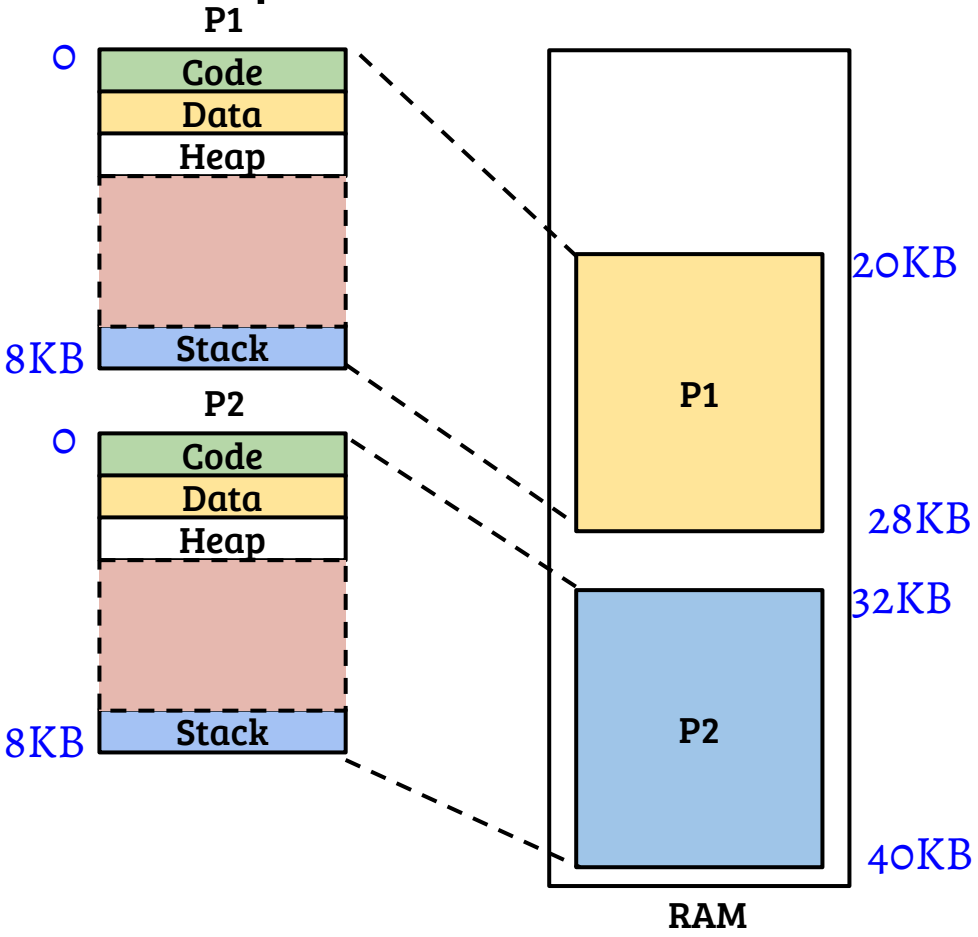
CS330: Operating Systems

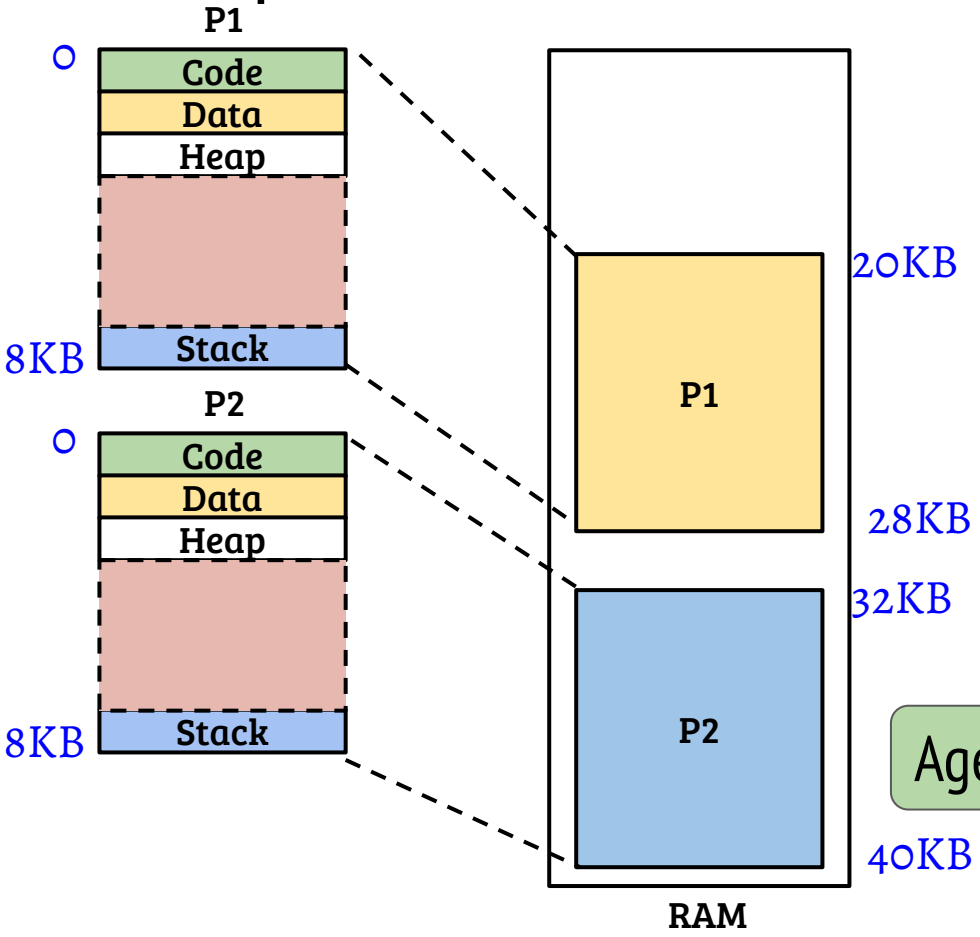
Virtual memory: Segmentation

Recap: Translation at address space granularity



- Physical memory of same size as the address space size is allocated to each process
- Issues: Memory inefficient, inflexible

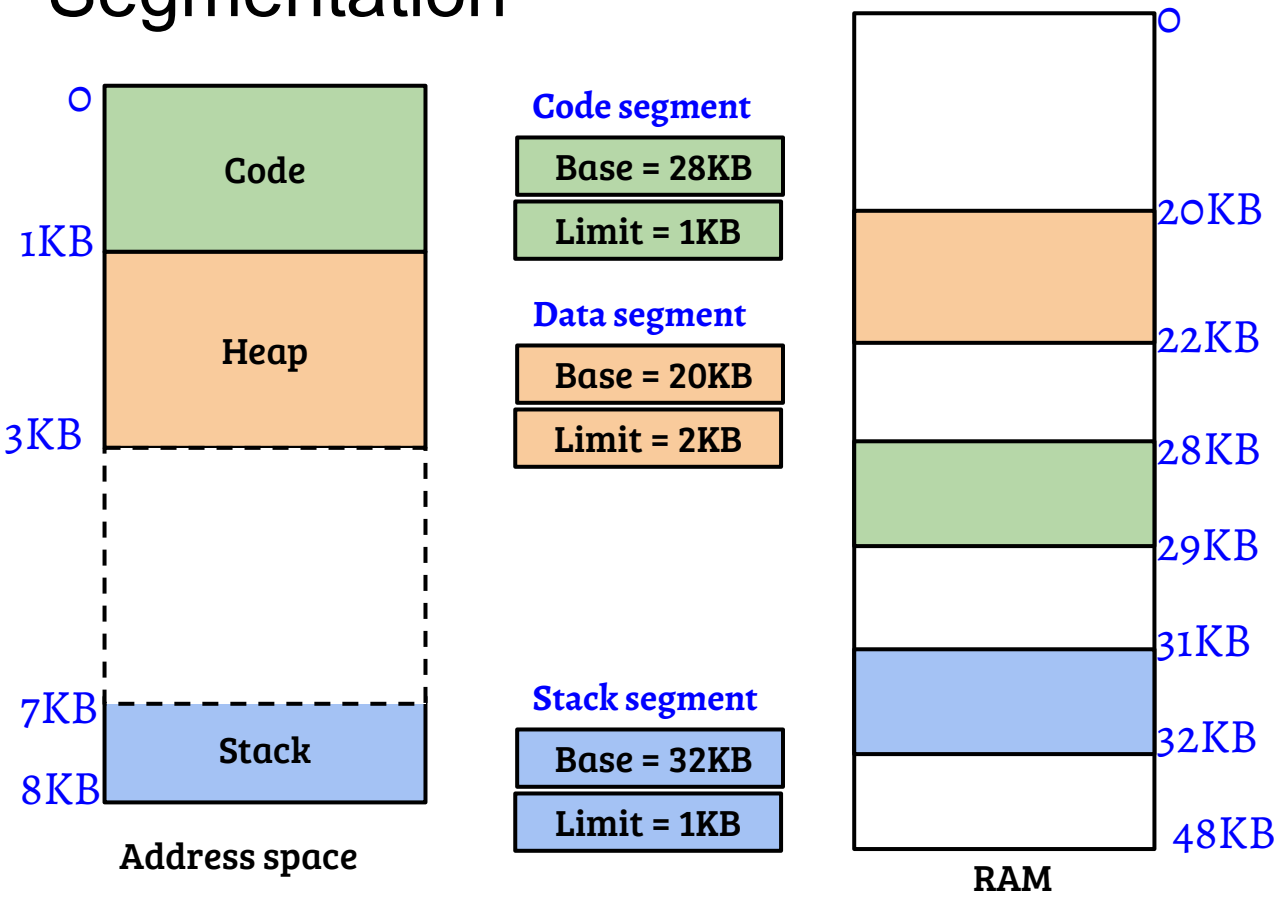
Recap: Translation at address space granularity



- Physical memory of same size as the address space size is allocated to each process
- Issues: Memory inefficient, inflexible

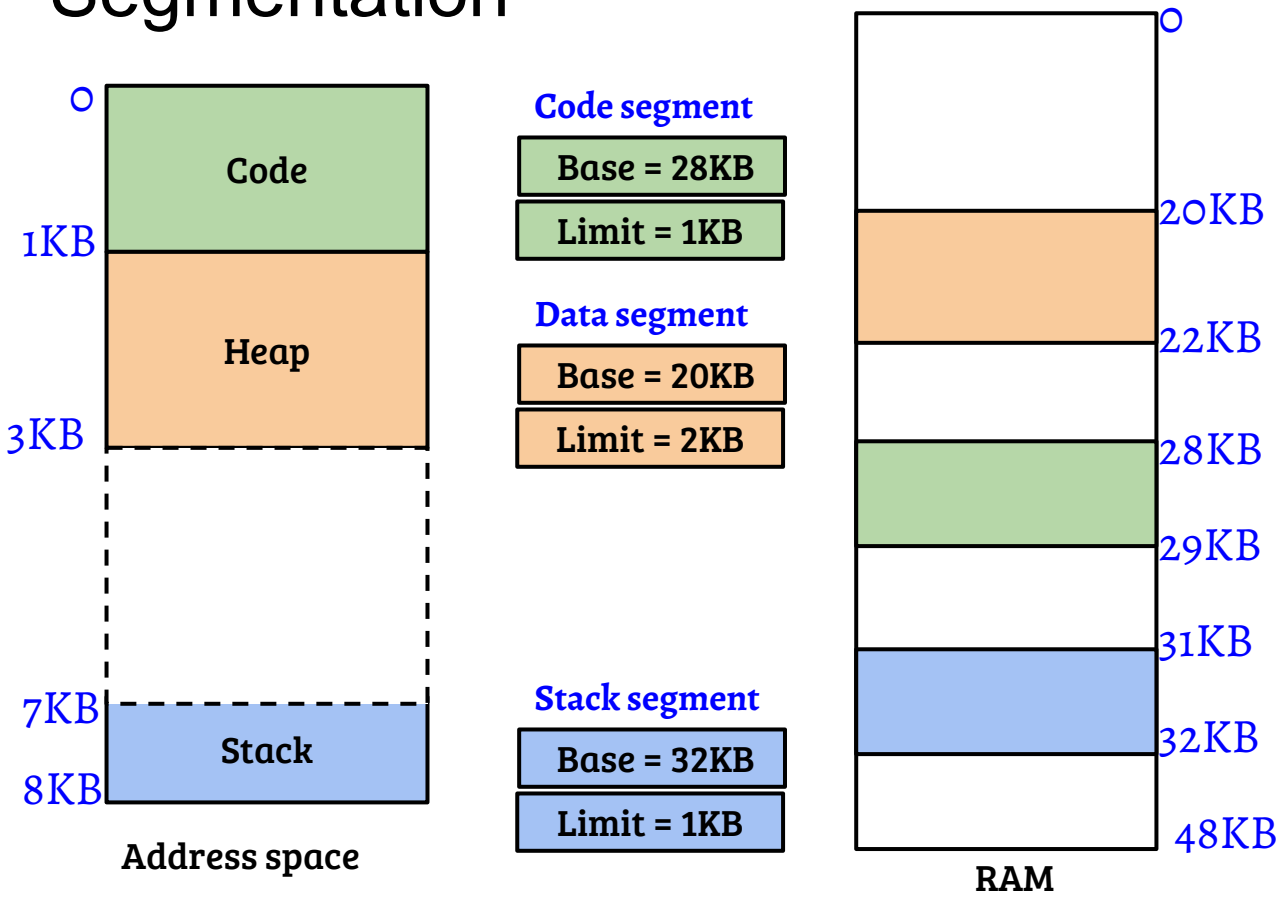
Agenda: Translation at segment granularity

