Virtual Memory

Arkaprava Basu

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

Memory is virtual !

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

    Virtual address
  - Load, store instructions carries virtual addresses

Memory is virtual !

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

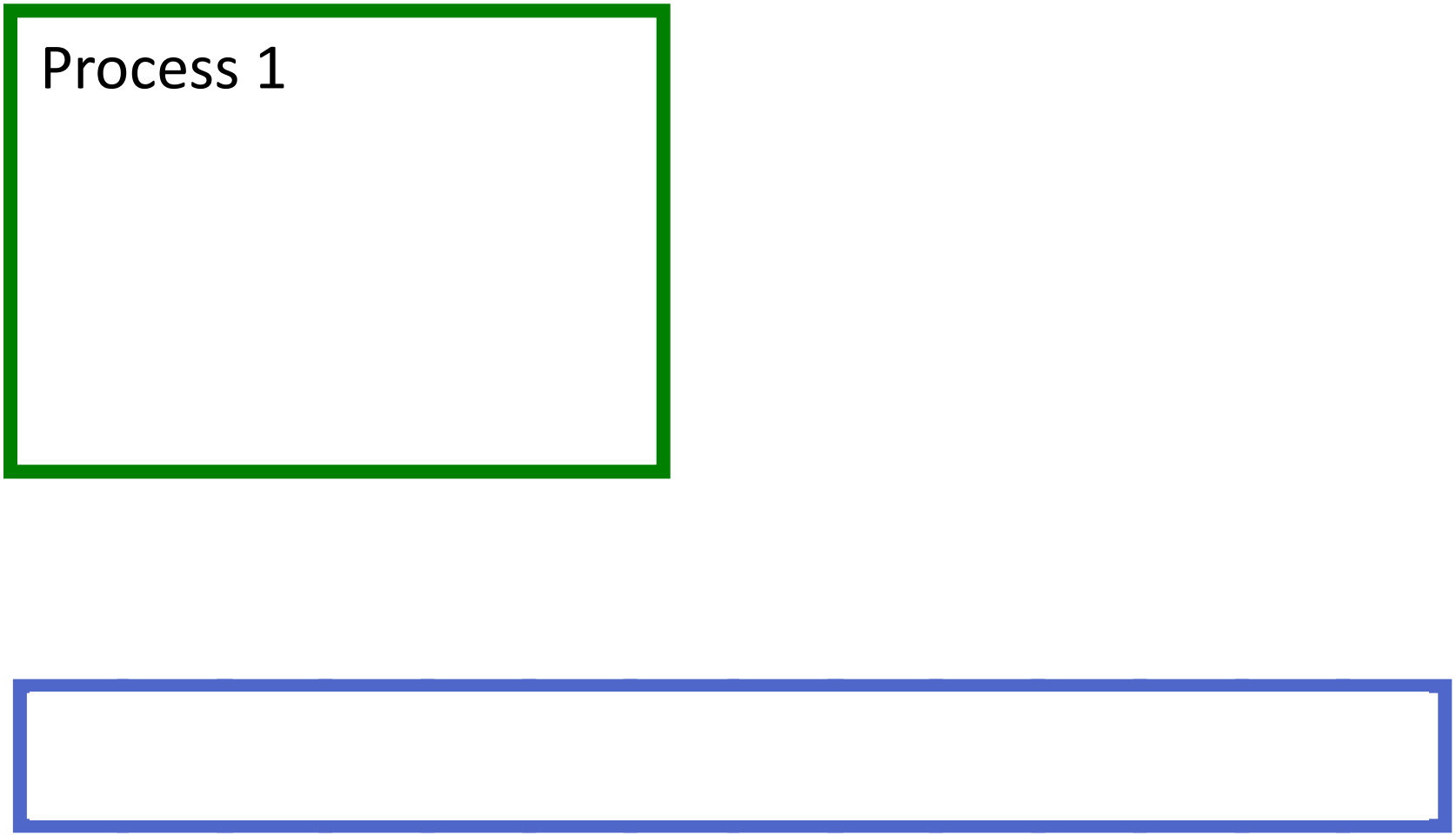
    Virtual address
  - Load, store instructions carries virtual addresses
- Hardware uses (real) **physical** address
  - e.g., to find data, lookup caches etc.
- When an application executes (a.k.a process) virtual addresses are translated to physical addresses, at runtime

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

Process 1

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

Physical Memory (e.g., DRAM)

Picture credit: Nima Honarmand, Stony Brook university

Bird's view of virtual memory

Process 1

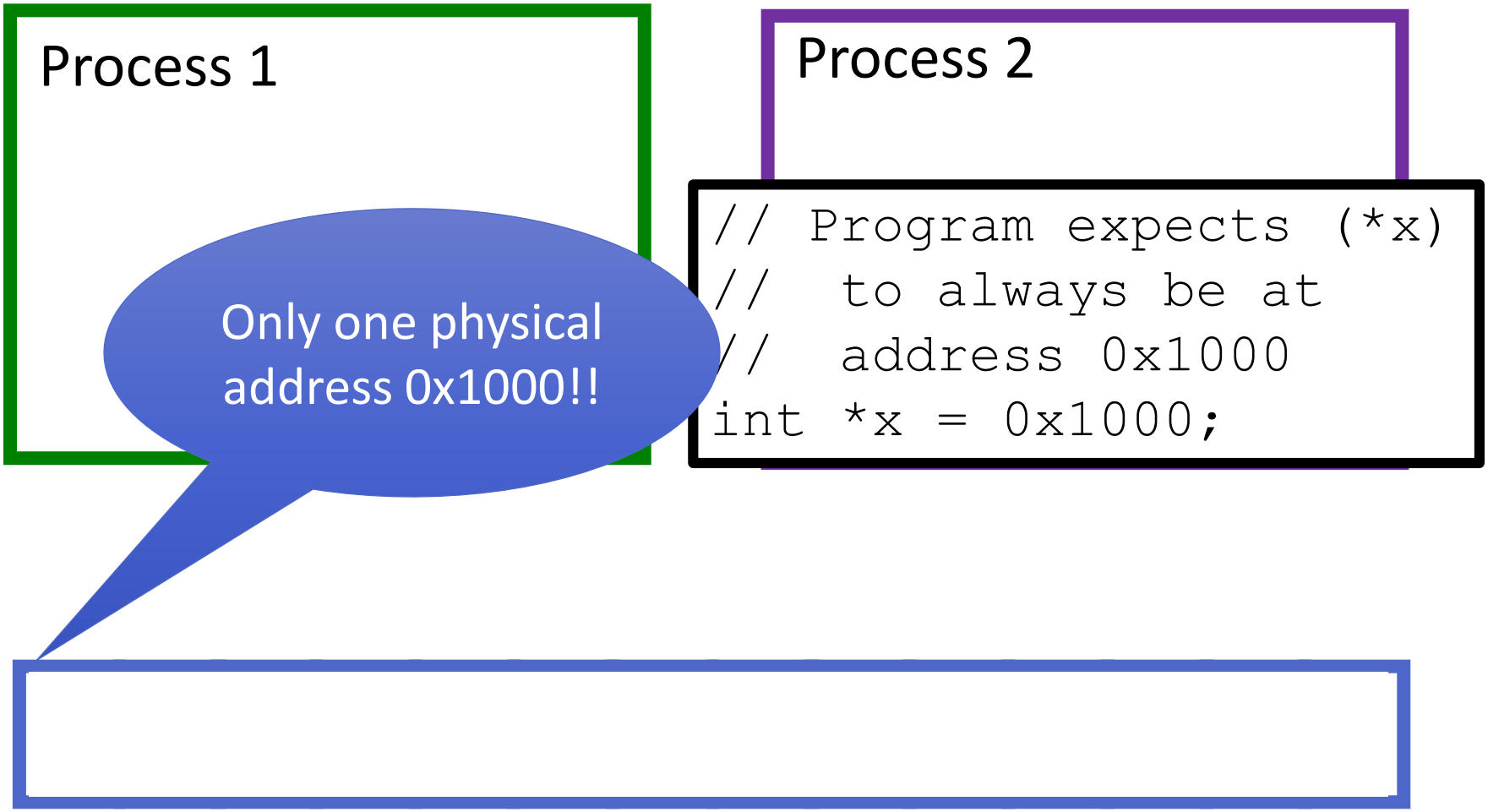
Process 2

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

Physical Memory (e.g., DRAM)

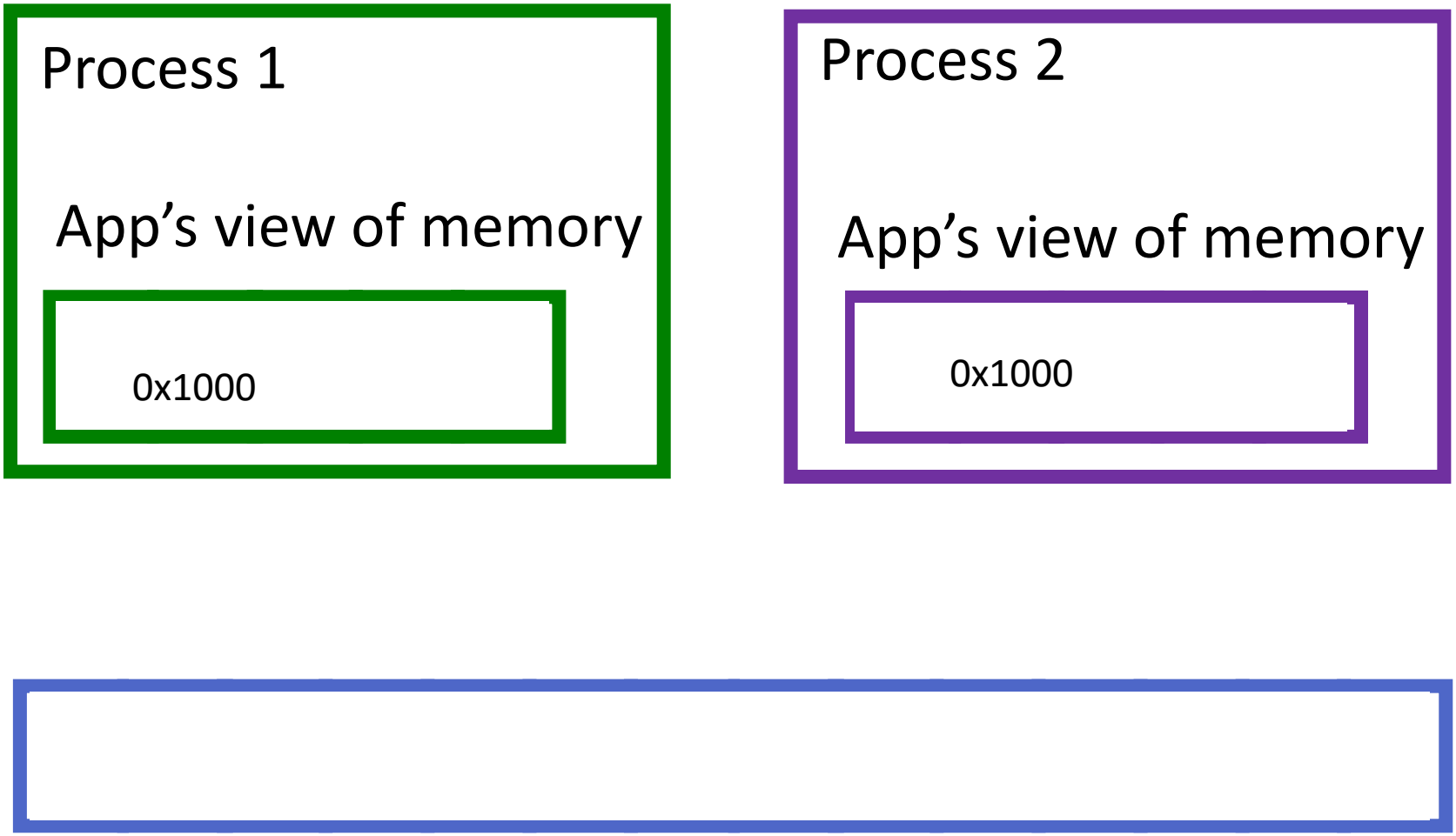
Picture credit: Nima Honarmand, Stony Brook university

