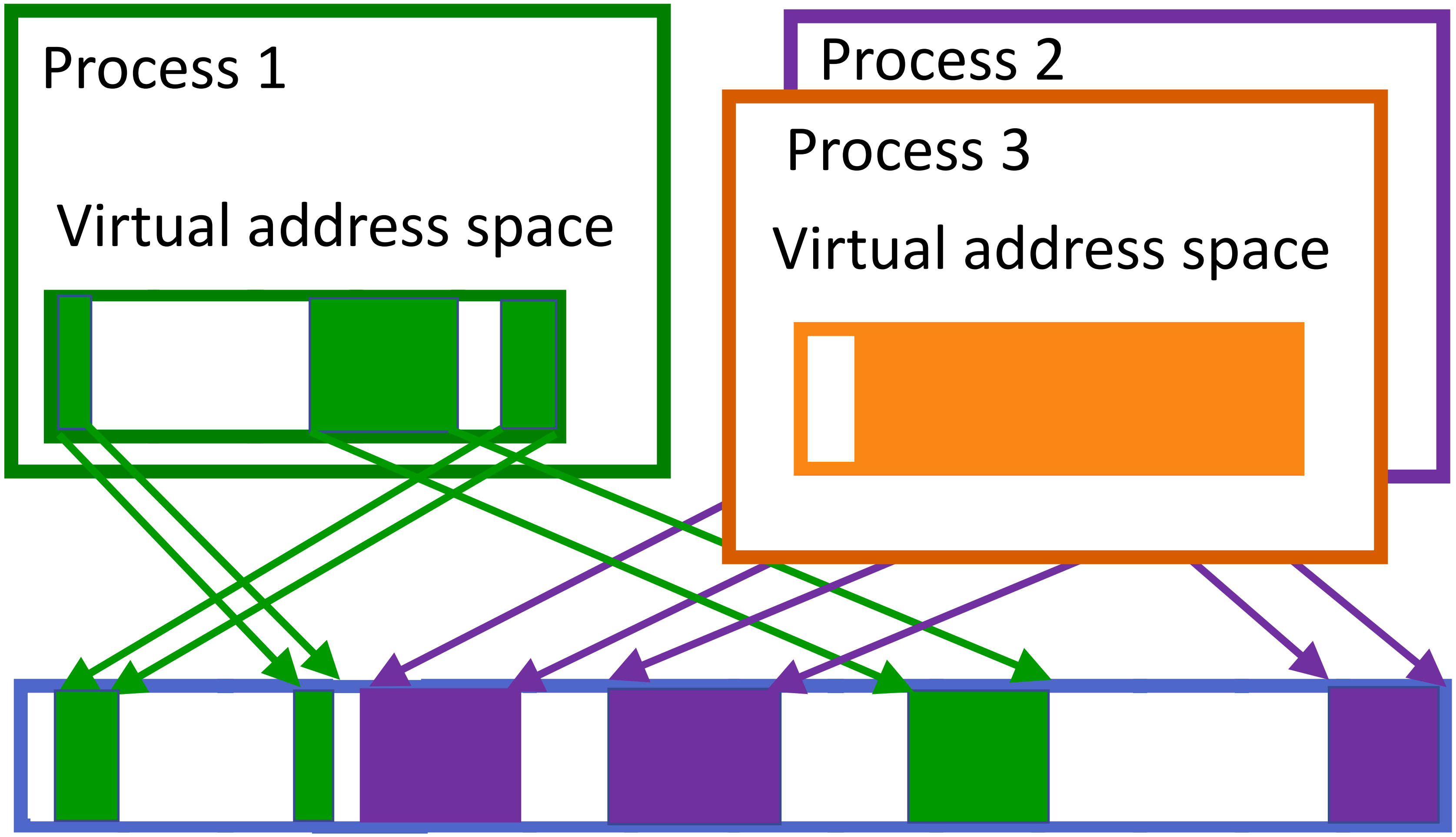
- How to select segmentation ID?
  - In most cases, the compiler can infer
    - For example, x86-32 bit has $\underbrace{\text{CS, SS, DS}}_{\text{Compiler selected}}, \underbrace{\text{ES, FS, GS}}_{\text{Programmer selected}}$

- Advantages segmentation:
  - Multiple memory regions with different protections
  - No need to have all segments in memory all the time
  - Ability to share segments across processes

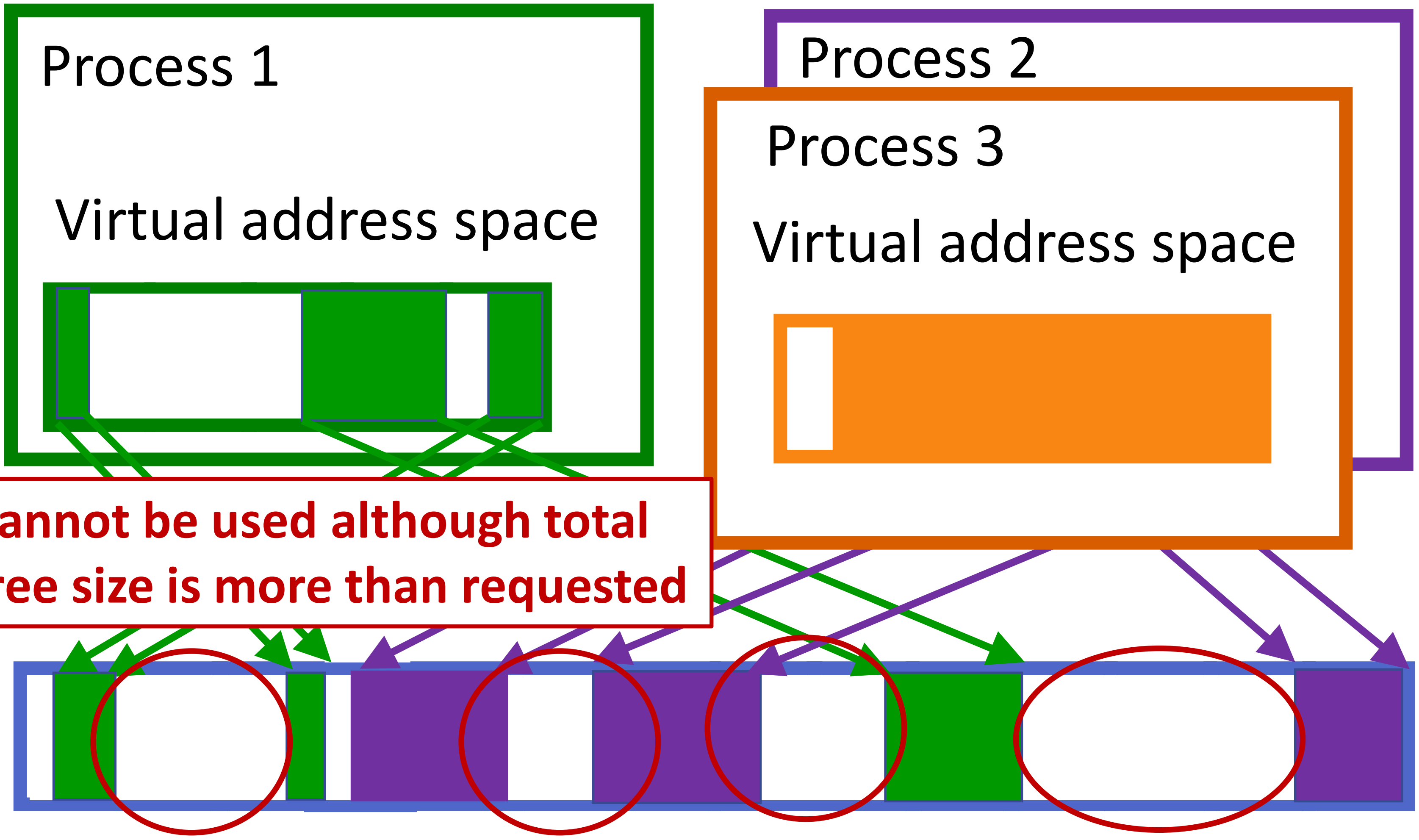
Disadvantages of Segmentation



Disadvantages of Segmentation



Disadvantages of Segmentation



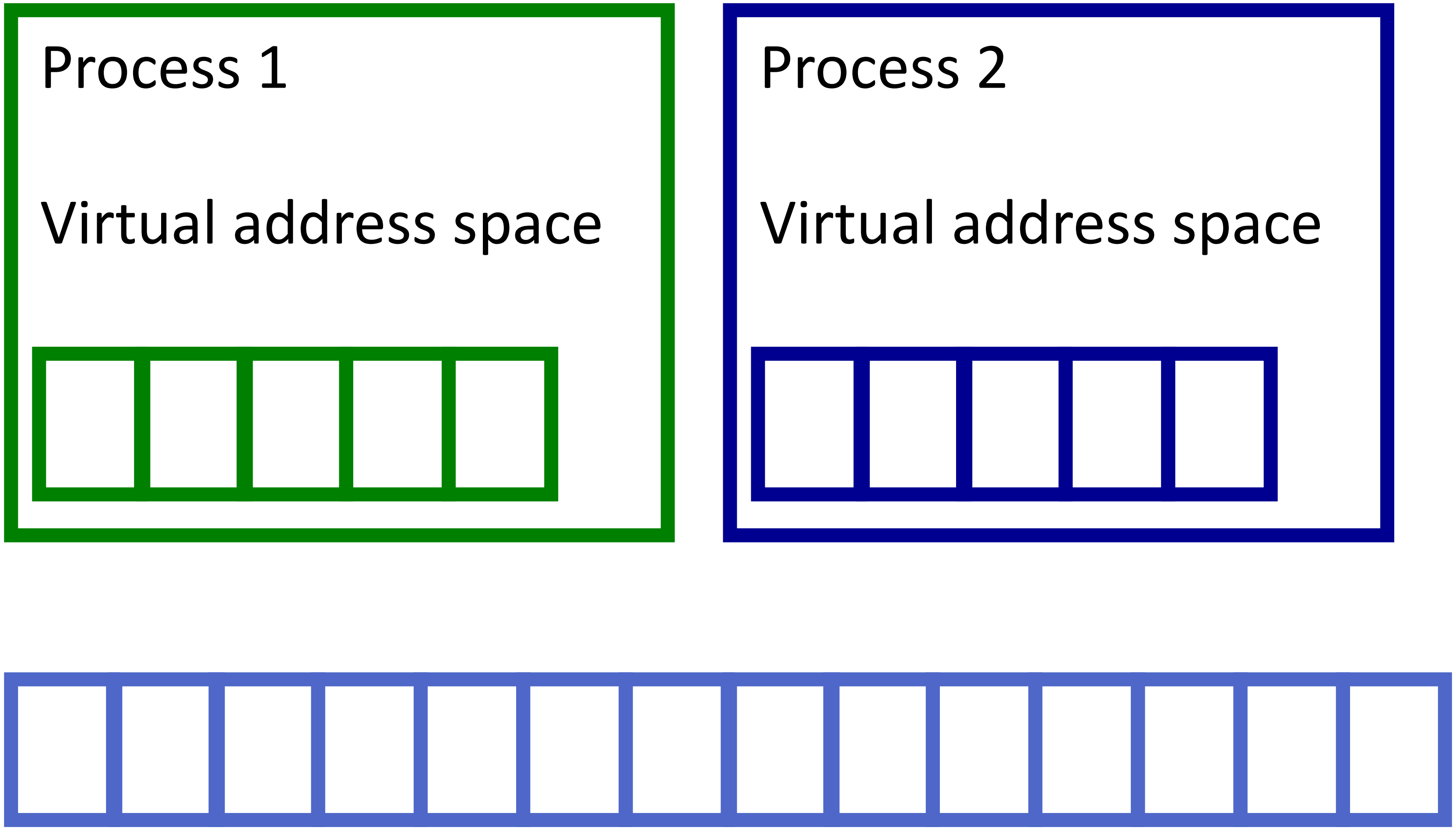
Disadvantages of Segmentation

- External fragmentation: Inability to use free memory as it is non-contiguous (fragmentation)
  - Allocating different sized segments leaves free memory fragmented
- A segment of size *n* bytes requires *n* bytes long contiguous physical memory
- Not completely transparent to applications