Segmentation



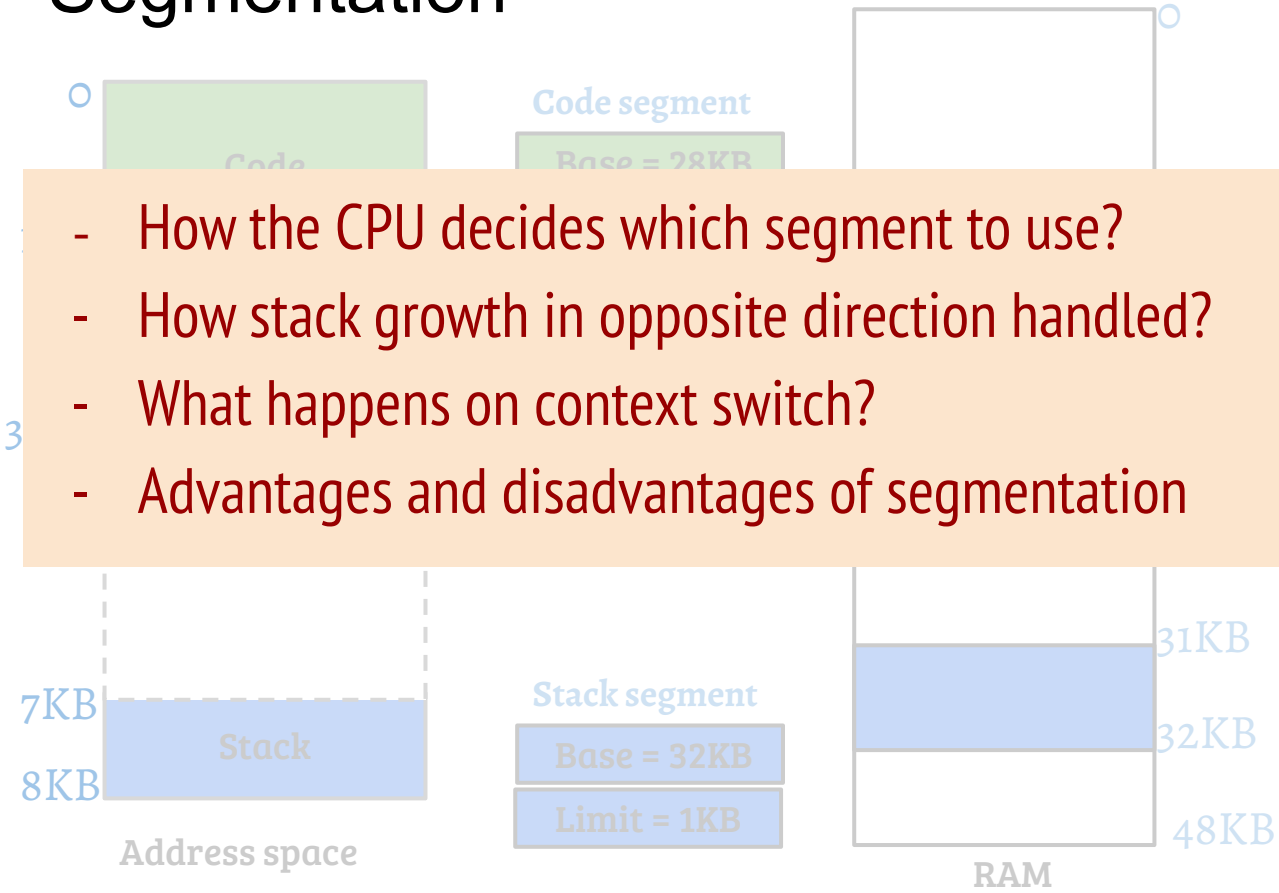
- Extension of the basic scheme with more base-limit register pairs

Segmentation



- Example
  - Code address
  - Data address

Segmentation



- How the CPU decides which segment to use?
- How stack growth in opposite direction handled?
- What happens on context switch?
- Advantages and disadvantages of segmentation

Segmentation: Explicit addressing

- Part of the address is used to explicitly specify segments
- In our example,
  - virtual address space = 8KB, address length = 13 bits and there are three segments
  - Two MSB bits used to specify the segment: “00” for code, “01” for data and “11” for stack
  - The hardware selects the segment register based on the value of two MSB bits and rest of the bits are used as the offset
  - Max. size of each segment = 2KB

Issues with explicit addressing

- Inflexible
  - Data and stack can not be sized dynamically
- Wastage of virtual address space
  - In our example, 2KB virtual address is unusable
- Note: Physical allocation is still done in an on-demand basis

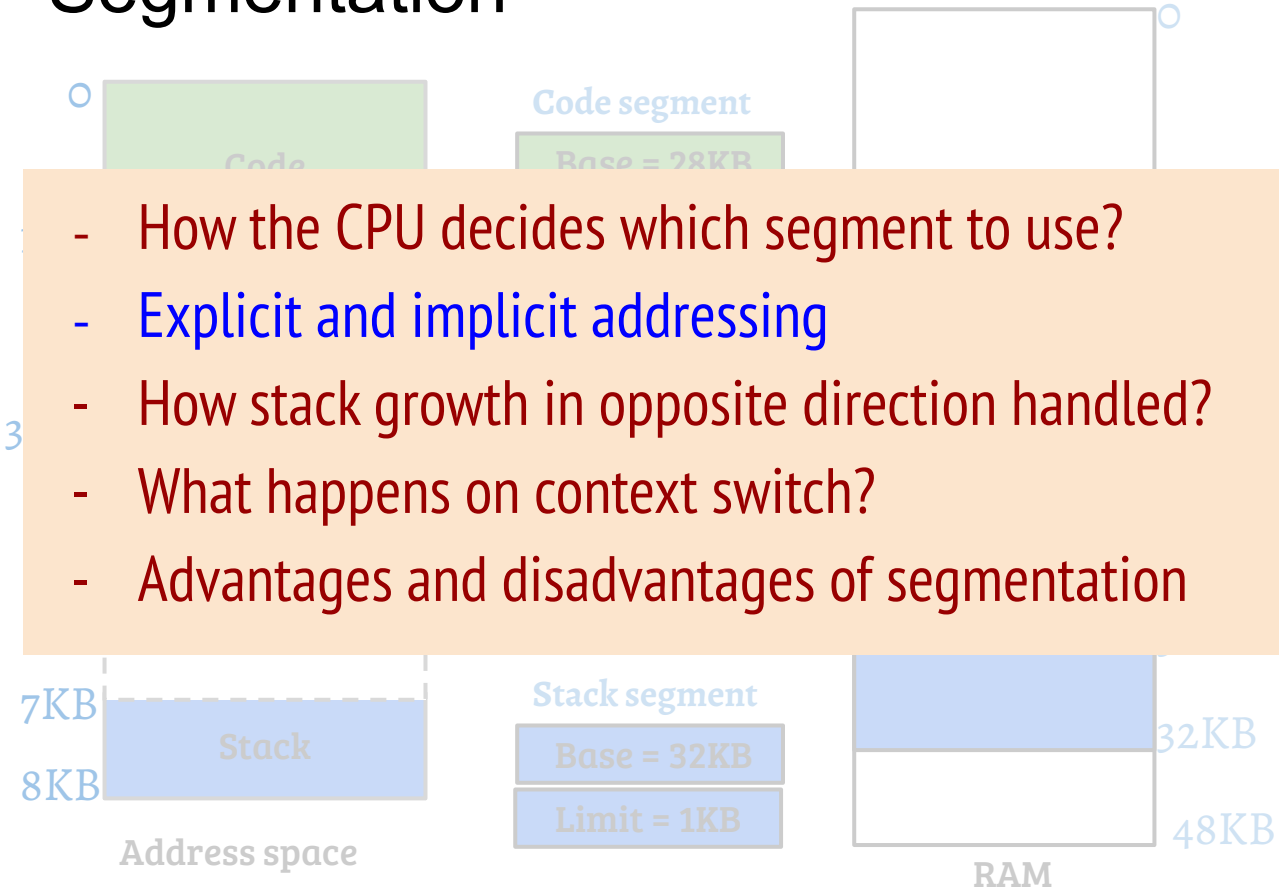
Segmentation: Implicit addressing

- The hardware selects the segment register based on the operation
- Code segment for instruction access
  - Fetch address, jump target, call address

Segmentation: Implicit addressing

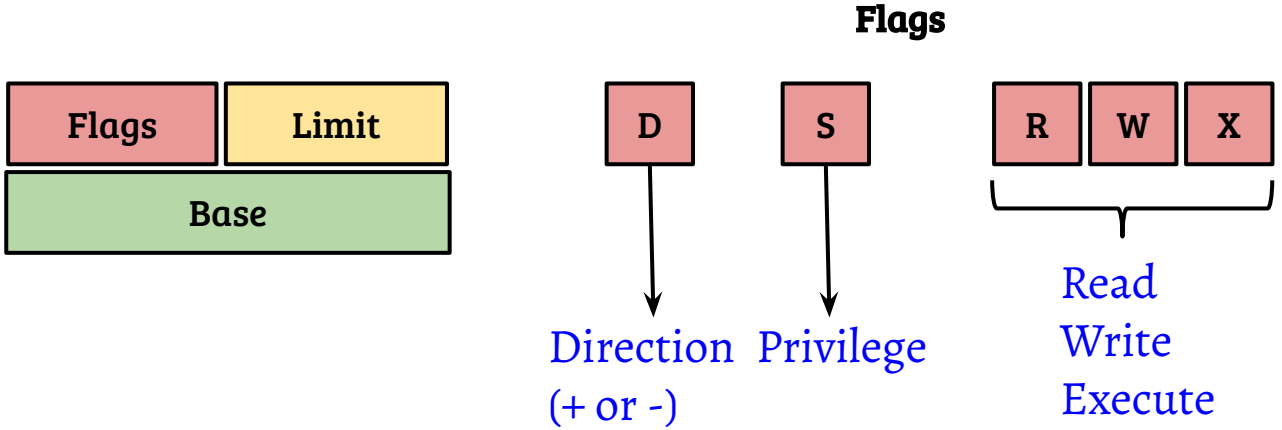
- The hardware selects the segment register based on the operation
- Code segment for instruction access
  - Fetch address, jump target, call address
- Stack segment for stack operations
  - Arguments for push and pop, indirect addressing with SP, BP
- Data segment for other addresses

Segmentation



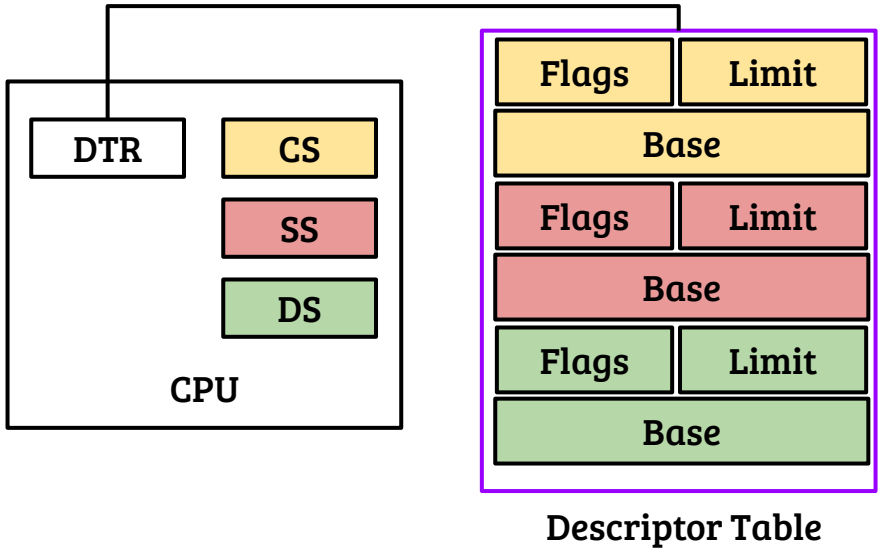
- How the CPU decides which segment to use?
- Explicit and implicit addressing
- How stack growth in opposite direction handled?
- What happens on context switch?
- Advantages and disadvantages of segmentation

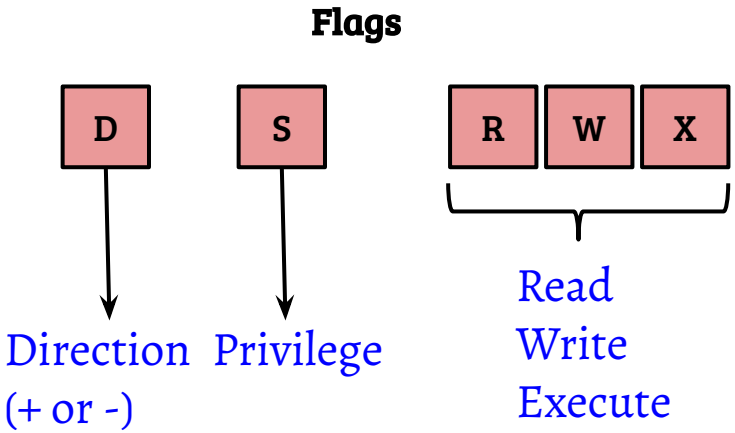
Segmentation (protection and direction)



- For stack, direction is -ve, used by hardware to calculate physical address
- “S” bit can be used to specify privilege, specifically useful in code segment
- R, W and X can be used to enforce isolation and sharing