Bird's view of virtual memory

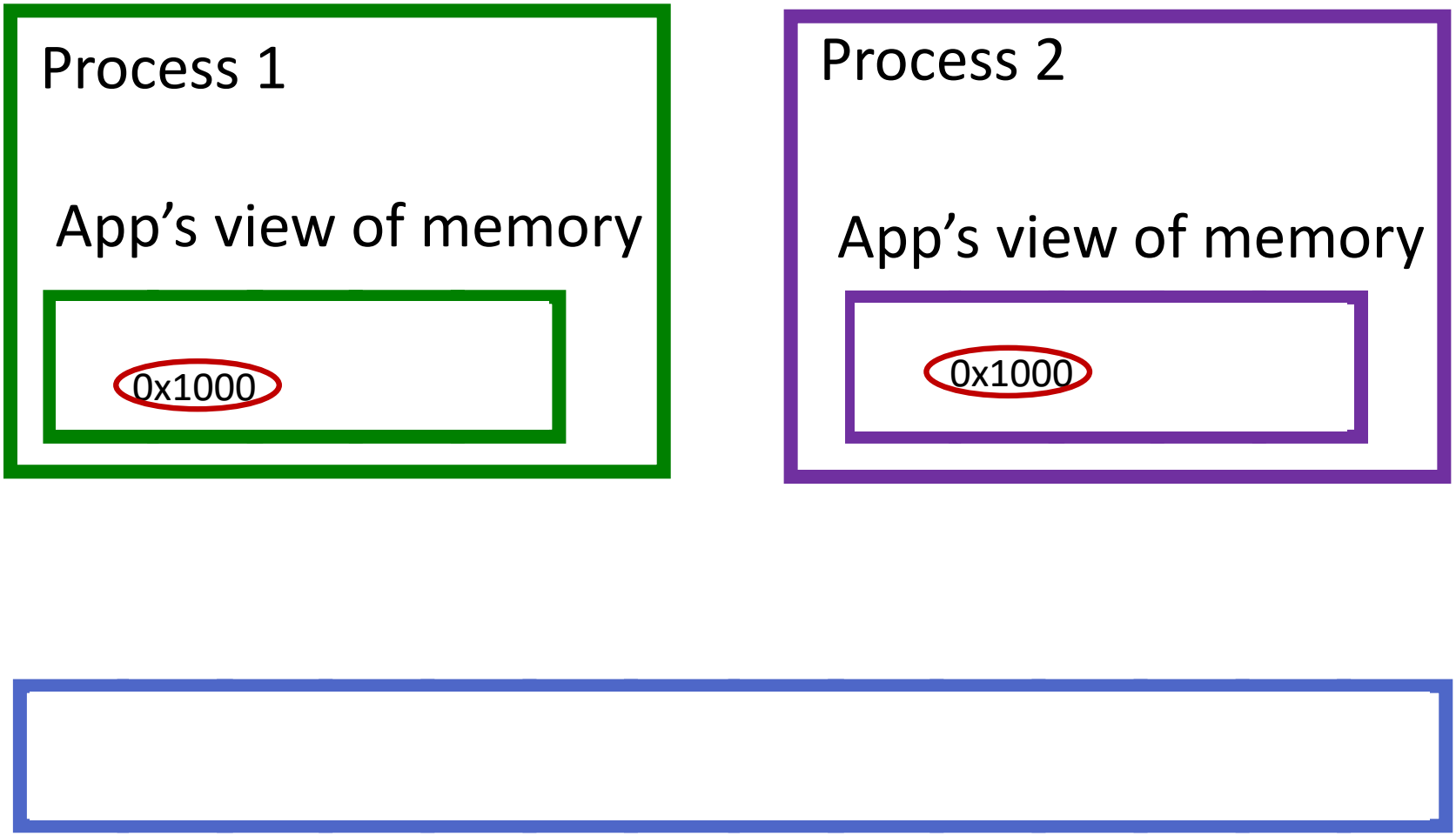


Picture credit: Nima Honarmand, Stony Brook university

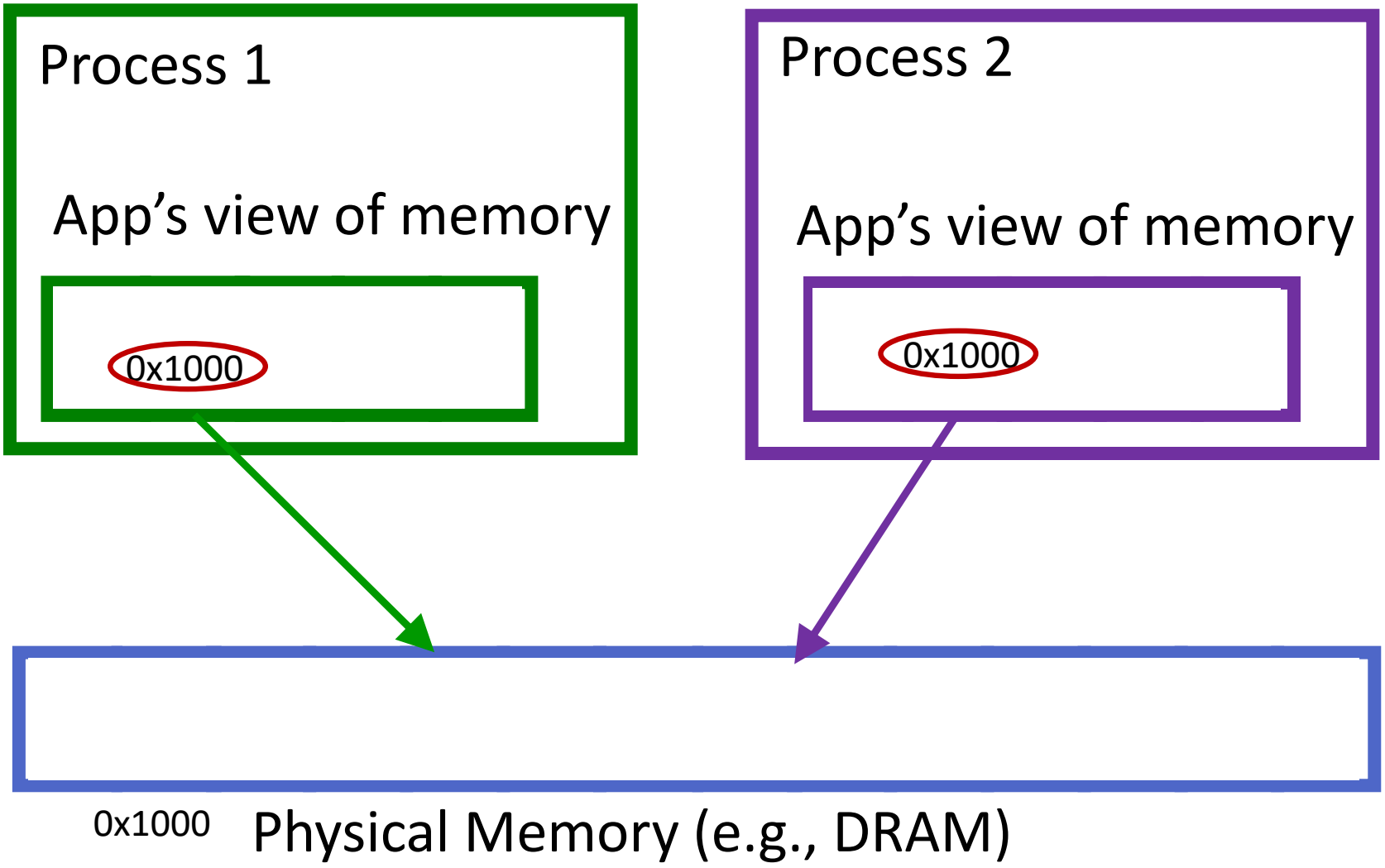
Bird's view of virtual memory



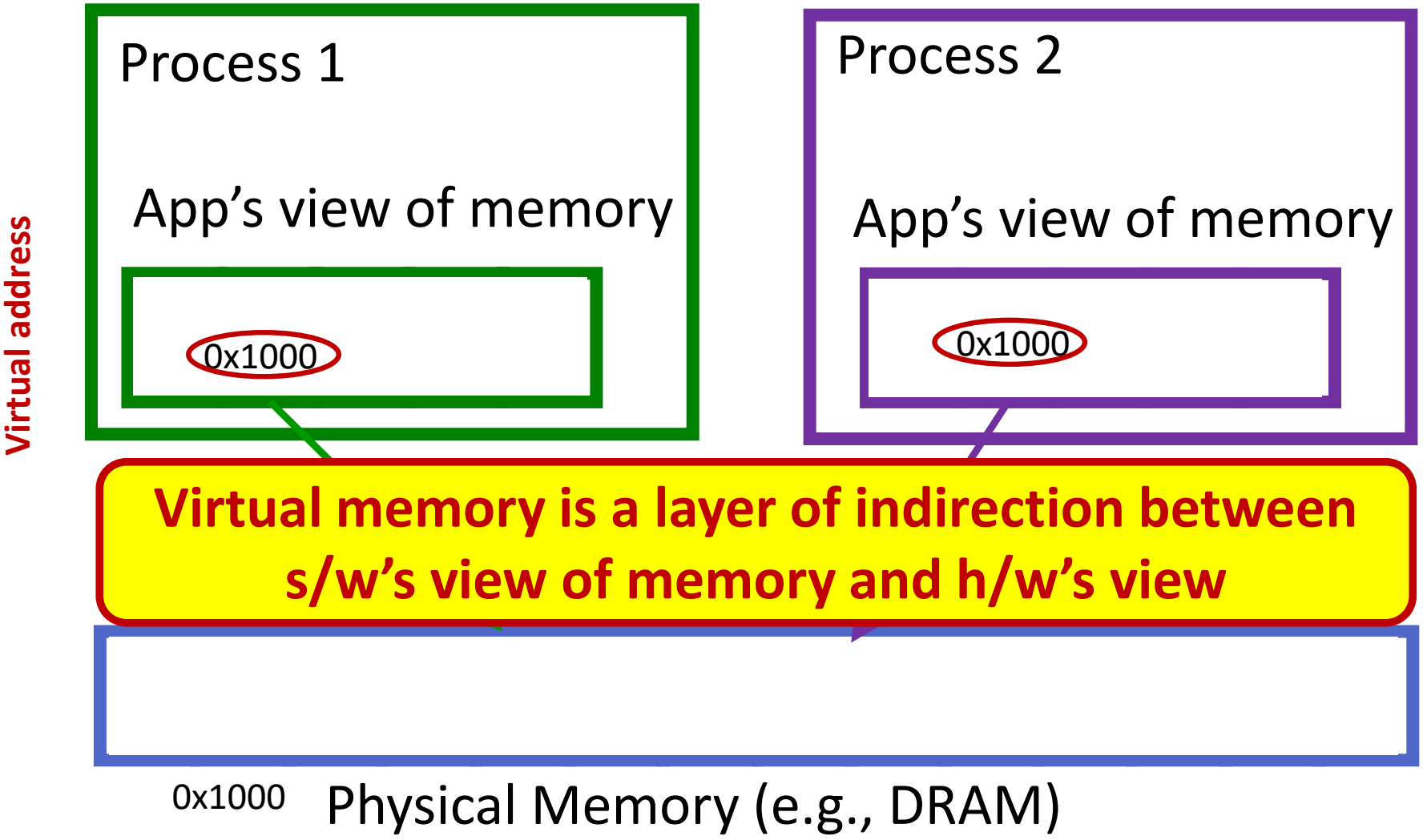
Bird's view of virtual memory



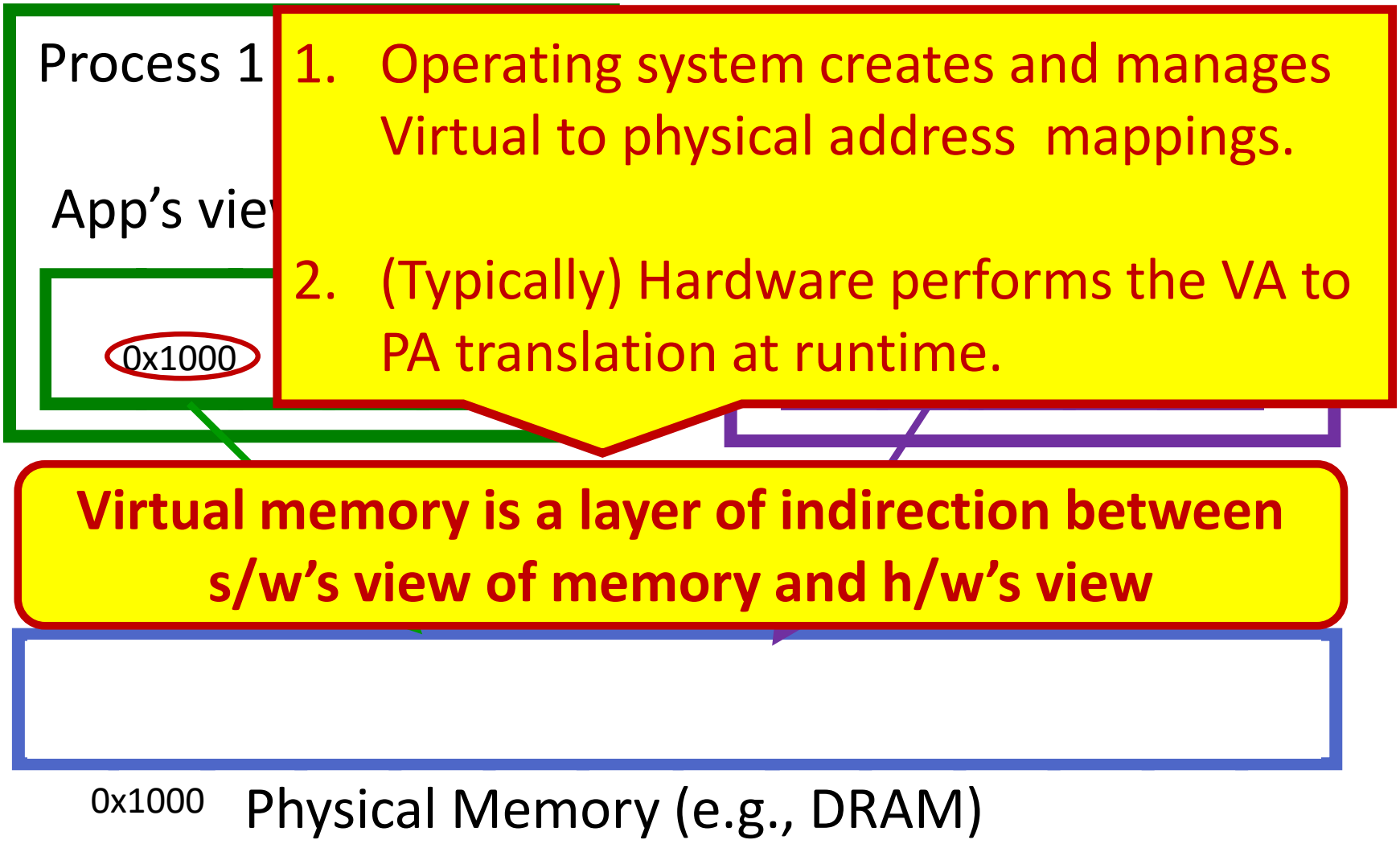
Bird's view of virtual memory



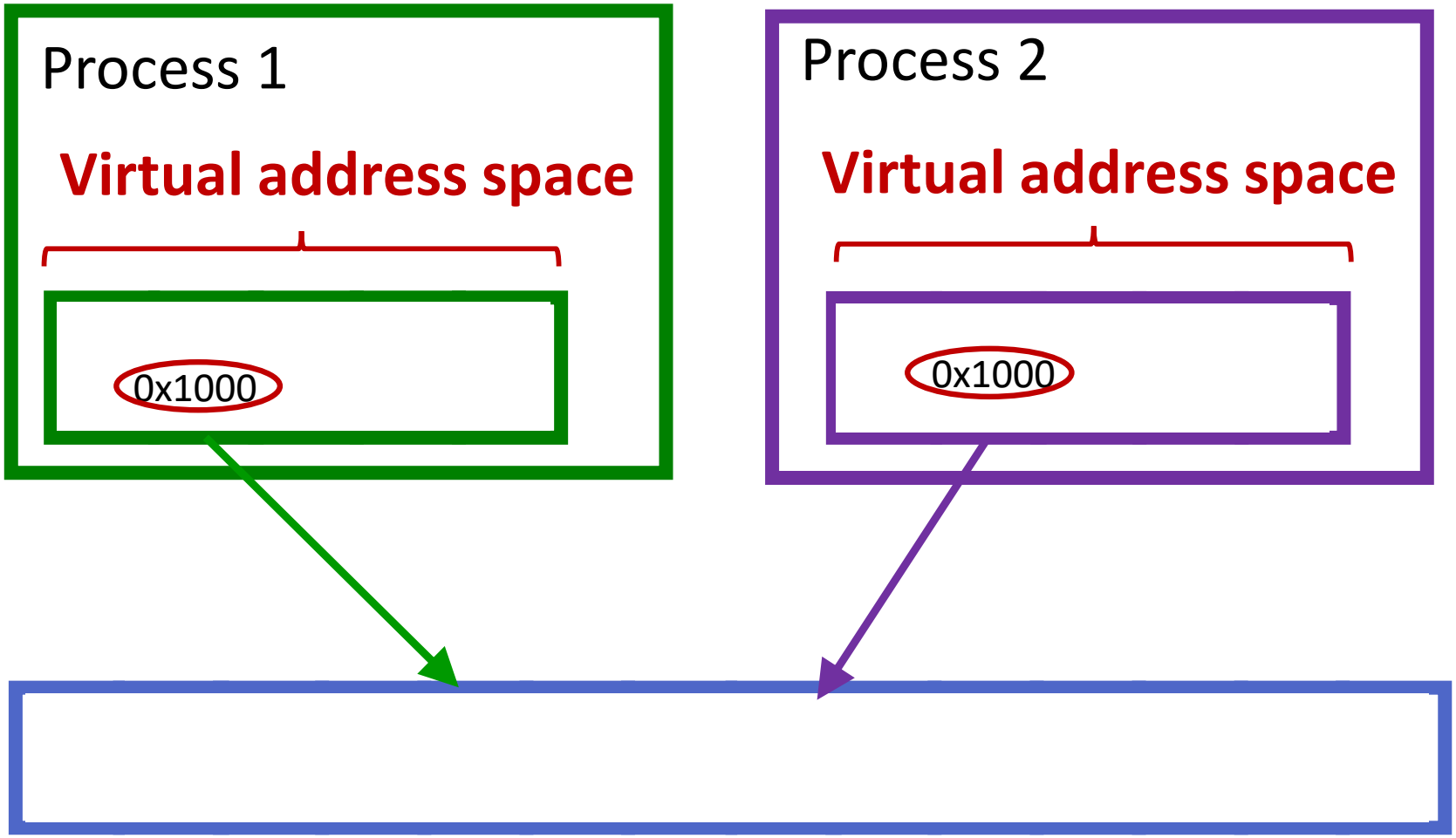
Bird's view of virtual memory



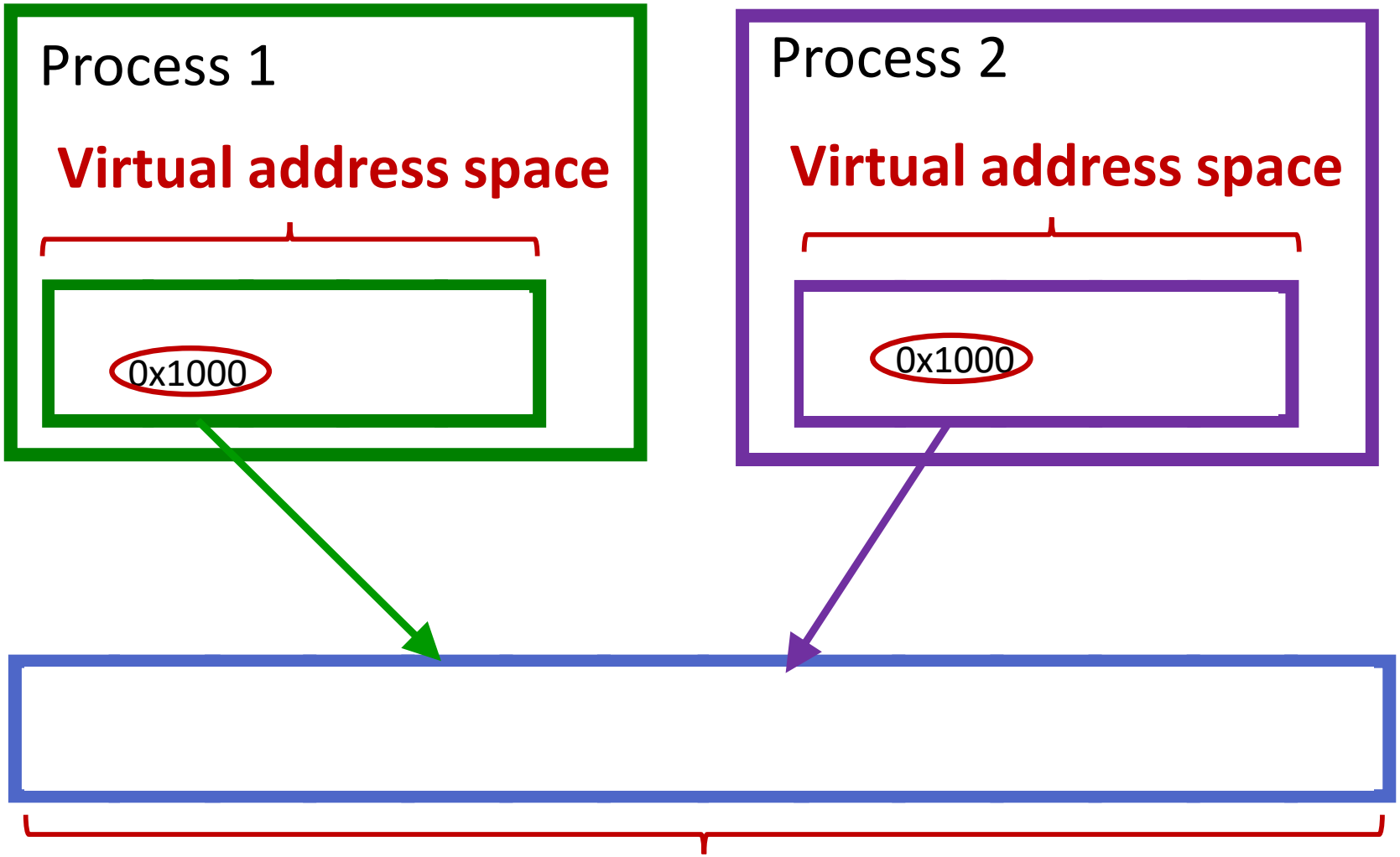
Roles of software and hardware



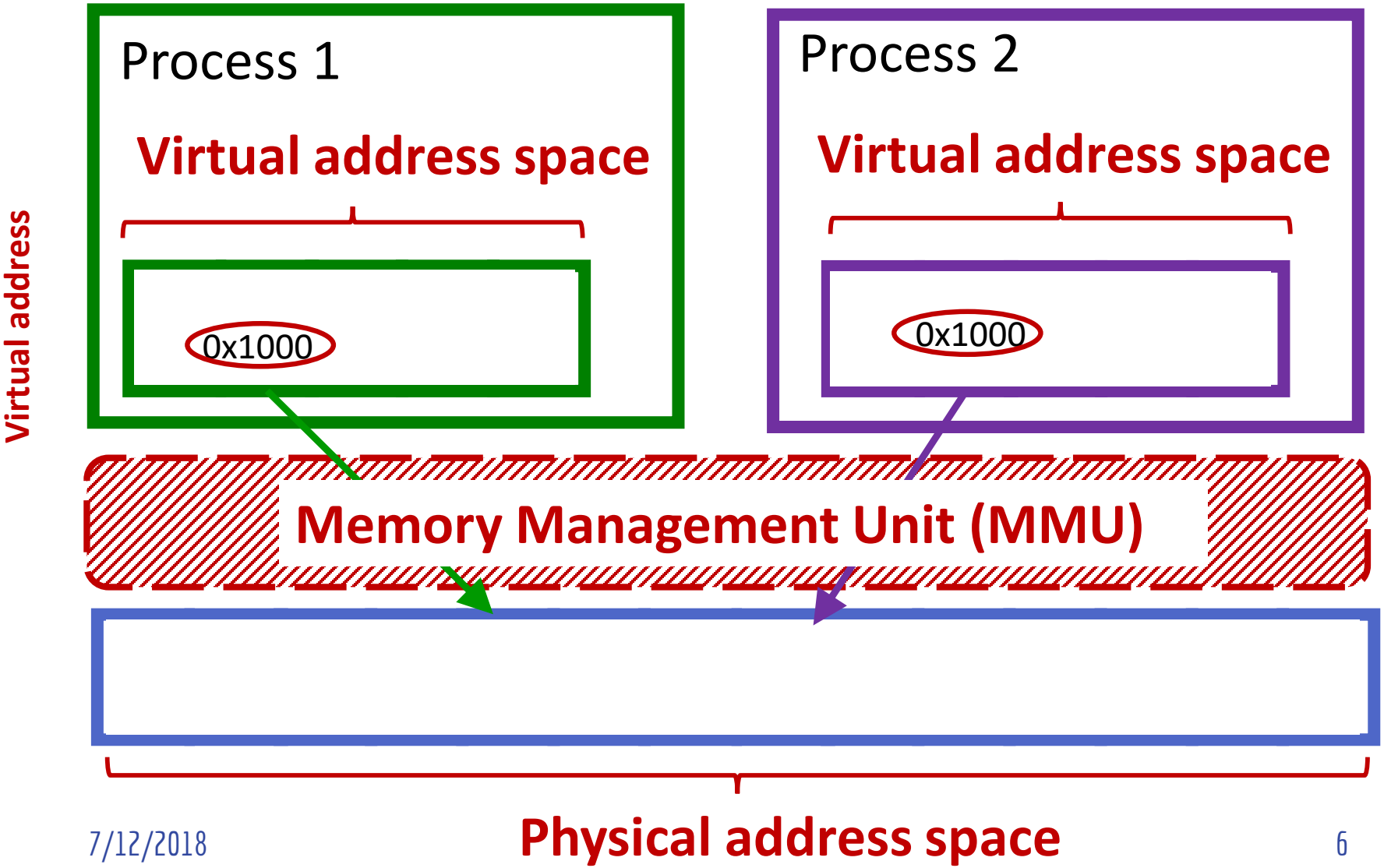
Virtual memory terminology



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
  - Provide each process with separate virtual address space
- **Protection:** Enforce rules on what memory a process can or cannot access

Agenda

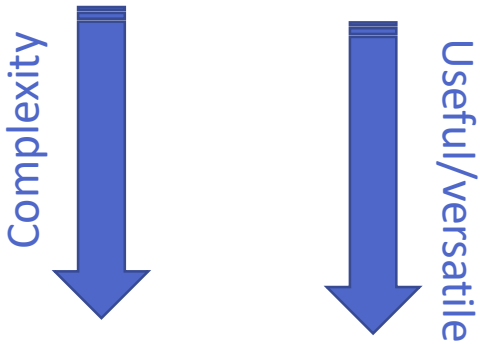
- What is virtual memory?
- Hardware implementations of virtual memory
- Software management of virtual memory
- Research opportunity in virtual memory

Agenda

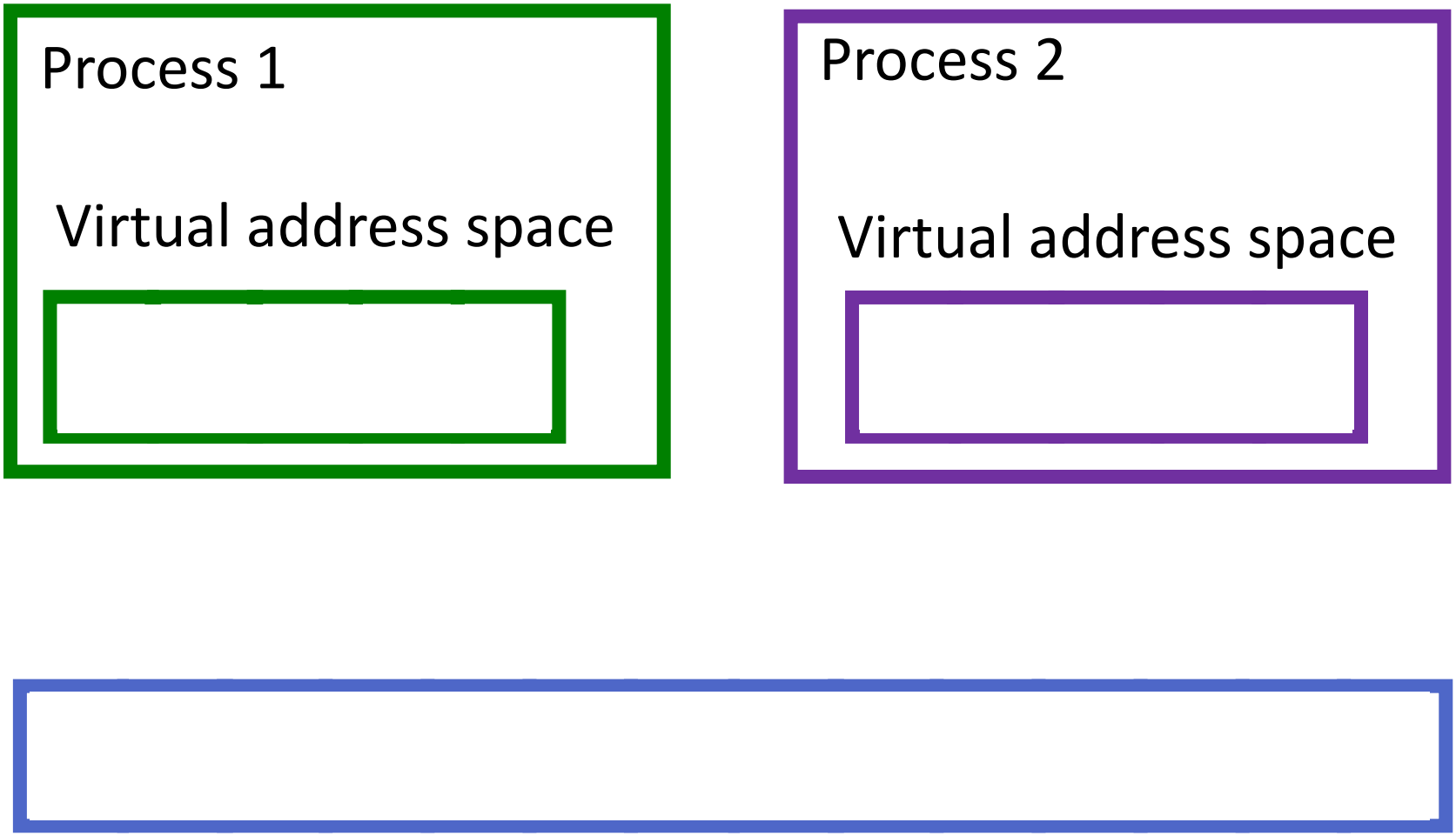
- What is virtual memory?
- Hardware implementations of virtual memory
- Software management of virtual memory
- Research opportunity in virtual memory

Implementing virtual memory

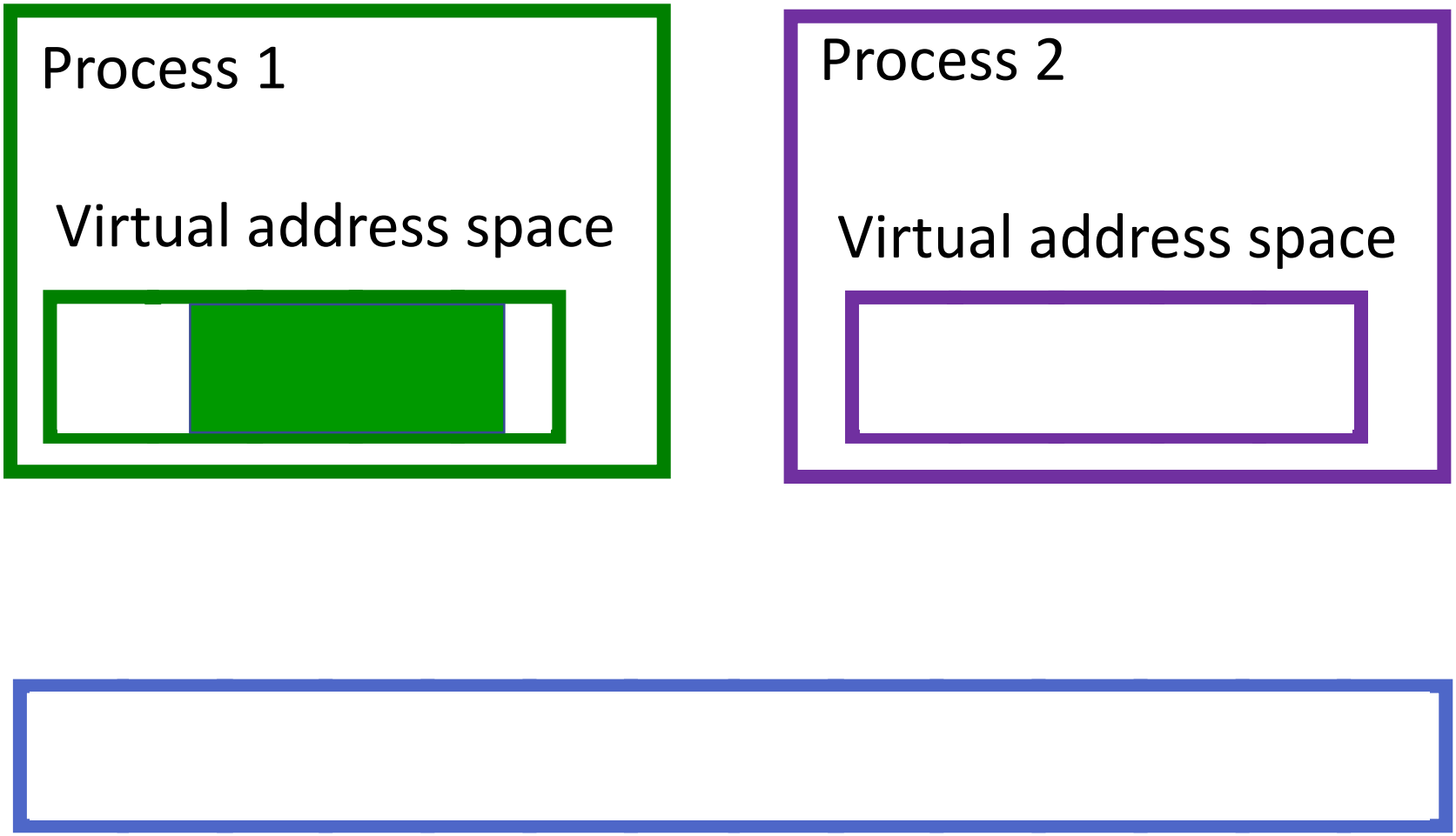
- Many ways to implement virtual memory
  - Base and bound register
  - Segmentation
  - Paging