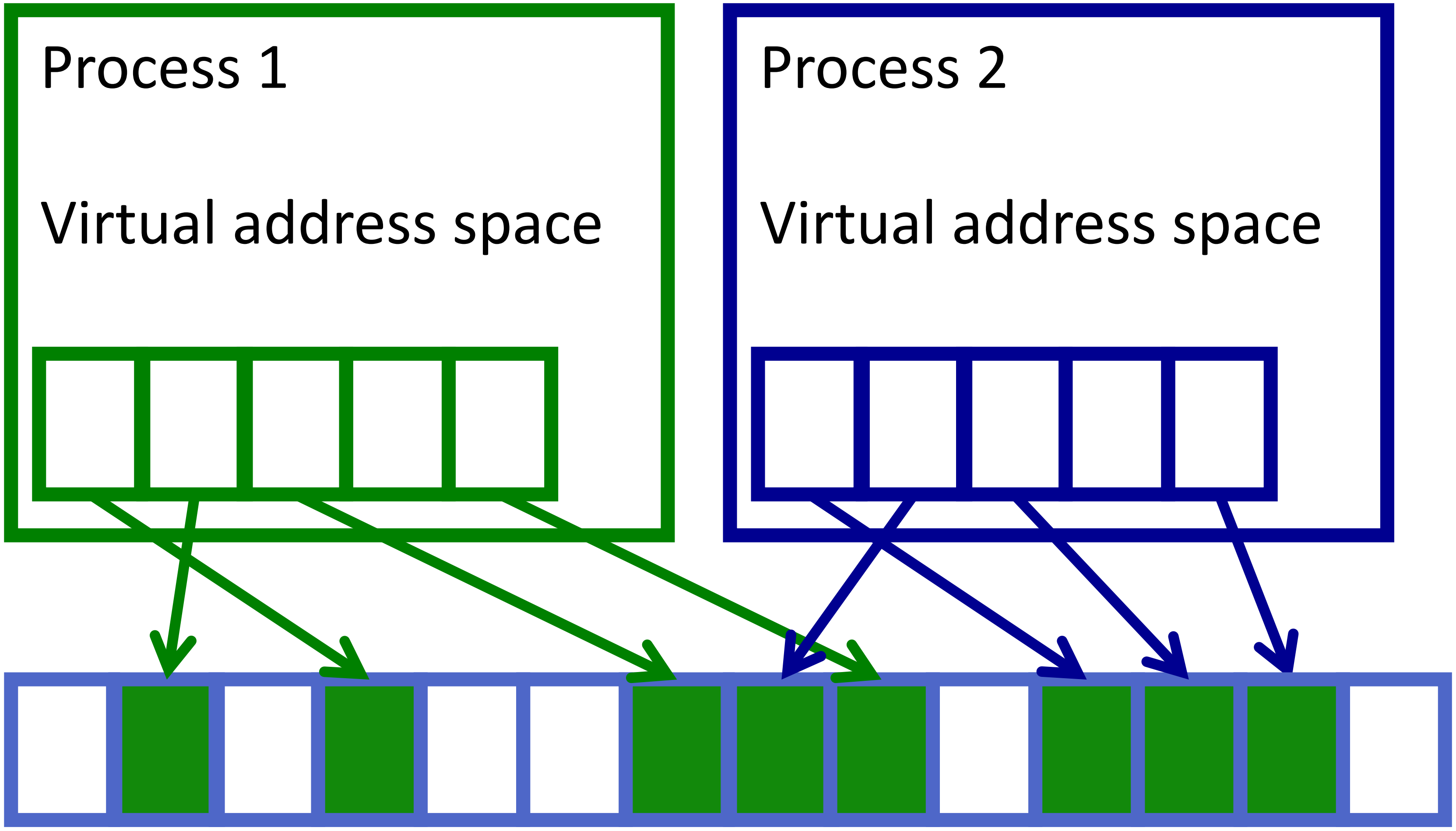
Paging

- Idea: Allocate/de-allocate memory in same size chunks (pages)
  - No external fragmentation (Why?)
  - Page sizes are typically small, e.g., 4KB

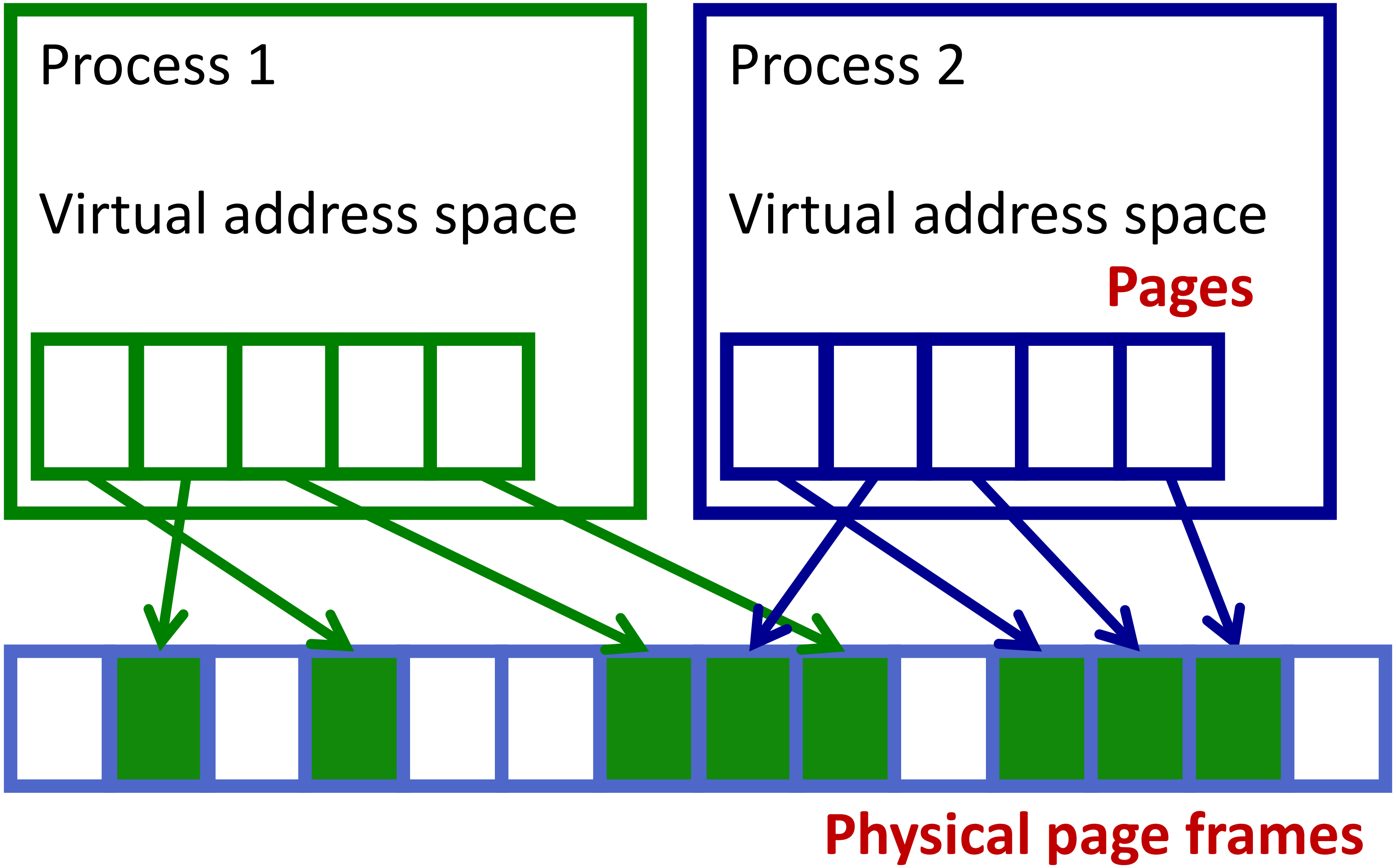
Paging


Process 1
Virtual address space
Process 2
Virtual address space

Paging


Process 1
Virtual address space
Process 2
Virtual address space

Paging


Process 1
Virtual address space
Process 2
Virtual address space
Pages
Physical page frames

Advantages of paging

- **No** external fragmentation !
- Simplifies allocation, deallocation
- Provides application a contiguous view of memory
  - But, physical memory backing it could very scattered

Advantages of paging

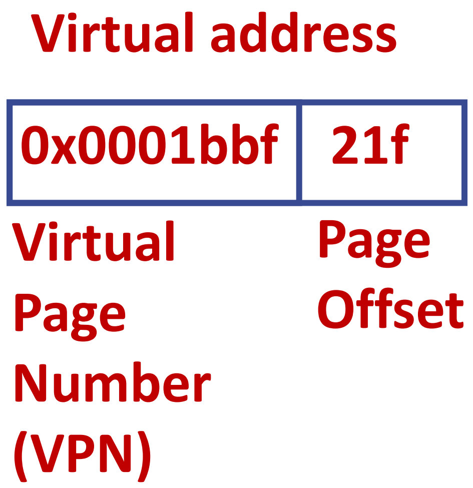
- **No** external fragmentation !
- Simplifies allocation, deallocation
- Provides application a contiguous view of memory
  - But, physical memory backing it could very scattered

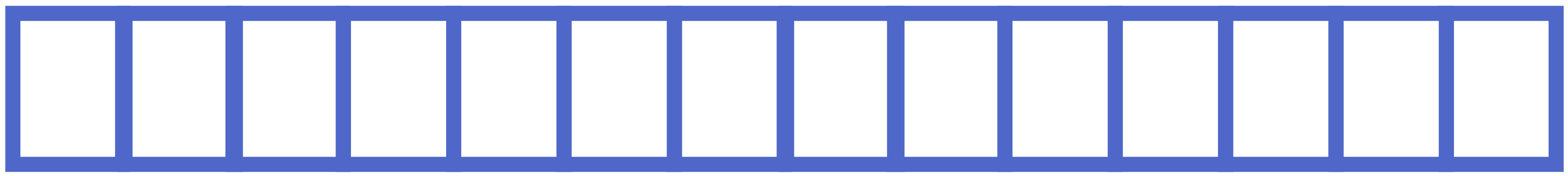
Almost all processors today employs paging

Disadvantages of paging

- **Internal fragmentation:** Memory wastage due to allocation only in page sizes (e.g., 4KB)
  - e.g., for an allocation 128 bytes a 4KB page will be allocated
  - Potentially waste (4KB – 128) bytes.

Paging: How it works (simplistic)?





Paging: How it works (simplistic)?