Segmentation in reality

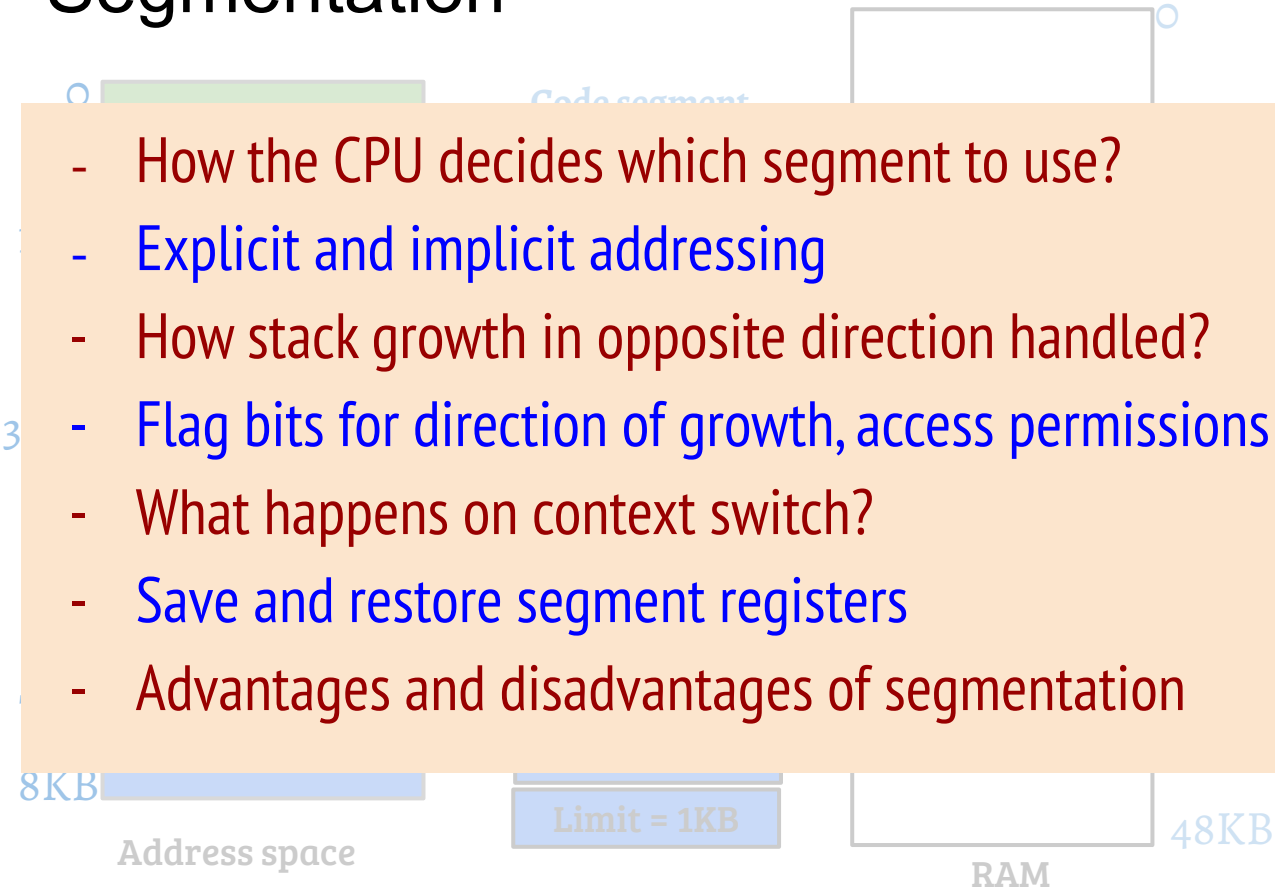




- Descriptor table register (DTR) is used to access the descriptor table
- # of descriptors depends on architecture
- Separate descriptors used for user and kernel mode

Segmentation

- How the CPU decides which segment to use?
- Explicit and implicit addressing
- How stack growth in opposite direction handled?
- Flag bits for direction of growth, access permissions
- What happens on context switch?
- Save and restore segment registers
- Advantages and disadvantages of segmentation



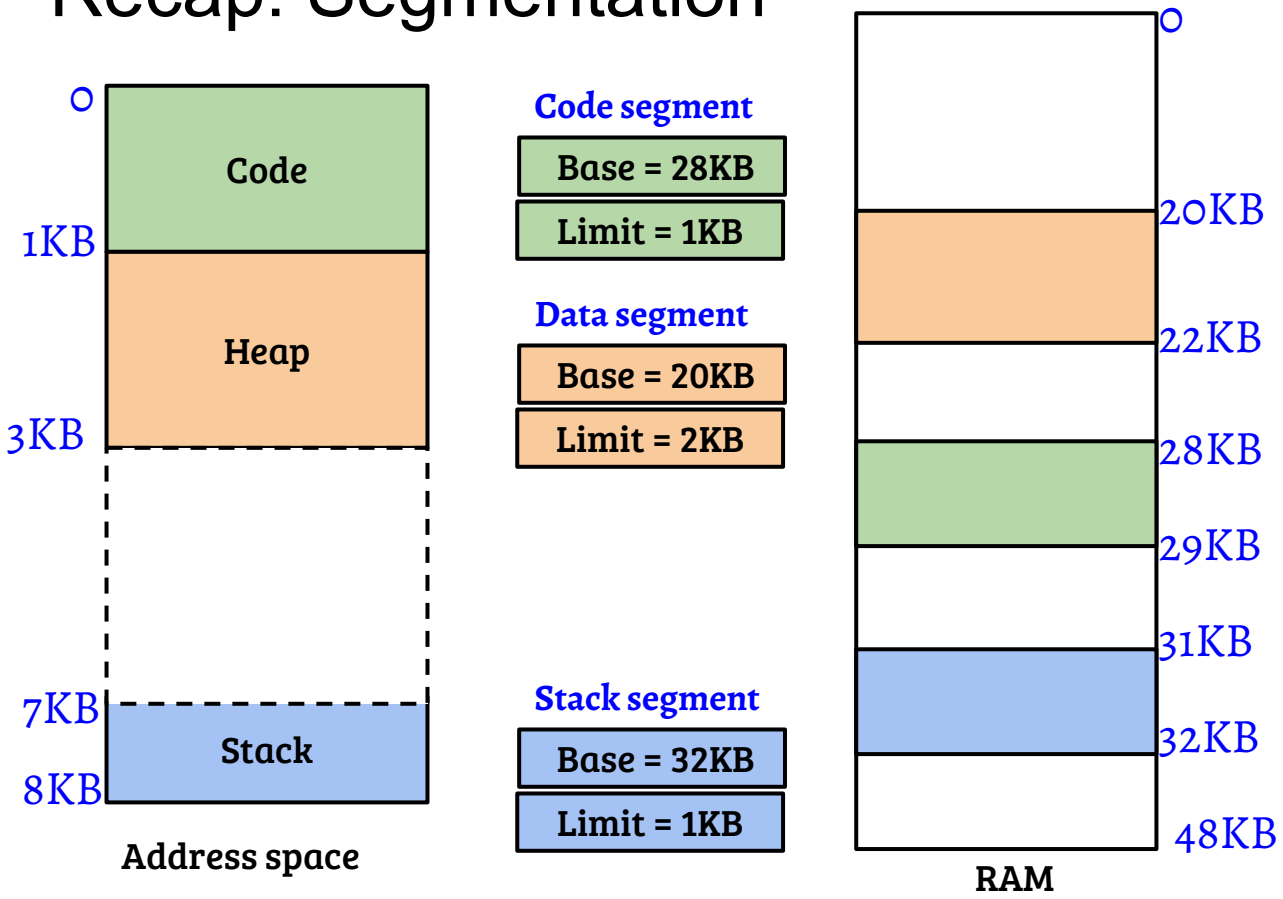
Advantages and disadvantages of segmentation

- Advantages
  - Easy and efficient address translation
  - Save memory wastage for unused addresses
- Disadvantages
  - External fragmentation
  - Can not support discontiguous sparse mapping

CS330: Operating Systems

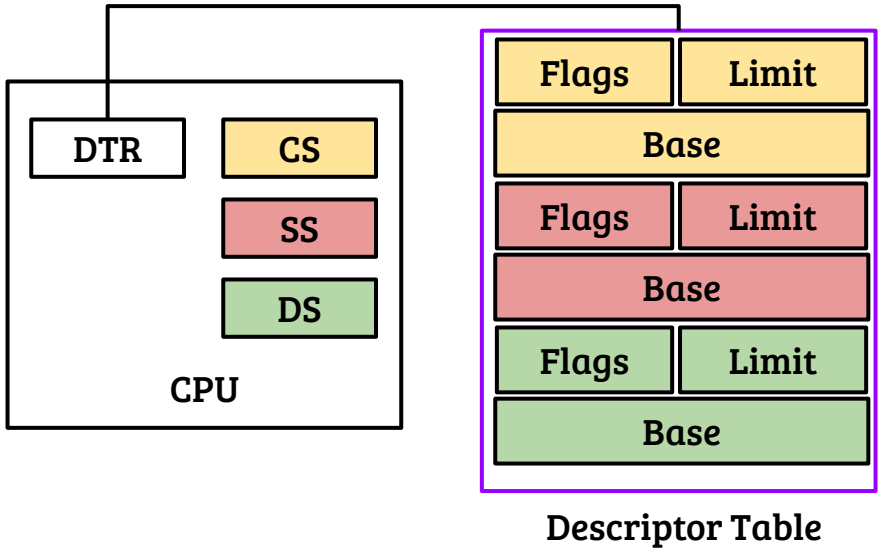
Virtual memory: Paging

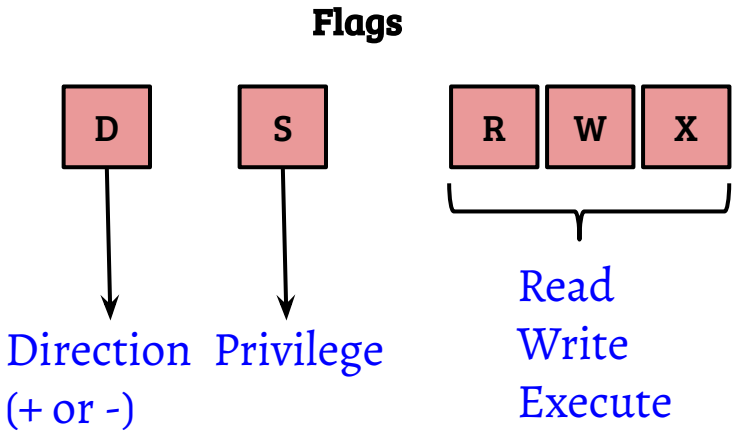
Recap: Segmentation



- Extension of the scheme for translation ar address space granularity
- Base-limit register pairs per segment

Recap: Segmentation in reality





- Descriptor table register (DTR) is used to access the descriptor table
- # of descriptors depends on architecture
- Separate descriptors used for user and kernel mode

Paging

- Paging addresses the following issues with segmentation
  - External fragmentation caused due to variable sized segments
  - No support for discontinuous/sparse mapping

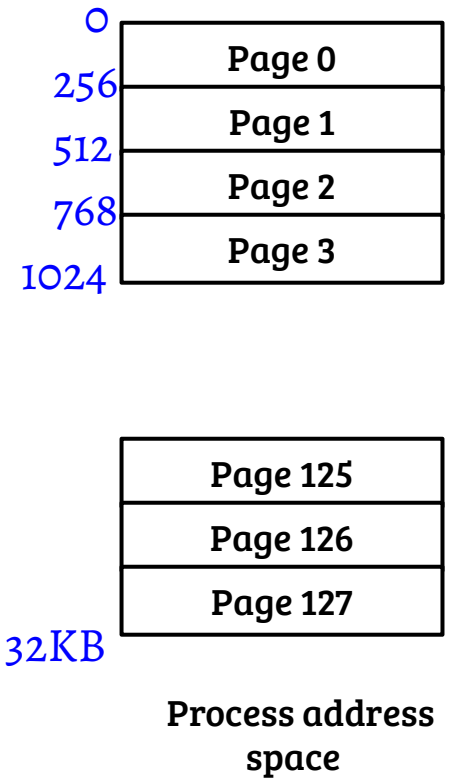
Paging

- Paging addresses the following issues with segmentation
  - External fragmentation caused due to variable sized segments
  - No support for discontinuous/sparse mapping
- The idea of paging
  - Partition the address space into fixed sized blocks (call it page)
  - Physical memory partitioned in a similar way (call it page frame)

Paging

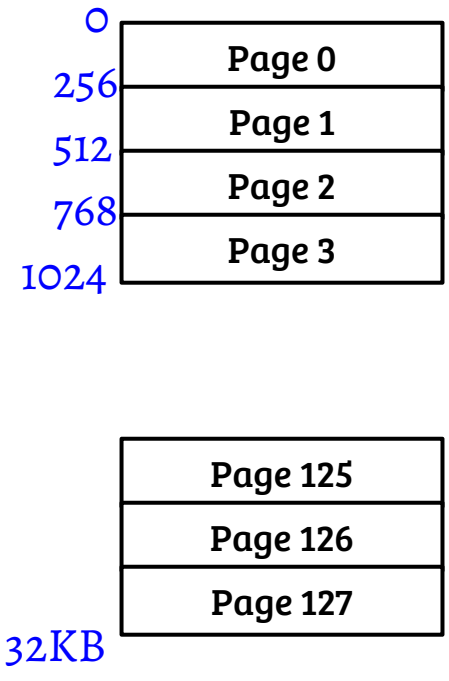
- Paging addresses the following issues with segmentation
  - External fragmentation caused due to variable sized segments
  - No support for discontinuous/sparse mapping
- The idea of paging
  - Partition the address space into fixed sized blocks (call it pages)
  - Physical memory partitioned in a similar way (call it page frames)
  - OS creates a mapping between *page* to *page frame*
  - H/W uses the mapping to translate VA to PA

Paging example (pages)

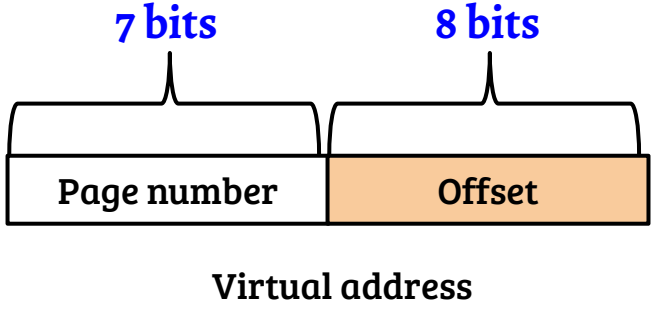


- Virtual address size = 32KB, Page size = 256 bytes
- Address length = 15 bits {0x0 - 0x7FFF}
- # of pages = 128