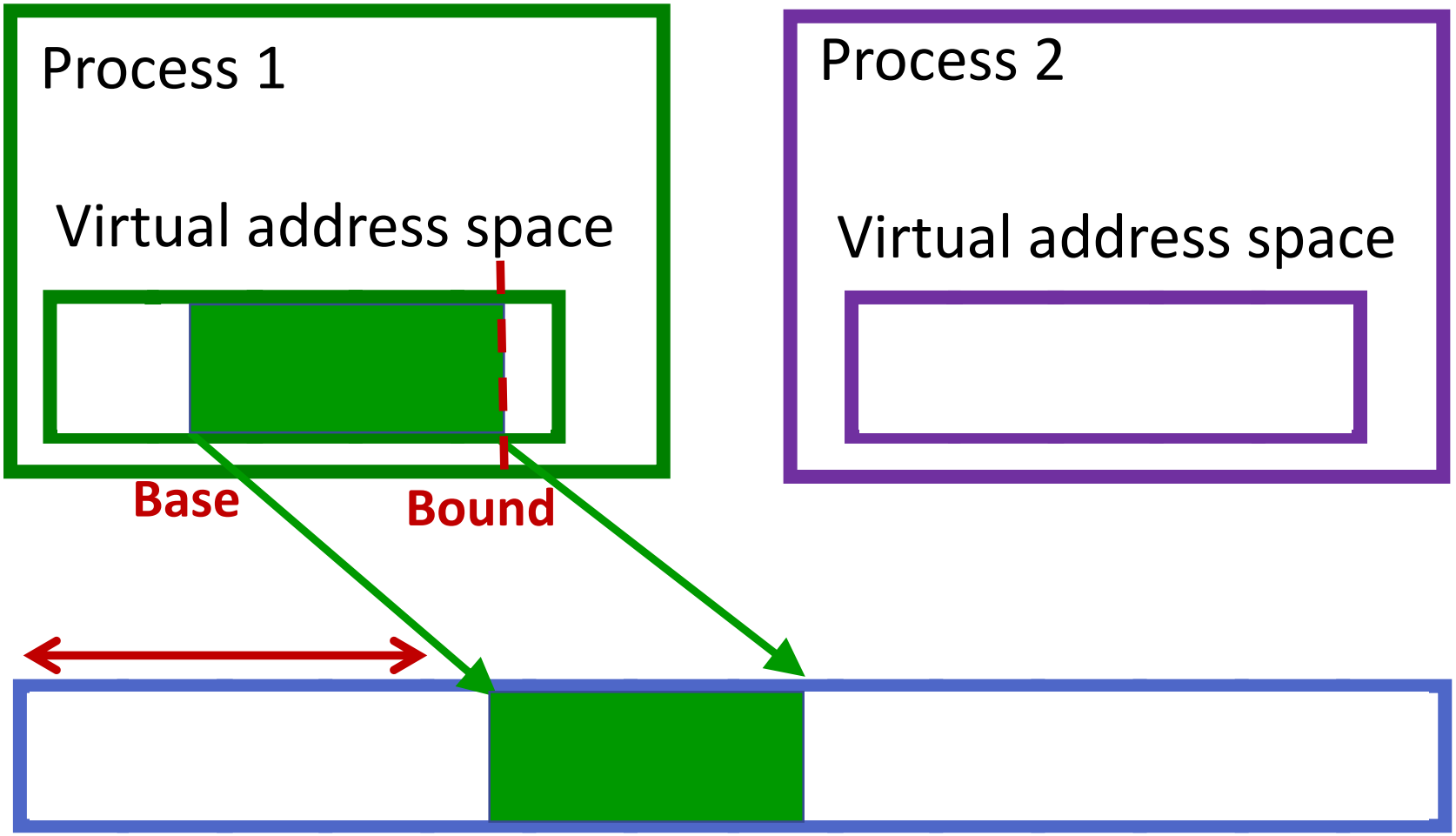
Base and bound registers



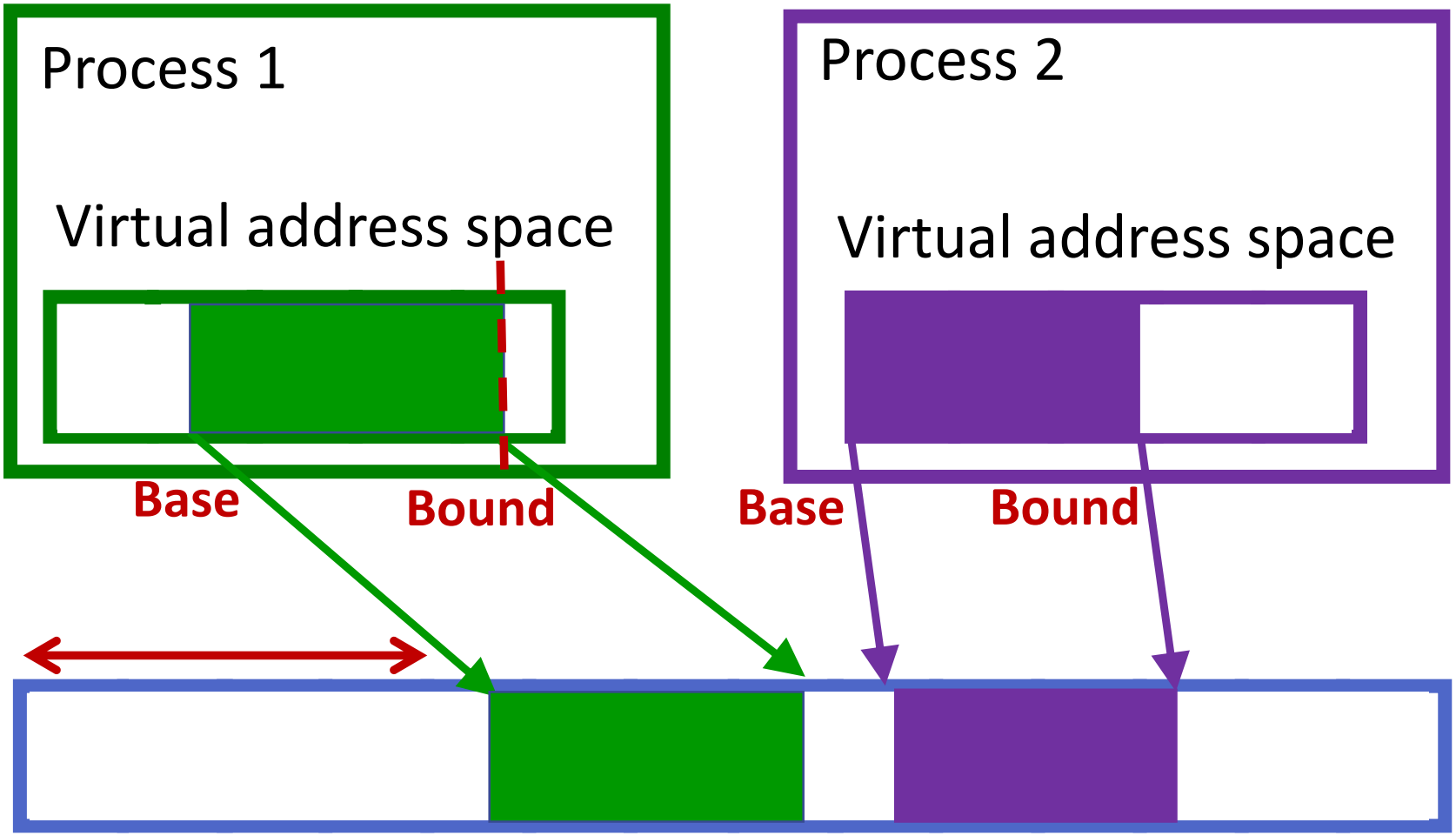
Base and bound registers



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

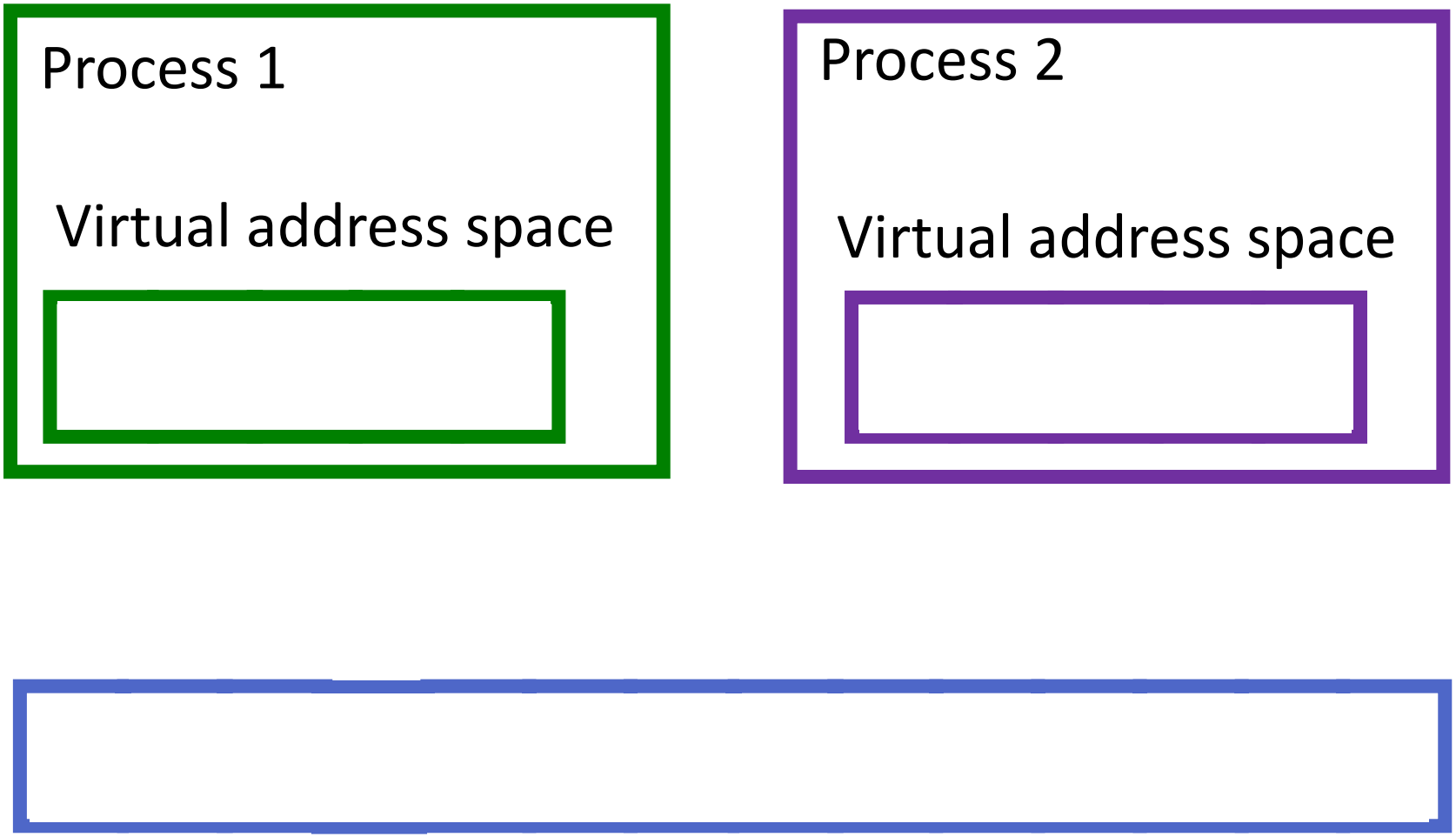
Base and bound registers

- 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

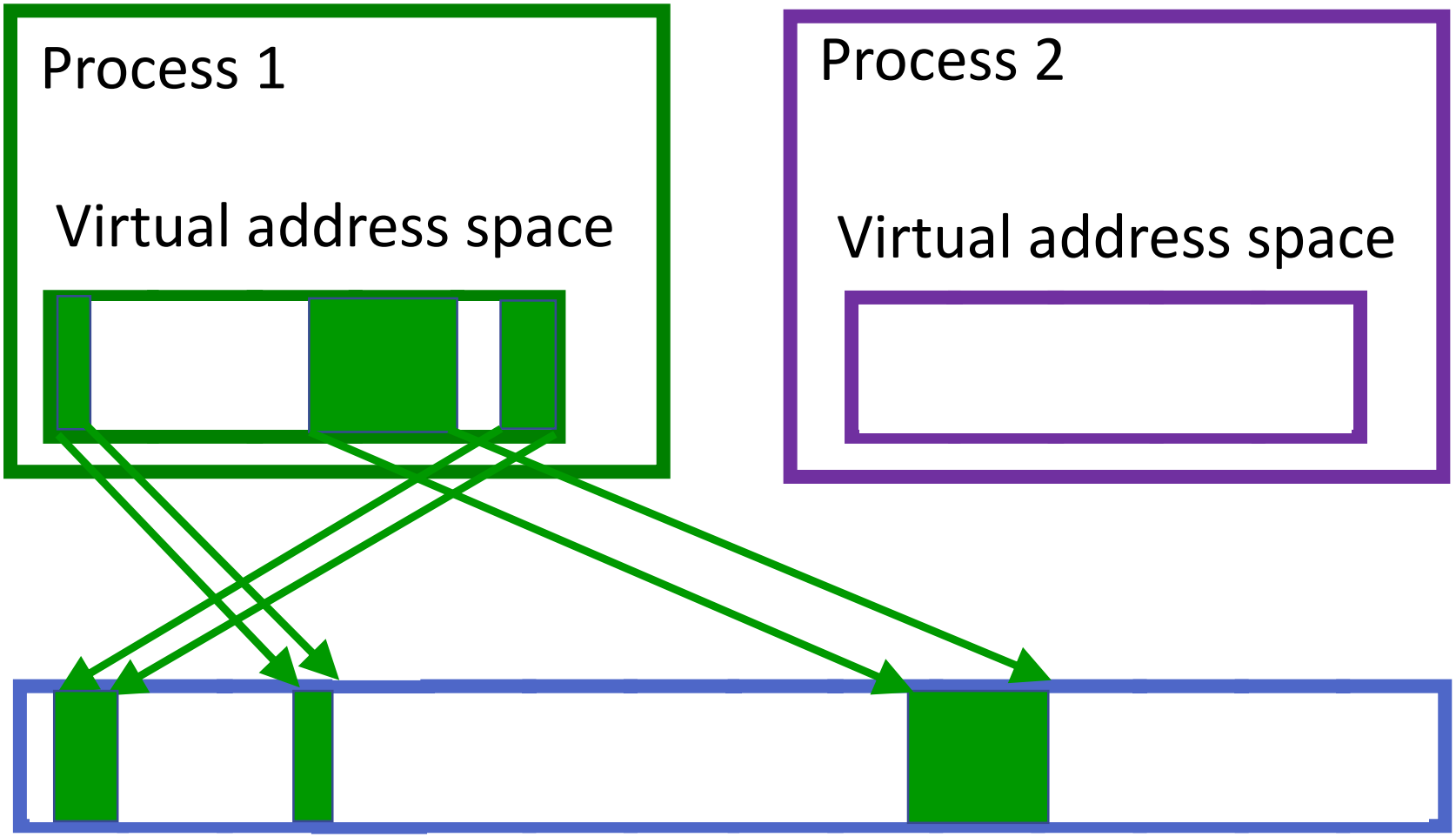
Base and bound registers

- 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 is no free memory at the edge

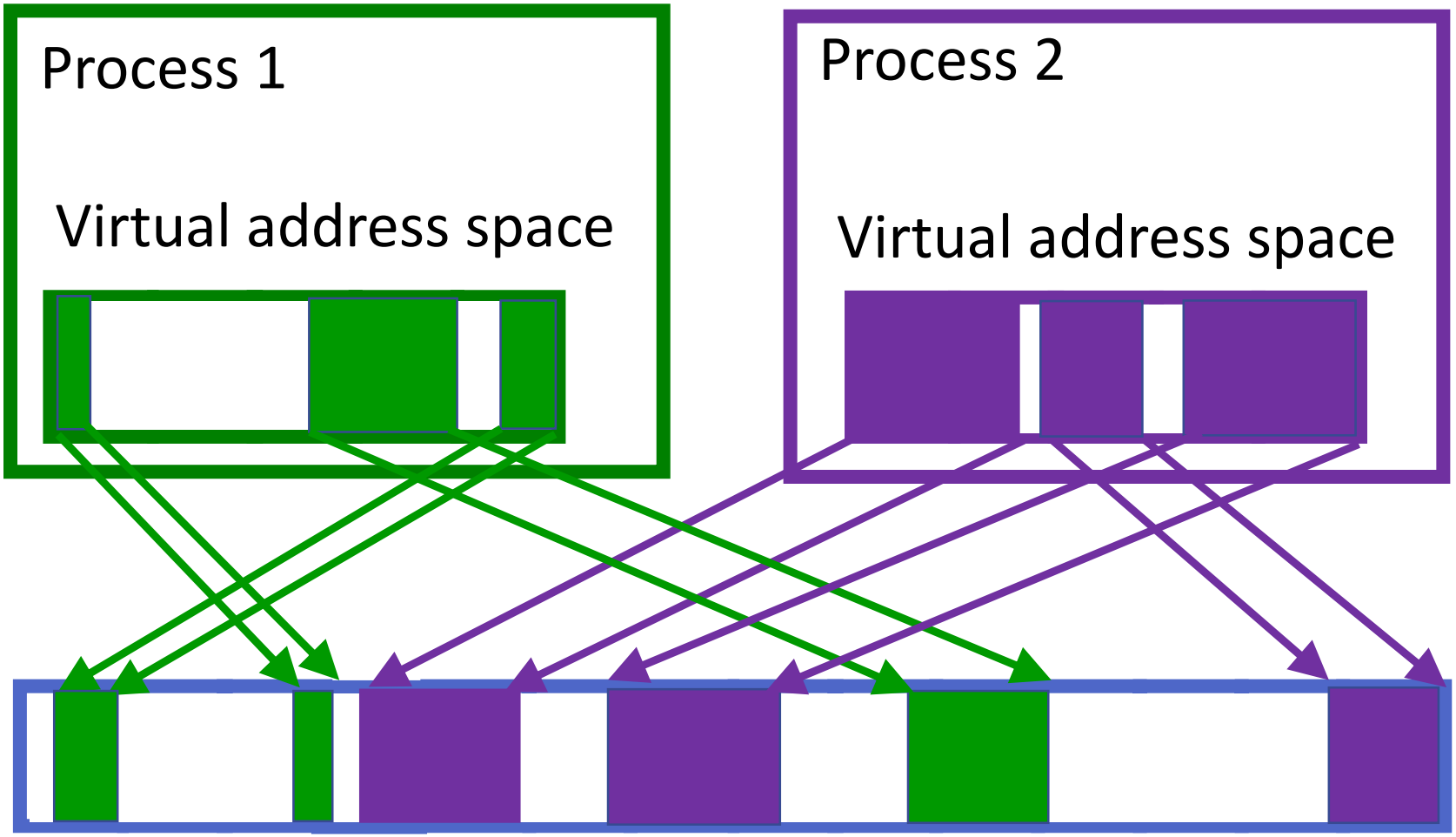
Segmentation



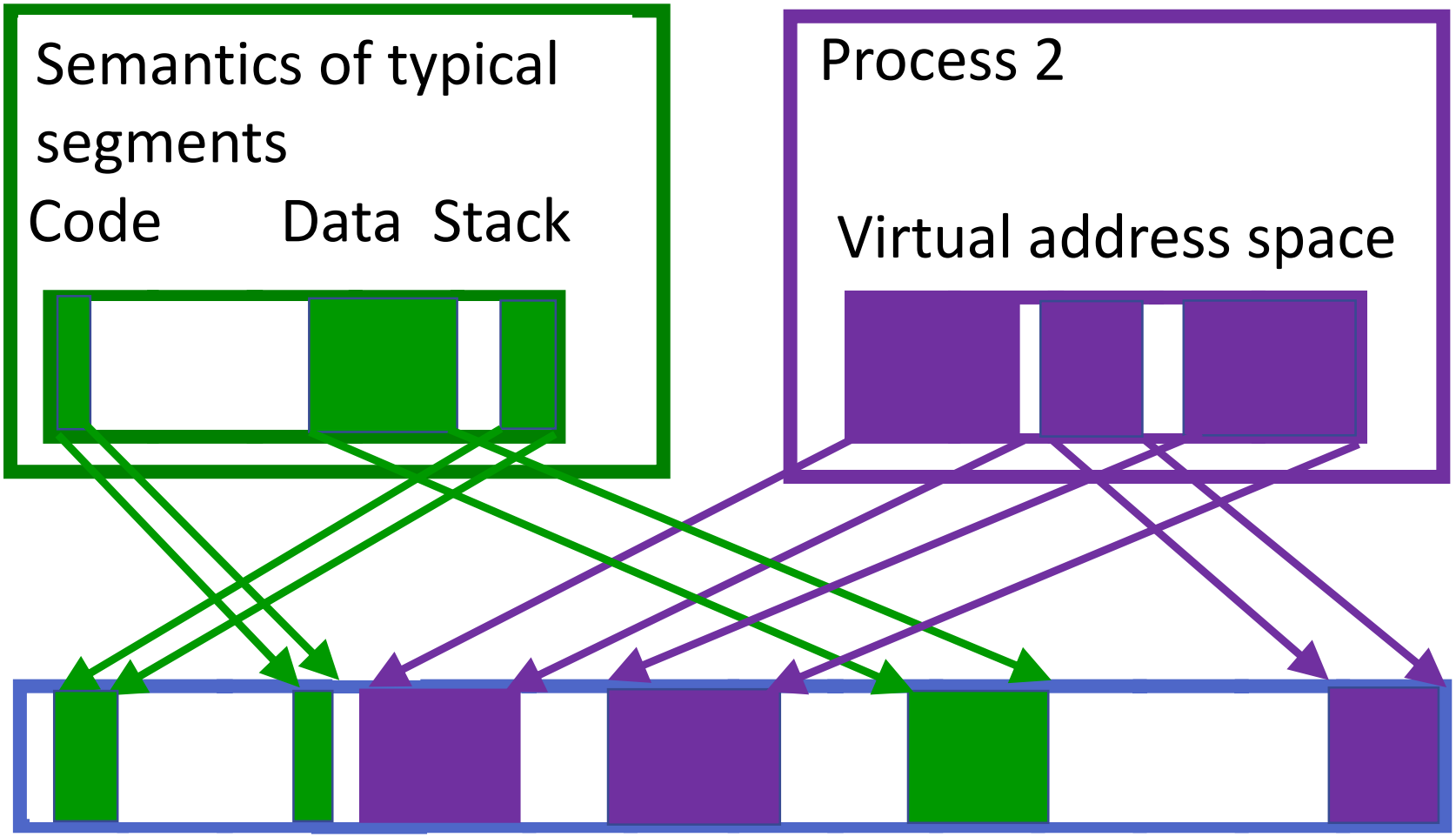
Segmentation



Segmentation



Segmentation



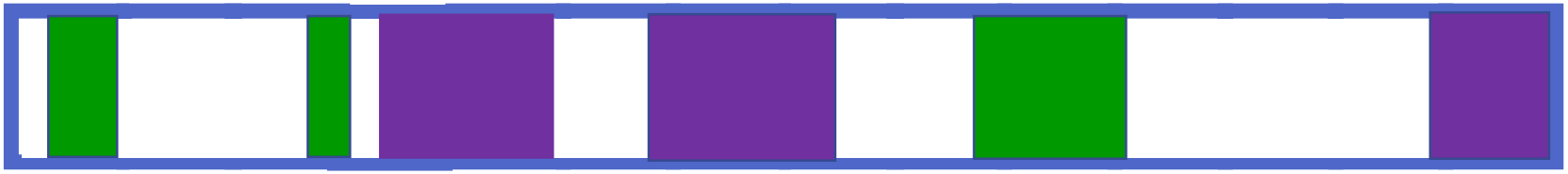
Segmentation: How it works?

Virtual address

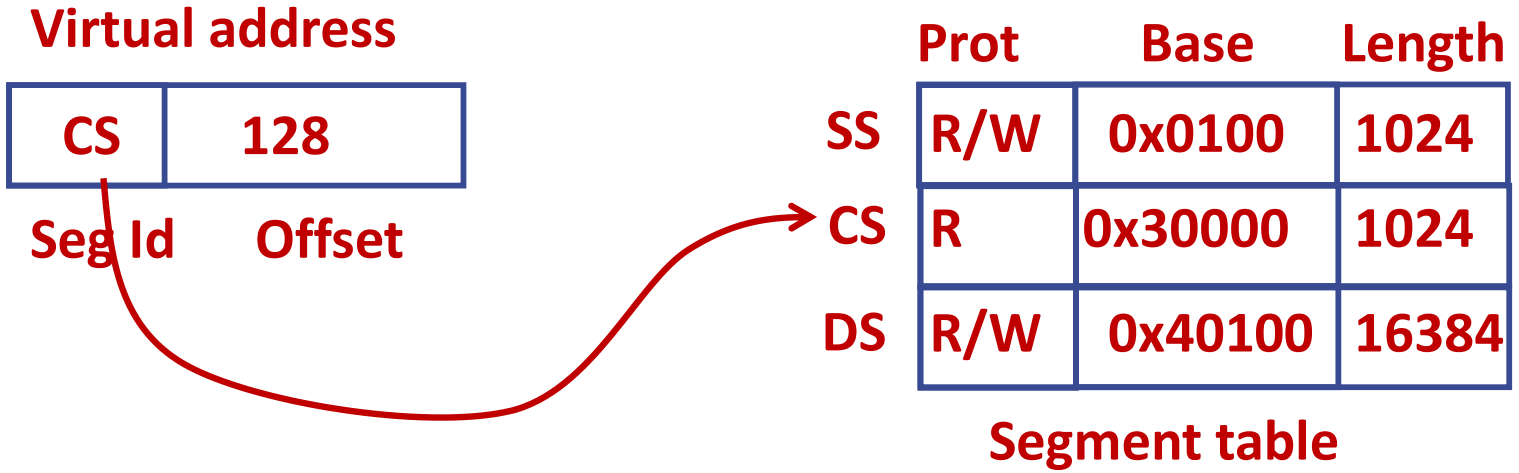
CS	128
Seg Id	Offset

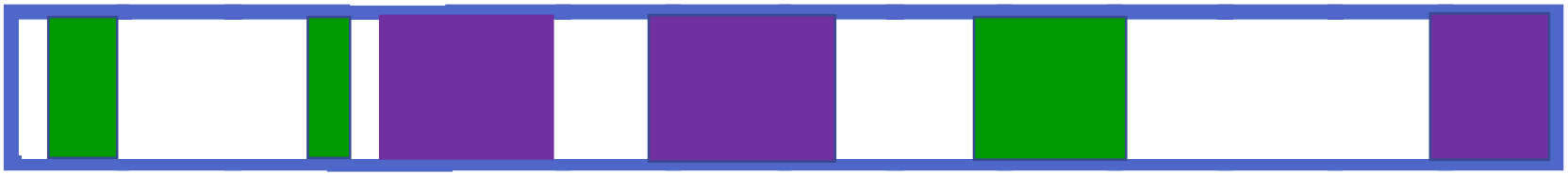
	Prot	Base	Length
SS	R/W	0x0100	1024
CS	R	0x30000	1024
DS	R/W	0x40100	16384

Segment table



Segmentation: How it works?





Segmentation: How it works?

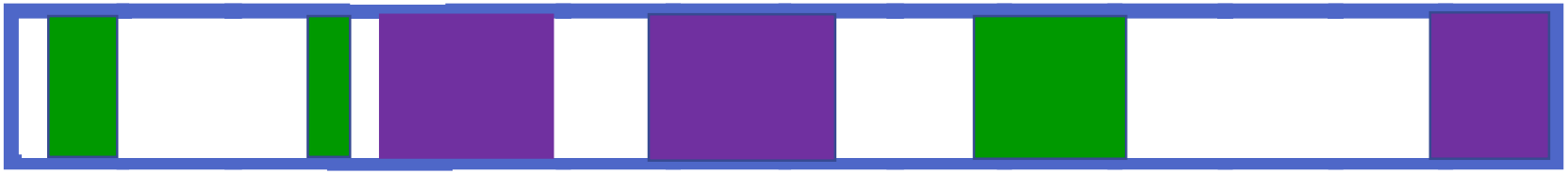
Virtual address

CS	128
Seg Id	Offset

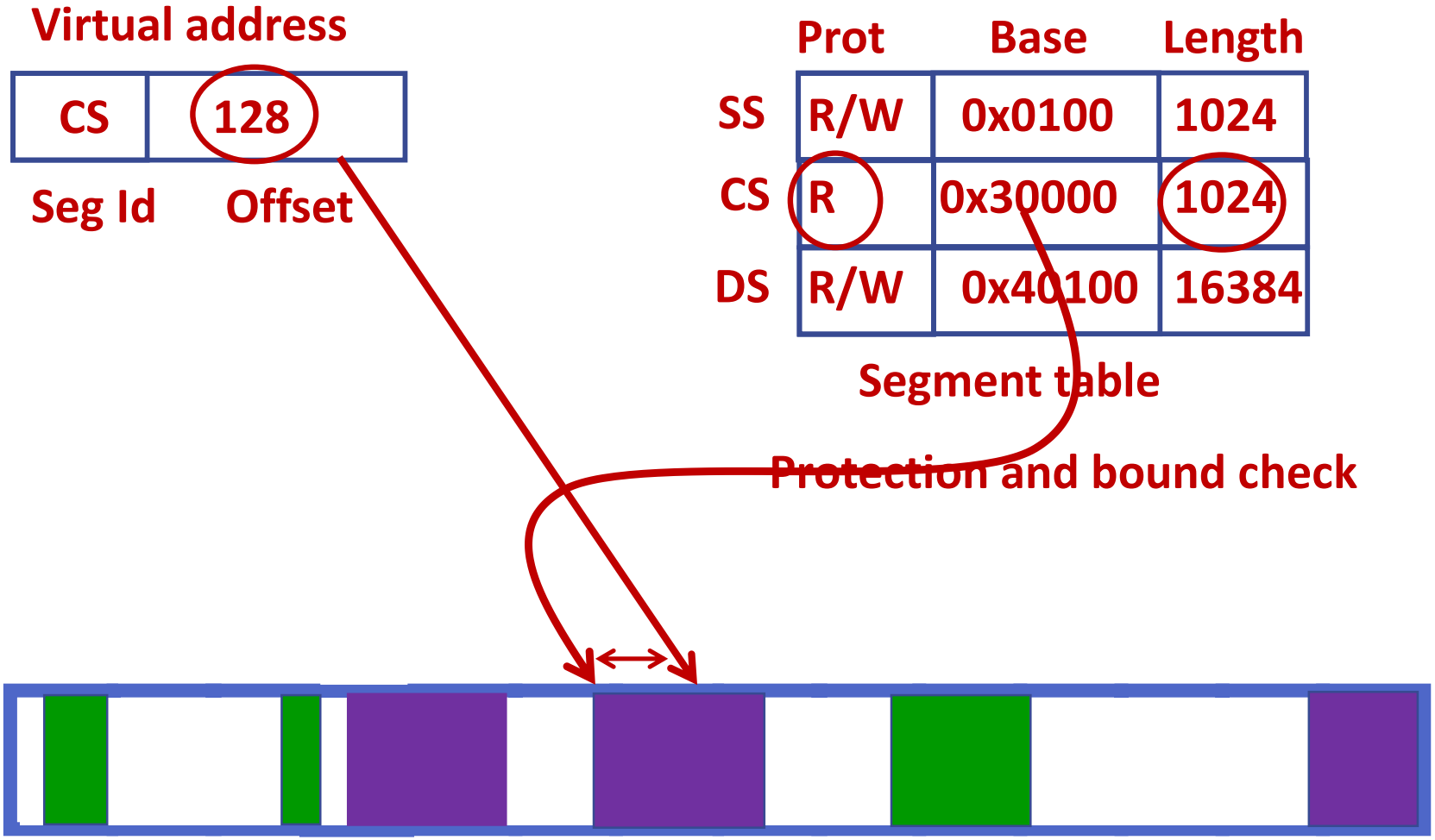
	Prot	Base	Length
SS	R/W	0x0100	1024
CS	R	0x30000	1024
DS	R/W	0x40100	16384

Segment table

Protection and bound check



Segmentation: How it works?