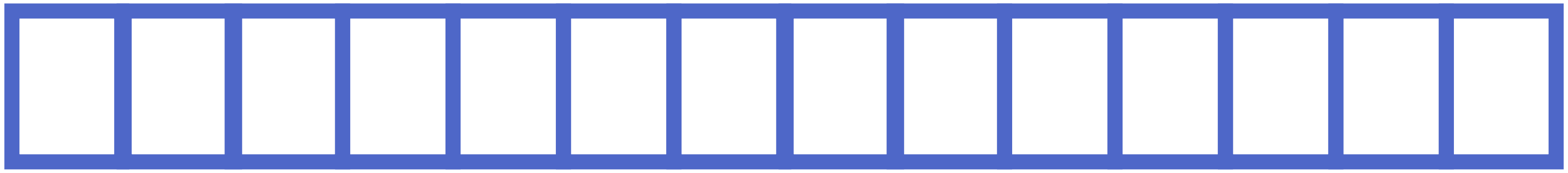
Virtual address

0x0001bbf	21f
Virtual Page Number (VPN)	Page Offset

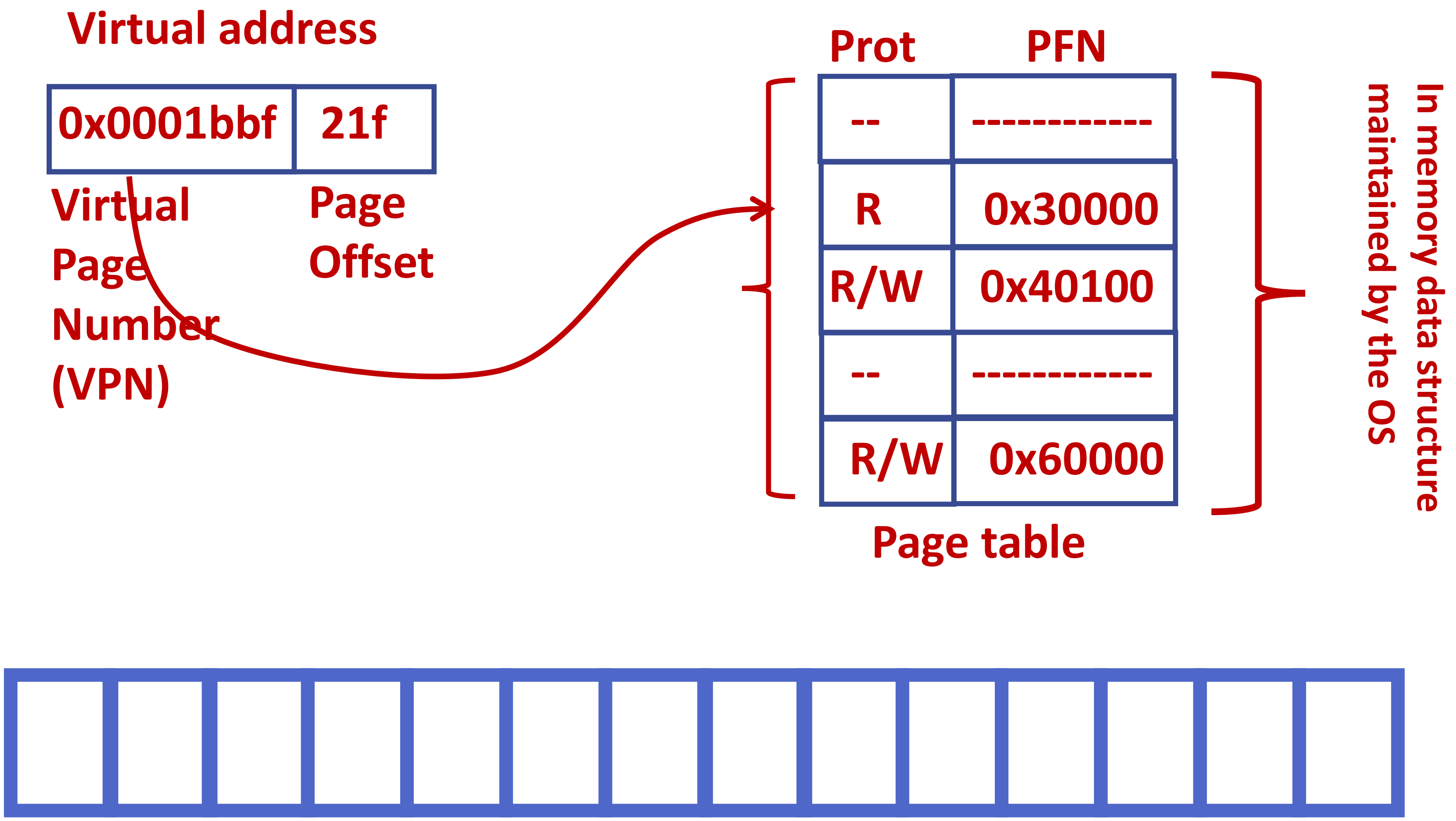
Prot	PFN
--	------------
R	0x30000
R/W	0x40100
--	------------
R/W	0x60000

Page table

In memory data structure maintained by the OS



Paging: How it works (simplistic)?



Paging: How it works (simplistic)?

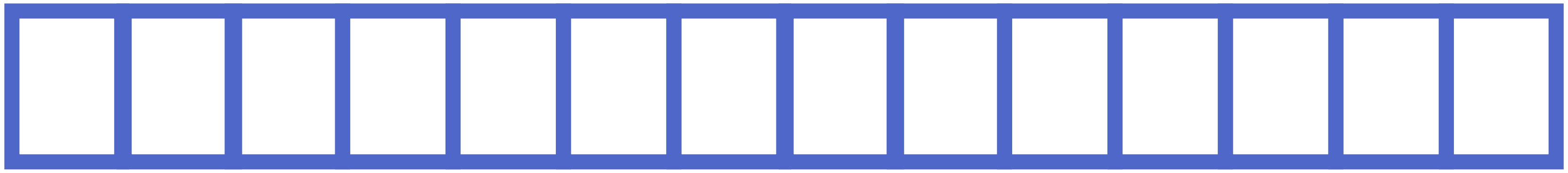
Virtual address

0x0001bbf	21f
Virtual Page Number (VPN)	Page Offset

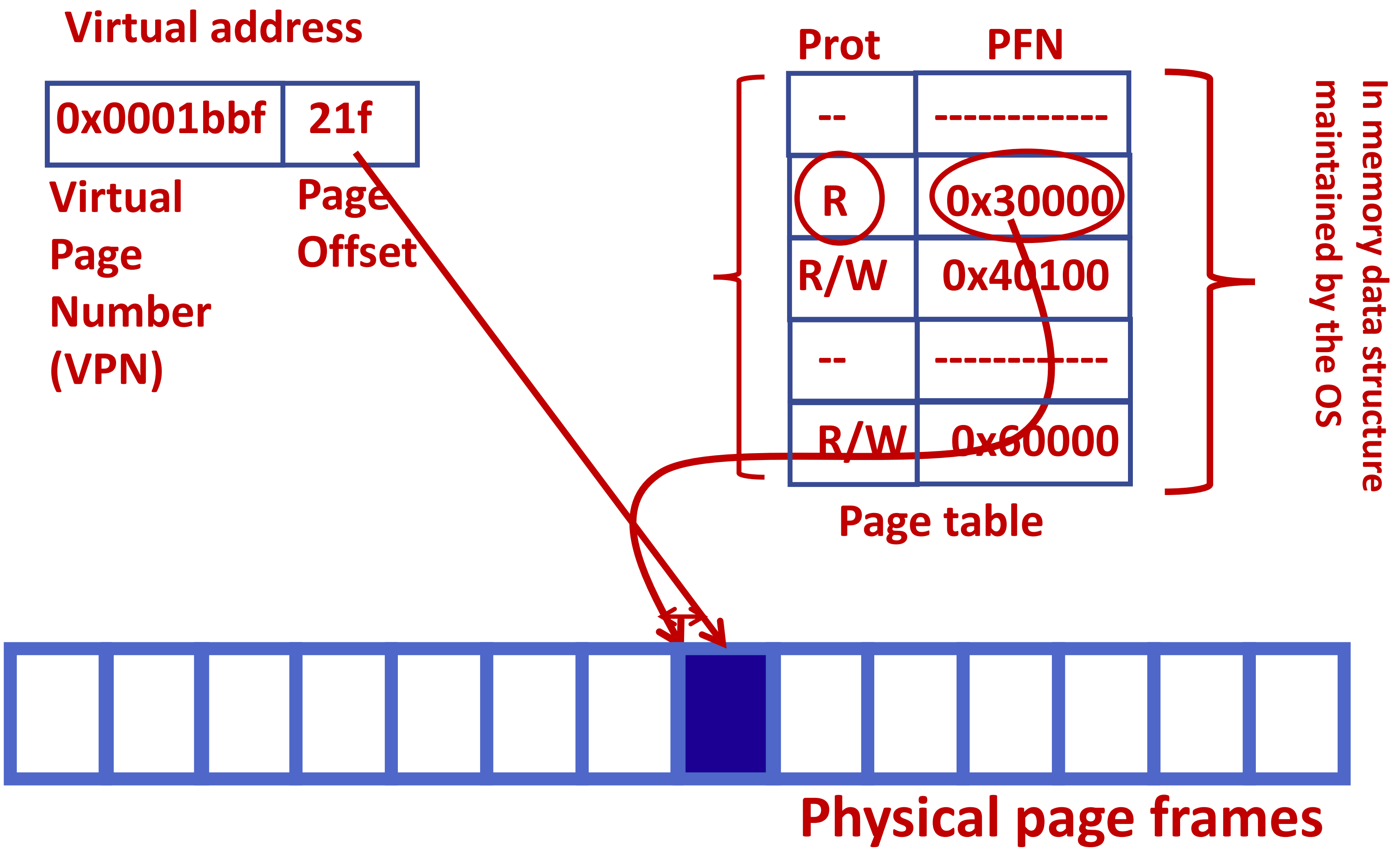
Prot	PFN
--	------------
R	0x30000
R/W	0x40100
--	------------
R/W	0x60000

Page table

In memory data structure maintained by the OS



Paging: How it works (simplistic)?



Paging: How it *really* works?

- Single level page table adds too much overhead
  - Consider a typical 48-bit virtual address space, 4KB page size and 8 byte long page table entry (PTE)
  - **Each** page table will be 512GB !
  - There could be many processes → many page tables

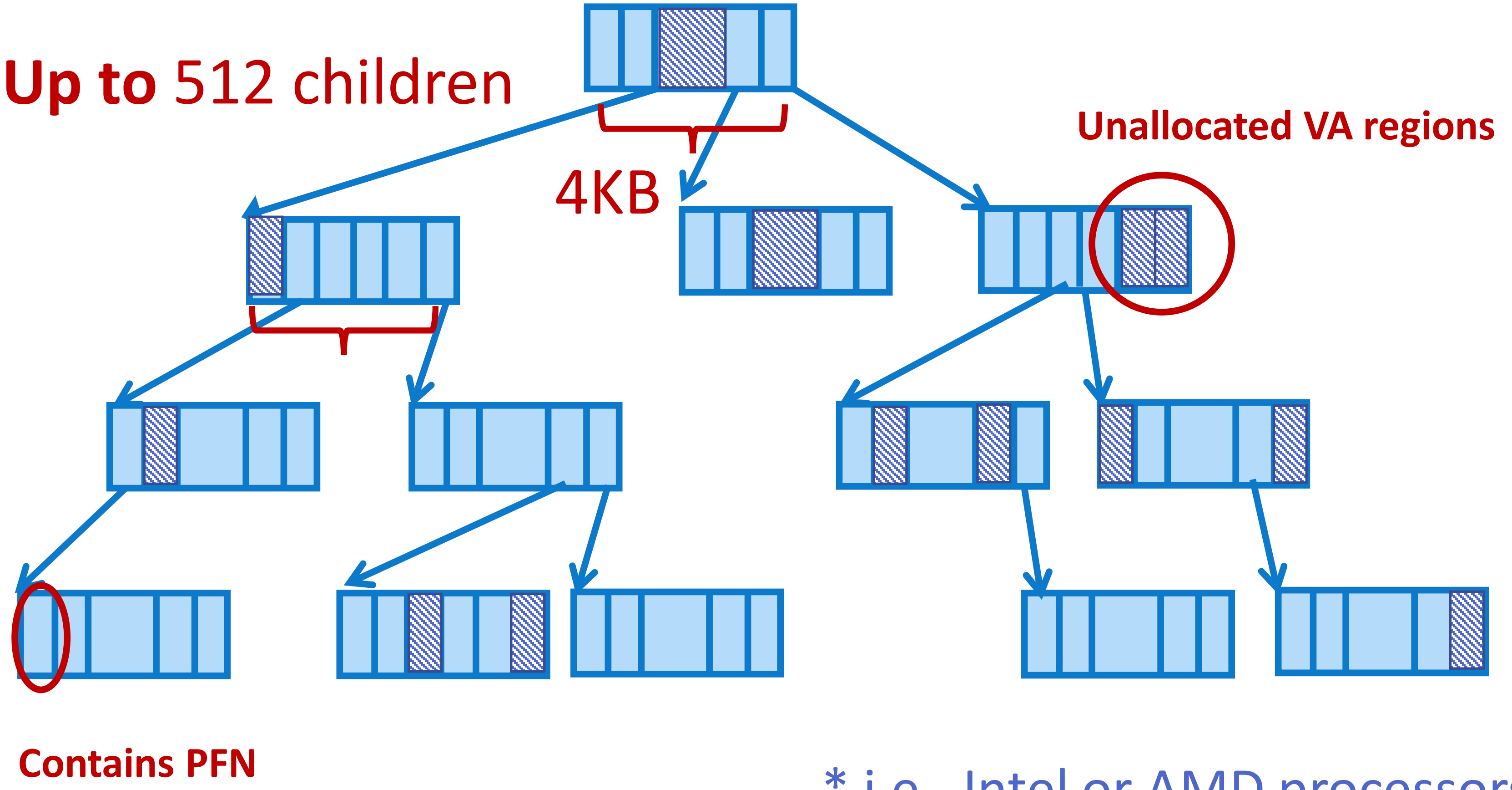
- Often virtual address space is sparsely allocated
  - Entire address space is typically not allocated