Paging example (pages)

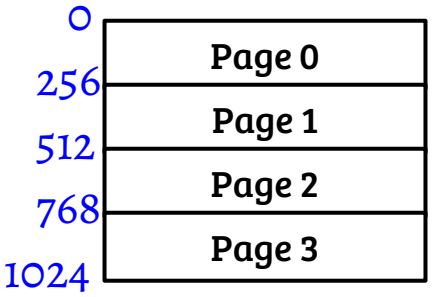


- Virtual address size = 32KB, Page size = 256 bytes
- Address length = 15 bits {0x0 - 0x7FFF}
- # of pages = 128

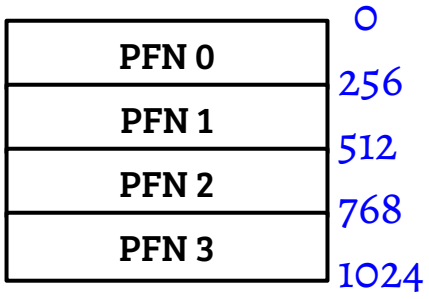


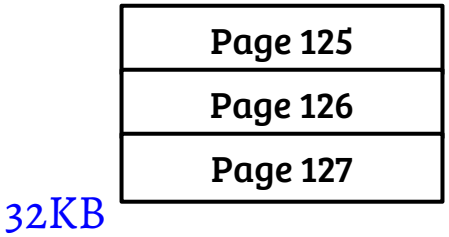
- Example: For Virtual address *0x0510*, Page number = 5, offset = 16

Paging example (page frames)

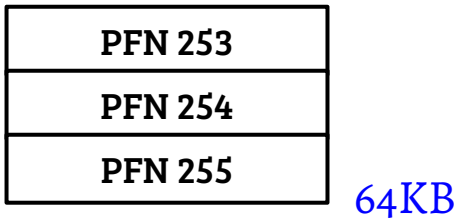


- Physical address size = 64KB
- Address length = 16 bits {0x0 - 0xFFFF}
- # of page frames = 256



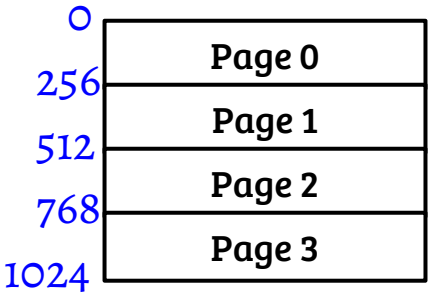


Process address space

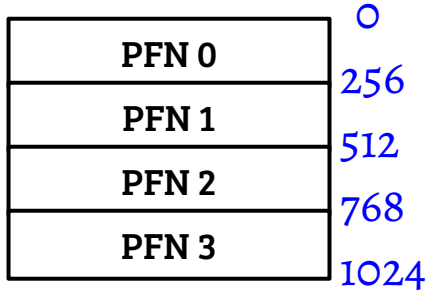


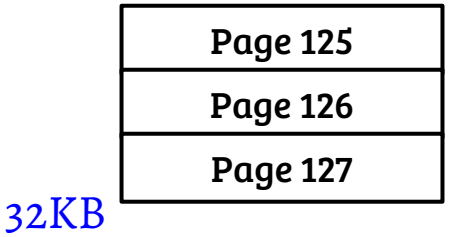
DRAM

Paging example (page frames)

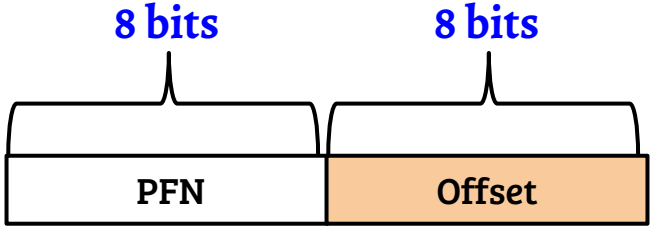


- Physical address size = 64KB
- Address length = 16 bits {0x0 - 0xFFFF}
- # of page frames = 256

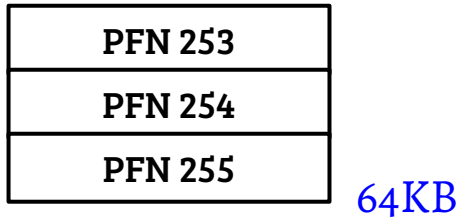




Process address space



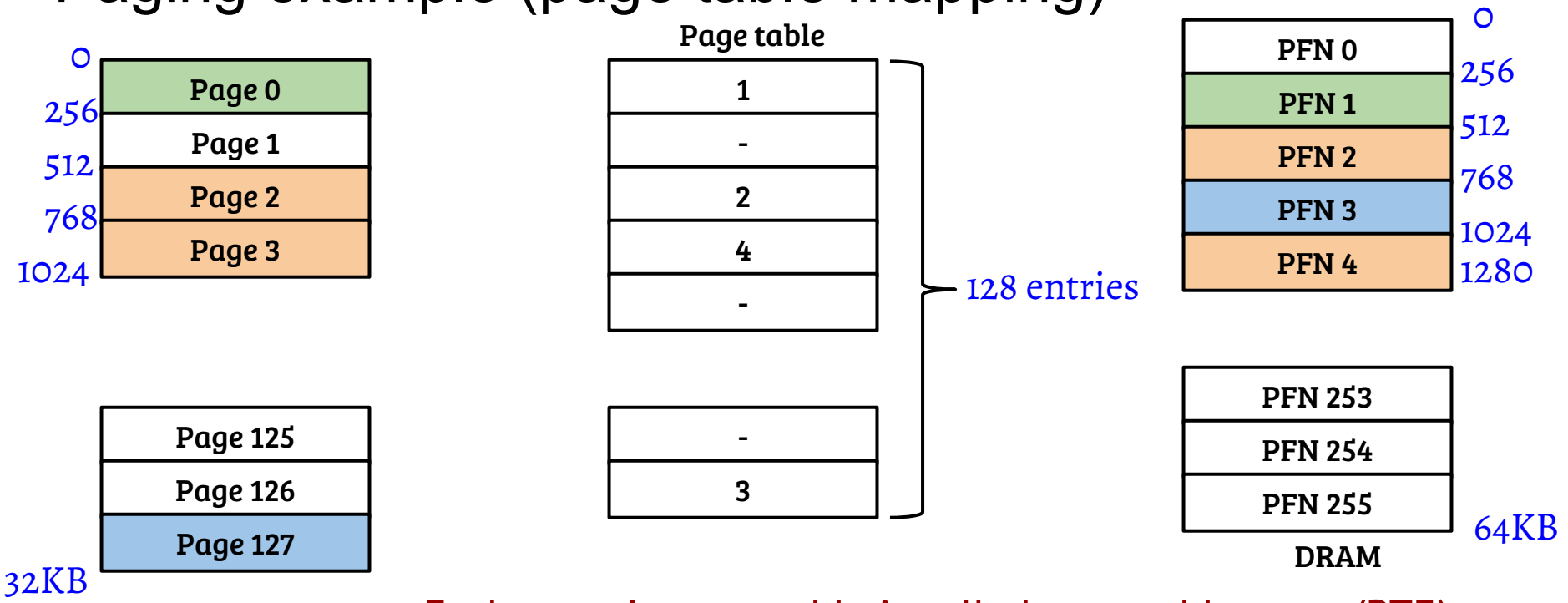
Physical address



DRAM

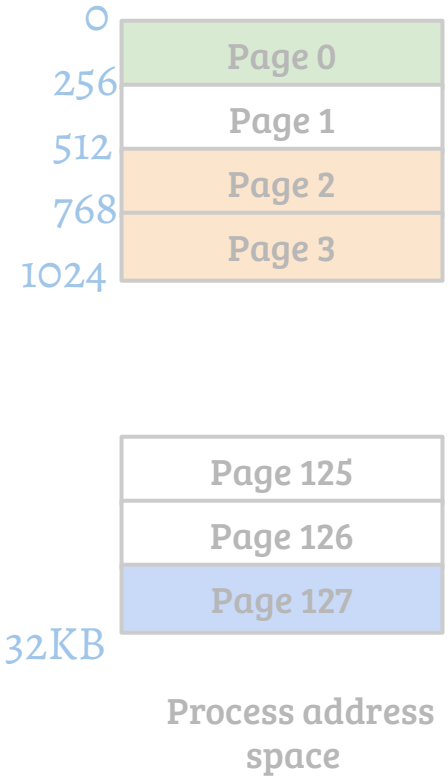
- Example: For physical address *0x1F51*, PFN = 31, offset = 81

Paging example (page table mapping)

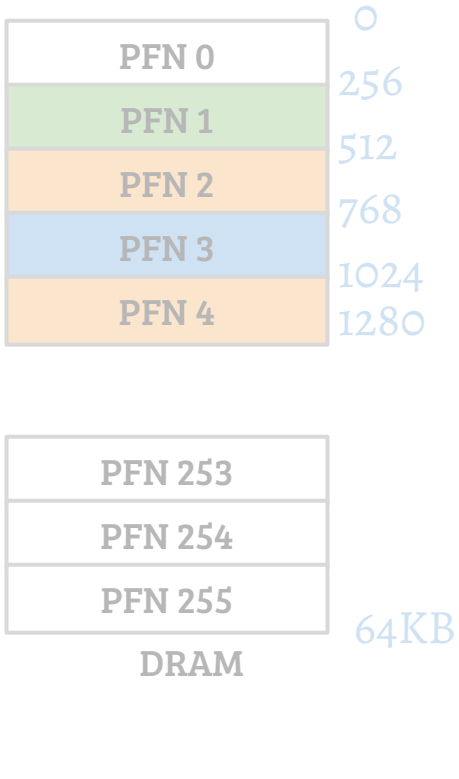


- Each entry in page table is called page table entry (PTE)
- Example mapping: page 0 ⇒ PFN 1, page 2 ⇒ PFN 2 and so on

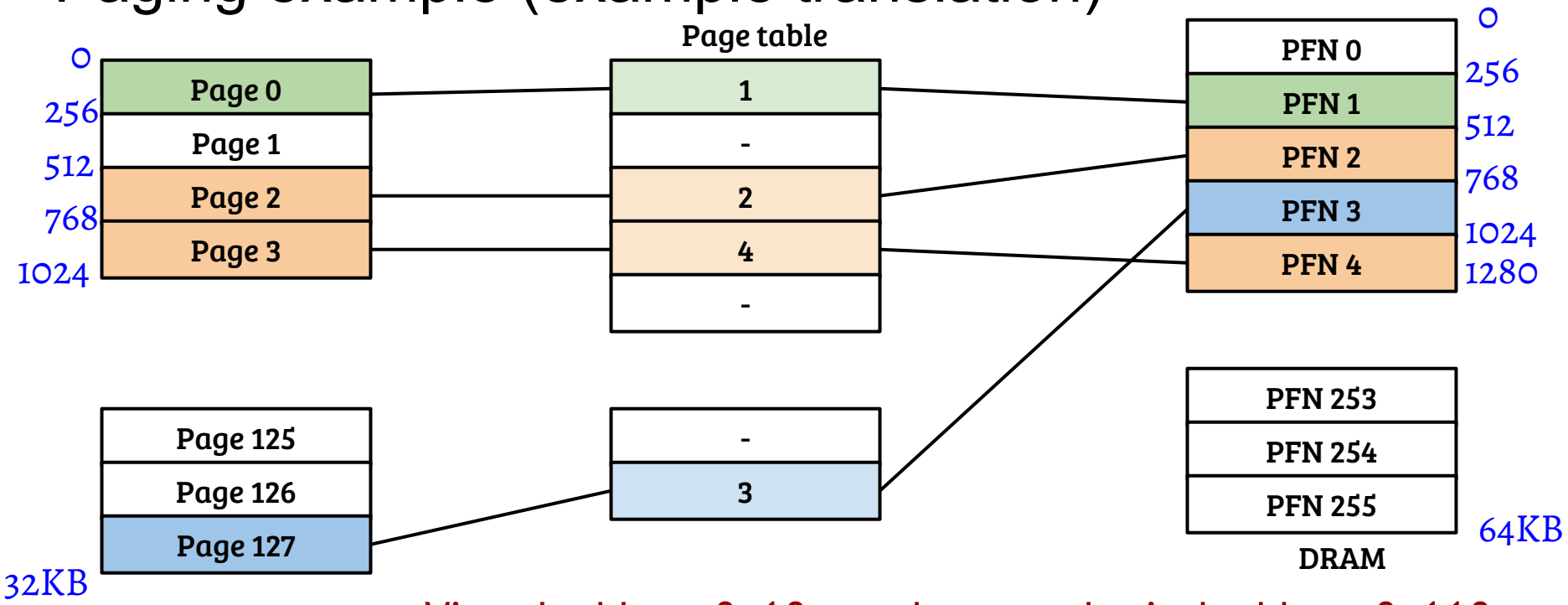
Paging example (page table walk)




```
PTW (vaddr V, PTable P)
// Input: Virtual address, Page table
// Returns physical address
{
  Entry = P[V >> 8];
  if (Entry.present)
      return (Entry.PFN << 8) + (V & 0xFF);
   Raise PageFault;
}
```