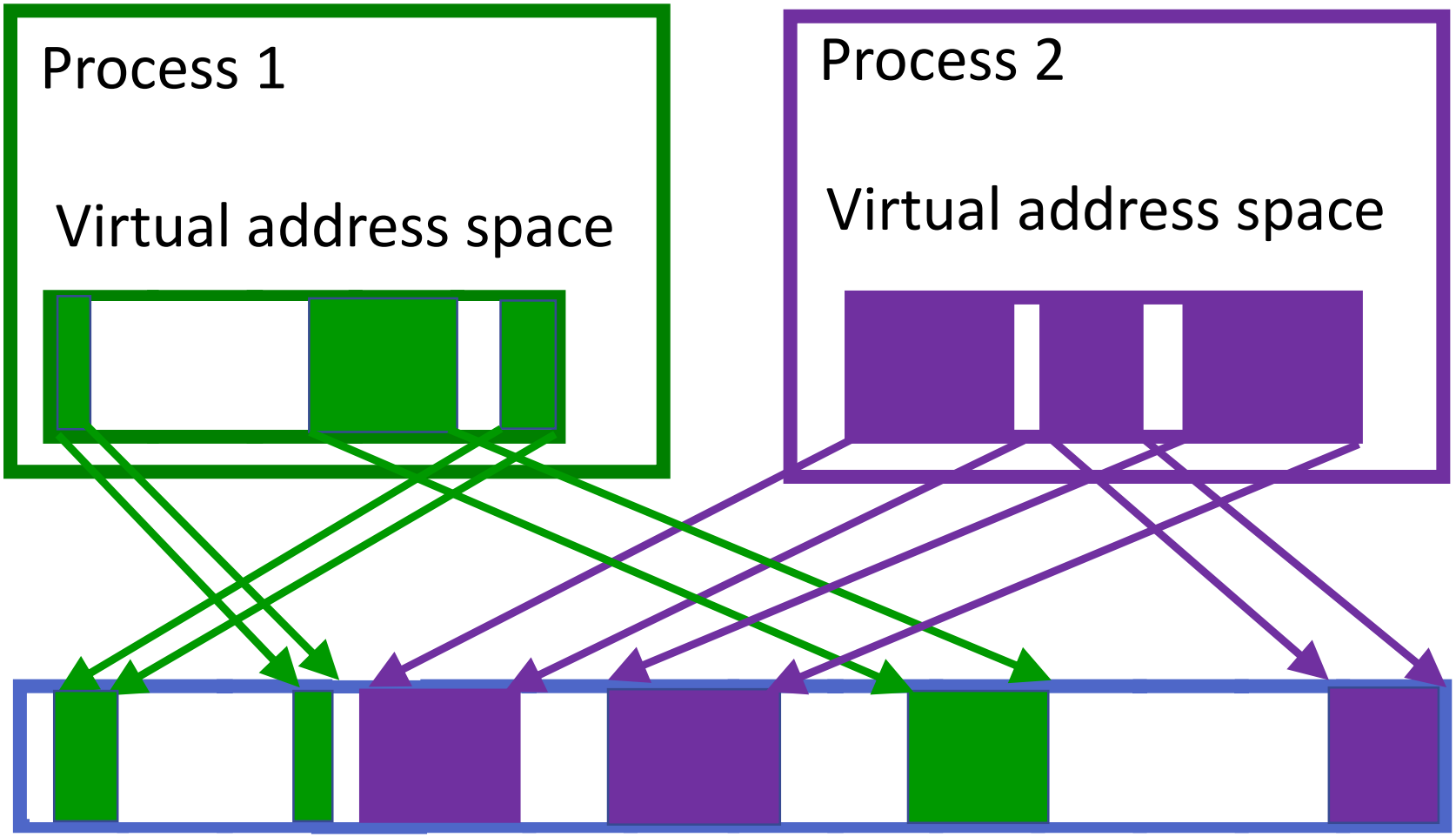
Segmentation: How it works?

- How to select segmentation ID?
  - In most cases, the compiler can infer
    - For example, x86-32 bit has CS, SS, DS, ES, FS, GS

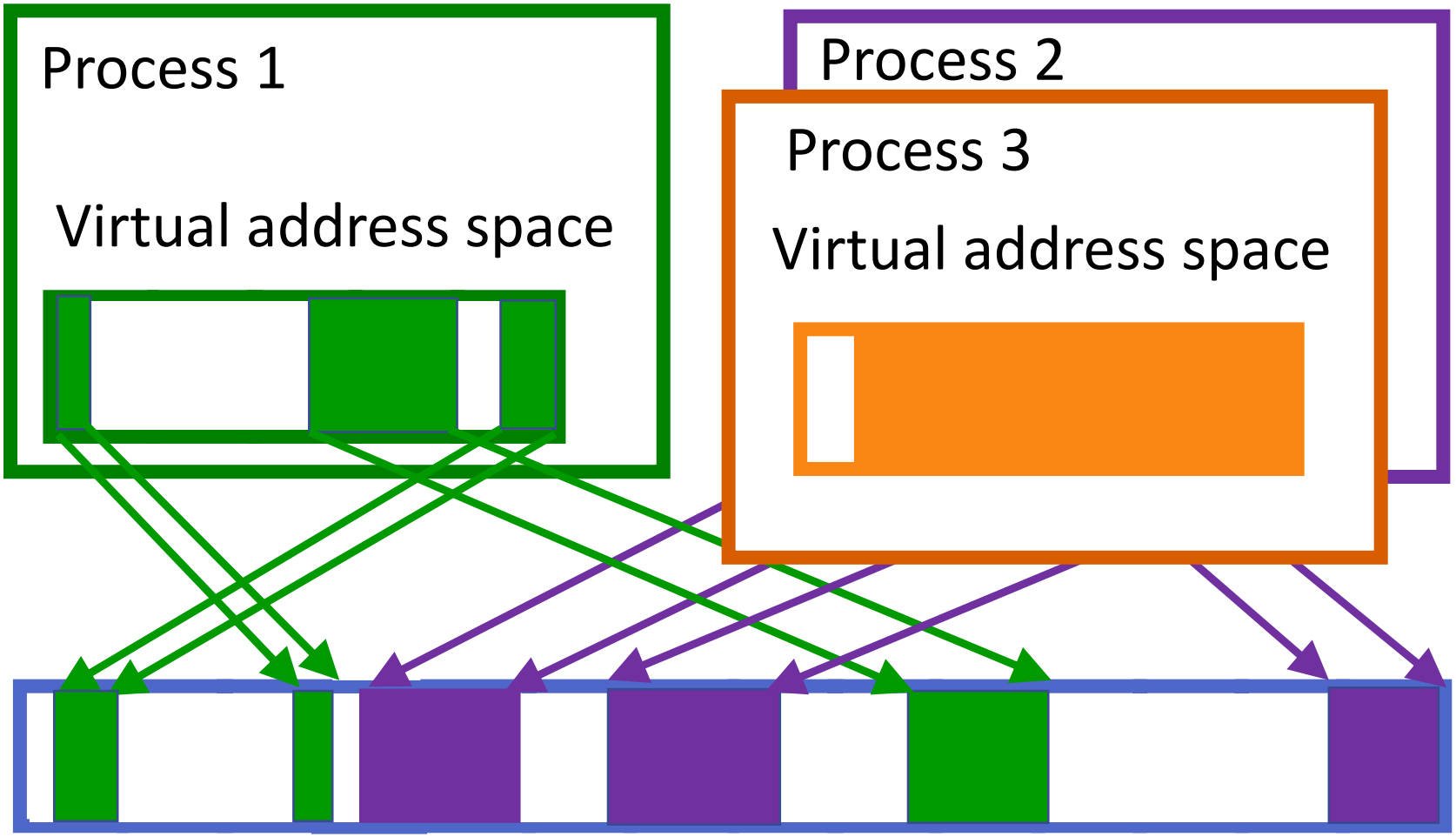
Compiler selected | Programmer selected

- 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

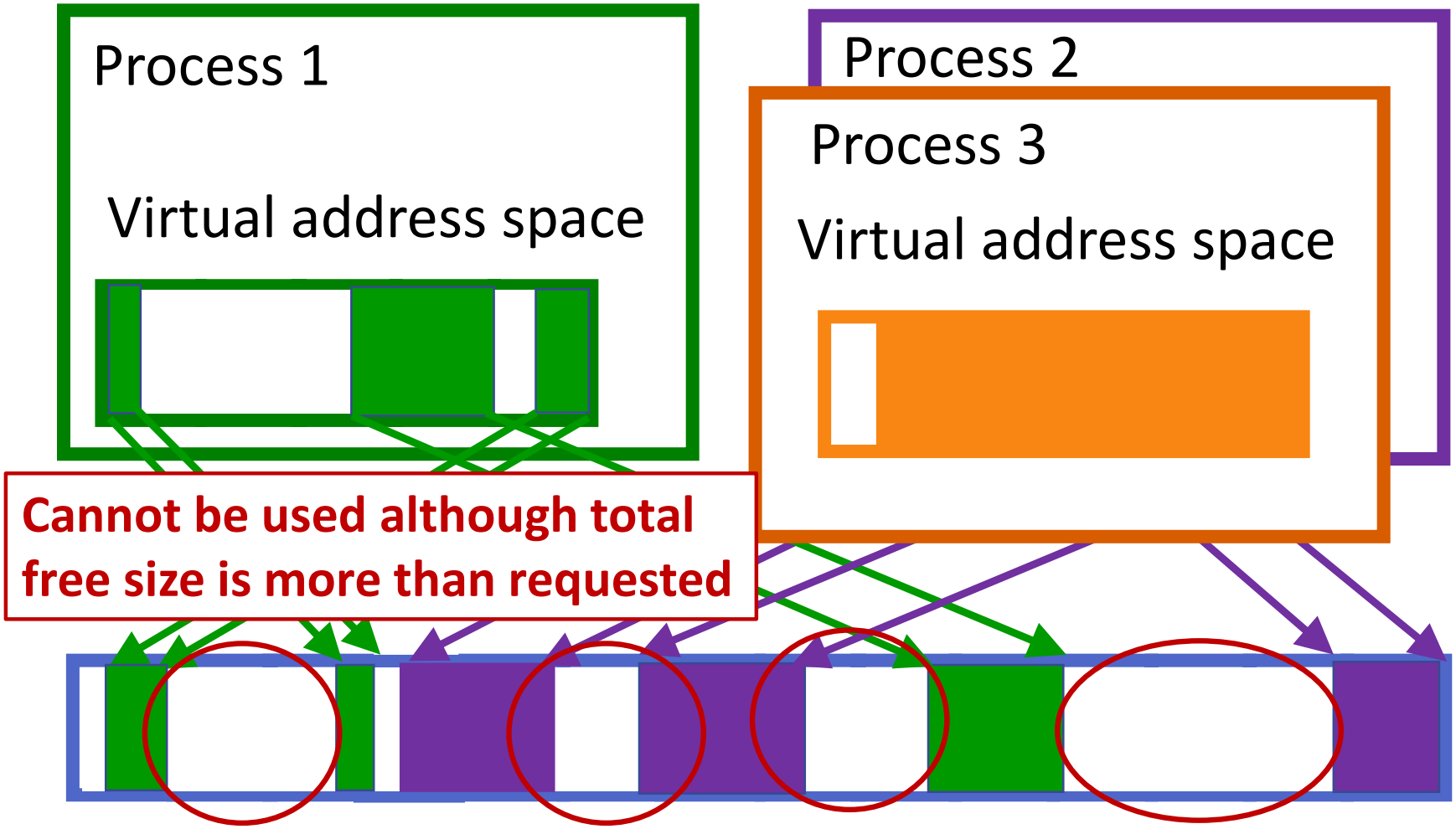
Disadvantages of Segmentation



Disadvantages of Segmentation



Disadvantages of Segmentation



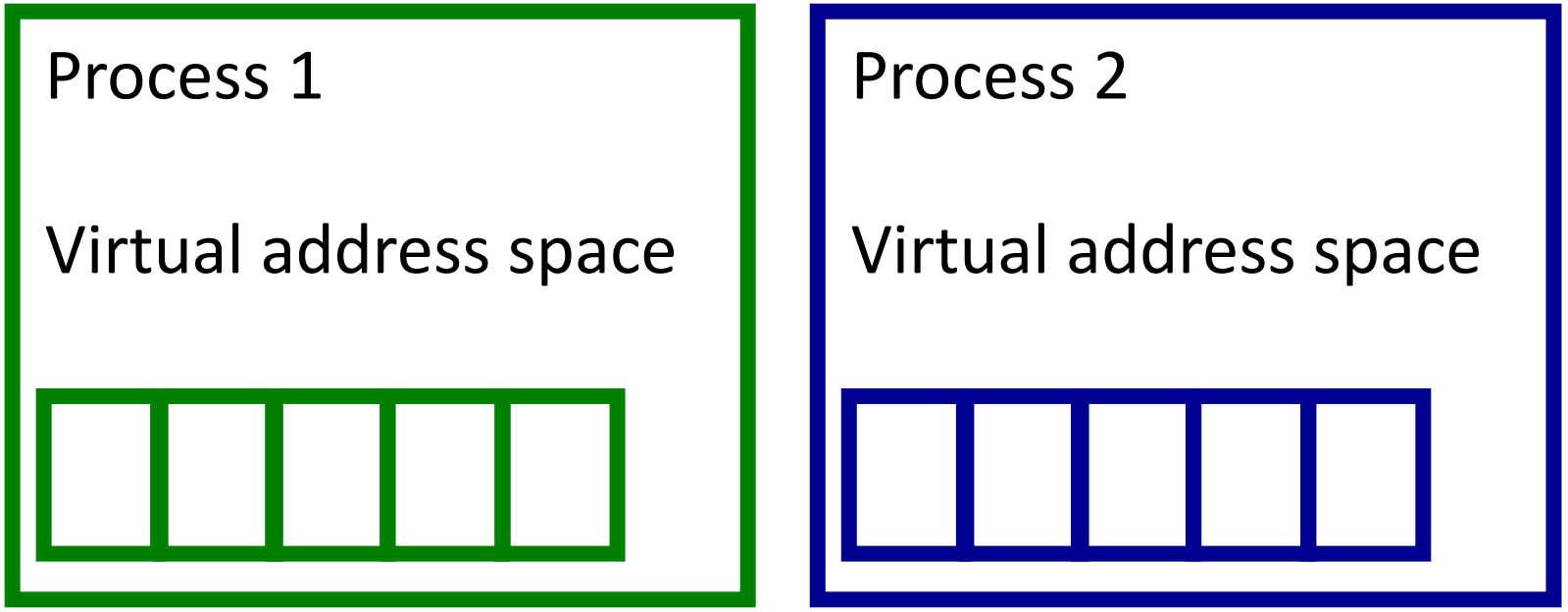
Disadvantages of Segmentation

- 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

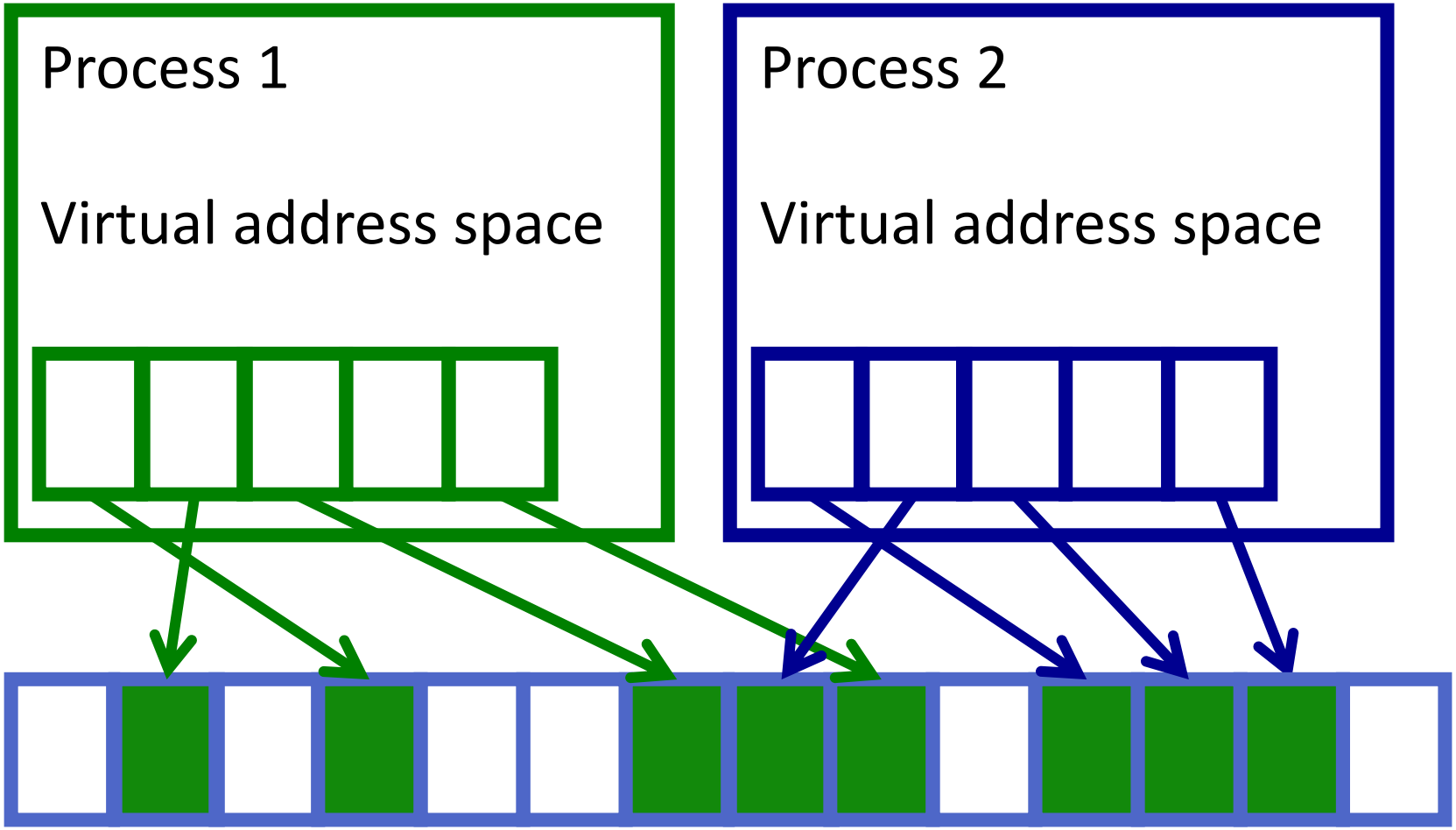
Paging

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

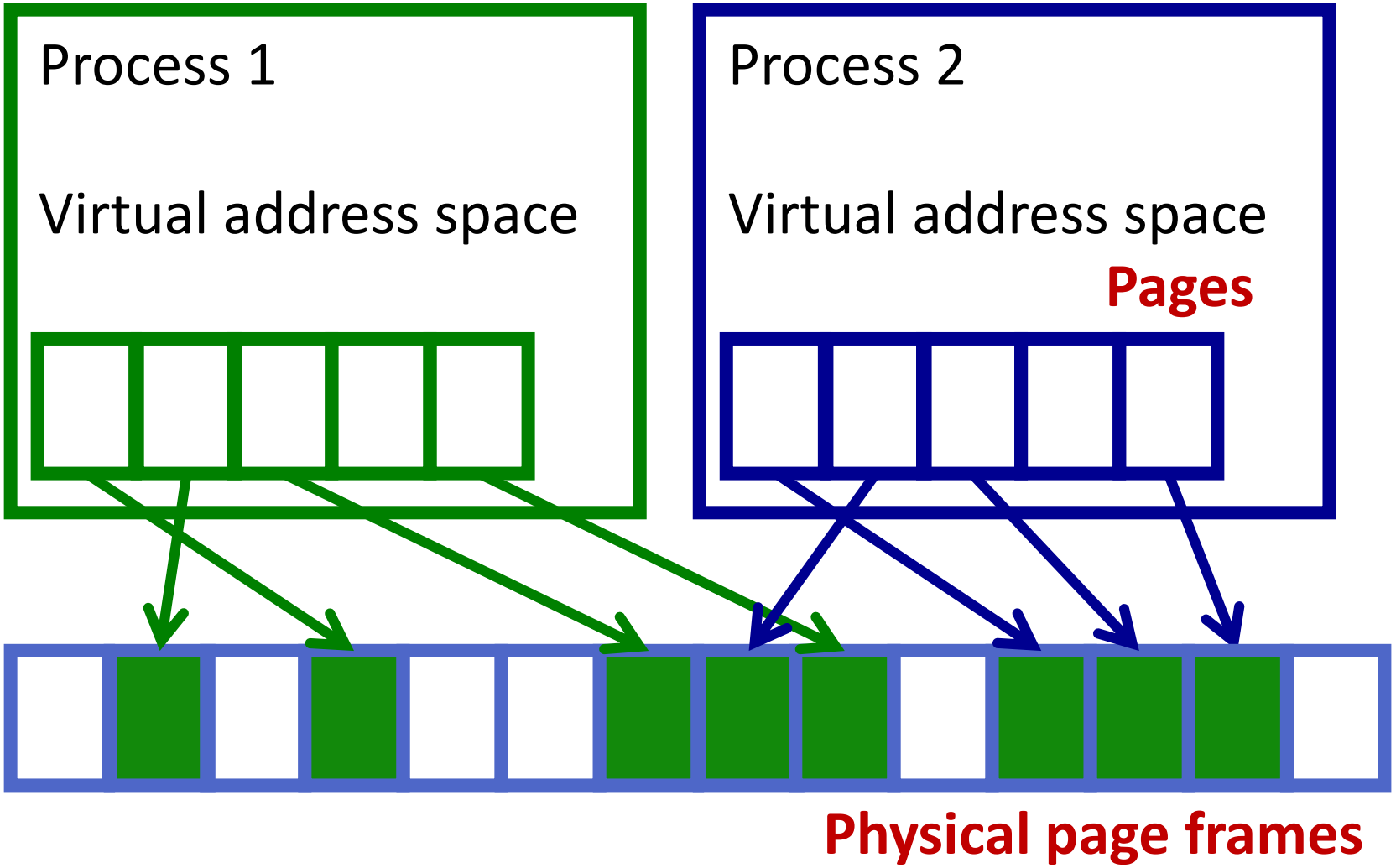
Paging


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

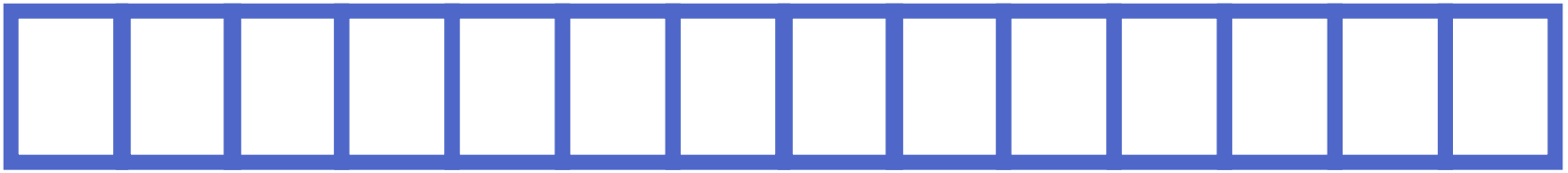
Almost all processors today employs paging

Disadvantages of paging

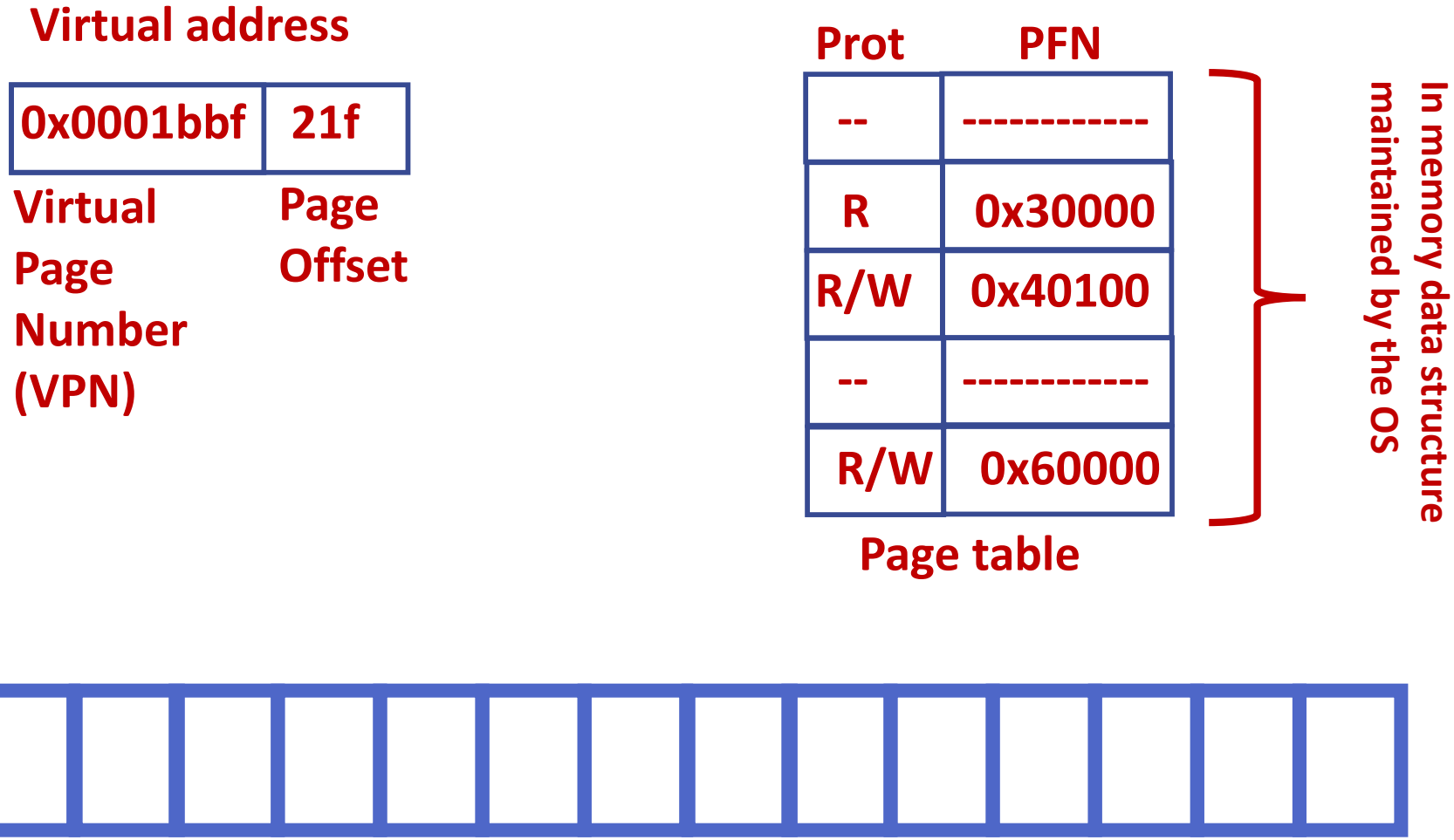
- **Internal fragmentation:** Memory wastage due to allocation only in page sizes (e.g., 4KB)
  - e.g., for an allocation 128 bytes a 4KB page will be allocated
  - Potentially waste (4KB – 128) bytes.