Paging: How it *really* works?

- Single level page table adds too much overhead
  - Consider a typical 48-bit virtual address space, 4KB page size and 8 byte long page table entry (PTE)
  - **Each** page table will be 512GB !
  - There could be many processes → many page tables
- Often virtual address space is sparsely allocated
  - Entire address space is typically not allocated
- Solution: **Multilevel radix tree** for page table

Paging: 64bit x86* page table

4 level 512-ary radix tree indexed by virtual page number
(latest x86-64 processors use 5-levels)



* i.e., Intel or AMD processors

Paging: 64bit x86 page table

- Operating system maintains one page table per process (a.k.a., per virtual address space)
  - Allocated in normal cacheable memory – just like any other data structure
- Operating system creates, updates, deletes page table entries (PTEs)
  - Creates on demand/request and/or on process creation

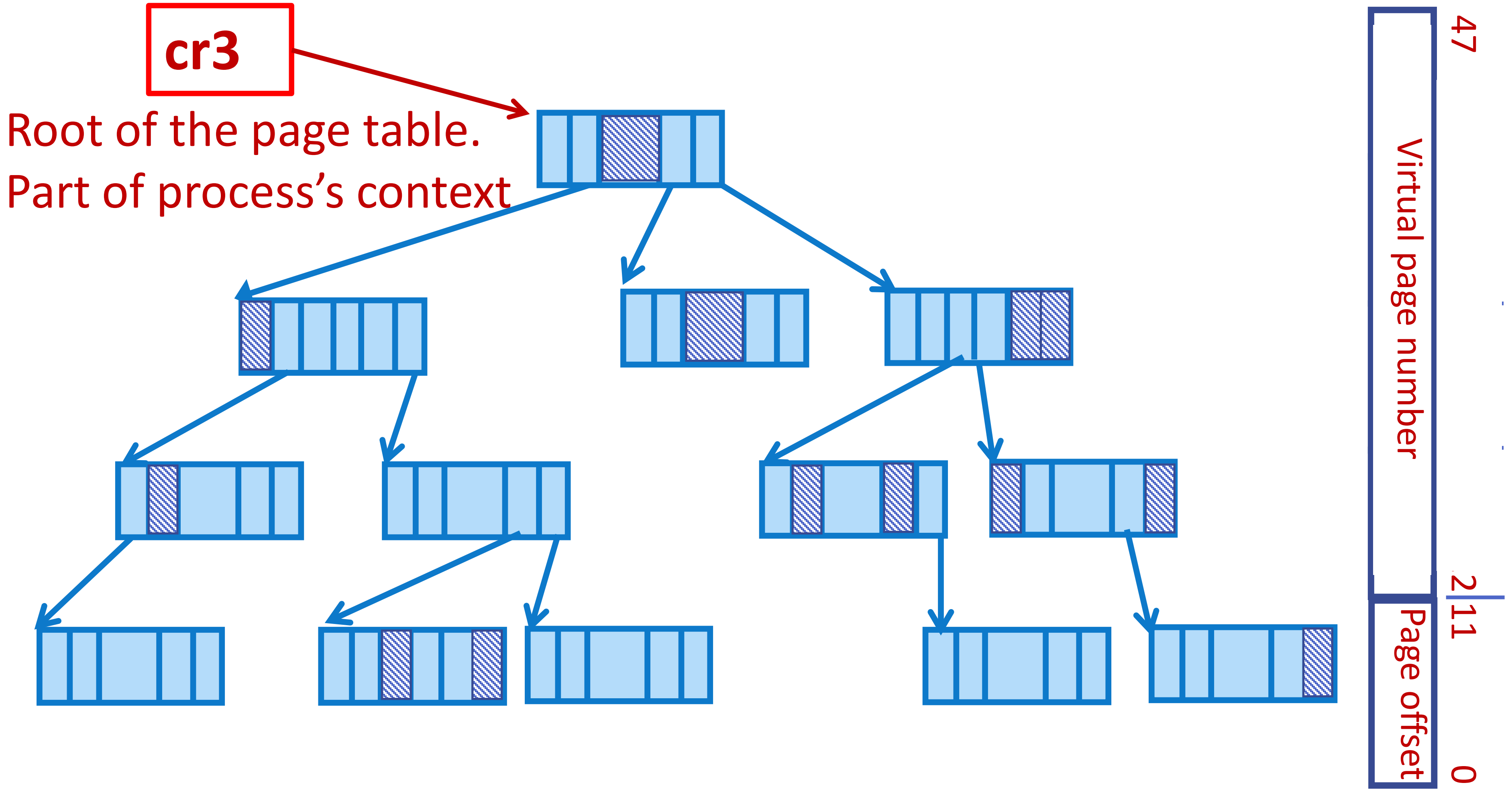
Paging: 64bit x86 page table

- Operating system maintains one page table per process (a.k.a., per virtual address space)
  - Allocated in normal cacheable memory – just like any other data structure
- Operating system creates, updates, deletes page table entries (PTEs)
  - Creates on demand/request and/or on process creation
- Page table structure is part of agreement between the OS and the hardware → page table structure is part of ISA (typically, e.g., x86 or ARM)

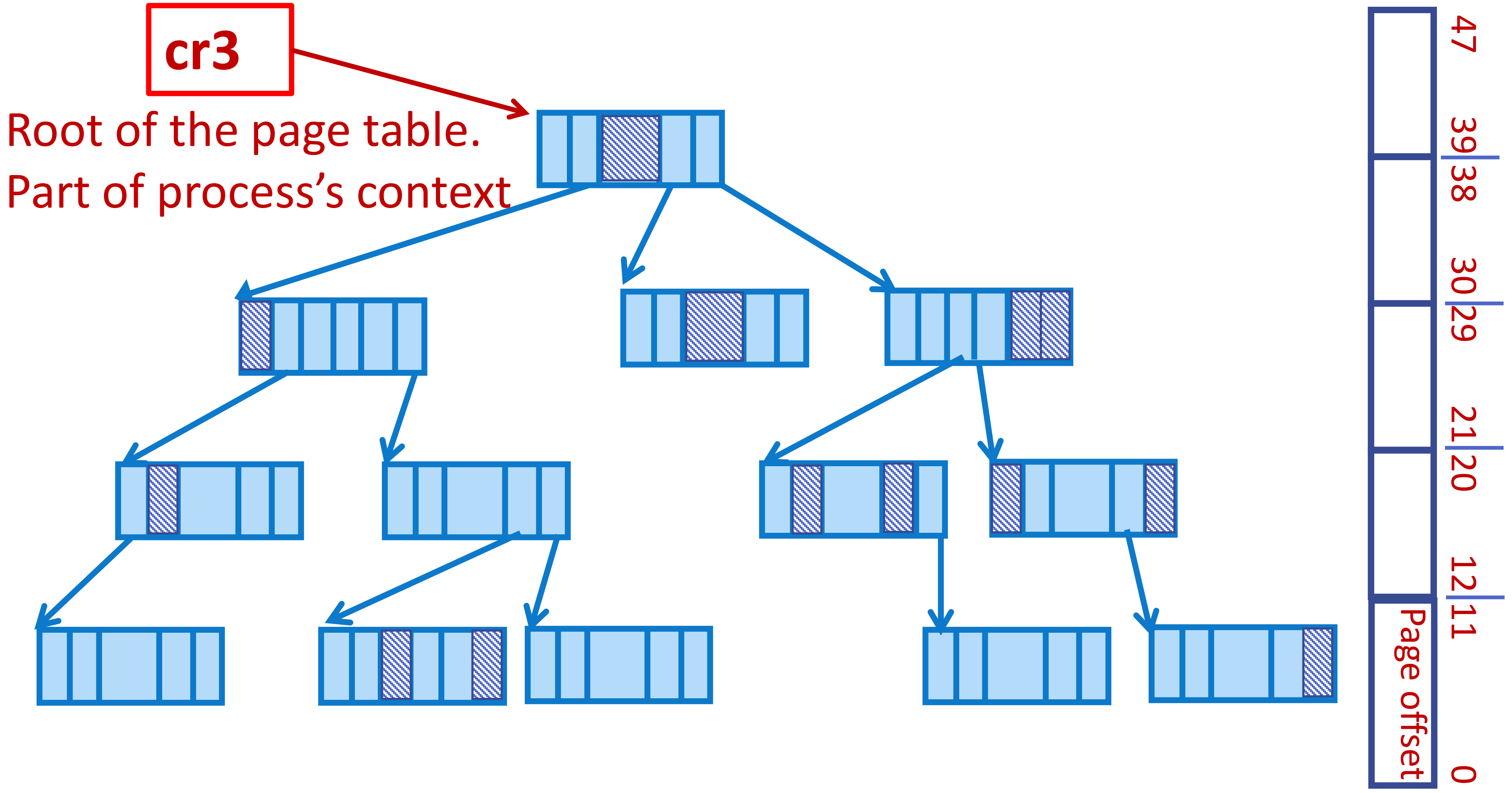
Paging: Virtual to physical address translation

- Called ***page table walk***.
  - Purpose: given a virtual address find the physical address?