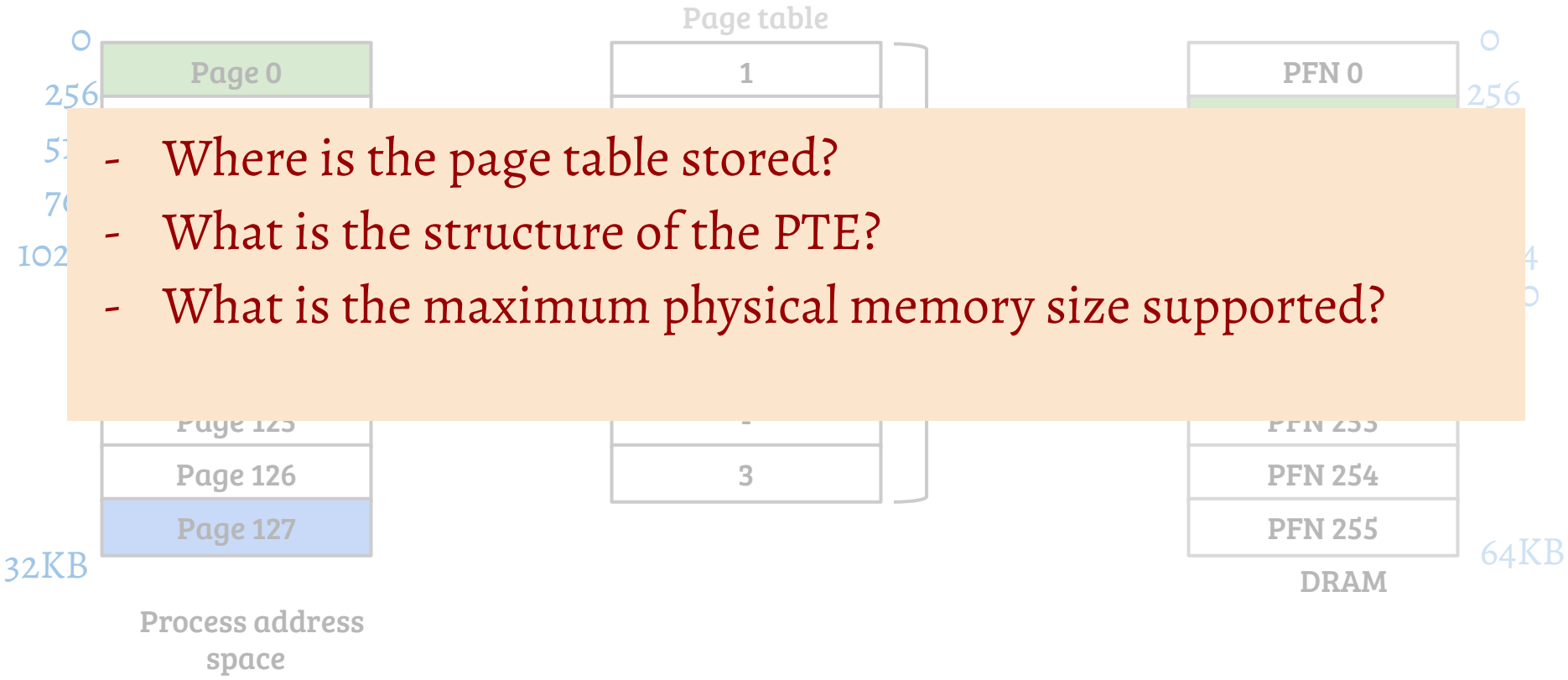


Paging example (example translation)



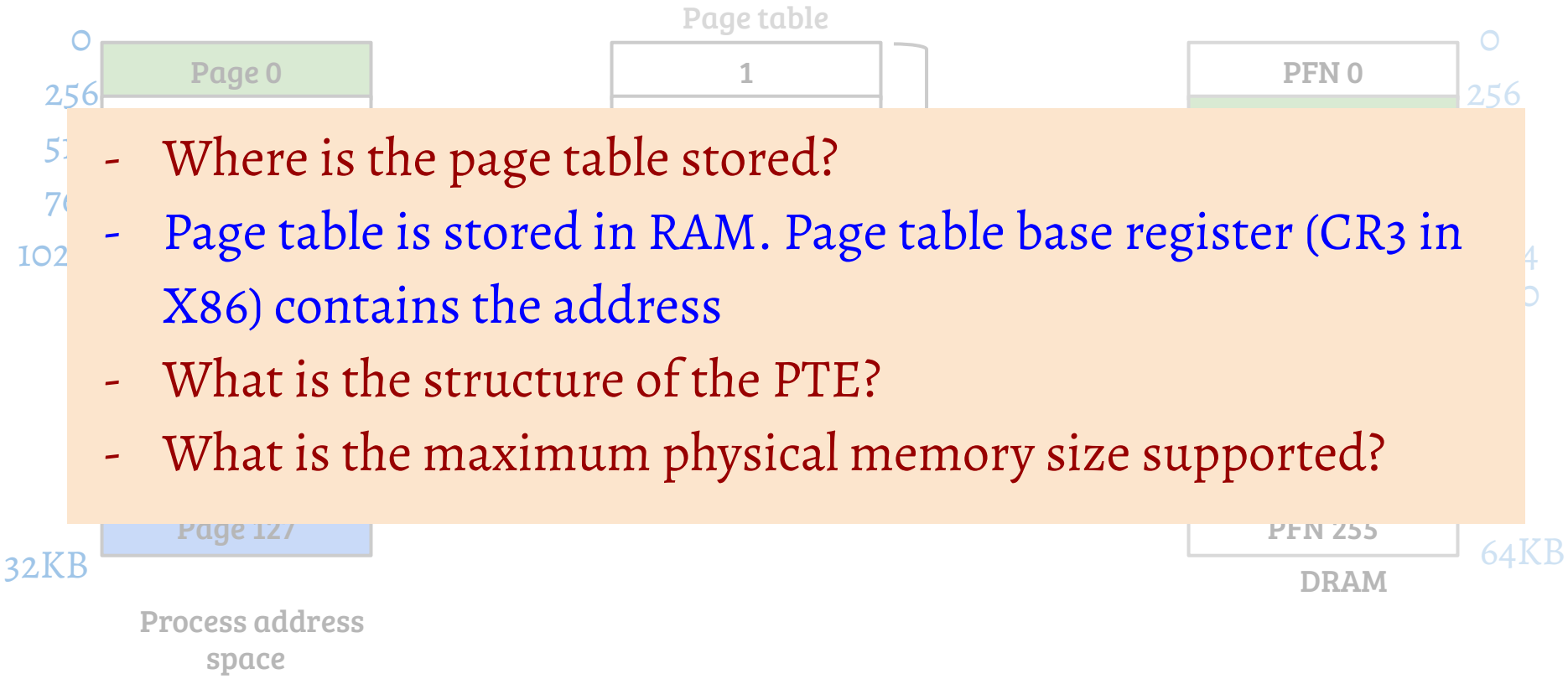
- Virtual address 0x10 translates to physical address 0x110
- Virtual address 0x7FF0 translates to physical address 0x3F0

Paging example (page table walk)



- Where is the page table stored?
- What is the structure of the PTE?
- What is the maximum physical memory size supported?

Paging example (page table walk)



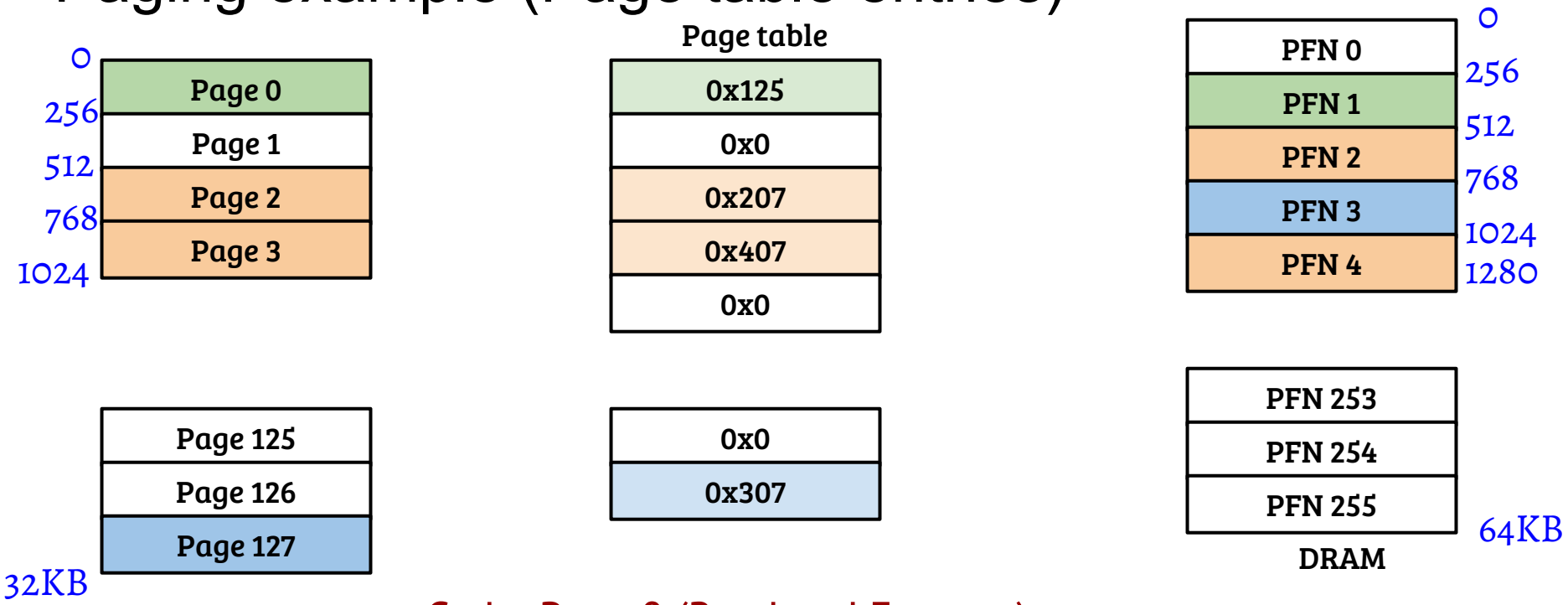
- Where is the page table stored?
- Page table is stored in RAM. Page table base register (CR3 in X86) contains the address
- What is the structure of the PTE?
- What is the maximum physical memory size supported?

Paging example (structure of an example PTE)

- PFN occupies a significant portion of PTE entry (8 bits in this example)

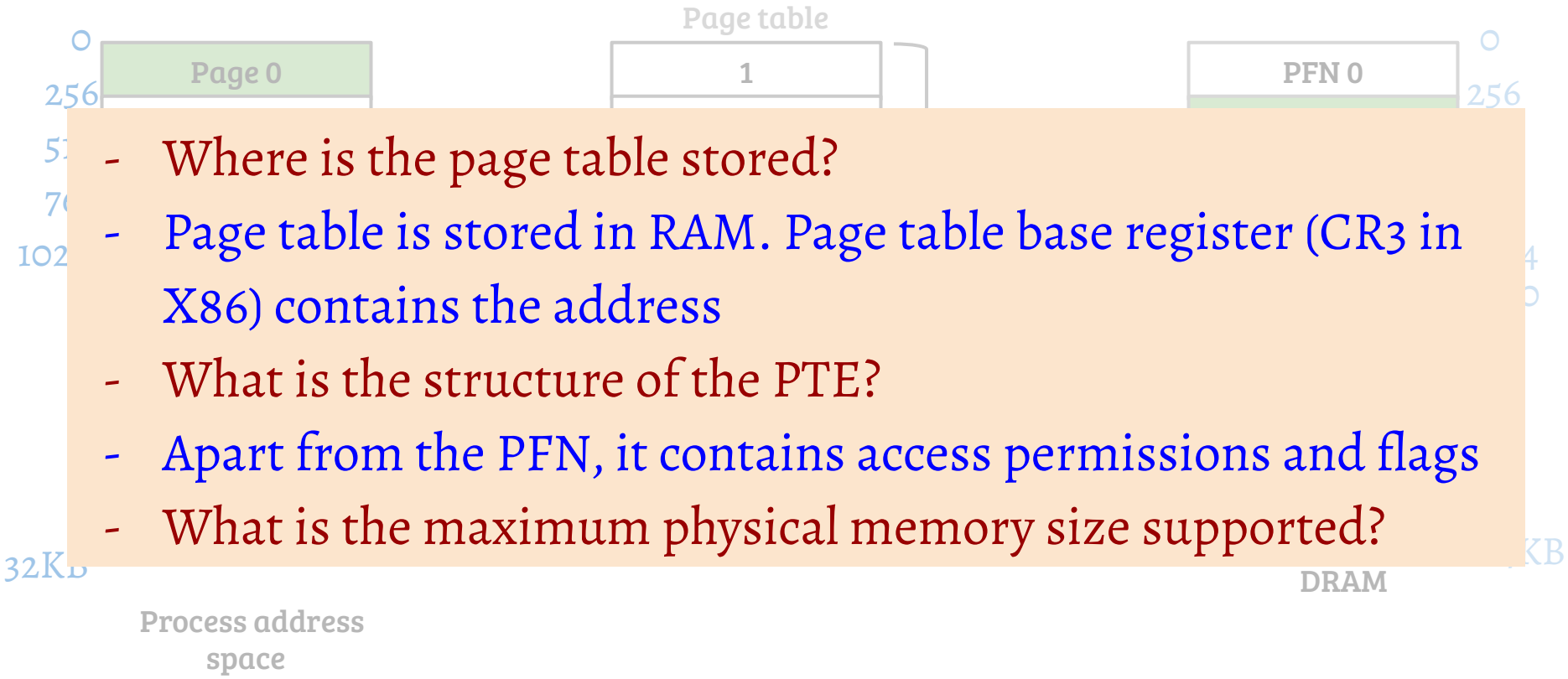
P	Present bit, 1 ⇒ entry is valid
W	Write bit, 1 ⇒ Write allowed
S	Privilege bit, 0 ⇒ only kernel mode access is allowed
A	Accessed bit, 1 ⇒ Address accessed (set by H/W during walk)
D	Dirty bit, 1 ⇒ Address written (set by H/W during walk)
X	Execute bit, 1 ⇒ Instruction fetch allowed for this page
	Reserved/unused bits

Paging example (Page table entries)



- Code: Page 0 (Read and Execute)
- Data: Page 2 and Page 3 (Read and Write)
- Stack: Page 127 (Read and Write)

Paging example (page table walk)



- Where is the page table stored?
- Page table is stored in RAM. Page table base register (CR3 in X86) contains the address
- What is the structure of the PTE?
- Apart from the PFN, it contains access permissions and flags
- What is the maximum physical memory size supported?