Paging: How it works (simplistic)?

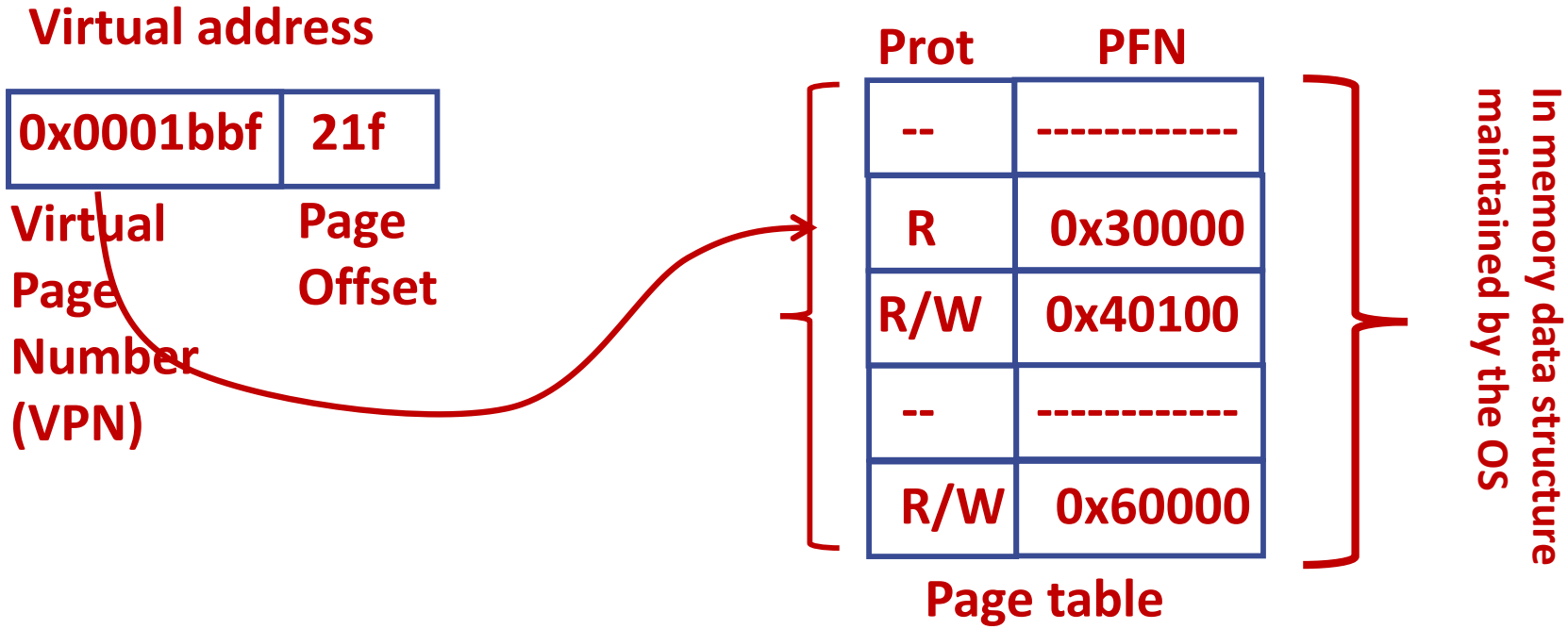
Virtual address



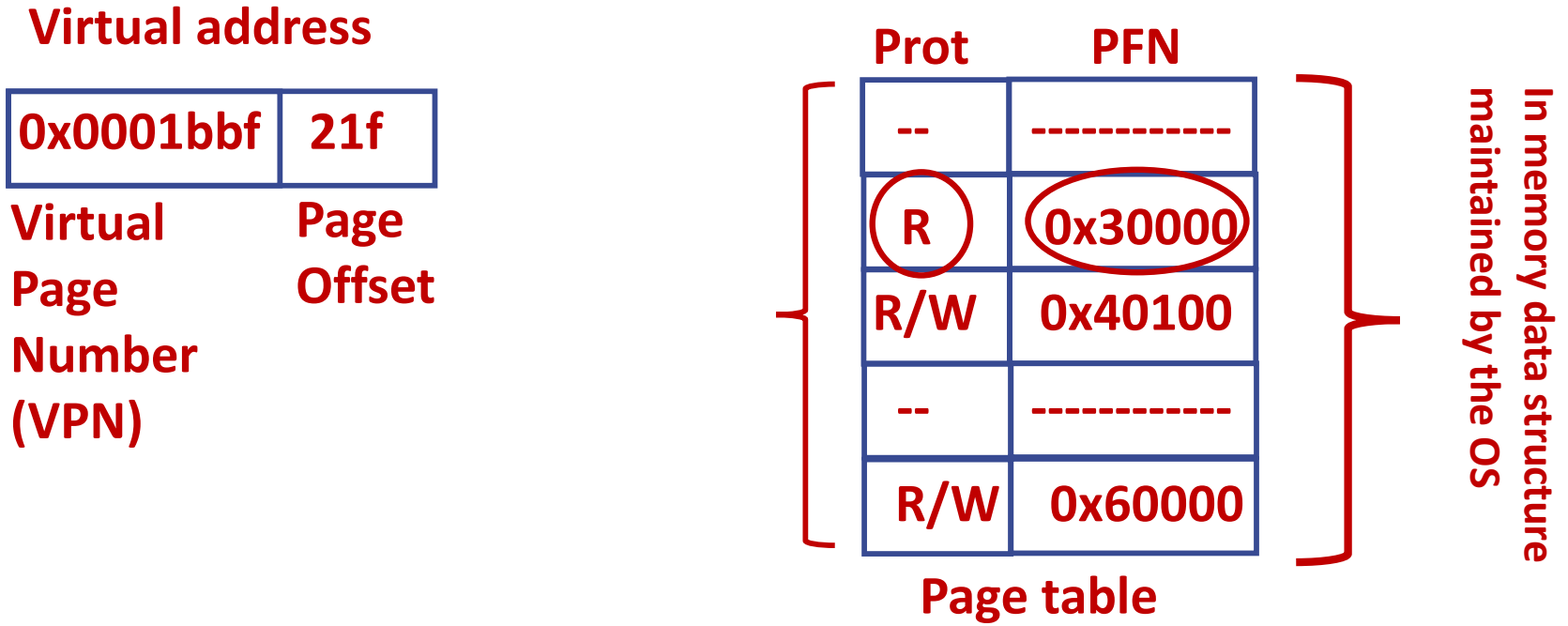
Paging: How it works (simplistic)?



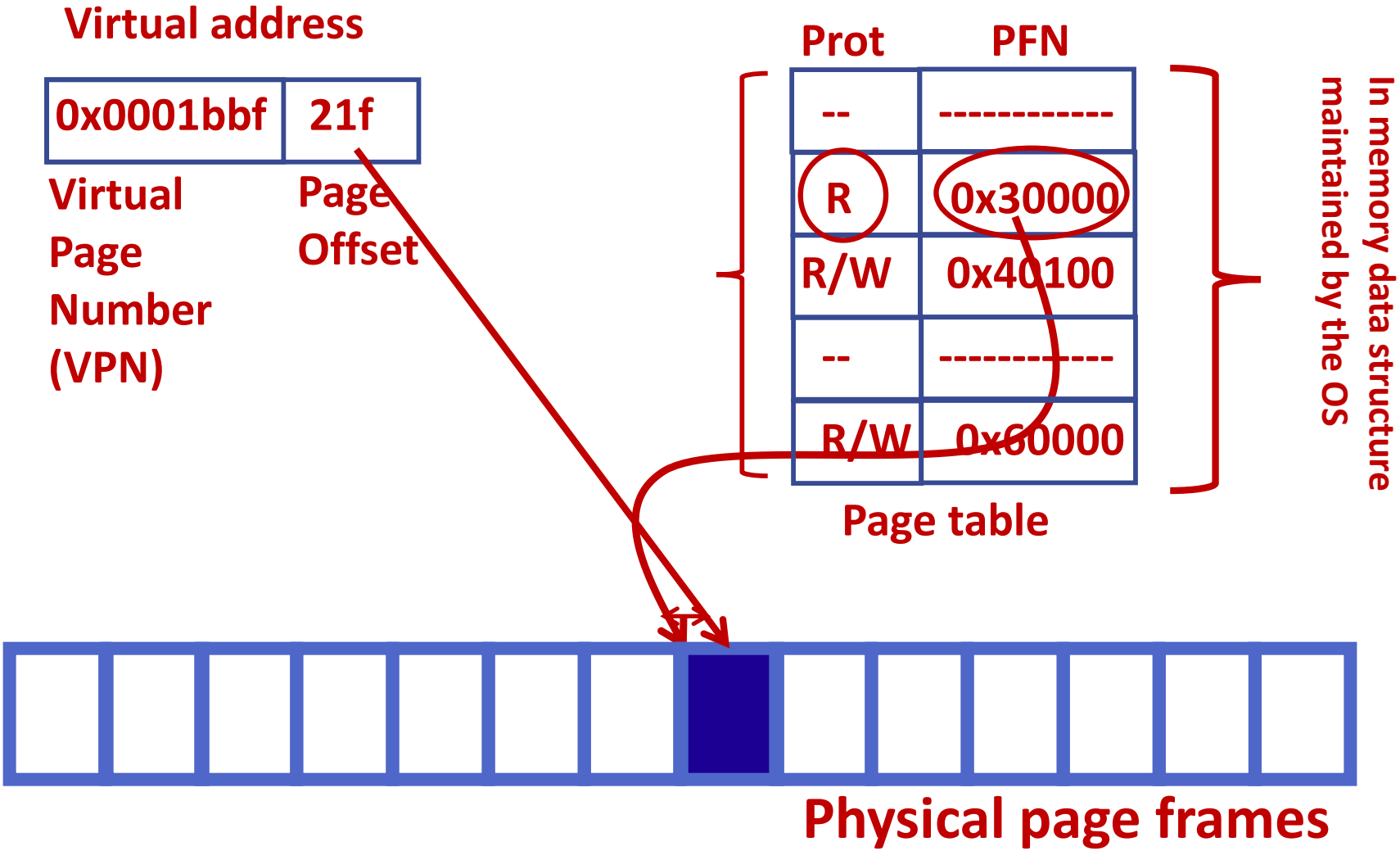
Paging: How it works (simplistic)?



Paging: How it works (simplistic)?



Paging: How it works (simplistic)?



Paging: How it *really* works?

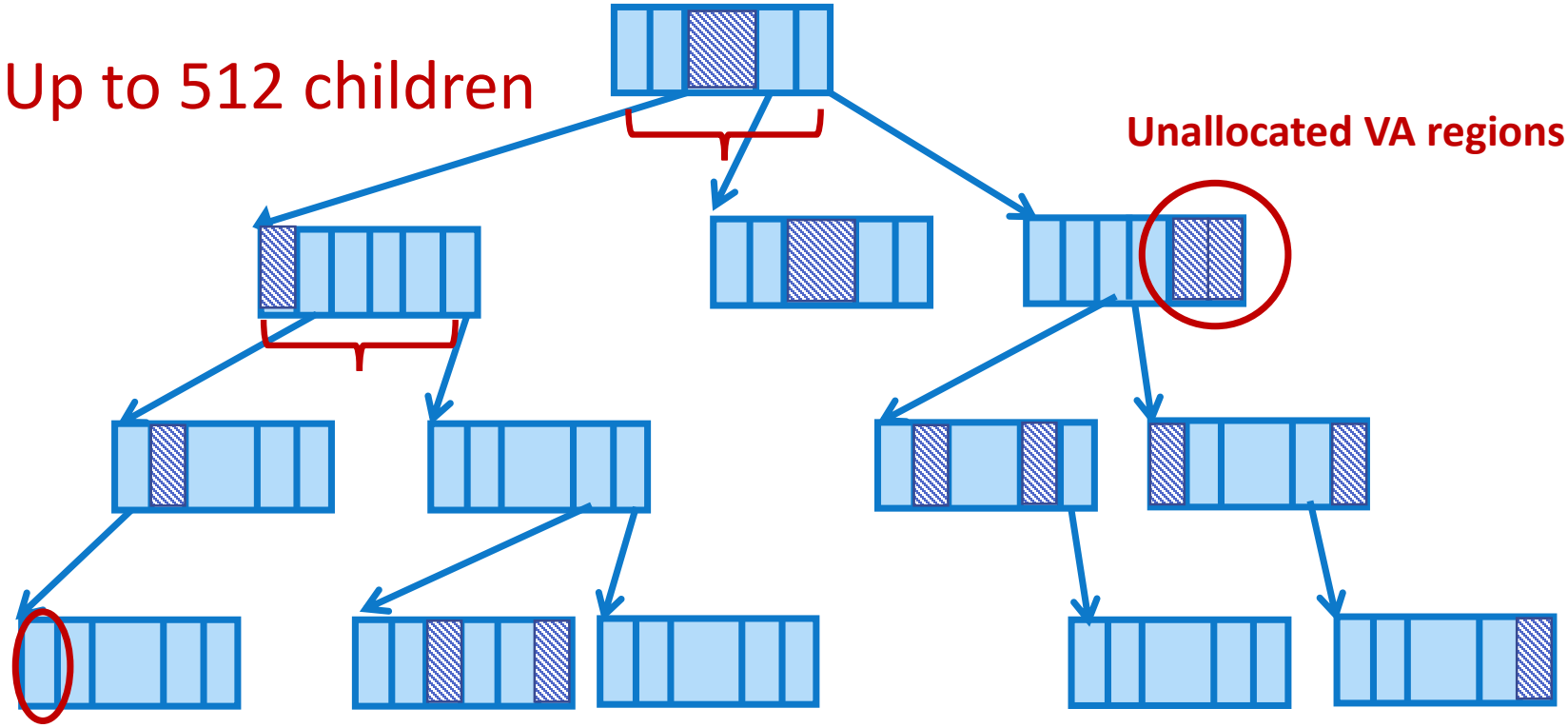
- Single level page table adds too much overhead
  - Consider a typical 48-bit virtual address space, 4KB page size and 8 byte long page table entry (PTE)
  - **Each** page table will be 512GB
  - There could be many processes → many page tables
- Often virtual address space is sparsely allocated
  -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



* i.e., Intel or AMD processors

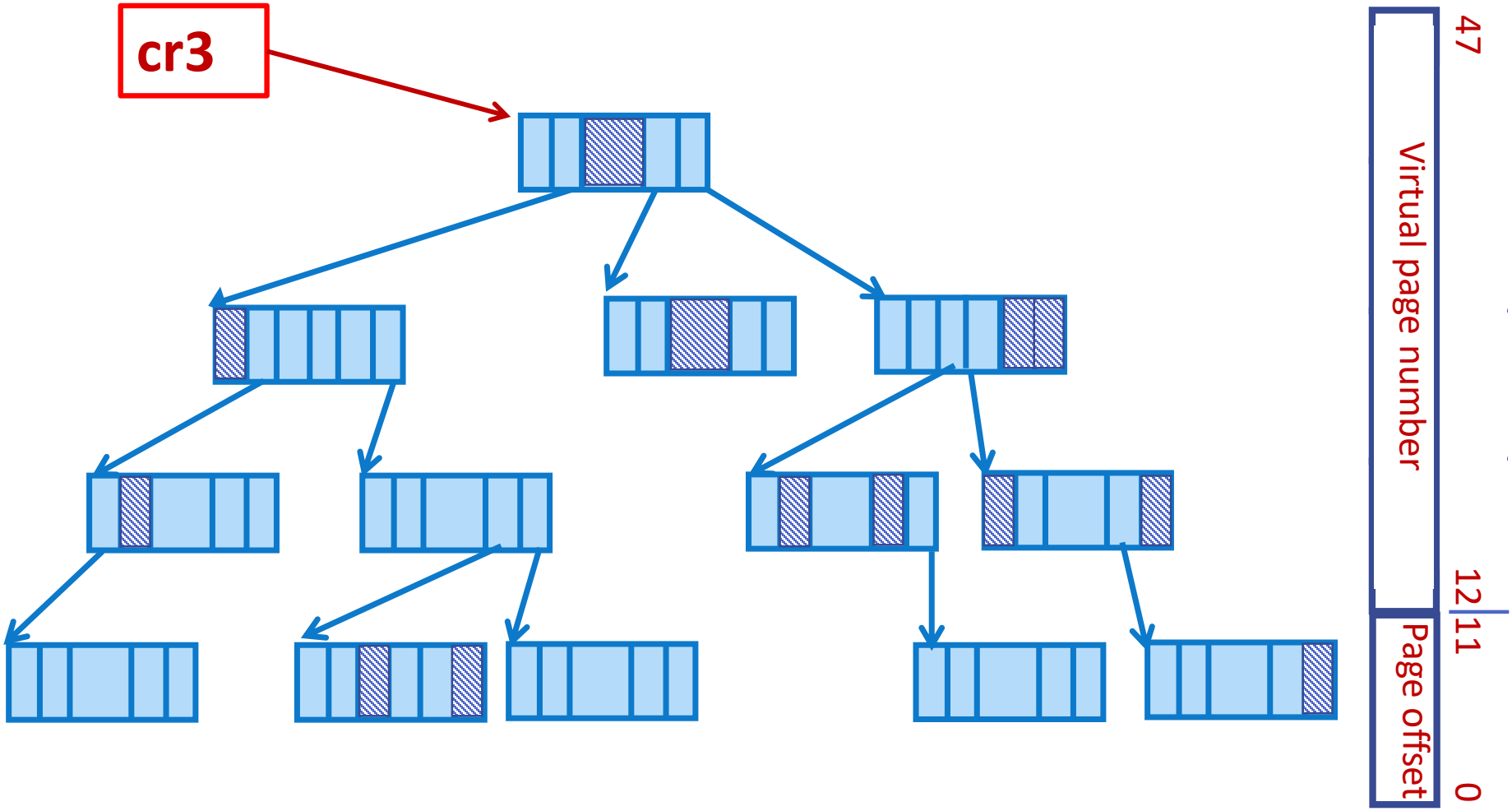
Paging: 64bit x86 page table

- Operating system maintains one page table per process (a.k.a., per virtual address space)
- Operating system creates, updates, deletes page table entries (PTEs)

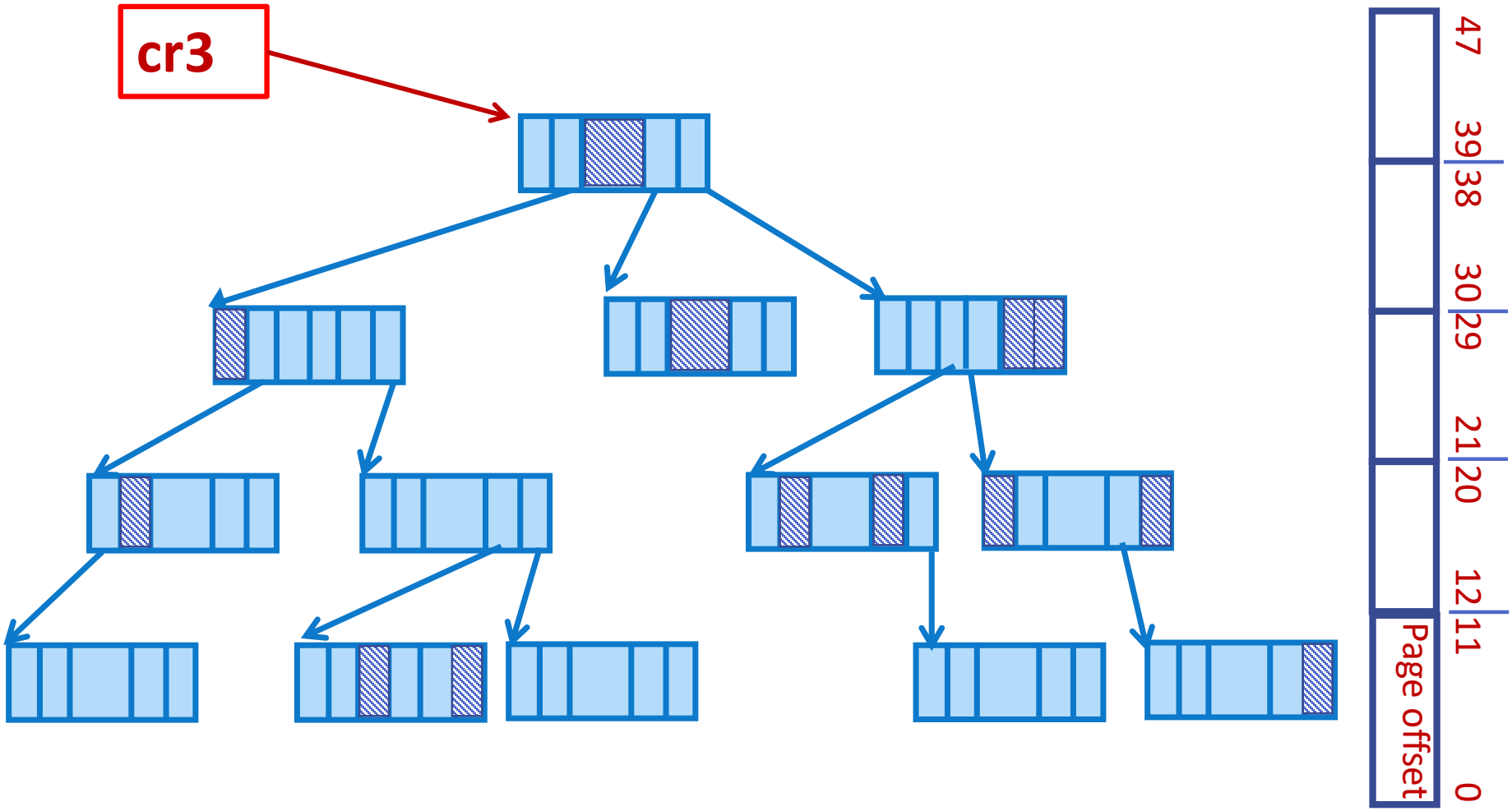
Paging: 64bit x86 page table

- Operating system maintains one page table per process (a.k.a., per virtual address space)
- Operating system creates, updates, deletes page table entries (PTEs)
- Page table structure is part of agreement between the OS and the hardware → page table structure ISA specific