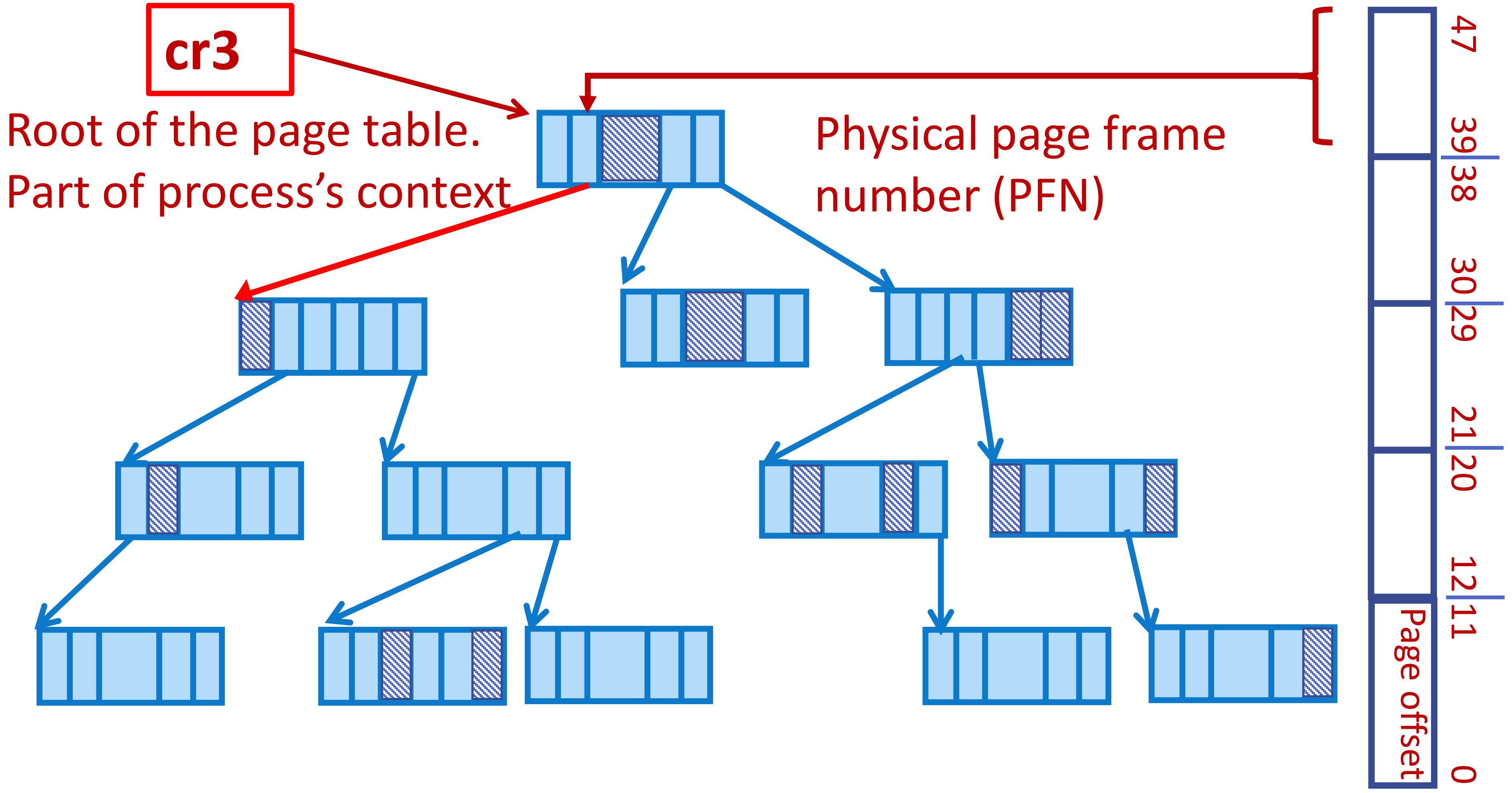
Paging: 64bit x86 page *walk*



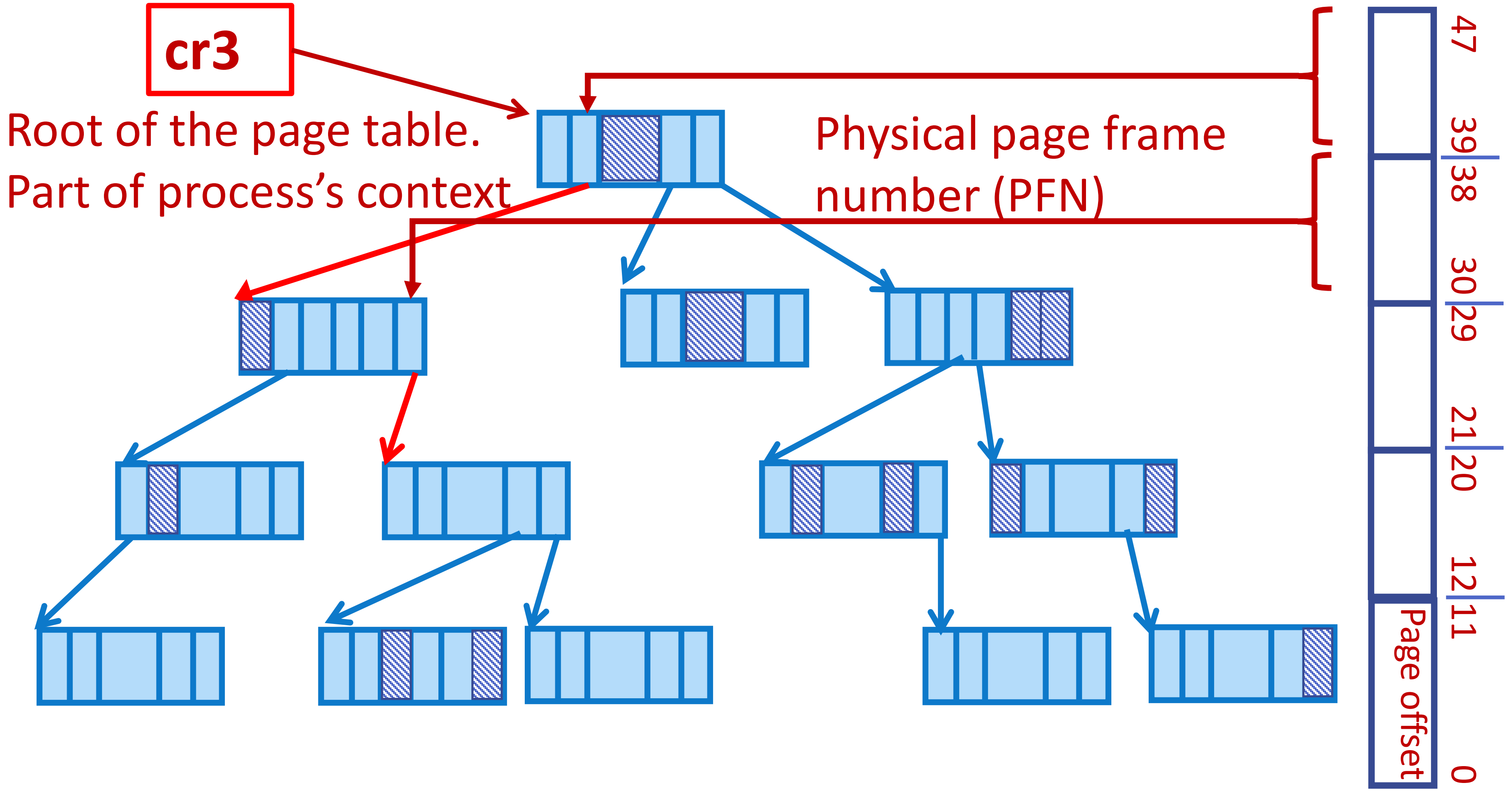
Paging: 64bit x86 page *walk*



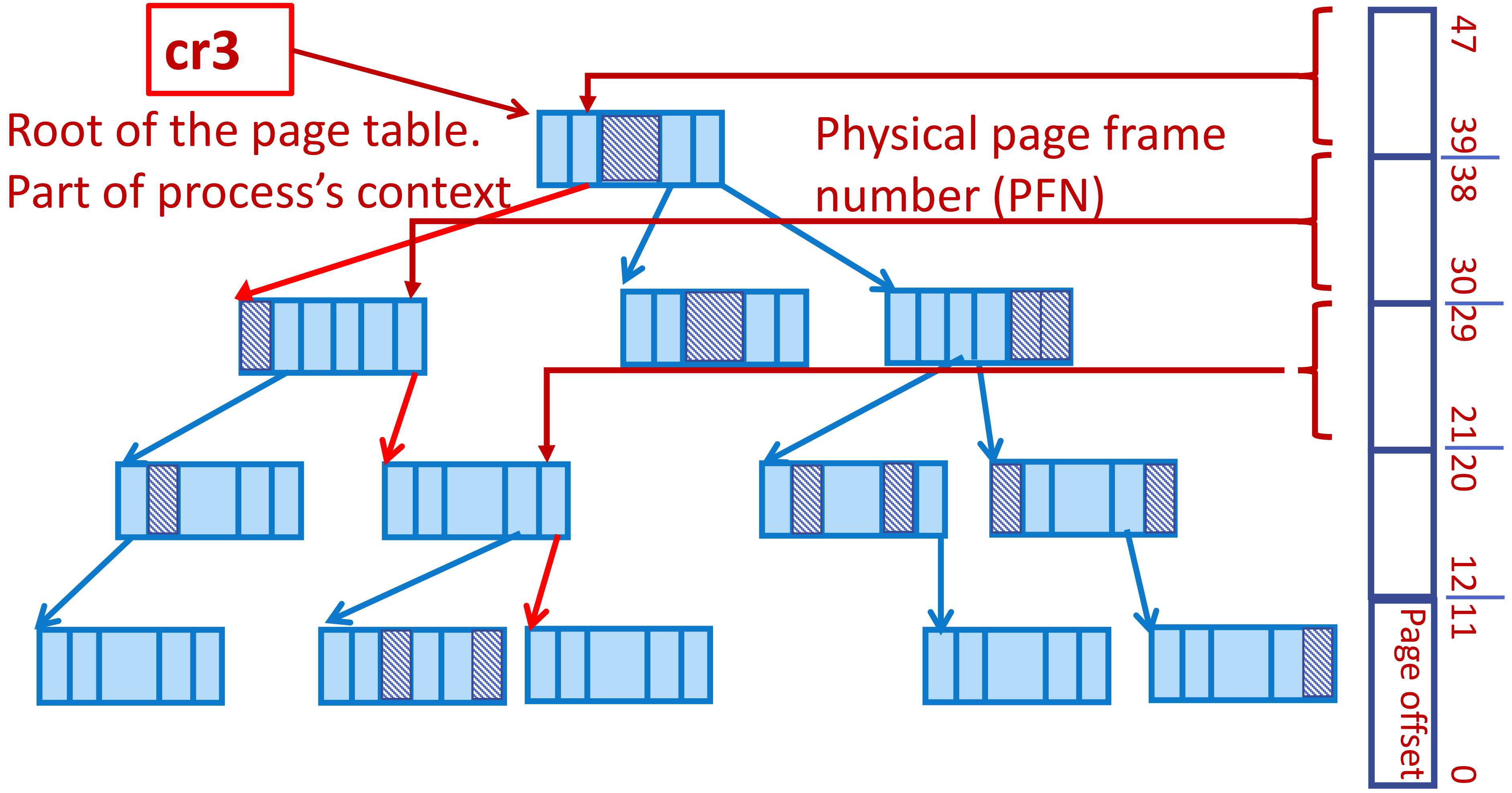
Paging: 64bit x86 page *walk*



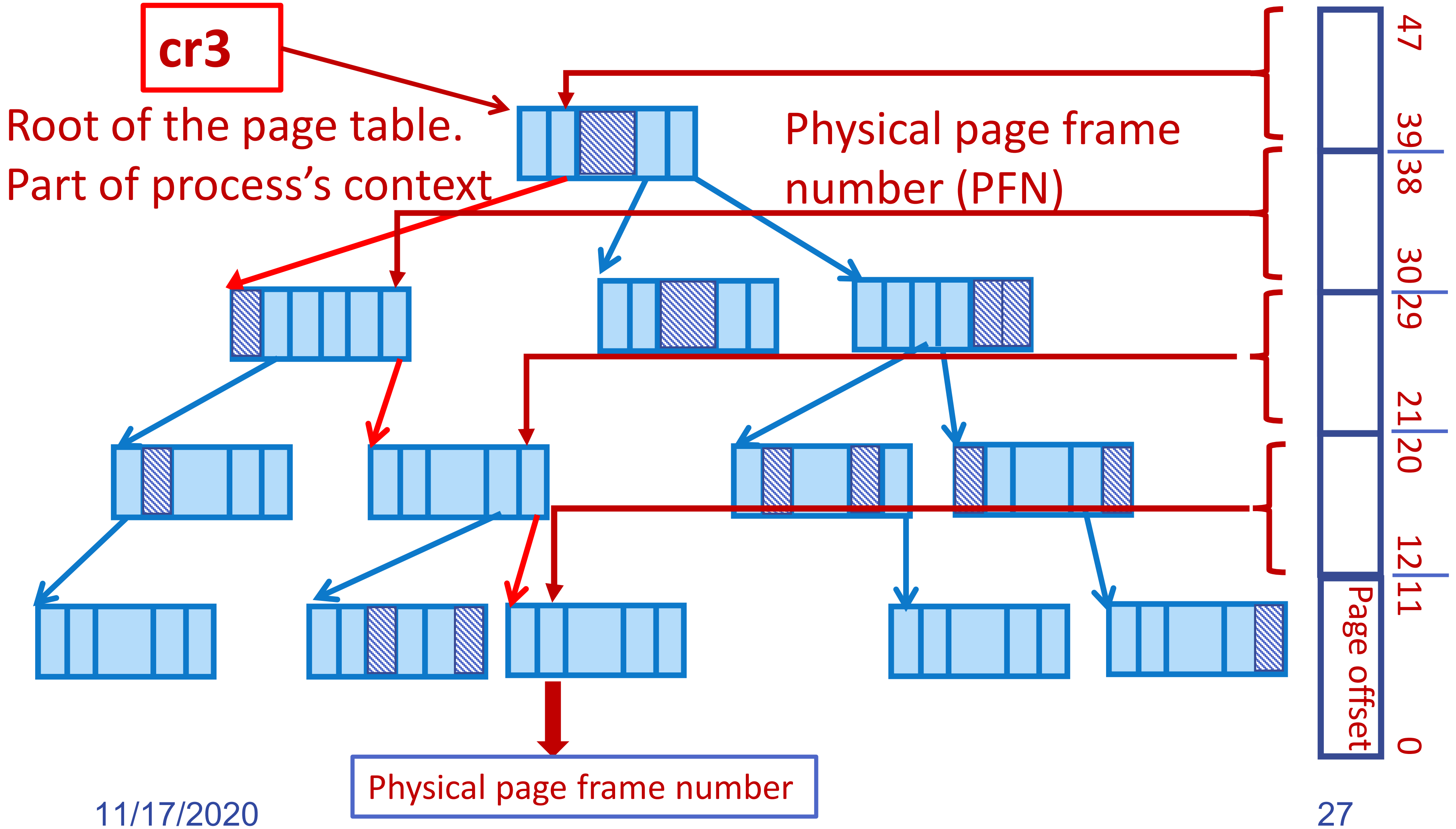
Paging: 64bit x86 page *walk*



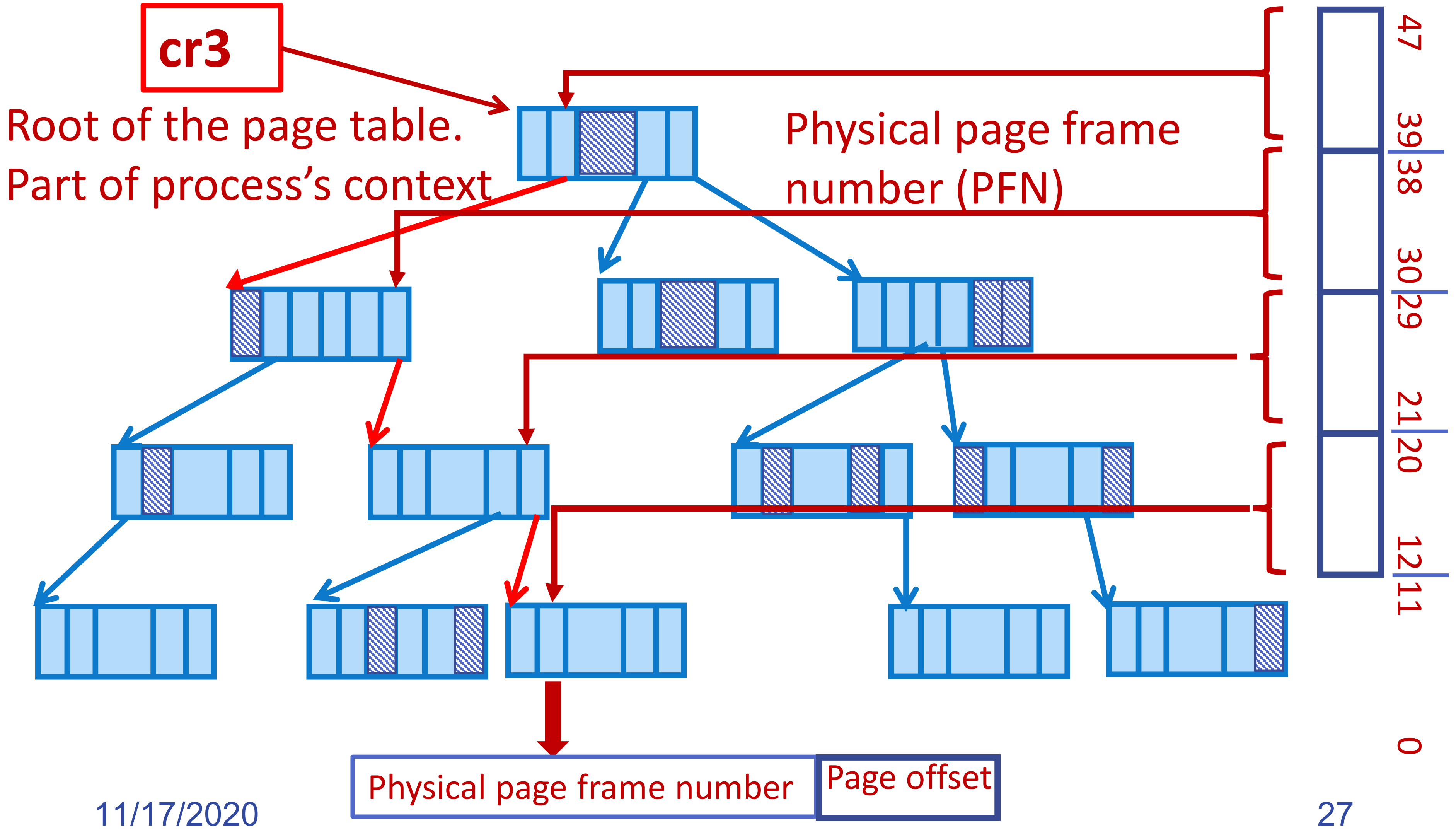
Paging: 64bit x86 page *walk*



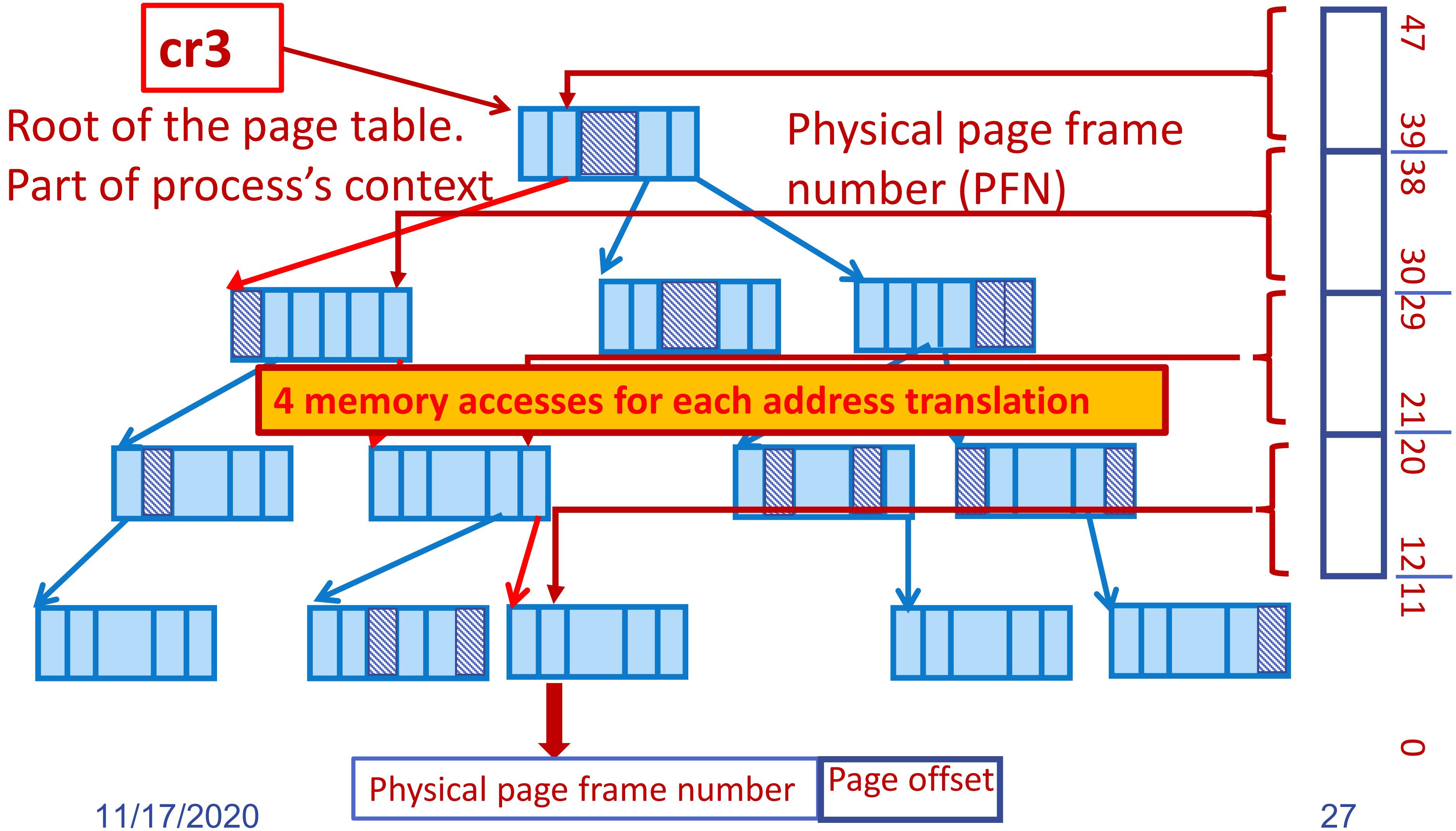
Paging: 64bit x86 page *walk*


cr3
Root of the page table.
Part of process's context
Physical page frame number (PFN)
47
39
38
30
29
21
20
12
11
0
Page offset
Physical page frame number

Paging: 64bit x86 page *walk*


cr3
Root of the page table.
Part of process's context
Physical page frame number (PFN)
47
39
38
30
29
21
20
12
11
0
Physical page frame number
Page offset

Paging: 64bit x86 page *walk*


cr3
Root of the page table.
Part of process's context
Physical page frame number (PFN)
4 memory accesses for each address translation
47
39
38
30
29
21
20
12
11
0
Physical page frame number
Page offset

Who walks the page table?

- A hardware page table walker (PTW) walks the page table (e.g., in Intel, AMD or ARM processors)
  - Input: Root of page table (cr3) and VPN
  - Output: Physical page frame number or a page fault
  - A hardware fine-state-automata in each CPU core
  - Generates **load-like "instructions"** to access addresses containing the memory holding desired page table entries

Who walks the page table?

- A hardware page table walker (PTW) walks the page table (e.g., in Intel, AMD or ARM processors)
  - Input: Root of page table (cr3) and VPN
  - Output: Physical page frame number or a page fault
  - A hardware fine-state-automata in each CPU core
  - Generates **load-like "instructions"** to access addresses containing the memory holding desired page table entries
- Alternatives: Software page table walker
  - A OS handler walks the page table

Who walks the page table?