Paging example (page table walk)

- Where is the page table stored?
- Page table is stored in RAM. Page table base register (CR3 in X86) contains the address
- What is the structure of the PTE?
- Apart from the PFN, it contains access permissions and flags
- What is the maximum physical memory size supported?
- For this example, 8-bits can be used to specify 256 page frames. Maximum RAM size = 256 * 256 = 64KB

Paging: one level of page table may not be feasible!

- Consider a 32-bit address space (=4GB)
- What should be the page size for this system?

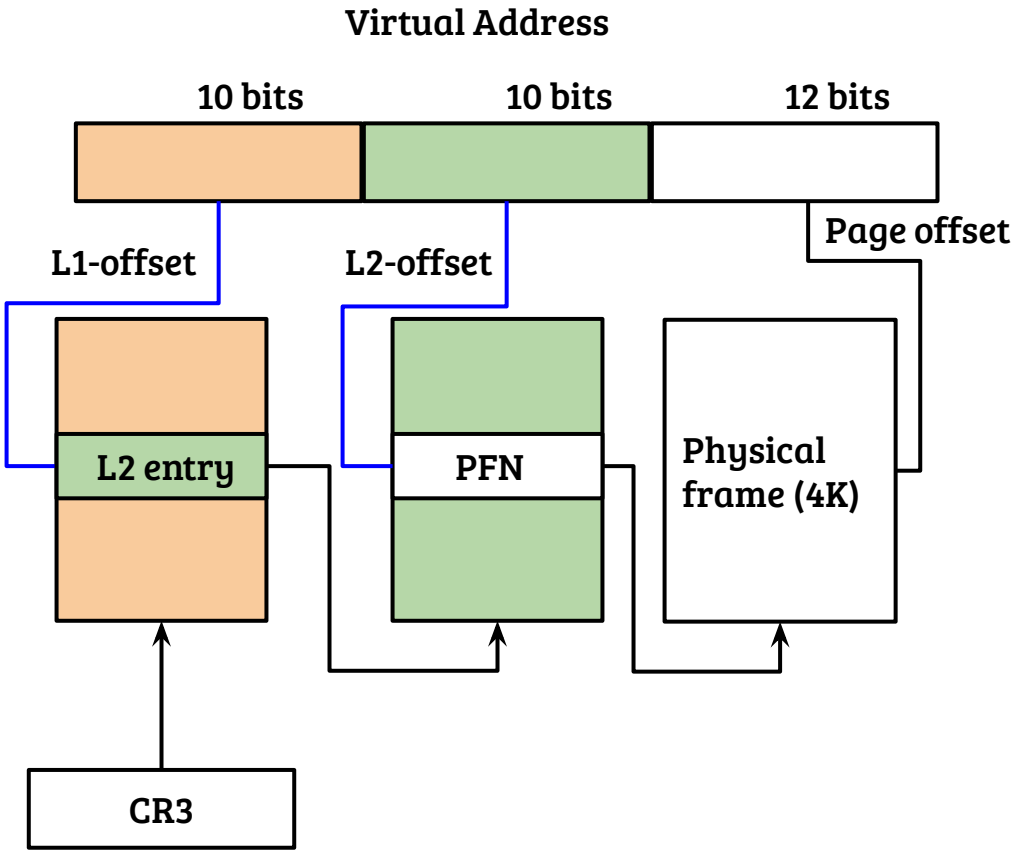
Paging: one level of page table may not be feasible!

- Consider a 32-bit address space (=4GB)
- What should be the page size for this system?
- Large page size results in *internal fragmentation*
- Assuming page size = 4KB, How many entries are required in a one-level paging system?

Paging: one level of page table may not be feasible!

- Consider a 32-bit address space (=4GB)
- What should be the page size for this system?
- Large page size results in *internal fragmentation*
- Assuming page size = 4KB, How many entries are required in a one-level paging system? (2^{20} entries)
- Not possible to hold 2^{20} entries in a single page
- Therefore, multi-level page tables are used in modern systems

Two-level page tables (32-bit virtual address)



- Two-level page table
- Level-1 page table contains entries pointing to Level-2 page table structures
- Level-2 entry contains PFN along with flags

CS330: Operating Systems

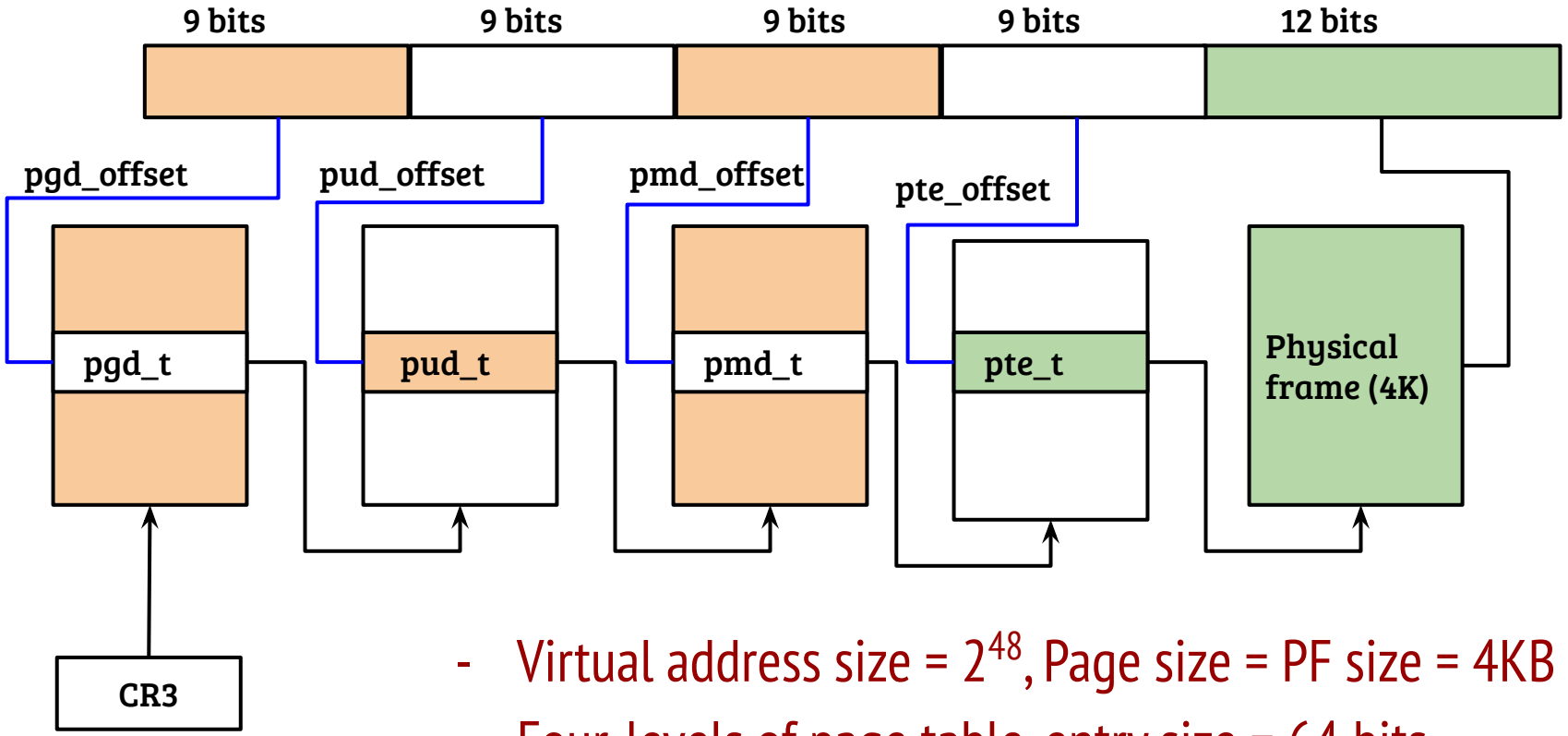
Virtual memory: Multilevel paging and TLB

Recap: Paging

- The idea of paging
  - Partition the address space into fixed sized blocks (call it pages)
  - Physical memory partitioned in a similar way (call it page frames)
  - OS creates a mapping between *page* to *page frame* , H/W uses the mapping to translate VA to PA
- With increased address space size, single level page table entry is not feasible, because
  - Increasing page size (= frame size) increases internal fragmentation
  - Small pages may not be suitable to hold all mapping entries

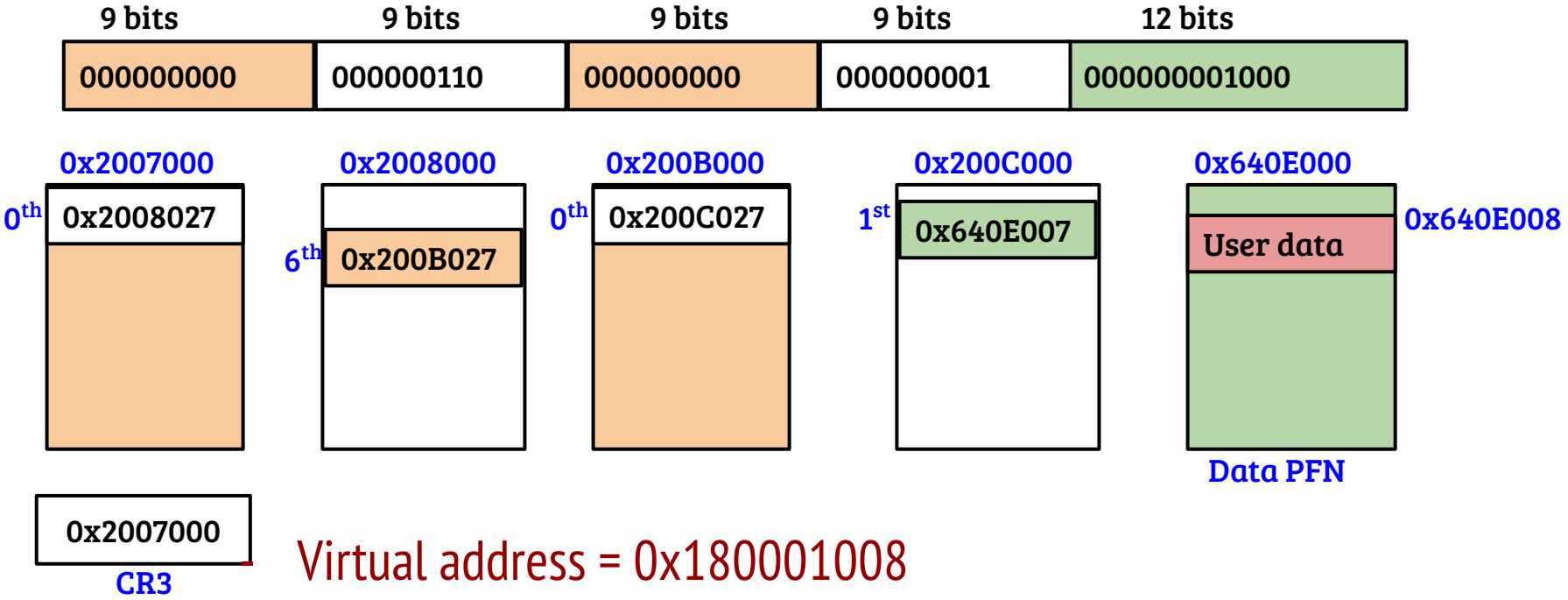
Agenda: Multi-level pages tables and their efficiency implications

4-level page tables: 48-bit VA (Intel x86_64)



- Virtual address size = 2^{48}, Page size = PF size = 4KB
- Four-levels of page table, entry size = 64 bits

4-level page tables: example translation



Virtual address = 0x180001008

- Hardware translation by repeated access of page table stored in physical memory
- Page table entry: 12 bits LSB is used for access flags

Paging: translation efficiency

```
sum = 0;
for(ctr=0; ctr<10; ++ctr)
    sum += ctr;
```

```
0x20100:  mov $0, %rax;
0x20102:  mov %rax, (%rbp);     // sum=0
0x20104:  mov $0, %rcx;         // ctr=0
0x20106:  cmp $10, %rcx;        // ctr < 10
0x20109:  jge  0x2011f;         // jump if >=
0x2010f:  add %rcx, %rax;
0x20111:  mov %rax, (%rbp);     // sum += ctr
0x20113:  inc %rcx              // ++ctr
0x20115:  jmp 0x20106           // loop
0x2011f:  ..............
```

- Considering four-level page table, how many memory accesses are required (for translation) during the execution of the above code?

Paging: translation efficiency

```
0x20100:  mov $0, %rax;
0x20102:  mov %rax, (%rbp):     // sum=0
```

- Instruction execution: Loop = 10 * 6, Others = 2 + 3
  - Memory accesses during translation = 65 * 4 = 260
- Data/stack access: Initialization = 1, Loop = 10
  - Memory accesses during translation = 11 * 4 = 44
- A lot of memory accesses (> 300) for address translation
- How many distinct pages are translated? Assume stack address range 0x7FFF000 - 0x8000000
- Considering four-level page table, how many memory accesses are required (for translation) during the execution of the above code?

Paging: translation efficiency

0x20100: mov $0, %rax;

- Instruction execution: Loop = 10 * 6, Others = 2 + 3
  - Memory accesses during translation = 65 * 4 = 260
- Data/stack access: Initialization = 1, Loop = 10
  - Memory accesses during translation = 11 * 4 = 44
- A lot of memory accesses (> 300) for address translation
- How many distinct pages are translated? Assume stack address range 0x7FFF000 - 0x8000000
- One code page (0x20) and one stack page (0x7FFF). Caching these translations, will save a lot of memory accesses.