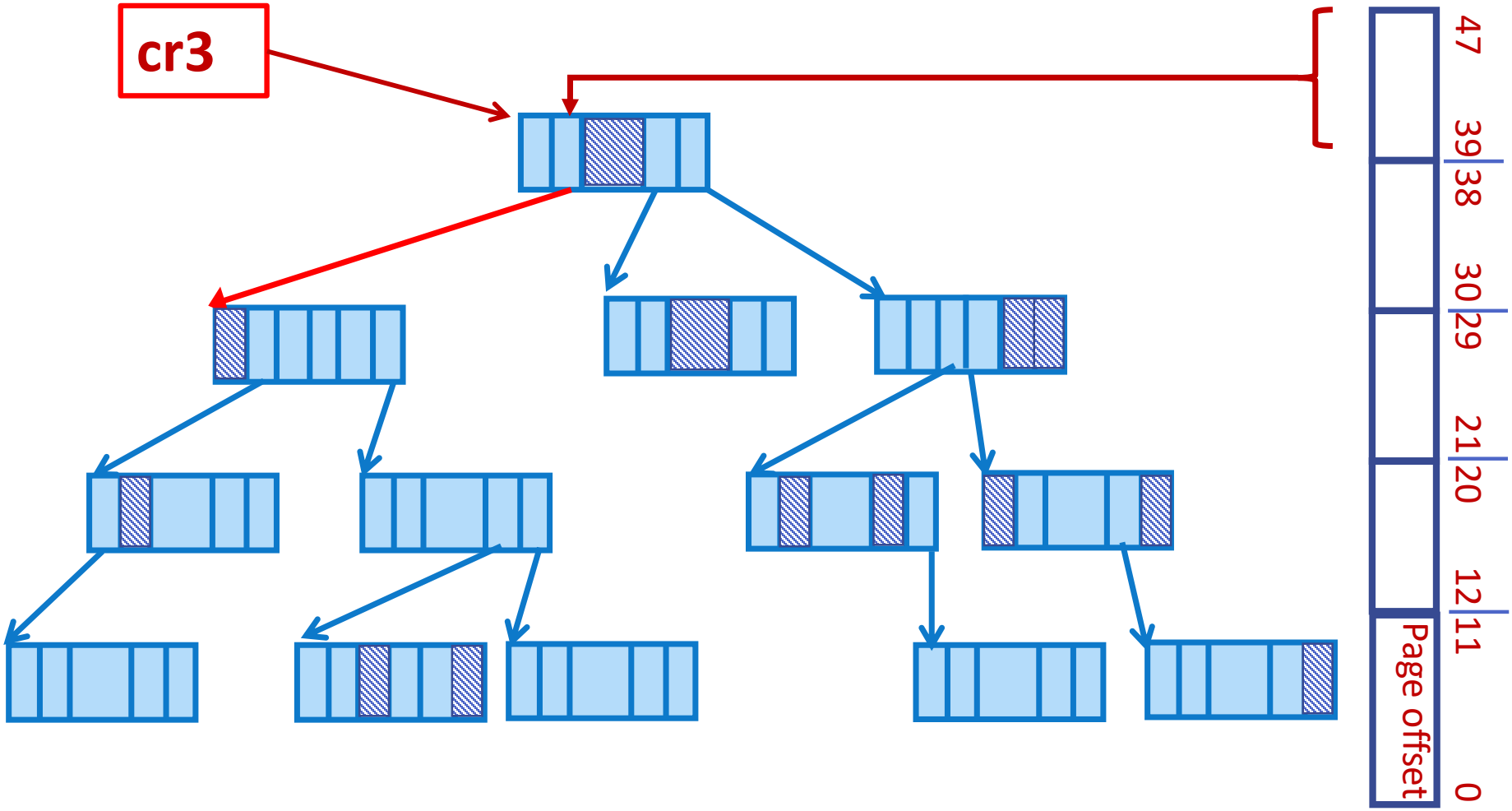
Paging: 64bit x86 page *walk*



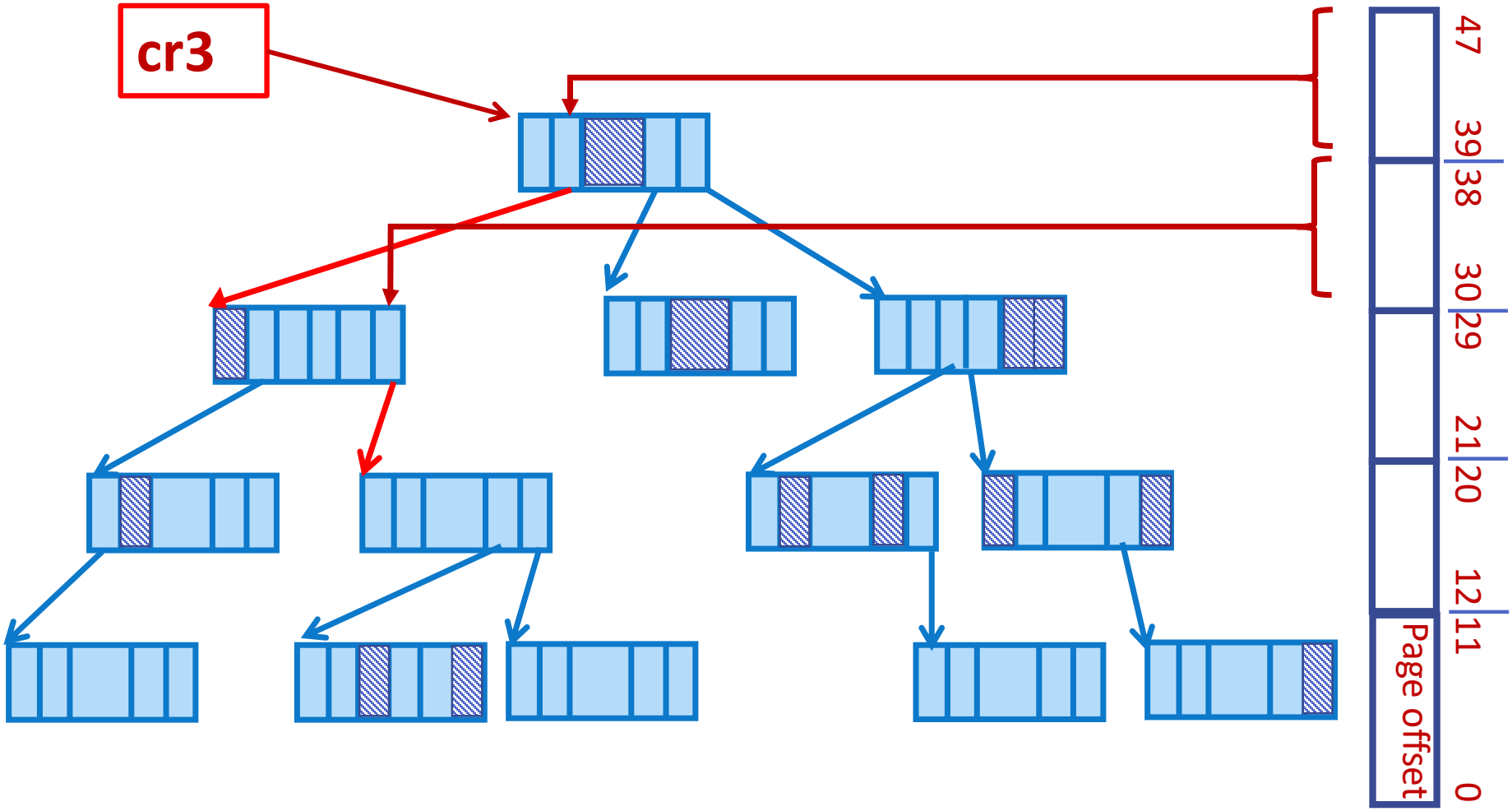
Paging: 64bit x86 page *walk*



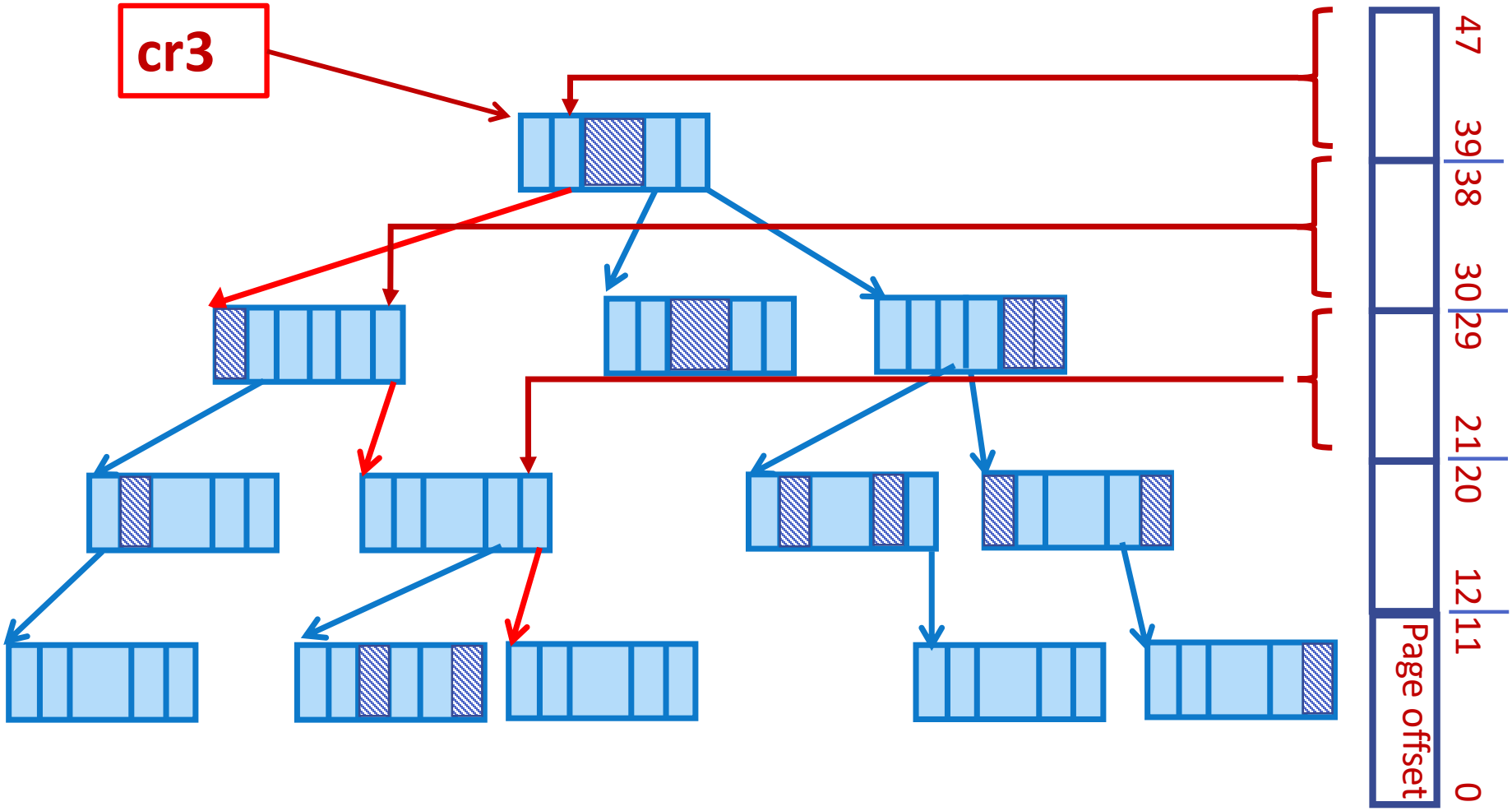
Paging: 64bit x86 page *walk*



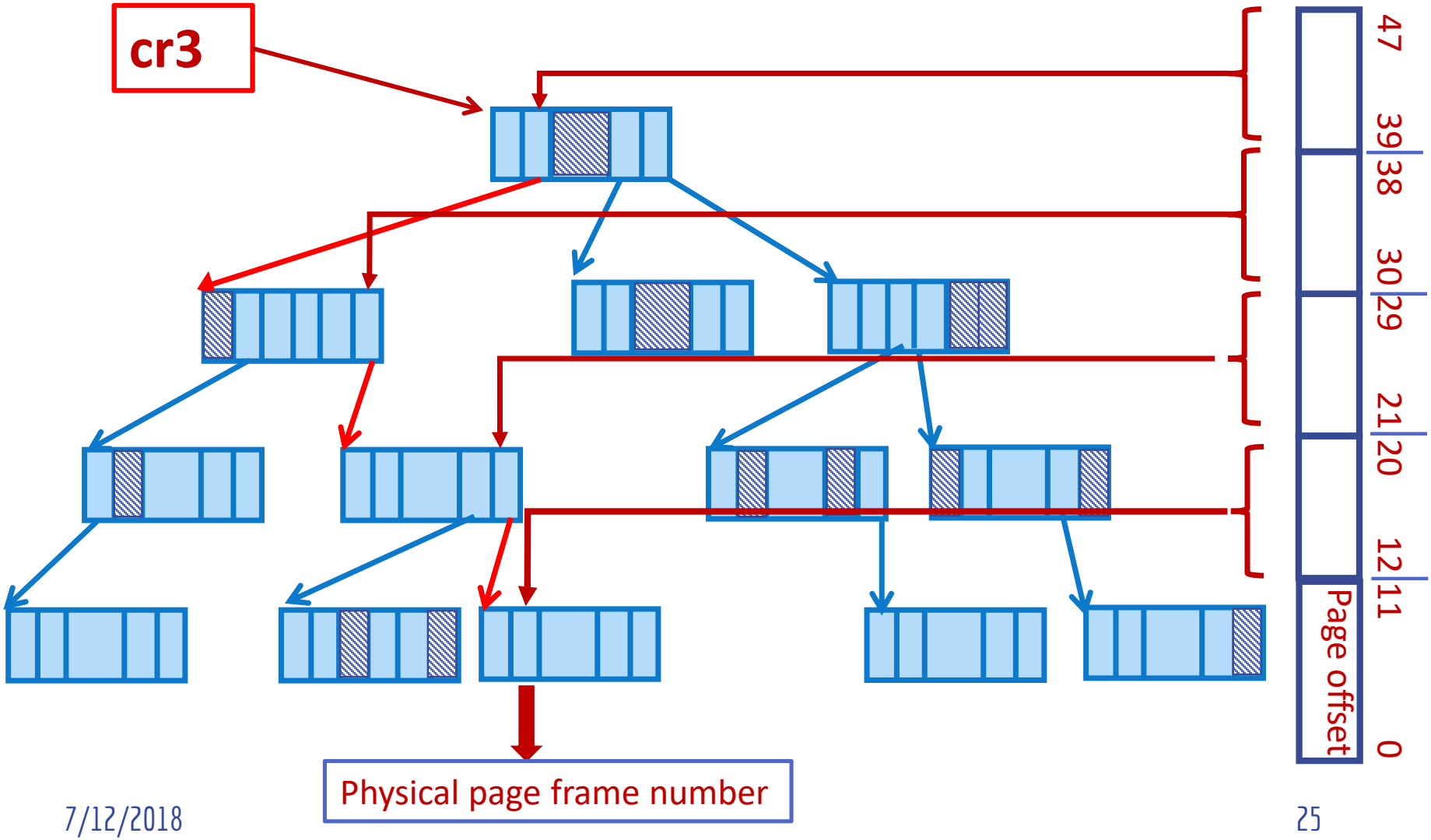
Paging: 64bit x86 page *walk*



Paging: 64bit x86 page *walk*



Paging: 64bit x86 page *walk*



Who walks the page table?

- A hardware page table walker (PTW) walks the page table
  - Input: Root of page table (cr3) and VPN
  - Output: Physical page frame number or fault
  - A hardware fine-state-automata in each CPU core

Who walks the page table?

- A hardware page table walker (PTW) walks the page table
  - Input: Root of page table (cr3) and VPN
  - Output: Physical page frame number or fault
  - A hardware fine-state-automata in each CPU core
  - Generates load-like “instructions” to access page table
  - Typical in x86 and ARM processors

Who walks the page table?

- A hardware page table walker (PTW) walks the page table
  - Input: Root of page table (cr3) and VPN
  - Output: Physical page frame number or fault
  - A hardware fine-state-automata in each CPU core
  - Generates load-like “instructions” to access page table
  - Typical in x86 and ARM processors
- 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
  - Input: Root of page table (cr3) and VPN
  - Output: Physical page frame number or fault
  - A hardware fine-state-automata in each CPU core
  - Generates load-like "instructions" to access page table
  - Typical in x86 and ARM processors
- 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 up to 4 memory accesses!

TLB: Making page walks faster

- **Disadvantage:** A single address translation can take up to 4 memory accesses!

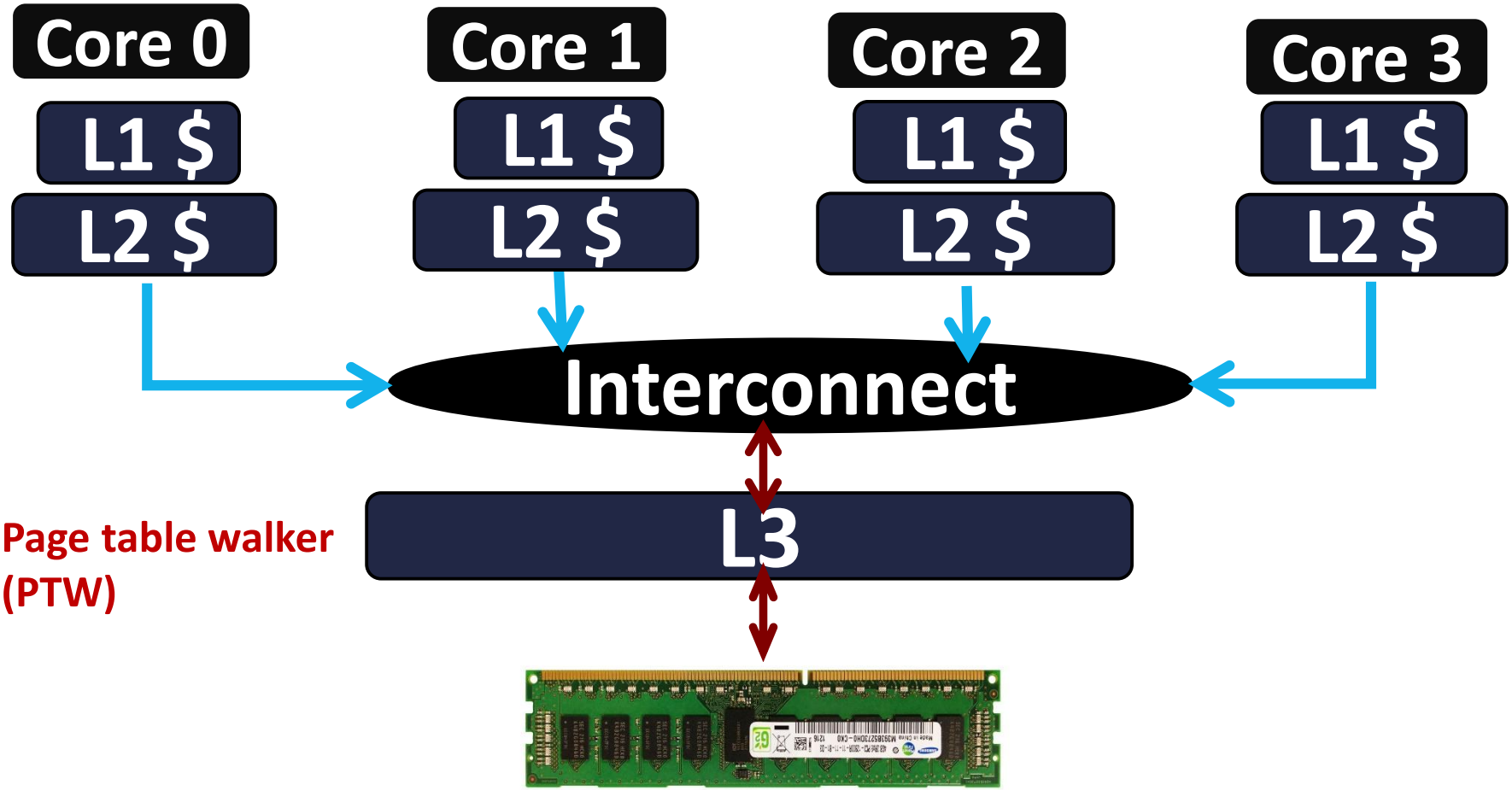
- How to make address translation fast?
  - **Translation Lookaside Buffer** or TLB to cache recently used virtual to physical address mappings

TLB: Making page walks faster

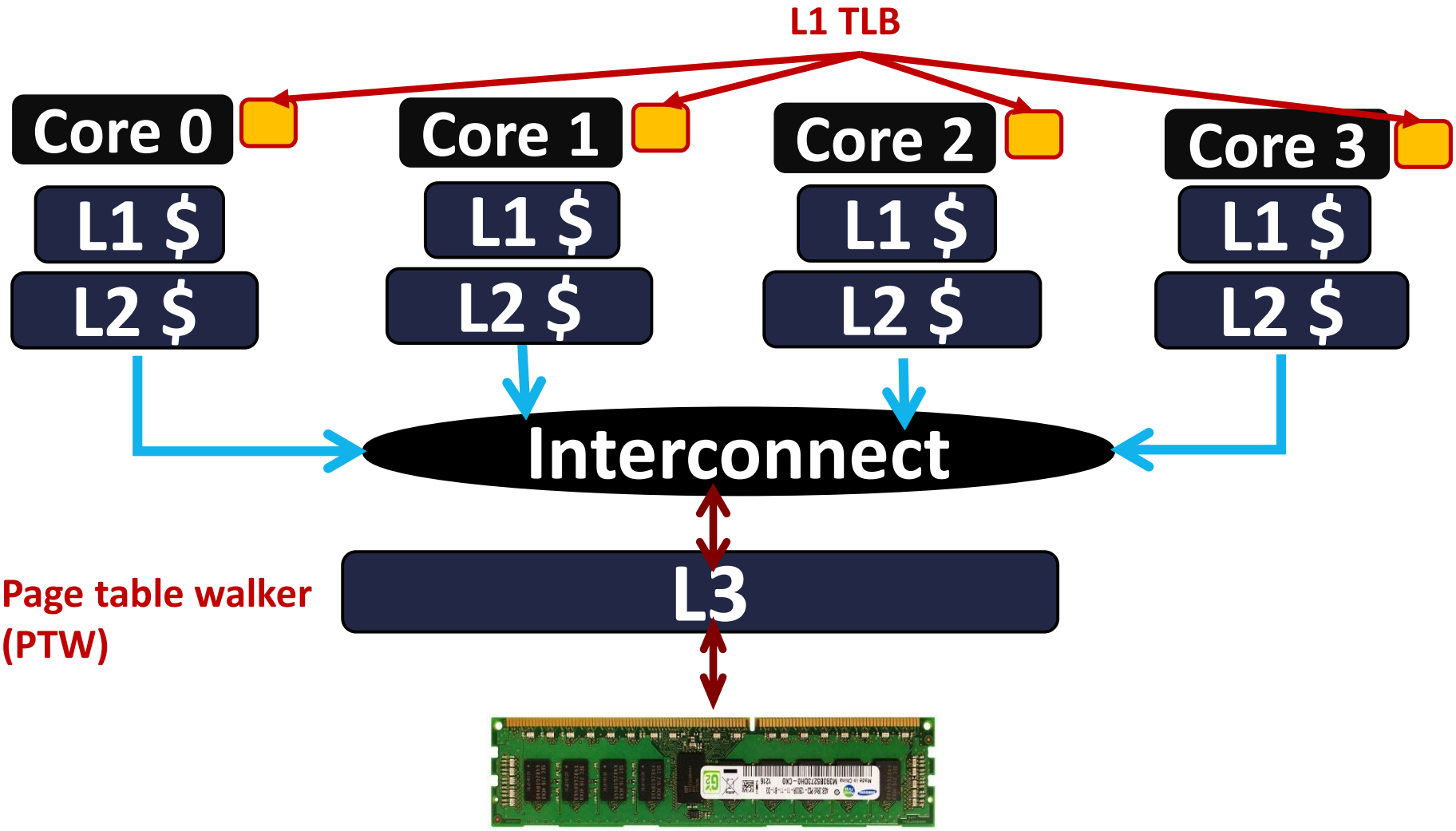
- **Disadvantage:** A single address translation can take up to 4 memory accesses!

- 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 page table entries
  - For address translation, TLB is first looked up
  - On a TLB miss page walk is performed
  - A TLB hit is fast but page walk is slow

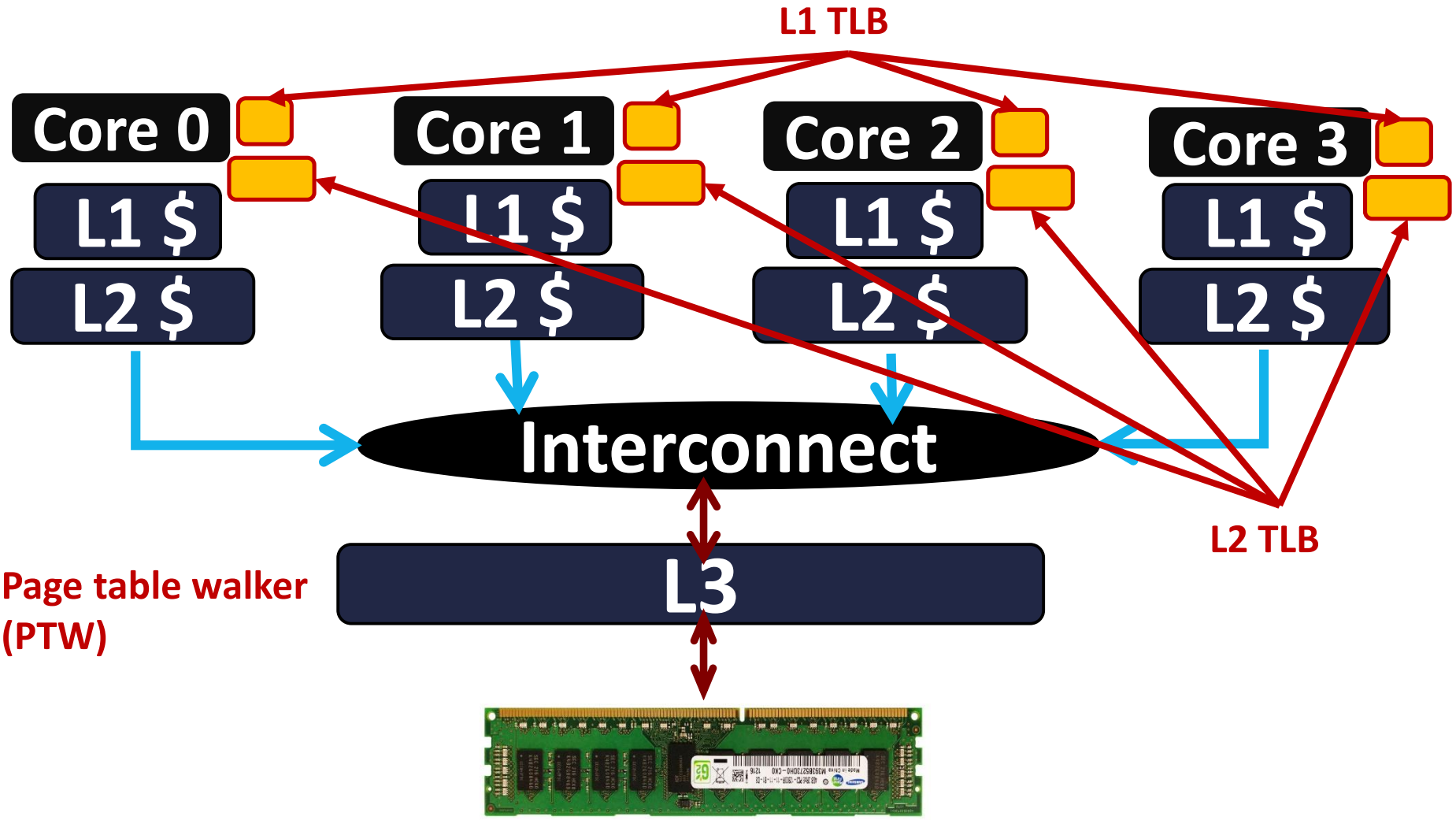
Where are TLBs & Page table walkers?