- A hardware page table walker (PTW) walks the page table (e.g., in Intel, AMD or ARM processors)
    - Input: Root of page table (cr3) and VPN
    - Output: Physical page frame number or a page fault
    - A hardware fine-state-automata in each CPU core
    - Generates **load-like "instructions"** to access addresses containing the memory holding desired page table entries
- Alternatives: Software page table walker
    - A OS handler walks the page table
    - Advantage: Free to choose page table format
    - Disadvantage: Slow → Large address translation overhead
    - Example: SPARC (Sun/Oracle) machines

TLB: Making page walks faster

- **Disadvantage:** A single address translation can take **4 memory accesses**!
  - To access one byte, 4 memory accesses needed for address translation

TLB: Making page walks faster

- **Disadvantage:** A single address translation can take **4 memory accesses**!
  - To access one byte, 4 memory accesses needed for address translation
- How to make address translation fast?
  - **Translation Lookaside Buffer** or TLB to cache recently used virtual to physical address mappings

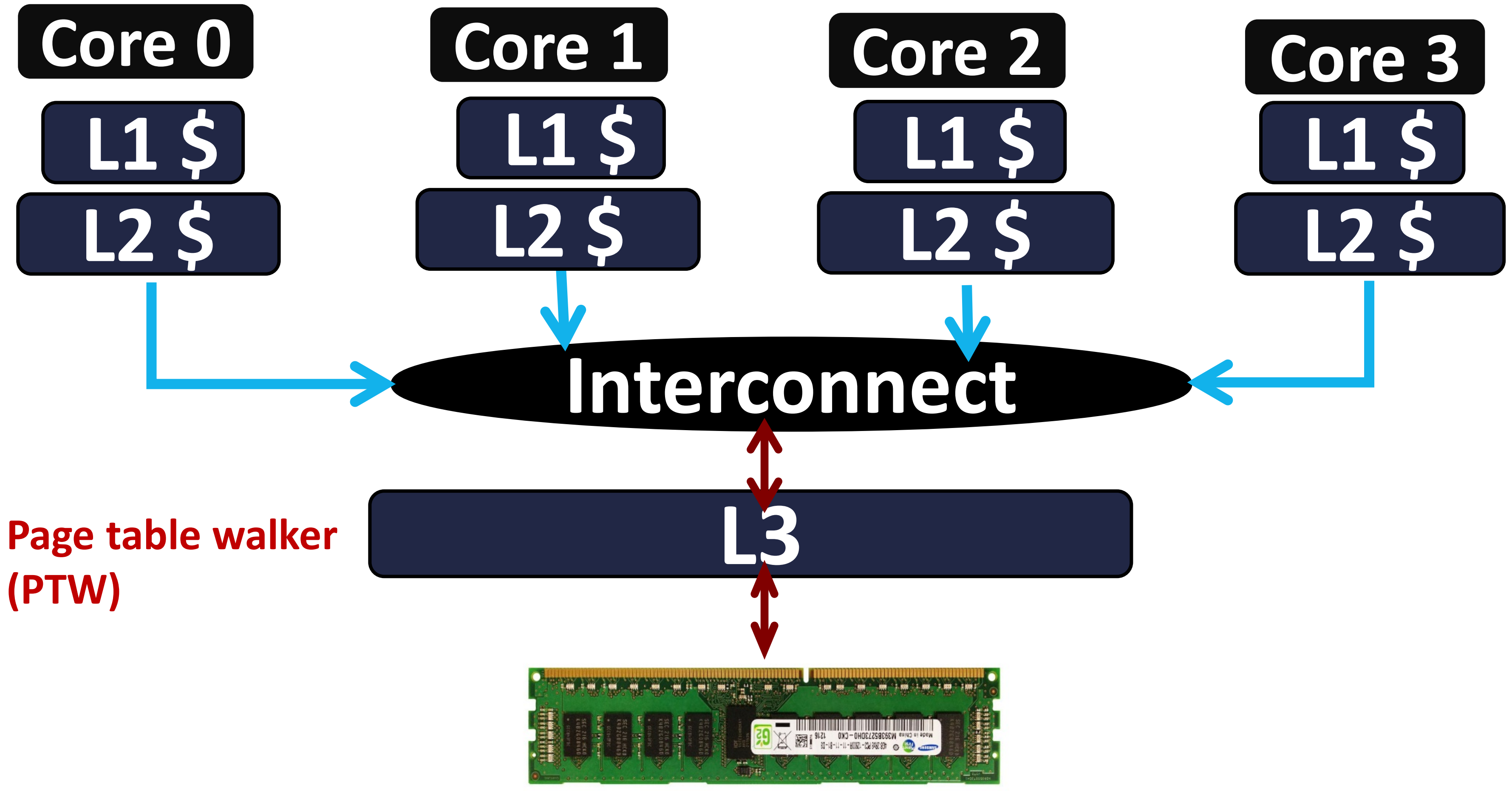
TLB: Making page walks faster

- **Disadvantage:** A single address translation can take **4 memory accesses**!
  - To access one byte, 4 memory accesses needed for address translation
- How to make address translation fast?
  - **Translation Lookaside Buffer** or TLB to cache recently used virtual to physical address mappings
  - A read-only cache of contents of page table entries
  - For address translation of every load/store, TLB is first looked up
  - On a TLB miss page walk is performed
  - A TLB hit is fast but page walk is slow

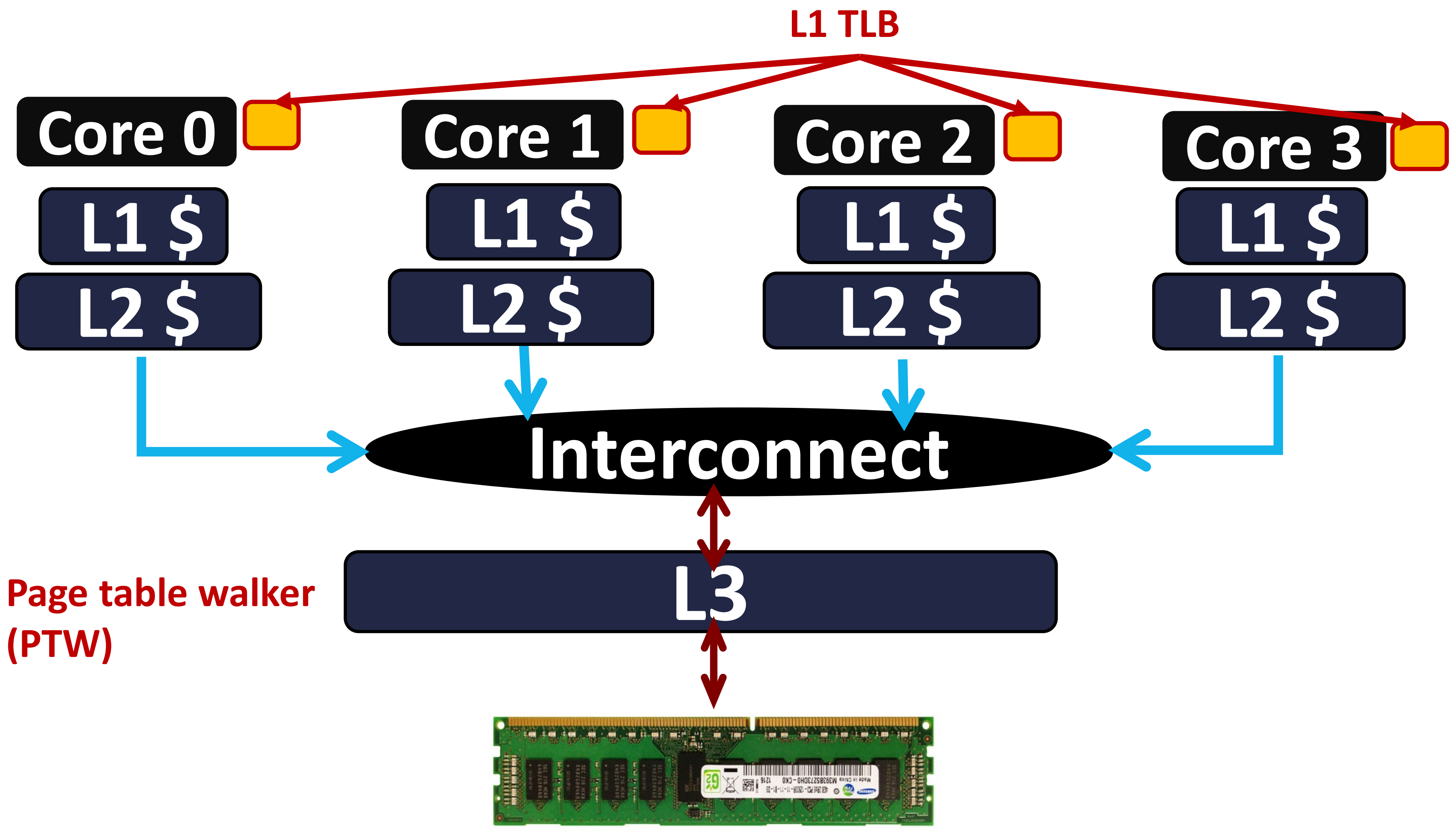
Typical TLB hierarchy of modern processors

- **Each core** typically has:
  - 32-64-entry L1 TLB (fully associative/4-8 way set-associative)
  - 1500-2500 entry L2 TLB (8-16 way set-associative)
  - Typically one page table walker (but latest Intel processor have two of them per core)
- Note: entire TLB hierarchy is private to a core (unlike cache subsystem)
  - Why?

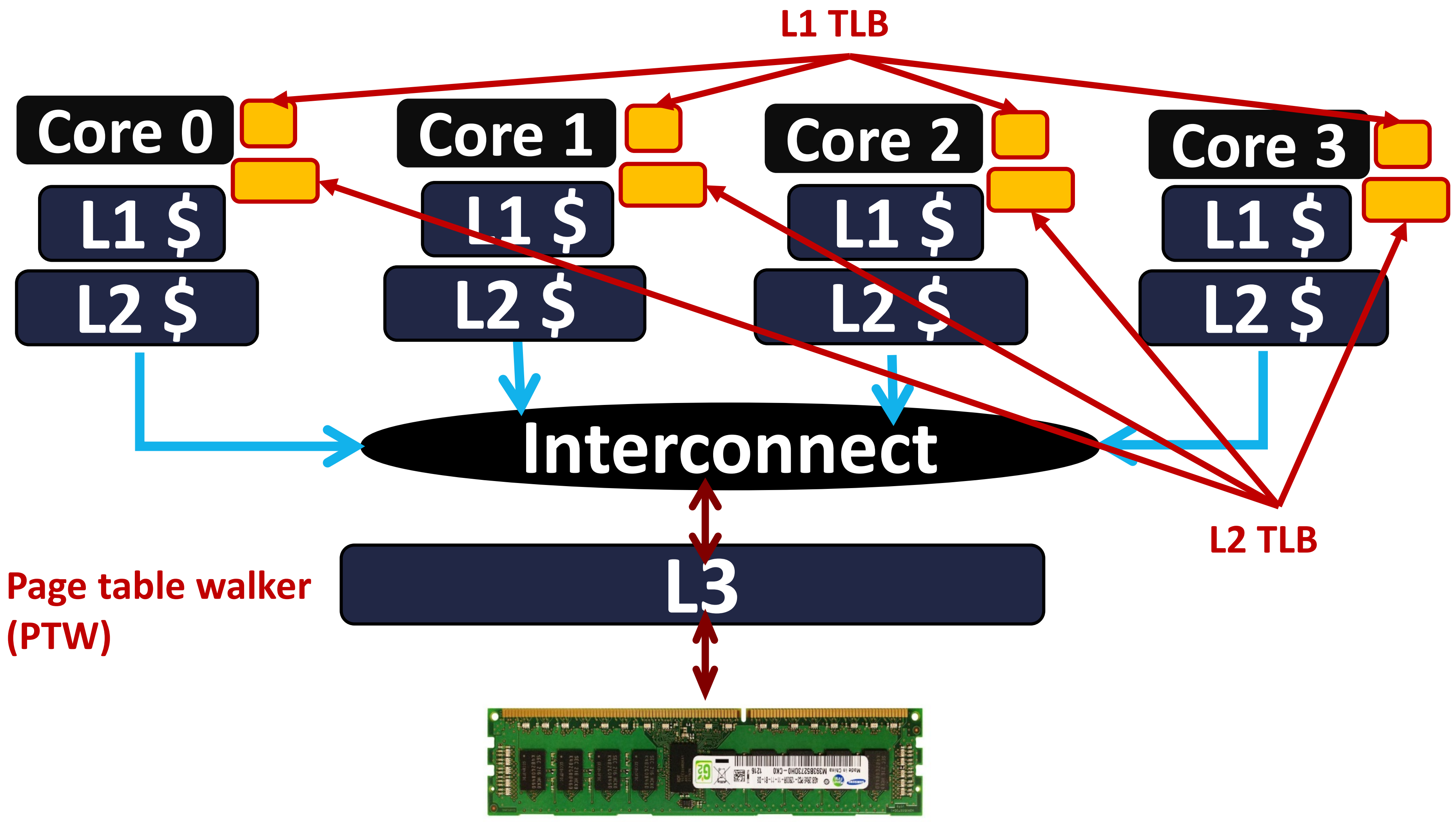
Where are TLBs & Page table walkers?