Paging with TLB: translation efficiency

TLB

Page	PTE
0x20	0x750
0x7FFF	0x890

- TLB is a hardware cache which stores *Page* to *PFN* mapping
- For code, after first miss for instruction fetch, all accesses hit the TLB
- Similarly, considering the stack virtual address range as 0x7FFF000 - 0x8000000, one entry in TLB avoids page table walk after first miss

Paging with TLB: translation efficiency

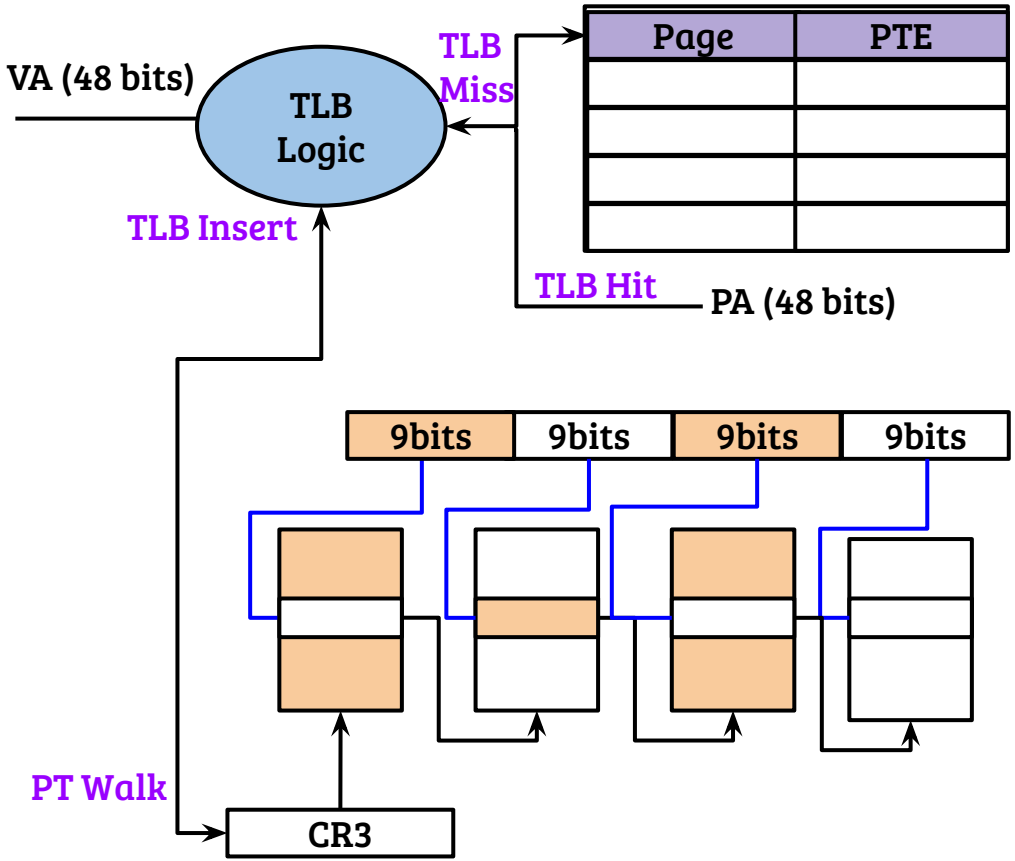
TLB

Page	PTE
0x20	0x750
0x7FFF	0x890

```
Translate(V){
    PageAddress P = V >> 12;
    TLBEntry entry = lookup(P);
    if (entry.valid) return entry.pte;
    entry = PageTableWalk(V);
    MakeEntry(entry);
    return entry.pte;
}
```

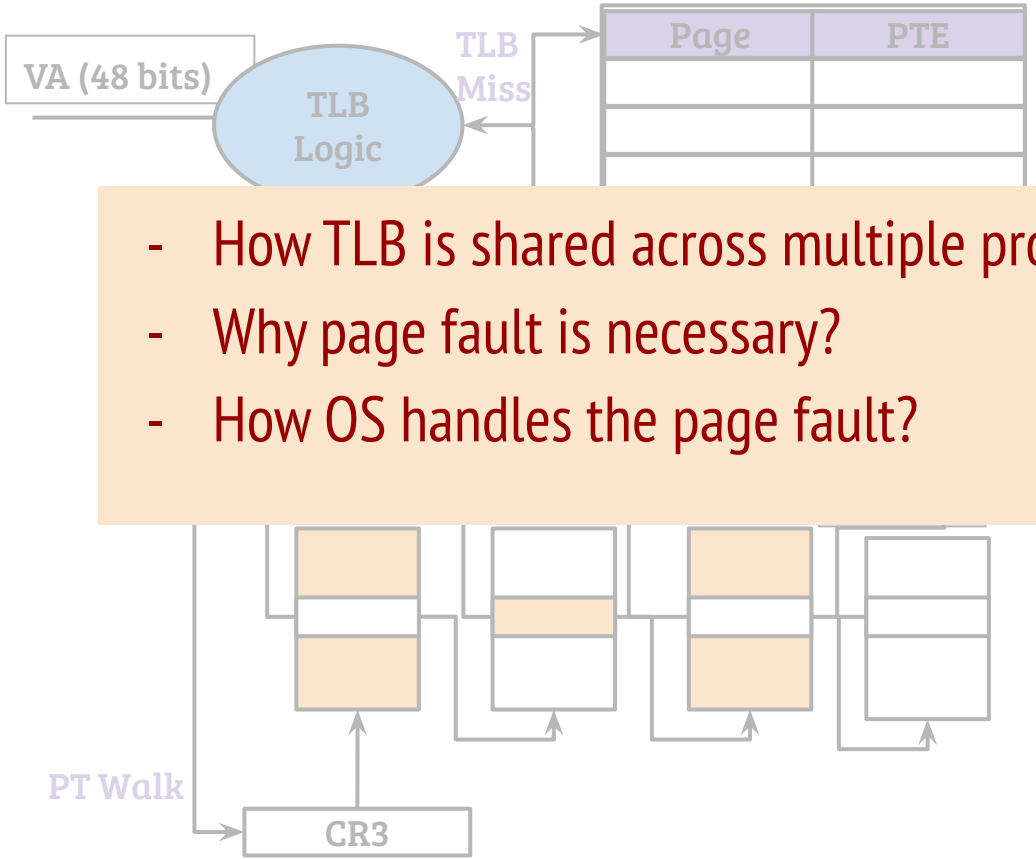
- TLB is a hardware cache which stores *Page* to *PFN* mapping
- After first miss for instruction fetch address, all others result in a TLB hit
- Similarly, considering the stack virtual address range as 0x7FFF000 - 0x8000000, one entry in TLB avoids page table walk after first miss

Address translation (TLB + PTW)



- TLB in the path of address translation
- Separate TLBs for instruction and data, multi-level TLBs
- In X86, OS can not make entries into the TLB directly, it can flush entries

Address translation (TLB + PTW)



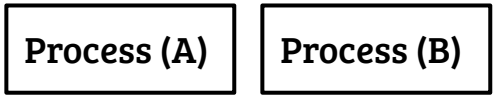
- TLB in the path of address

- How TLB is shared across multiple processes?
- Why page fault is necessary?
- How OS handles the page fault?

d

into the TLB directly, it can flush entries

TLB: Sharing across applications

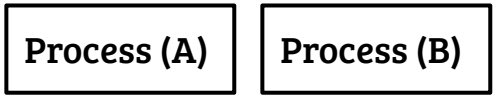


Page	PTE
0x100	0x200007
0x101	0x205007

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution

TLB: Sharing across applications

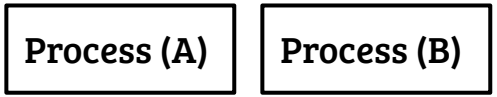


Page	PTE
0x100	0x200007
0x101	0x205007

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution
- Process B may be using the same addresses used by A. Result: Wrong translation

TLB: Sharing across applications

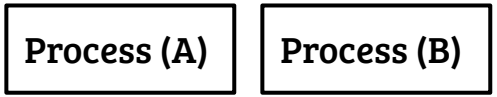


Page	PTE
~~0x100~~	~~0x200007~~
~~0x101~~	~~0x205007~~

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution
- Correctness ensured. Performance is an issue (with frequent context switching)

TLB: Sharing across applications

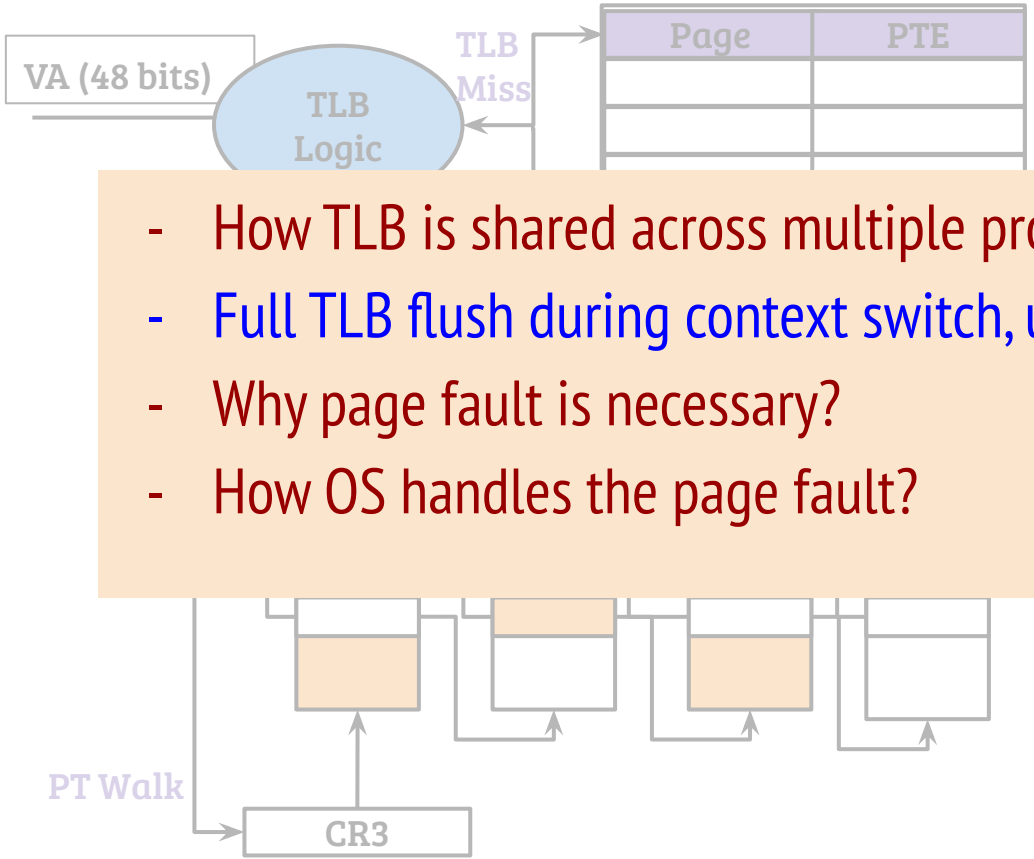


ASID	Page	PTE
A	0x100	ox200007
A	0x101	ox205007
B	0x100	0x301007
B	0x101	0x302007

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution
- Address space identified (ASID) along with each TLB entry to identify the process

Address translation (TLB + PTW)



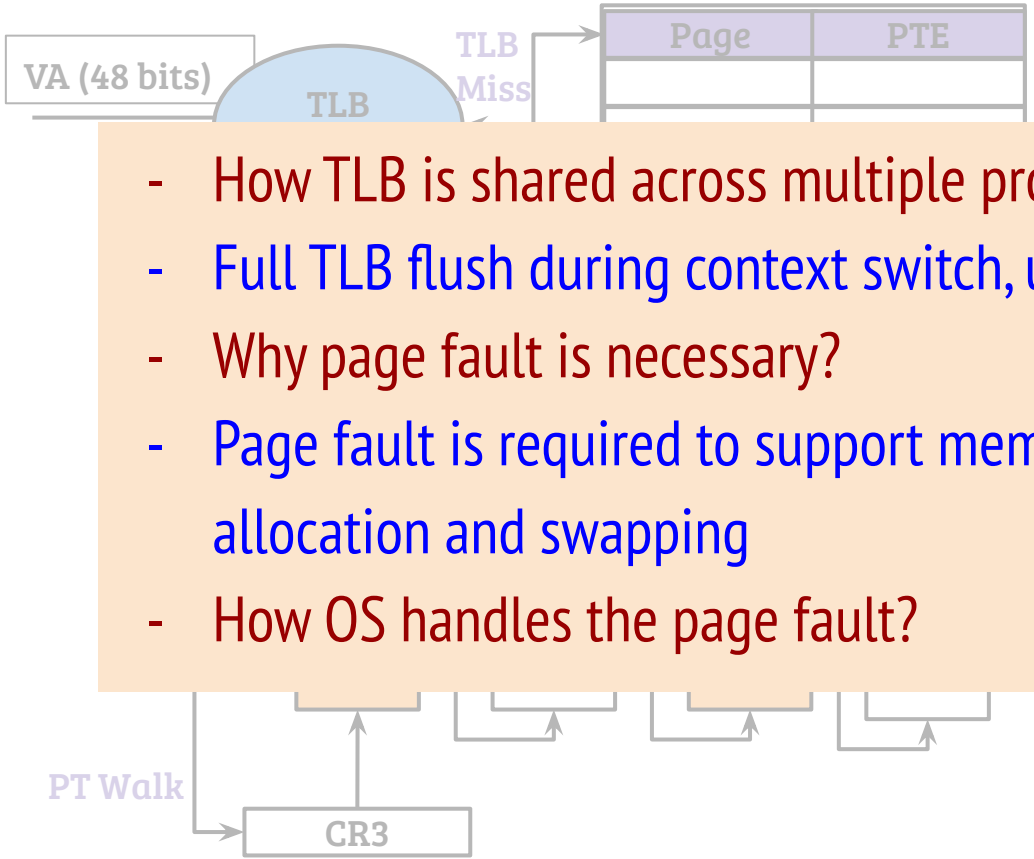
- TLB in the path of address

- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- How OS handles the page fault?

d

entries

Address translation (TLB + PTW)



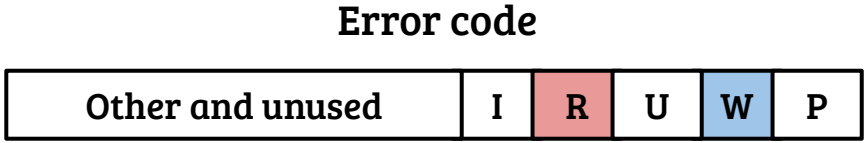
- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- Page fault is required to support memory over-commitment through lazy allocation and swapping
- How OS handles the page fault?