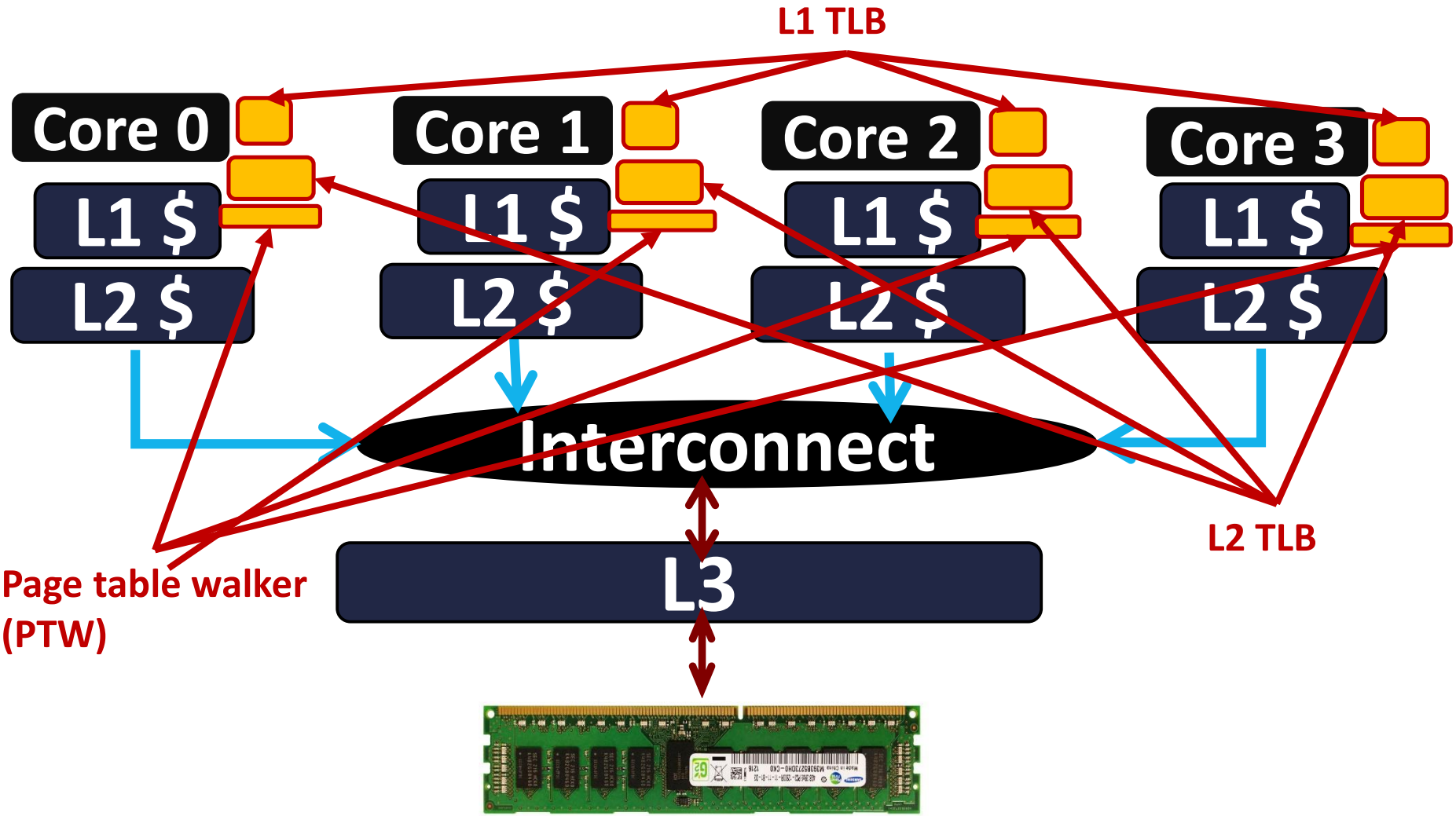
Where are TLBs & Page table walkers?



Where are TLBs & Page table walkers?



Where are TLBs & Page table walkers?



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)
  - One to two page table walker

Putting it all together

- Steps of address translation
  - Load/store instructions carries virtual address
  - Virtual address divided into virtual page number (VPN) and page offset

Putting it all together

- Steps of address translation
  - Load/store instructions carries virtual address
  - Virtual address divided into virtual page number (VPN) and page offset
  - TLBs are looked for the desired VPN to page frame number mapping (PFN)
  - On a hit, VPN is concatenated with offset and cache is looked up using the physical address

Putting it all together

- Steps of address translation
  - Load/store instructions carries virtual address
  - Virtual address divided into virtual page number (VPN) and page offset
  - TLBs are looked for the desired VPN to page frame number mapping (PFN)
  - On a hit, VPN is concatenated with offset and cache is looked up using the physical address
  - On a miss, PTW starts walking the page table to get desired PFN
  - If found, the address mapping is returned to the TLB

Putting it all together

- Steps of address translation
  - Load/store instructions carries virtual address
  - Virtual address divided into virtual page number (VPN) and page offset
  - TLBs are looked for the desired VPN to page frame number mapping (PFN)
  - On a hit, VPN is concatenated with offset and cache is looked up using the physical address
  - On a miss, PTW starts walking the page table to get desired PFN
  - If found, the address mapping is returned to the TLB
  - If there exits no entry for the VPN in the page table then page fault is raised to the OS

Miscellaneous paging topic

Superpages: Reducing TLB misses

- TLB miss are long latency operation
  - Address translation overhead == ~ TLB miss overhead
- TLB reach = # of TLB entries ✖ page size
  - Higher TLB reach → (Typically) Lower TLB miss rate

Superpages: Reducing TLB misses

- TLB miss are long latency operation
  - Address translation overhead == ~ TLB miss overhead
- TLB reach = # of TLB entries ✖ page size
  - Higher TLB reach → (Typically) Lower TLB miss rate
- How to increase TLB reach?
  - Increase TLB size.
    - Not always possible due to h/w overhead
  - Increase page size
    - Increases internal fragmentation

Superpages: Reducing TLB misses

- TLB miss are long latency operation
  - Address translation overhead == ~ TLB miss overhead
- TLB reach = # of TLB entries ✖ page size
  - Higher TLB reach → (Typically) Lower TLB miss rate
- How to increase TLB reach?
  - Increase TLB size.
    - Not always possible due to h/w overhead
  - Increase page size
    - Increases internal fragmentation
  - Have multiple page sizes
    - Larger page sizes beyond base page size (e.g., 4KB)

Superpages: Reducing TLB misses

- Larger page sizes called superpages
- Example page sizes in x86-64 (Intel, AMD) machines
  - 4KB (base page size), 2MB, 1GB (superpages)

Superpages: Reducing TLB misses

- Larger page sizes called superpages
- Example page sizes in x86-64 (Intel, AMD) machines
  - 4KB (base page size), 2MB, 1GB (superpages)
- **Challenge**: How to decide which page size to use for mapping a given virtual address?
  - Use of larger pages increase internal fragmentation → memory bloat
  - Using smaller page size can increase address translation overhead

Superpages: Reducing TLB misses

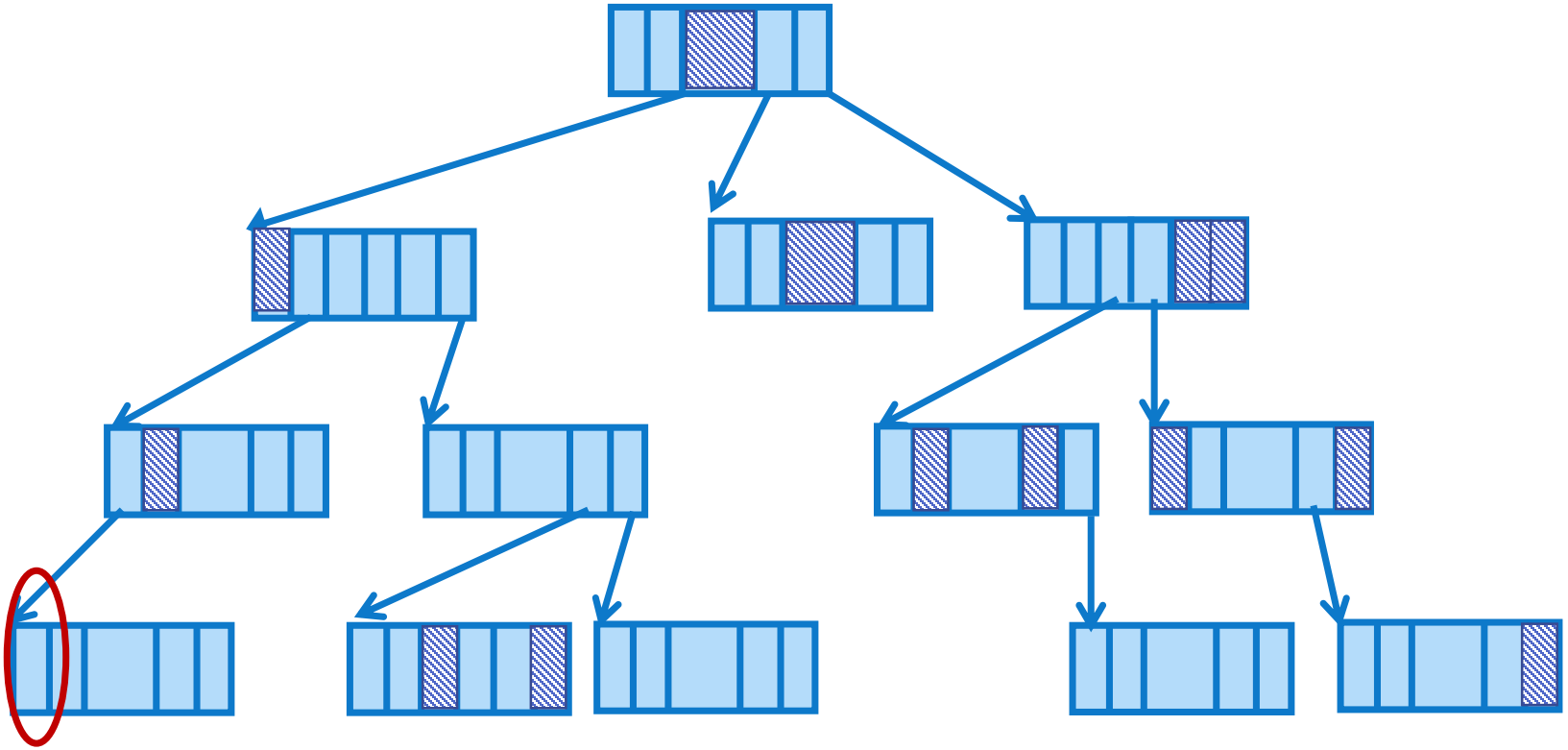
- Larger page sizes called superpages
- Example page sizes in x86-64 (Intel, AMD) machines
  - 4KB (base page size), 2MB, 1GB (superpages)

Will revisit

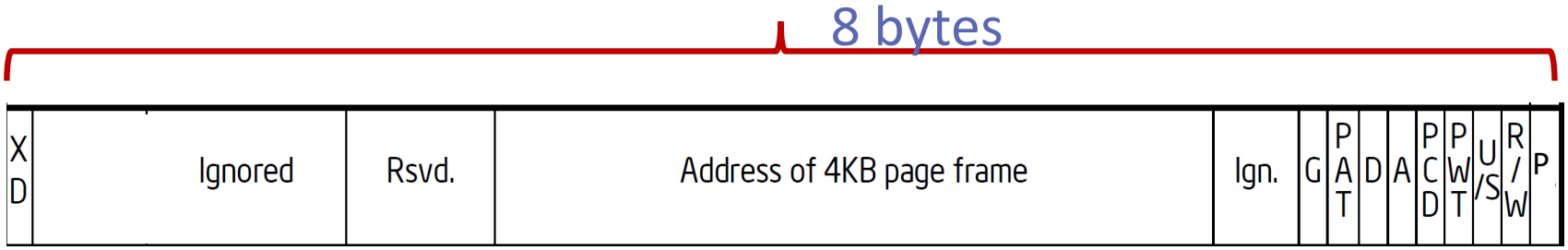
- **Challenge**: How to decide which page size to use for mapping a given virtual address?
  - Use of larger pages increase internal fragmentation → memory bloat
  - Using smaller page size can increase address translation overhead

Contents of a page table entry

4 level 512-ary radix tree



Contents of a 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?

Segmentation with paging

- 32-bit x86 machines allowed segmentation on top of paging
  - Virtual address is first translated via segment registers (CS,DS,ES etc.) to linear address
  - Linear address is then translated to physical address
  - Why?

Segmentation with paging

- 32-bit x86 machines allowed segmentation on top of paging
  - Virtual address is first translated via segment registers (CS,DS,ES etc.) to linear address
  - Linear address is then translated to physical address
  - Why?
- 64-bit x86 mostly got rid of segmenation

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

Journey so far….Next up

Mostly H/W

- Virtual memory basics
- Address mapping and translation

Mostly S/W

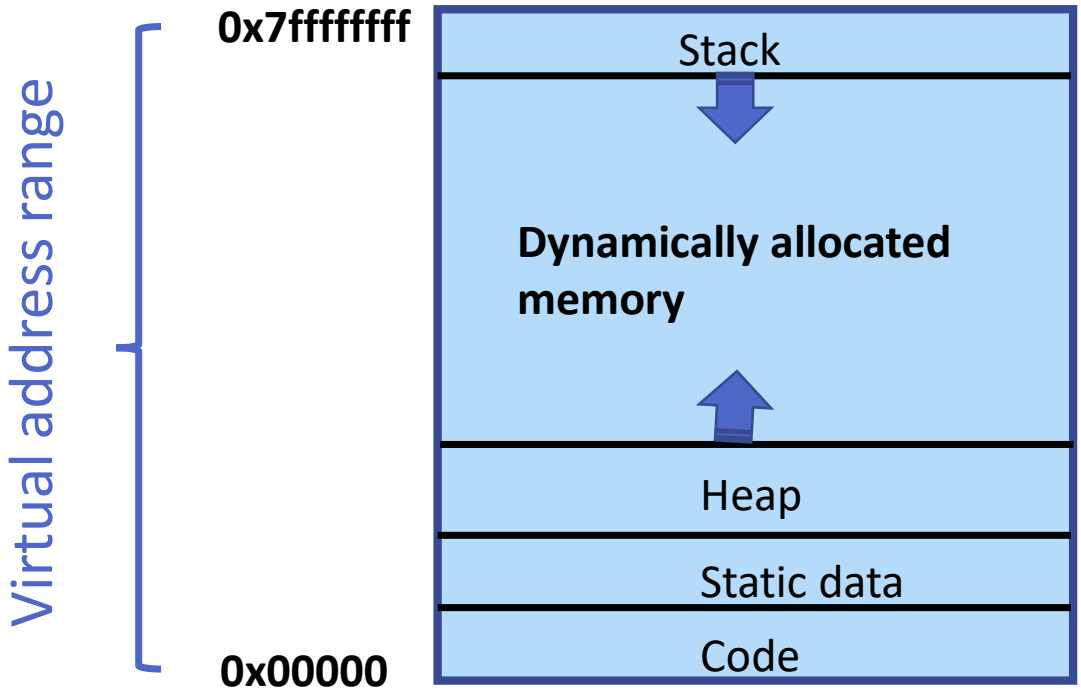
- Virtual address allocation
- Physical memory allocation and page table creation

Virtual address allocation

- "Memory allocation" is actually virtual address allocation
- Operating system is responsible for allocating virtual address for an application

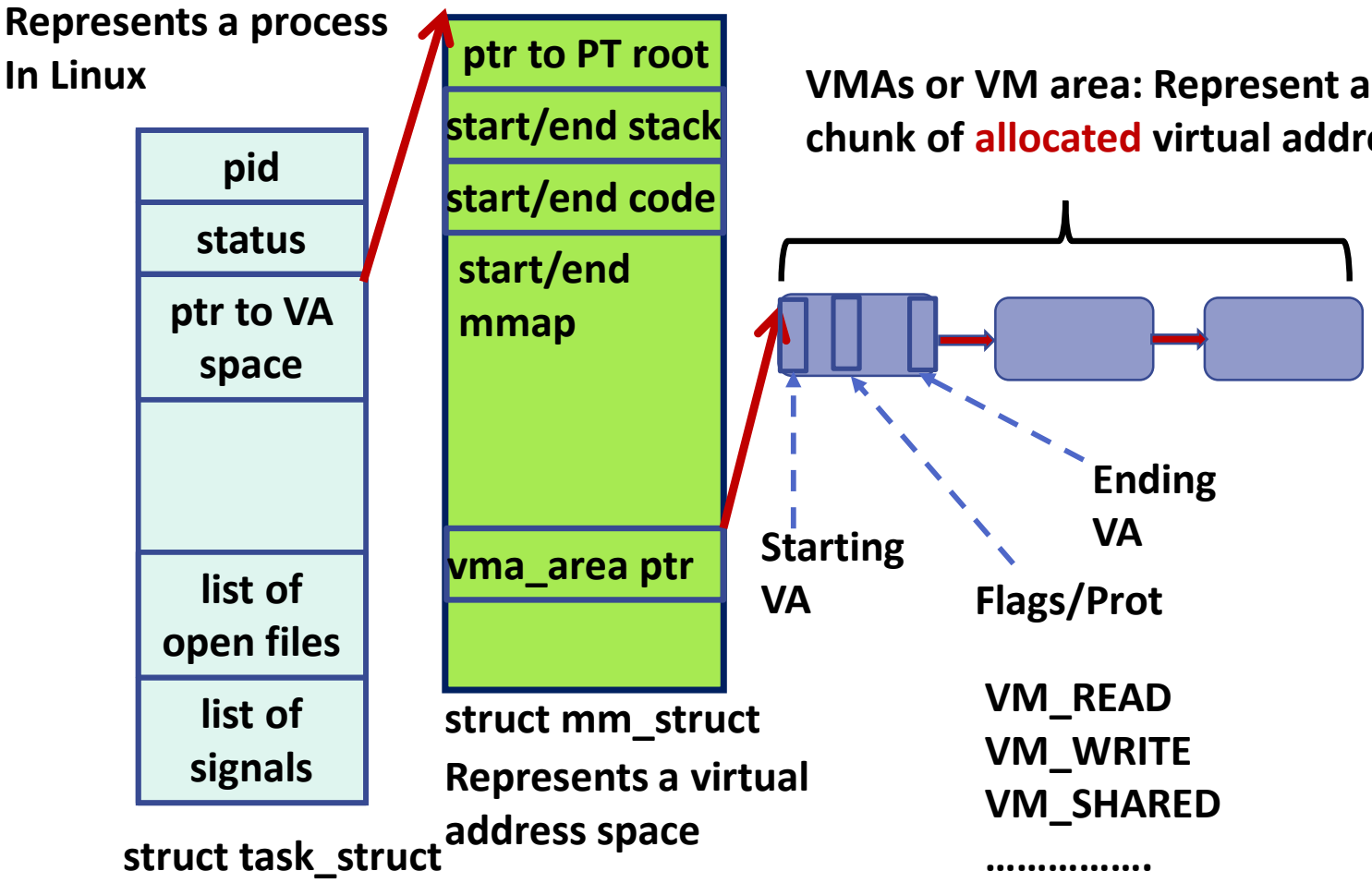
Virtual address allocation

- "Memory allocation" is actually virtual address allocation
- Operating system is responsible for allocating virtual address for an application



Typical virtual address layout of a process in Linux

Representation of VA (in Linux)



Dynamically allocating VA

- User application or library requests VA allocation via system calls

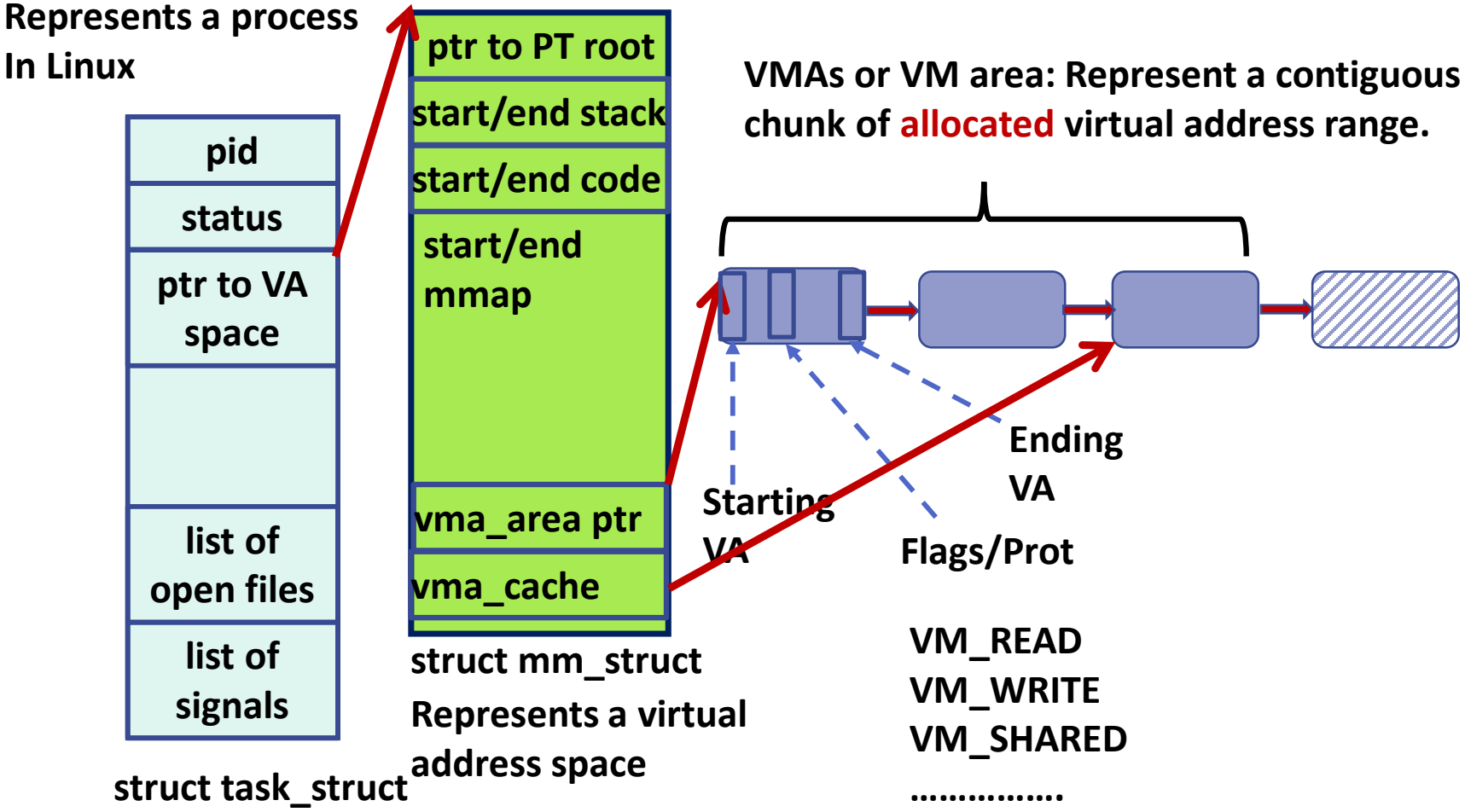
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

Length has to be multiple of 4KB

Prot ➔ PROT_NONE, PROT_READ, PROT_WRITE…

Flags➔ MAP_ANONYMOUS, MAP_SHARED, MAP_PRIVATE, MAP_SHARED

What happens on dynamic memory allocation (e.g., mmap())?

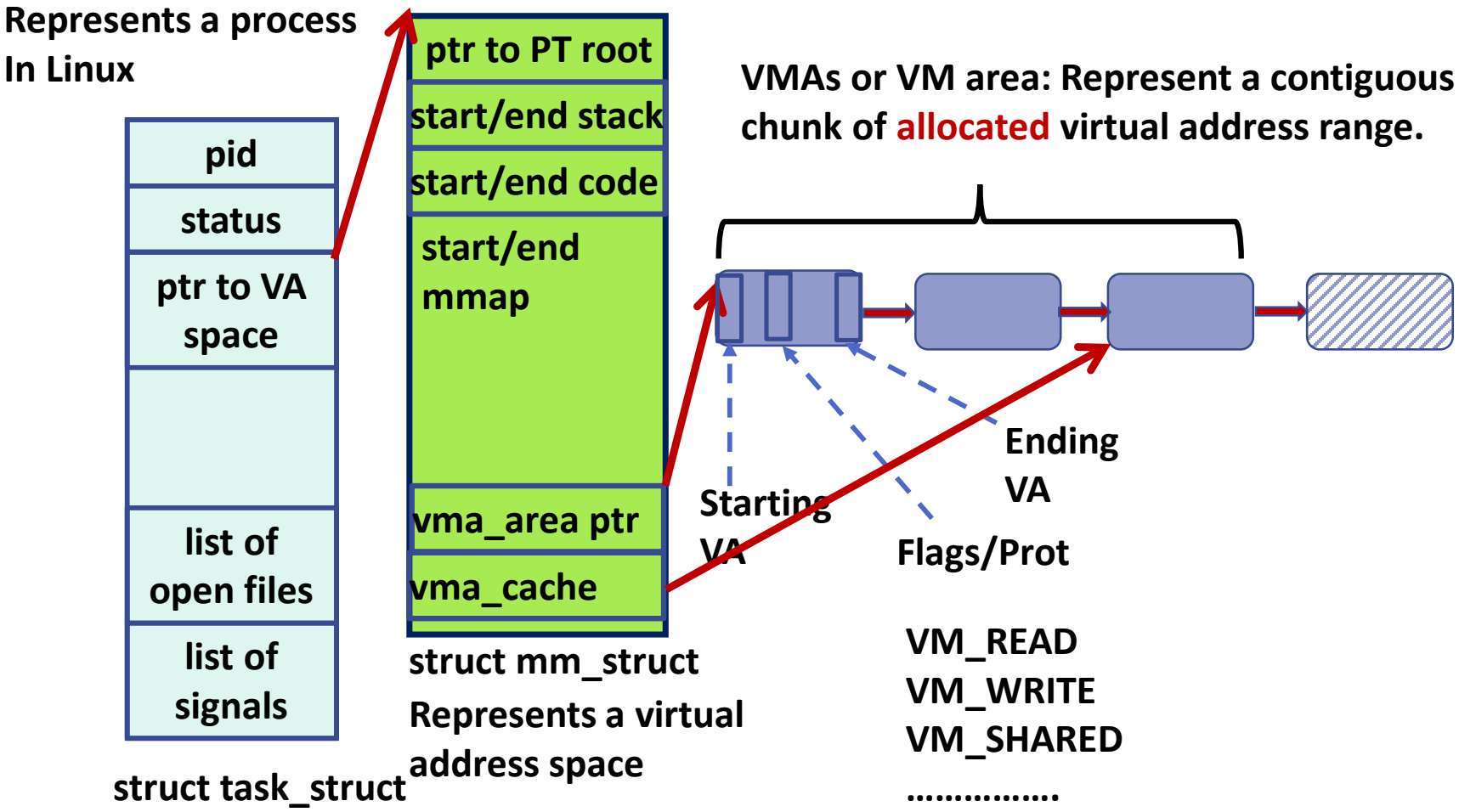


Extending heap memory

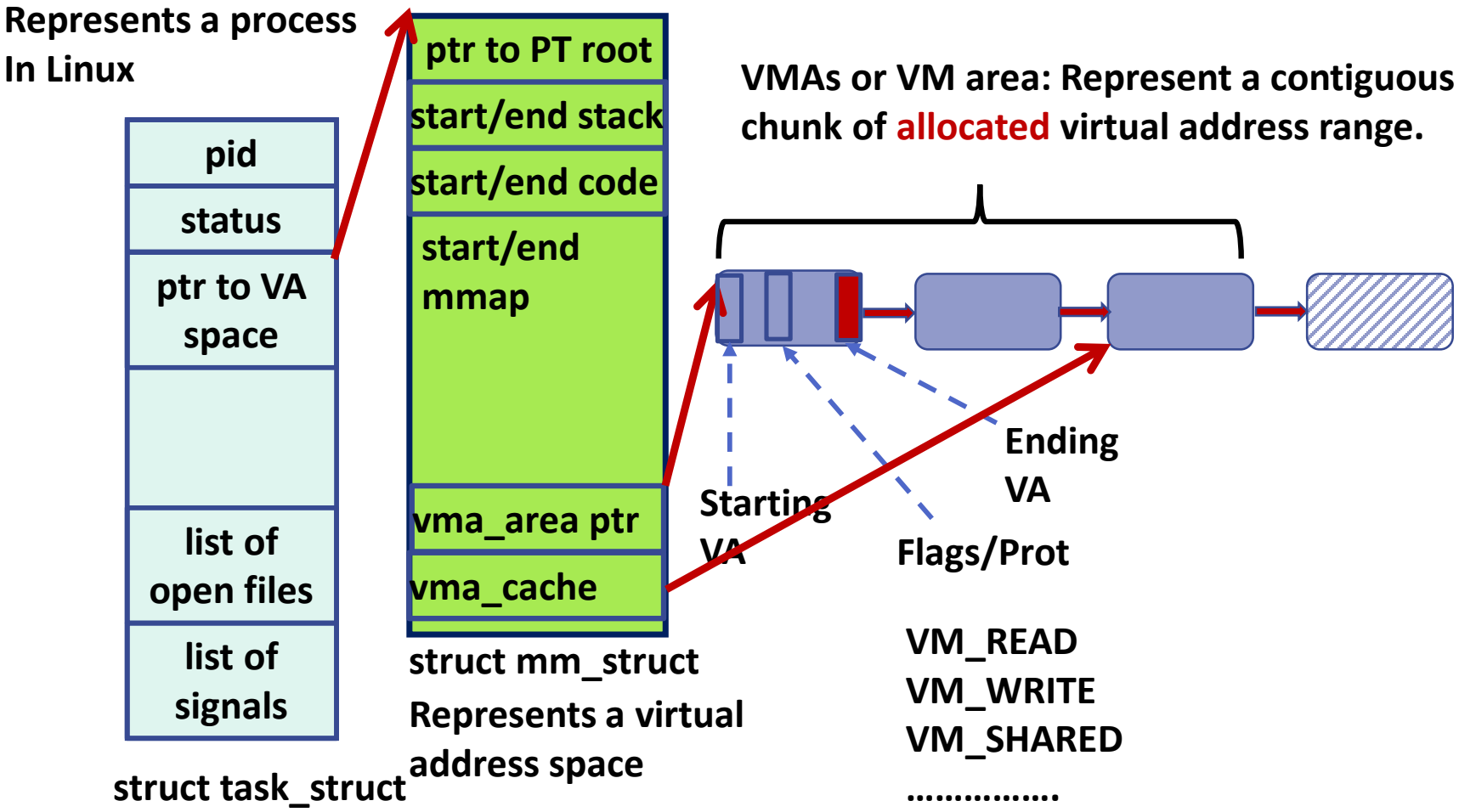
- **Heap**: Special type of dynamically allocated contiguous memory region that grows in upwards