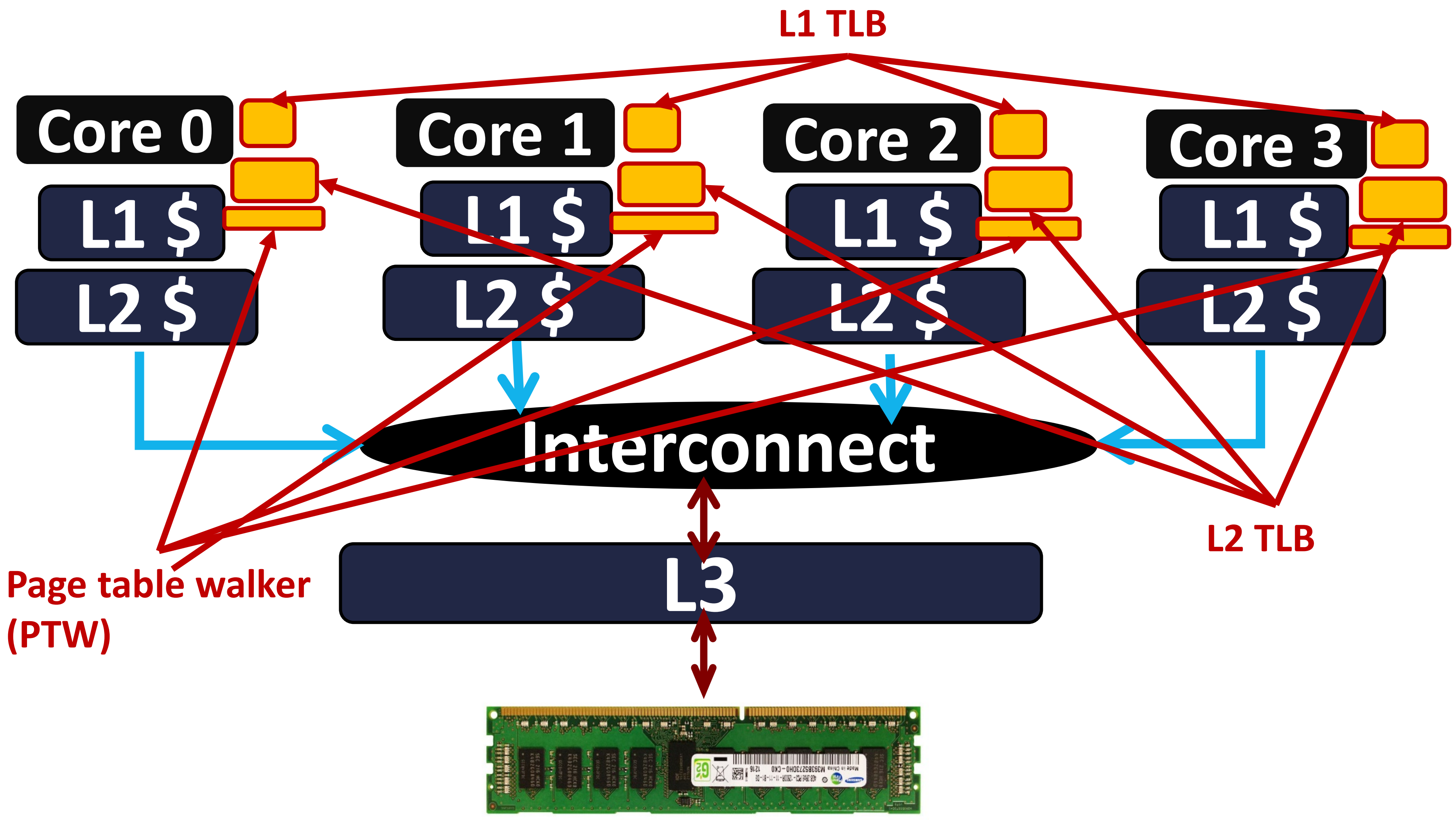
Where are TLBs & Page table walkers?



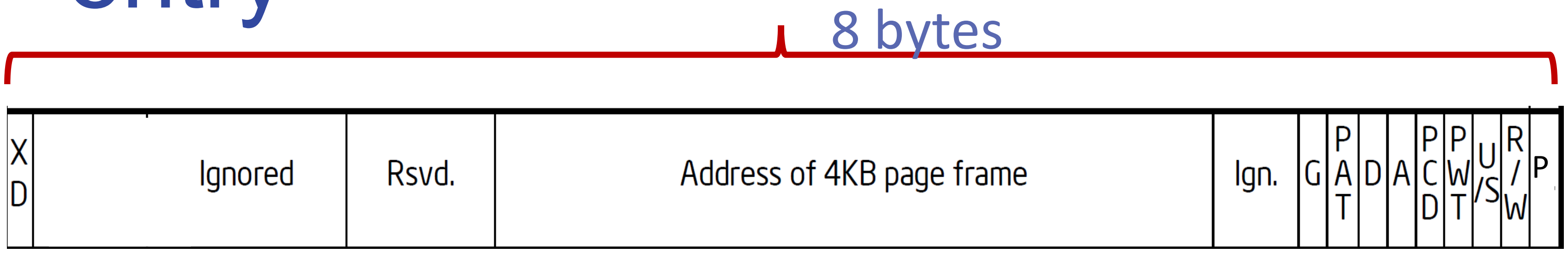
Where are TLBs & Page table walkers?



Where are TLBs & Page table walkers?



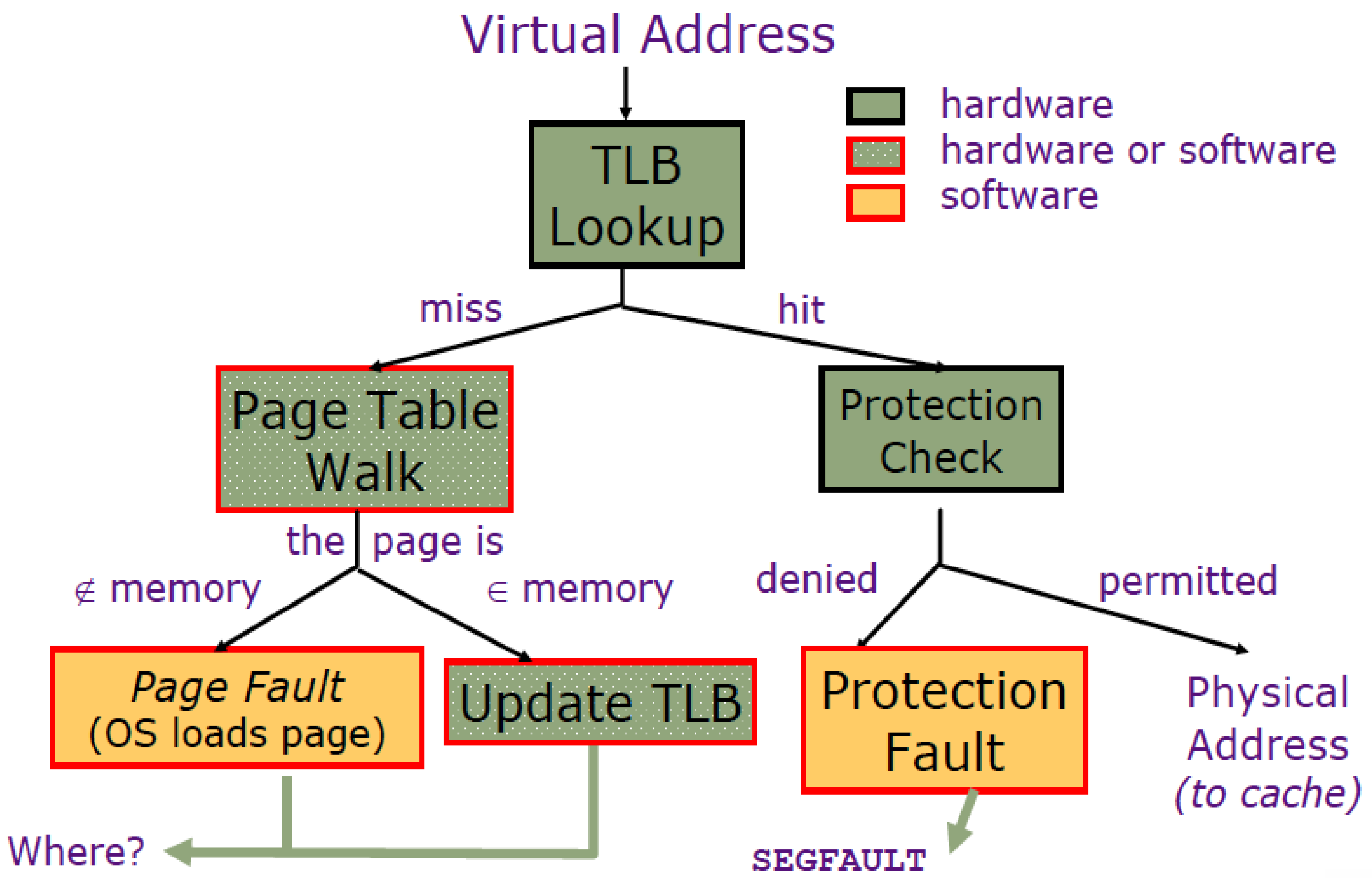
Contents of typical page table entry



Typical x86-64 PTE

- ‘P’ (Present bit): If the address present in memory
- ‘U/S’ (User/Supervisor bit): Is the page accessible in supervisor mode (e.g., by OS) only?
- ‘R/W’ (Read/Write): Is the page read-only?
- A (Access bit): Is this page has ever been accessed (load/stored to)?
- D (Dirty bit): Is the page has been written to?
- X/D (Executable bit): Does the page contains executable?

Putting it all Together


Virtual Address
hardware
hardware or software
software
TLB Lookup
miss
hit
Page Table Walk
Protection Check
the page is
∉ memory
∈ memory
denied
permitted
Page Fault (OS loads page)
Update TLB
Protection Fault
Physical Address *(to cache)*
Where?
SEGFAULT