Page fault handling in X86: Hardware

```
If(  !pte.valid ||
    (access == write && !pte.write) ||
    (cpl != 0 && pte.priv == 0)){
          CR2 = Address;
          errorCode = pte.valid
                            | access << 1
                            | cpl << 2;
           Raise pageFault;
} // Simplified
```

Page fault handling in X86: Hardware

```
If( !pte.valid ||
    (access == write && !pte.write) ||
    (cpl != 0 && pte.priv == 0)){
        CR2 = Address;
        errorCode = pte.valid
                    | access << 1
                    | cpl << 2;
        Raise pageFault;
} // Simplified
```



- **P** — Present bit, 1 ⇒ fault is due to protection
- **W** — Write bit, 1 ⇒ Access is write
- **U** — Privilege bit, 1 ⇒ Access is from user mode
- **R** — Reserved bit, 1 ⇒ Reserved bit violation
- **I** — Fetch bit, 1 ⇒ Access is Instruction Fetch

- Error code is pushed into the kernel stack by the hardware (X86)

Page fault handling in X86: OS fault handler

```
HandlePageFault( u64 address, u64 error_code)
{
    If ( AddressExists(current → mm_state, address) &&
        AccessPermitted(current → mm_state, error_code) {
            PFN = allocate_pfn( );
            install_pte(address, PFN);
            return;
        }
    RaiseSignal(SIGSEGV);
}
```

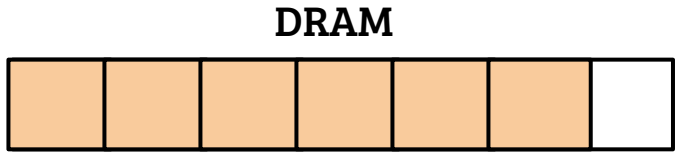
Address translation (TLB + PTW)

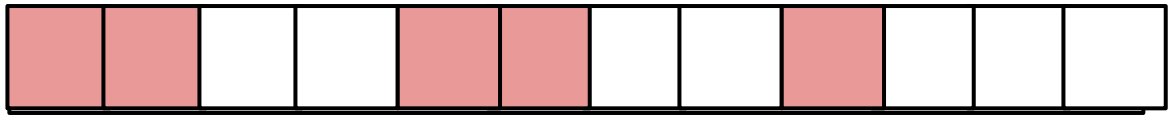
- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- Page fault is required to support memory over-commitment through lazy allocation and swapping
- How OS handles the page fault?
- The hardware invokes the page fault handler by placing the error code and virtual address. The OS handles the page fault either fixing it or raising a SEGFAULT.

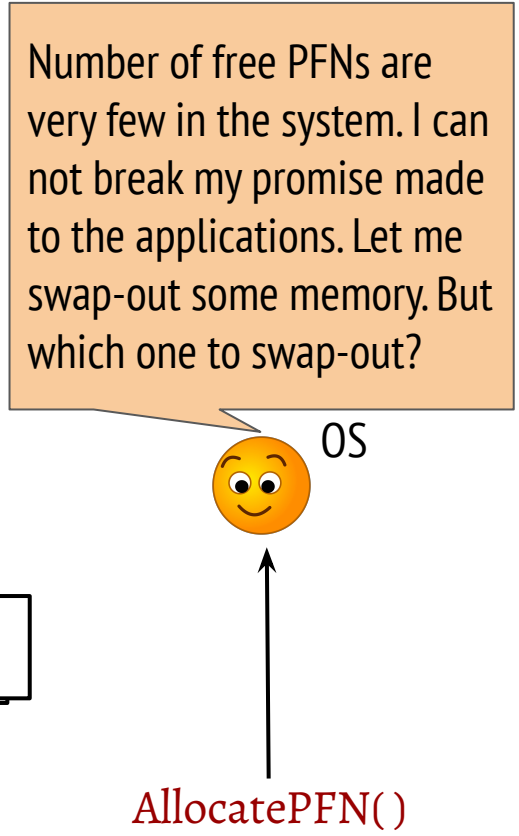
CS330: Operating Systems

Virtual memory: Page fault and Swapping

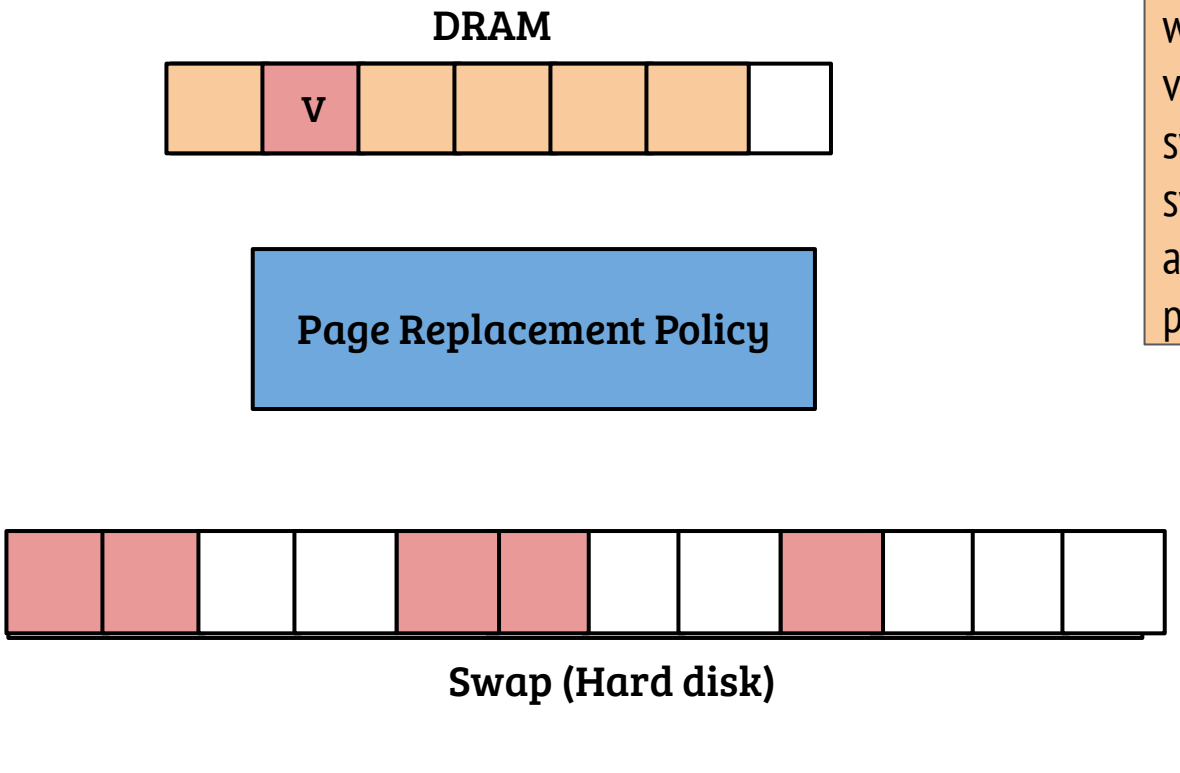
Swapping (swap-out)



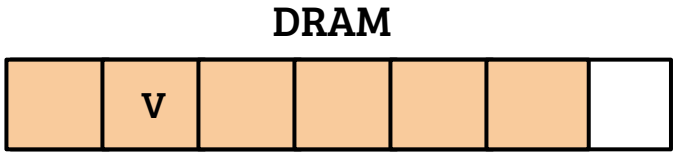




Swapping (swap-out)

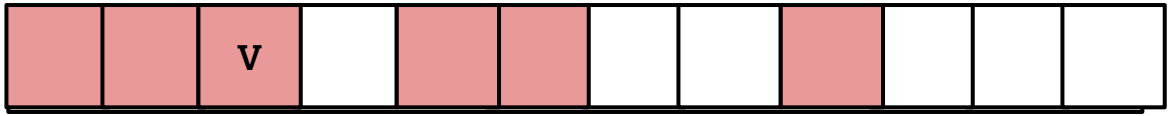


Swapping (swap-out)



PTE mapping the victim PFN (before swap)

PTE mapping the victim PFN (after swap)

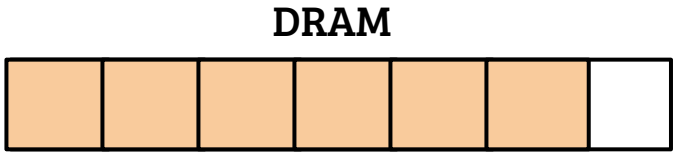


Swap (Hard disk)

Update the present-bit to 0 in the PTE such that any access to the page through the virtual address will result in a page fault. Also maintain the swap address in the PTE.

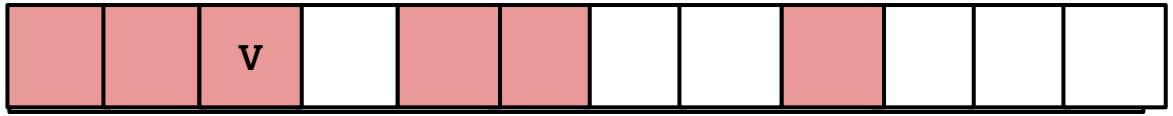
AllocatePFN( )

Swapping (swap-out)



PTE mapping the victim PFN (before swap)

PTE mapping the victim PFN (after swap)



Swap (Hard disk)

Content of the PFN is now in the swap device. In future, any translation using the PTE will result in a page fault. The page fault handler would copy it back from the swap device.

AllocatePFN( )

Page fault: Swap-in

```
HandlePageFault( u64 address, u64 error_code)
{
    If ( AddressExists(current → mm_state, address) &&
        AccessPermitted(current → mm_state, error_code) {
            PFN = allocate_pfn( );
            If ( is_swapped_pte(address) )      // Check if the PTE is swapped out
             swapin(getPTE(address), PFN);  // Copy the swap block  to PFN
            install_pte(address, PFN);          // and update the PTE
           return;
       }
   RaiseSignal(SIGSEGV);
}
```

Page replacement

- Objective: minimize number of page faults (due to swapping)
- We can model this problem with three parameters
  - A given sequence of access to virtual pages
  - # of memory pages (Frames)
  - Page replacement policy
- Metrics to measure the effectiveness: # of page faults, page fault rate, average memory access time

Belady's optimal algorithm (MIN)

- Strategy: Replace the page that will be referenced after the longest time
- Example:

  #of frames = 3

  Reference sequence (in temporal order)

  1, 3, 1, 5, 4, 1, 2, 5, 2, 2, 5, 3
- #of page faults = ?

Belady's optimal algorithm (MIN)

- Strategy: Replace the page that will be referenced after the longest time
- Example:

  #of frames = 3

  Reference sequence (in temporal order)

  1, 3, 1, 5, 4, 1, 2, 5, 2, 2, 5, 3
- #of page faults = 6 (3 cold-start misses result in page faults, no swapping)
- Belady's MIN is proven to be optimal, but impractical as it requires knowledge of future access

First In First Out (FIFO)

- Strategy: Replace the page that is in memory for the longest time
- Example:

  #of frames = 3

  Reference sequence (in temporal order)

  1, 3, 1, 5, 4, 1, 2, 5, 2, 2, 5, 3
- #of page faults = ?

First In First Out (FIFO)

- Strategy: Replace the page that is in memory for the longest time
- Example:

  #of frames = 3

  Reference sequence (in temporal order)

  1, 3, 1, 5, 4, 1, 2, 5, 2, 2, 5, 3
- #of page faults = 8 (3 cold-start misses)
- FIFO suffers from an anomaly known as Belady's anomaly
  - With increased #of frames, #of page fault may also increase!

First In First Out (FIFO)

- Strategy: Replace the page that is in memory for the longest time
- Example:

  #of frames = 3

  Reference sequence (in temporal order)

  1, 3, 1, 5, 4, 1, 2, 5, 2, 2, 5, 3
- #of page faults = 8 (3 cold-start misses)
- FIFO suffers from an anomaly known as Belady's anomaly
  - With increased #of frames, #of page fault may also increase!
  - Example access sequence: 0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4
  - #of page faults with 3 frames < #of page faults with 4 frames

Least recently used (LRU)

- Strategy: Replace the page that is not referenced for the longest time
- Example:

  #of frames = 3

  Reference sequence (in temporal order)

  1, 3, 1, 5, 4, 1, 2, 5, 2, 2, 5, 3
- #of page faults = ?

Least recently used (LRU)

- Strategy: Replace the page that is not referenced for the longest time
- Example:

  #of frames = 3

  Reference sequence (in temporal order)

  1, 3, 1, 5, 4, 1, 2, 5, 2, 2, 5, 3
- #of page faults = 7 (3 cold-start)
- LRU shown to be useful for workloads with access locality
- Implementation of LRU using the accessed-bit is not easy, approximated using CLOCK