- System calls in Linux to extend heap
  - int sbrk (increment _bytes)

Extending heap via *sbrk*



Extending heap via *sbrk*



Demand paging

- Note when “memory” (a.k.a, VA) is allocated no physical memory is allocated

Demand paging

- Note when “memory” (a.k.a, VA) is allocated no physical memory is allocated
- Why? Virtual address is in abundance; but physical memory is scarce resource.
- Allocate physical memory for when the virtual address is accessed first time

Demand paging

- Note when “memory” (a.k.a, VA) is allocated no physical memory is allocated
- Why? Virtual address is in abundance; but physical memory is scarce resource.
- Allocate physical memory for when the virtual address is accessed first time
- Lazily allocating physical memory is called *demand paging*
- Advantage: Commits physical memory only if used

Page fault in demand paging

- **Page fault:** When h/w page walker fails to find desired PTE the processor generates a general protection fault to the OS

Page fault in demand paging

- **Page fault:** When h/w page walker fails to find desired PTE the processor generates a general protection fault to the OS
- First access to an allocated memory generates a page fault

Page fault in demand paging

- **Page fault:** When h/w page walker fails to find desired PTE the processor generates a general protection fault to the OS
- First access to an allocated memory generates a page fault
- Page fault can also happen due to insufficient permission (e.g., write access to read-only page) → protection fault

Servicing a page fault

- Page fault routine is implemented inside the OS
- Argument routine are faulting VA and type of access (e.g., read or write)

Servicing a page fault

- Page fault routine is implemented inside the OS
- Argument routine are faulting VA and type of access (e.g., read or write)
- Steps of handling a page fault

Servicing a page fault

- Page fault routine is implemented inside the OS
- Argument routine are faulting VA and type of access (e.g., read or write)
- Steps of handling a page fault
  - Check VMA structures if the VA is allocated
  - If not allocated or insufficient permission, raise segmentation fault to application
  - If allocated with correct permission find a physical page frame to map the page containing the faulting VA
  - Update page table to note new VA->PA mapping
  - Return

Servicing a page fault

- Page fault routine is implemented inside the OS
- Argument routine are faulting VA and type of access (e.g., read or write)
- Steps of handling a page fault
  - Check VMA structures if the VA is allocated
  - If not allocated or insufficient permission, raise segmentation fault to application
  - If allocated with correct permission find a physical page frame to map the page containing the faulting VA
  - Update page table to note new VA->PA mapping
  - Return
- The faulting instruction retries after page fault returns

Servicing a page fault

- Page fault routine is implemented inside the OS
- Argument routine are faulting VA and type of access (e.g., read or write)
- Steps of handling a page fault
  - Check VMA structures if the VA is allocated
  - If not allocated or insufficient permission, raise segmentation fault to application
  - If allocated with correct permission find a physical page frame to map the page containing the faulting VA
  - Update page table to note new VA->PA mapping
  - Return
- The faulting instruction retries after page fault returns

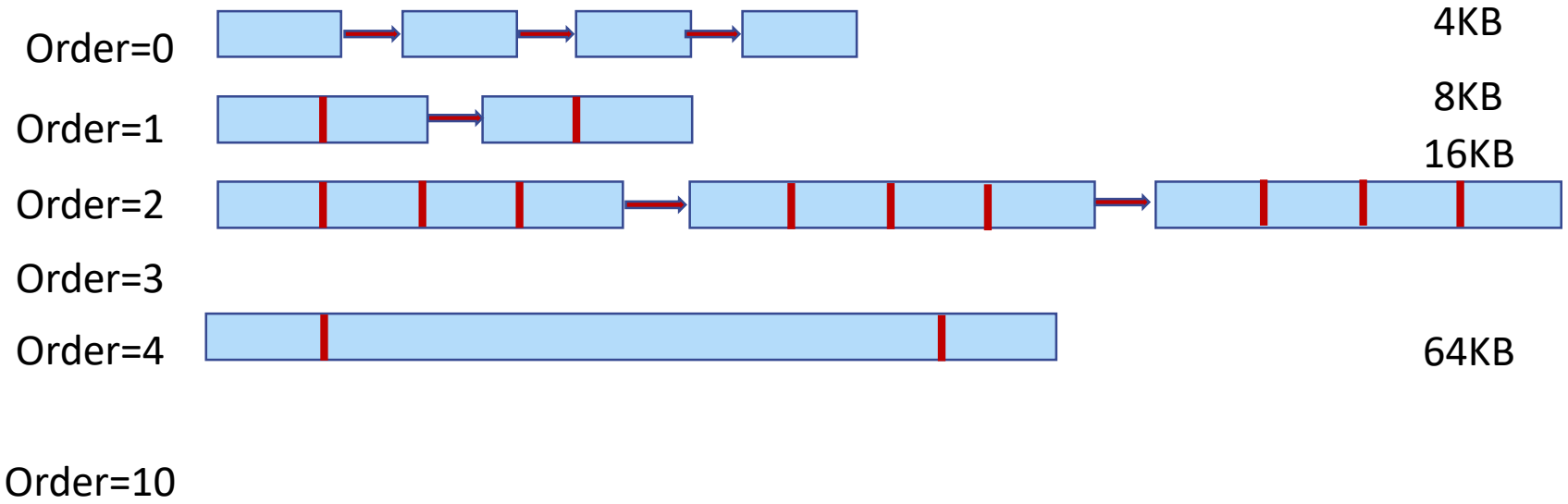
Next

Allocating physical memory

- Buddy allocator in Linux: Goal is to keep free physical memory as contiguous as possible (why?)
- It is a list of free list of contiguous physical pages of different sizes (2^{order} x 4KB)

Allocating physical memory

- Buddy allocator in Linux: Goal is to keep free physical memory as contiguous as possible (why?)
- It is a list of free list of contiguous physical pages of different sizes (2^{order} x 4KB)



Buddy allocator operation

- Allocate from a free list which is smallest that fits the requested allocation size
  - If no entry in the smallest list, go to next larger list and so on…
  - Put the leftover blocks in the lower order list

Buddy allocator operation

- Allocate from a free list which is smallest that fits the requested allocation size
  - If no entry in the smallest list, go to next larger list and so on…
  - Put the leftover blocks in the lower order list
- Merge two contiguous blocks of physical memory in a free list and add the merged block in next higher order free list

Where does *malloc()* fits in?

- *malloc()* is function call implemented in a library. It is **not** part of OS.
- malloc allocates virtual address range (like mmap())

Where does *malloc()* fits in?

- *malloc()* is function call implemented in a library. It is **not** part of OS.
- malloc allocates virtual address range (like mmap())
- Then, why malloc()/free()?
- Limitation of mmap
  - Minimum granularity of VA allocation is a page (4KB)
  - But applications often allocates in chunks less than 4KB

Where does *malloc()* fits in?

- *malloc()* is function call implemented in a library. It is **not** part of OS.
- malloc allocates virtual address range (like mmap())
- Then, why malloc()/free()?
- Limitation of mmap
  - Minimum granularity of VA allocation is a page (4KB)
  - But applications often allocates in chunks less than 4KB
- malloc() maintains free list of small allocations
- If free memory is available in malloc()'s free list, no need to got to the OS
- malloc() with large size (e.g., >32KB) converts to mmap

Where does *malloc() fits in*?

- Two key goals of a malloc library:
  - Reduce memory bloat = additional allocated memory than what application asked
  - Reduce number of system calls to OS (e.g., mmap())
  - System calls are slow

Where does *malloc() fits in*?

- Two key goals of a malloc library:
  - Reduce memory bloat = additional allocated memory than what application asked
  - Reduce number of system calls to OS (e.g., mmap())
  - System calls are slow
- Many approaches to create malloc library
  - Best fit, first fit, worst fit
- Many malloc() libraries
  - glibc-malloc, tc-malloc, dl-malloc
- You can write your own!

Putting everything together

1. Application requests VA allocation via mmap()
2. OS creates/extends VMA regions to allocate VA range
3. OS returns the starting VA of just-allocated VA range

Putting everything together

1. Application requests VA allocation via mmap()
2. OS creates/extends VMA regions to allocate VA range
3. OS returns the starting VA of just-allocated VA range
4. Application performs load/store on the address in the VA range
5. H/W looks up TLB; misses (Why?)
6. H/W page table walker walks the page table

Putting everything together

1. Application requests VA allocation via mmap()
2. OS creates/extends VMA regions to allocate VA range
3. OS returns the starting VA of just-allocated VA range
4. Application performs load/store on the address in the VA range
5. H/W looks up TLB; misses (Why?)
6. H/W page table walker walks the page table
7. Desired entry not found on PTE (why?)
8. H/W generates a page fault to OS

Putting everything together

9. OS'a page fault handler checks VMA regions to ensure it's a legal access
10. Page fault handler finds a free physical page frame to map the VA page from the buddy allocator
11. Updates page table to note new VA to PA mapping and return

Putting everything together

9. OS'a page fault handler checks VMA regions to ensure it's a legal access
10. Page fault handler finds a free physical page frame to map the VA page from the buddy allocator
11. Updates page table to note new VA to PA mapping and return
12. Application retries the same instruction
13. This time page table walker finds it in page table and loads it into TLB
14. Next time if same page is accessed it may hit in TLB (→ no page walk)

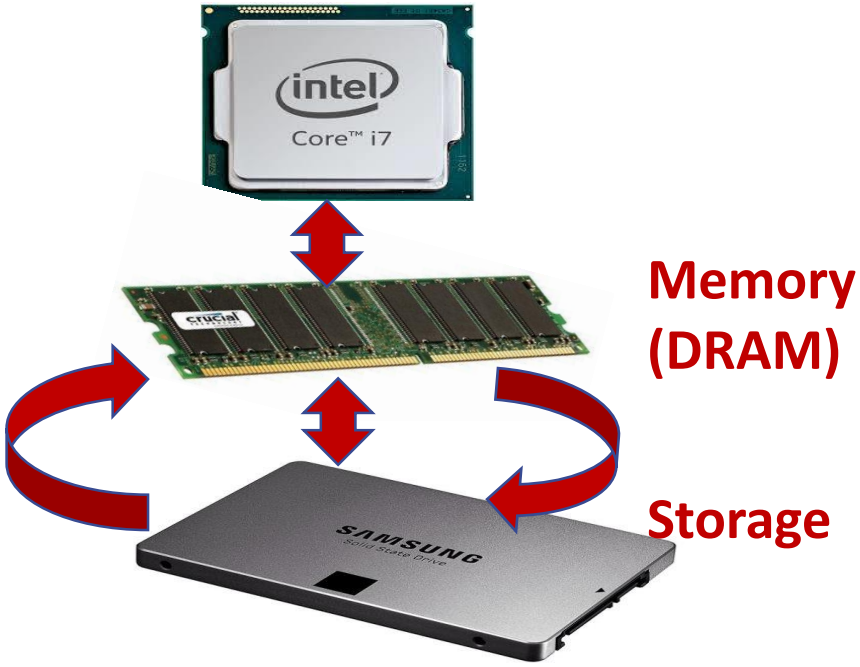
Miscellaneous related topics

Swapping

- **Goal**: Provide an illusion of larger memory than actually available

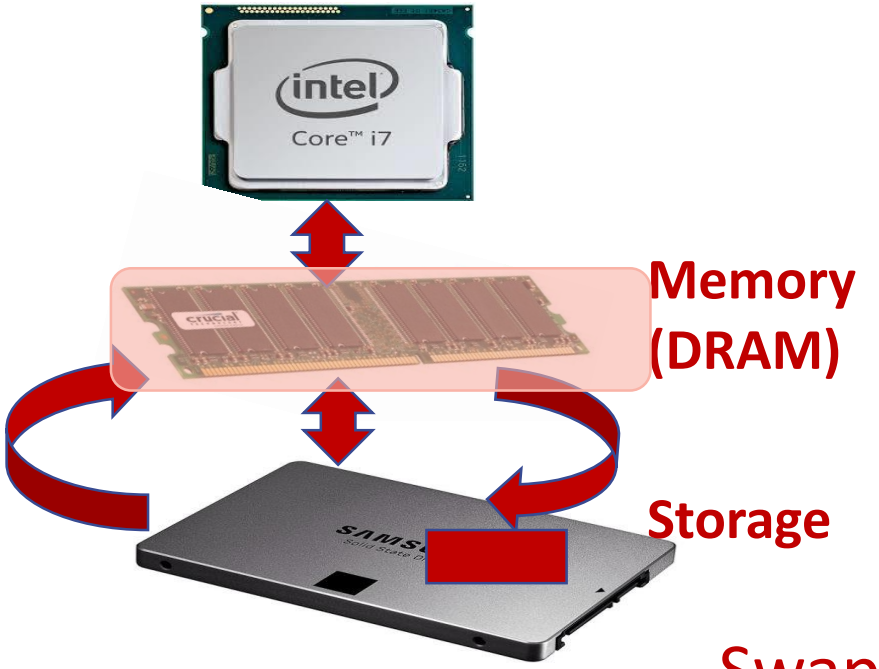
Swapping

- **Goal**: Provide an illusion of larger memory than actually available



Swapping

- **Goal**: Provide an illusion of larger memory than actually available



How swapping work?

- OS attempts to figure out which pages in memory are not actively used
  - Use access bit ("A" bit) in the PTE
  - OS periodically unsets "A" bits of pages in memory
  - After a little while, OS checks H/W has set "A" bit of those pages

How swapping work?

- OS attempts to figure out which pages in memory are not actively used
  - Use access bit ("A" bit) in the PTE
  - OS periodically unsets "A" bits of pages in memory
  - After a little while, OS checks H/W has set "A" bit of those pages
  - If "A" bit set for a page → actively used page
  - If "A" bit is unset for a page → not actively used → candidate for swapped out to storage

How swapping work?

- OS attempts to figure out which pages in memory are not actively used
  - Use access bit ("A" bit) in the PTE
  - OS periodically unsets "A" bits of pages in memory
  - After a little while, OS checks H/W has set "A" bit of those pages
  - If "A" bit set for a page → actively used page
  - If "A" bit is unset for a page → not actively used → candidate for swapped out to storage
- OS writes back data of the page to swap file
- Update PTE to unset present ("p") bit

How swapping work?

- There will be page fault if an application accesses a page that is swapped out
  - Because “p” bit is unset
- OS page fault handler figures out the page had been swapped out
- Page fault handler brings the page into memory
- Application retries the faulting instruction.

How to utilize of superpages?

- Recap: superpages increases TLB reach → decrease TLB miss rate
  - x86-64 page sizes: 4KB (base), 2MB, 1GB(superpages)
- **Challenge**: Which page size to use for mapping a given virtual address?
- Approach 1: Application writer tells when to use superpage
  - Mmap() syscall has flag MAP_2MB, MAP_1GB

How to utilize superpages?

- Approach 2: Let OS automatically decide which page size to use.
  - Advantage: No application modification required
  - Challenge: How should OS decide?

How to utilize superpages?

- Approach 2: Let OS automatically decide which page size to use.
  - Advantage: No application modification required
  - Challenge: How should OS decide?
- Transparent Huge Pages (THP) in Linux
  - Large allocations (>2MB) mapped with superpage
  - Periodically scans VA space of processes to find contiguous allocations
  - Periodically compacts physical memory to create contiguous physical memory
  - Map contiguous VA range to contiguous physical memory

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

Extending virtual memory to accelerators

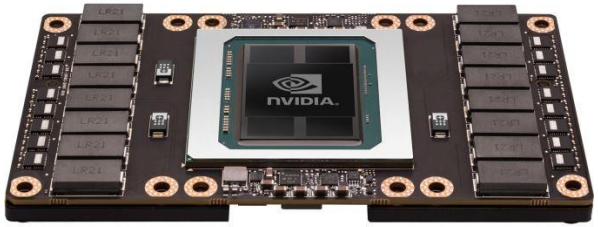
Accl. for massive parallel processing

NVDIA/AMD's GPU

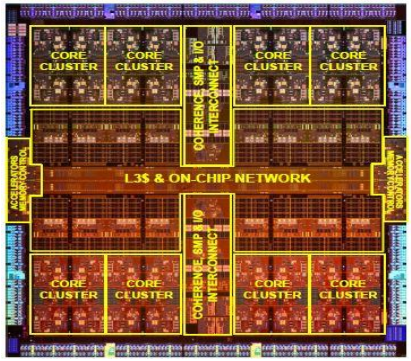
Extending virtual memory to accelerators

Accl. for cryptography

Accl. for massive parallel processing
NVDIA/AMD's GPU



Accl. For AI
Google's TPU, NVDIA's DGX-1



Accl. for database queries
Oracle's DAX

Extending virtual memory to accelerators

Accl. for cryptography

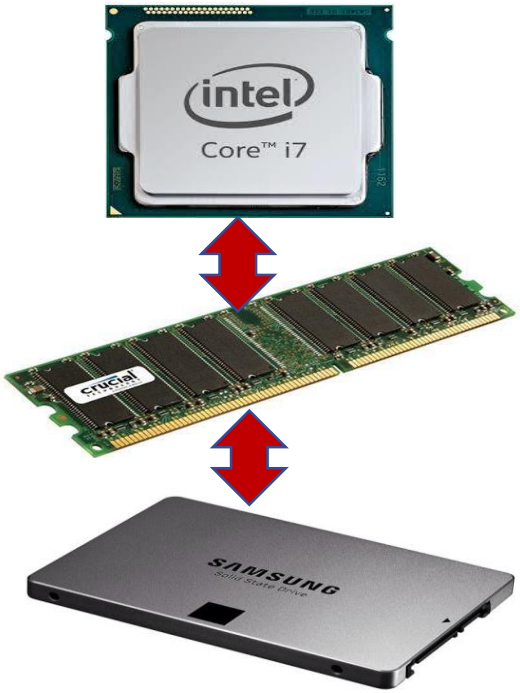
Accl. for massive parallel processing
NVDIA/AMD's GPU

How to efficiently extend benefits of virtual memory to accelerators?

Accl. For AI
Google's TPU, NVDIA's DGX-1

Accl. for database queries
Oracle's DAX

Virtual memory with Non volatile memory (NVM)



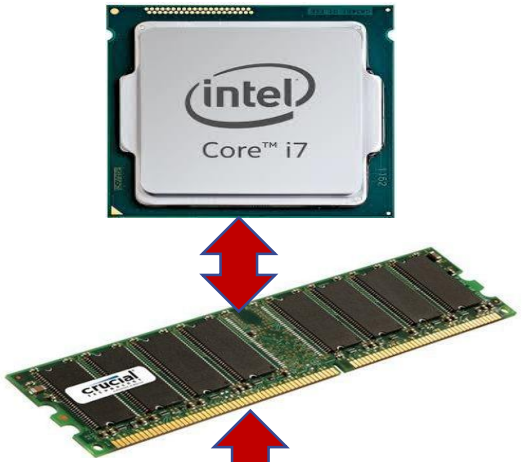
Memory (DRAM)

Storage

Traditional

Virtual memory with Non volatile memory (NVM)

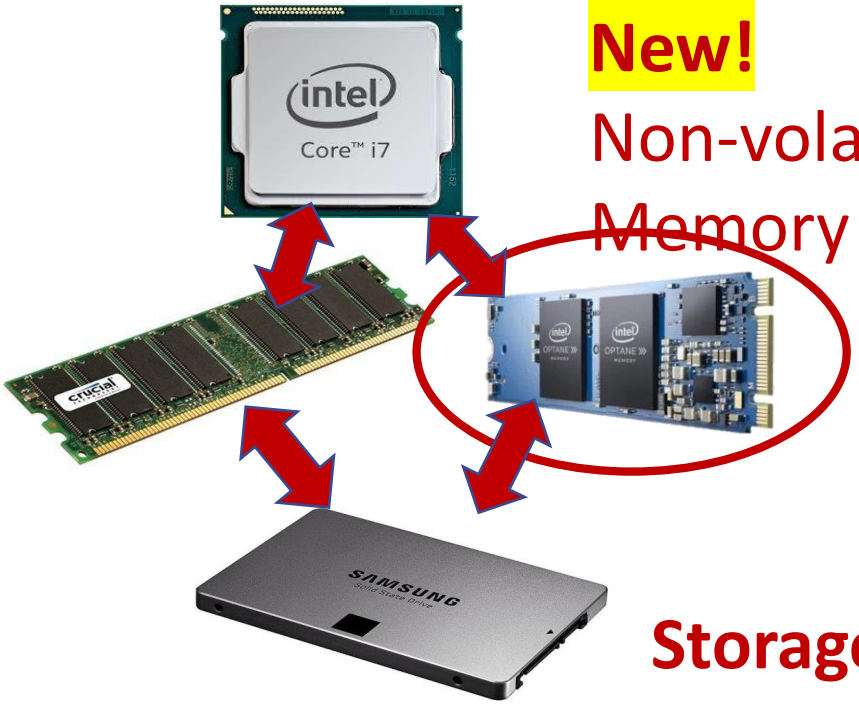
- NVM blurring boundary between memory and storage



Memory (DRAM)

Storage

Traditional



New!

Non-volatile ~~Memory~~

Storage

Emerging

Virtual memory with Non volatile memory (NVM)

- NVM blurring boundary between memory and storage

New!
Non-volatile Memory

How to incorporate notion on non-volatility in virtual memory?

Storage

Traditional

Storage

Emerging

Security implications of virtual memory

Security implications of virtual memory

- Speculative h/w state accessible to software

- OS's VA space attached to user's VA space

Security implications of virtual memory

- Speculative h/w state accessible to software

+

- OS's VA space attached to user's VA space

→ An user application can read OS's memory!

Security implications of virtual memory

Understanding and improving security implications of virtual memory

➡ An user application can read OS's memory!

Conclusion

- Virtual memory is a basic component of modern computing
- An example of true H/W-S/W co-design

- Many research opportunities!

Briefing for hands on session

- High level objectives
  1. Run a simple application (provided) with base pages only (4KB) and measure number of TLB misses, running time
  2. Run the same application with superpages and measure TLB misses and running time
     - A. Run with explicit application directed allocation of superpages
     - B. Run with Linux transparent huge pages (THP) that automatically uses superpages without application modification or pre-reservation of superpages
  3. Modify the given program/change its parameters to worsen number of TLB misses

Initial benchmark setup

- Download the compressed file from Google drive
- Unzip it [e.g., tar –xzvf microbenchmark.tar.gz]
- Go to directory benchmark/strided
- run make to compile
- Test that you can run the application by following command
  - ./strided 524288000 NONE STRIDED 256
  - This will take around 10 seconds or more to run
- Attached README explanation of the benchmark
- Take a look inside the strided benchmark

1 TLB misses with base page (4KB)

- Turn off Linux's transparent huge pages (could be on by default)
  - Switch to root privilege
    - Command "su – "
  - Command echo never > /sys/kernel/mm/transparent_hugepage/enabled
  - This will ensure your application will use 4KB base pages only
- Run the application using following command
  - ./strided 524288000 NONE STRIDED 256
  - Run at least three times and note down the average running time printed by the application

1 TLB misses with base page (4KB)

- Measure the number of TLB misses using perf tool
  - perf stat -e dTLB-load-misses ./strided 524288000 NONE STRIDED 256
  - Run at least three times and note down average dTLB misses (data TLB misses)

2A Explicit superpage allocation

- In this experiment the we will allocate superpages explicitly from applications
  - First, reserve superpages using following command
    - echo 2000 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
- Run the application to explicitly request superpages
  - ./strided 524288000 TLBFS STRIDED 256
  - Note down the execution time by averaging runtime of at least three runs
  - Measure the TLB misses as in experiment 1

2B Using transparent huge pages (THP)

- Free reserved huge pages
  - echo 0 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
- Turn on THP by following
  - echo always > /sys/kernel/mm/transparent_hugepage/enabled
- Measure the running and TLB misses as before

3 How to make TLB miss worse?

- Change parameters, modify application to make TLB miss worse