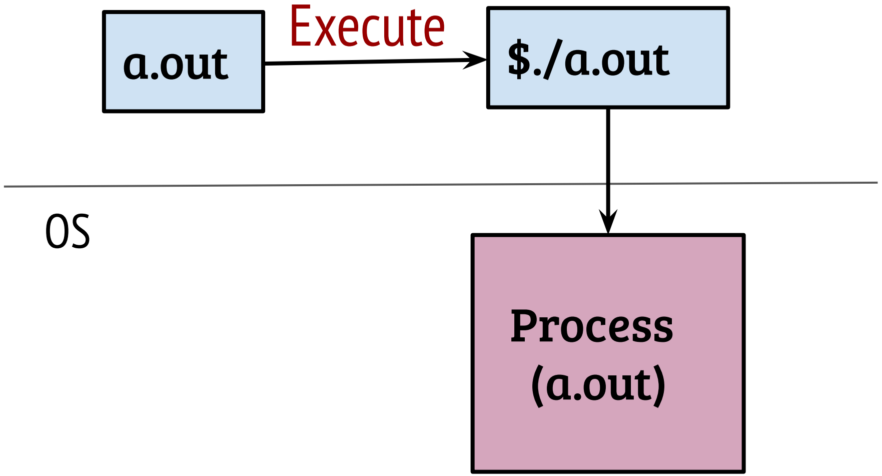
CS330: Operating Systems

Virtual memory: Address spaces

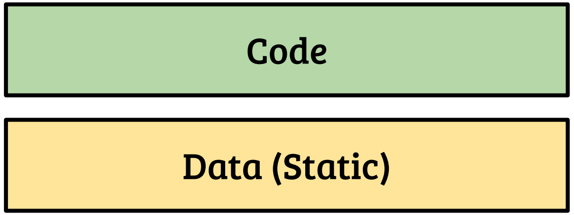
Recap: The process abstraction

- The OS creates a *process* when we run an *executable*



- Executable is a file, stored in a persistent storage (e.g., disk)
- To run, the process code and data should reside in memory
- Run-time memory allocation and deallocation should be supported

Executable file to process memory view



- A typical executable file contains code and statically allocated data
- Statically allocated: global and static variables
- Is loading the program (code and data) sufficient for program execution?

Executable file to process memory view

Code

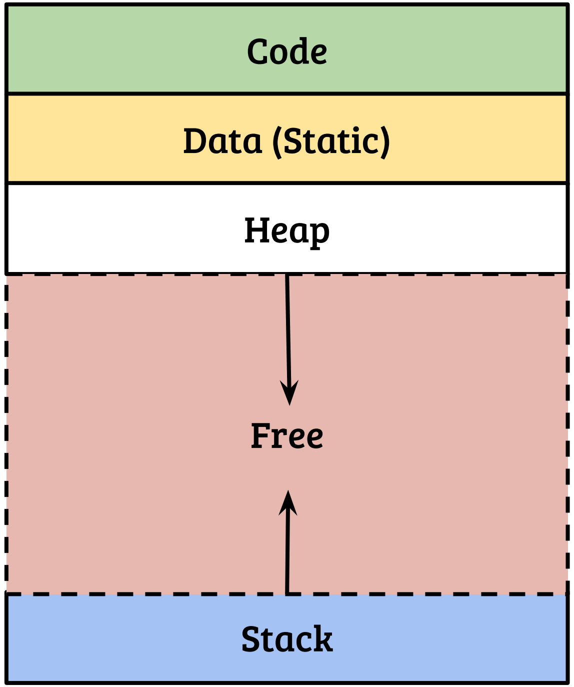
Data (Static)

Stack

Heap

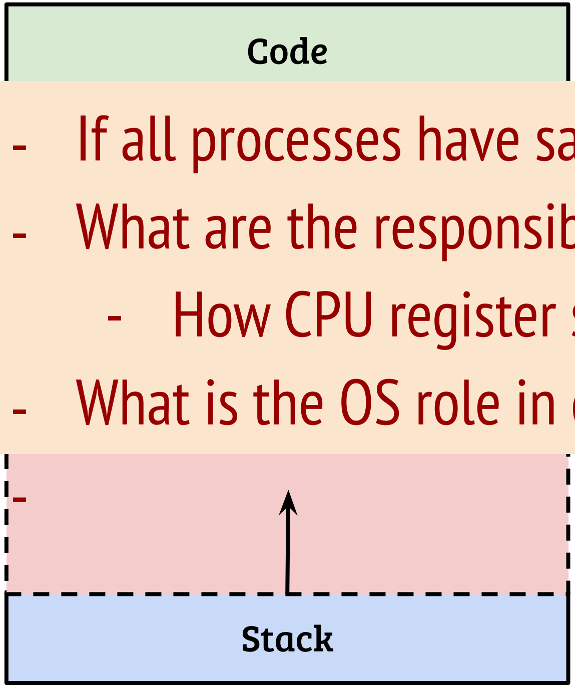
- A typical executable file contains code and statically allocated data
- Statically allocated: global and static variables
- Is loading the program (code and data) sufficient for program execution?
- No, we need memory for stack and dynamic allocation
- Stack: function call and return, store local (stack) variables
- Heap: dynamic memory allocation through APIs like *malloc( )*

The address space abstraction



- Address space represents memory state of a process
- Address space layout is same for all the processes (convinience)
- Exact layout can be decided by the OS, conventional layout is shown

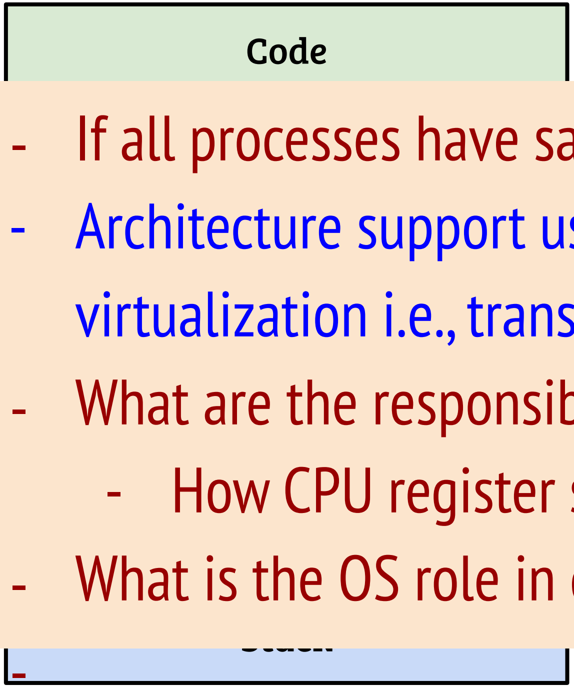
The address space abstraction



- If all processes have same address space, how they map to actual memory?
- What are the responsibilities of the OS during program load?
  - How CPU register state is changed?
- What is the OS role in dynamic memory allocation?

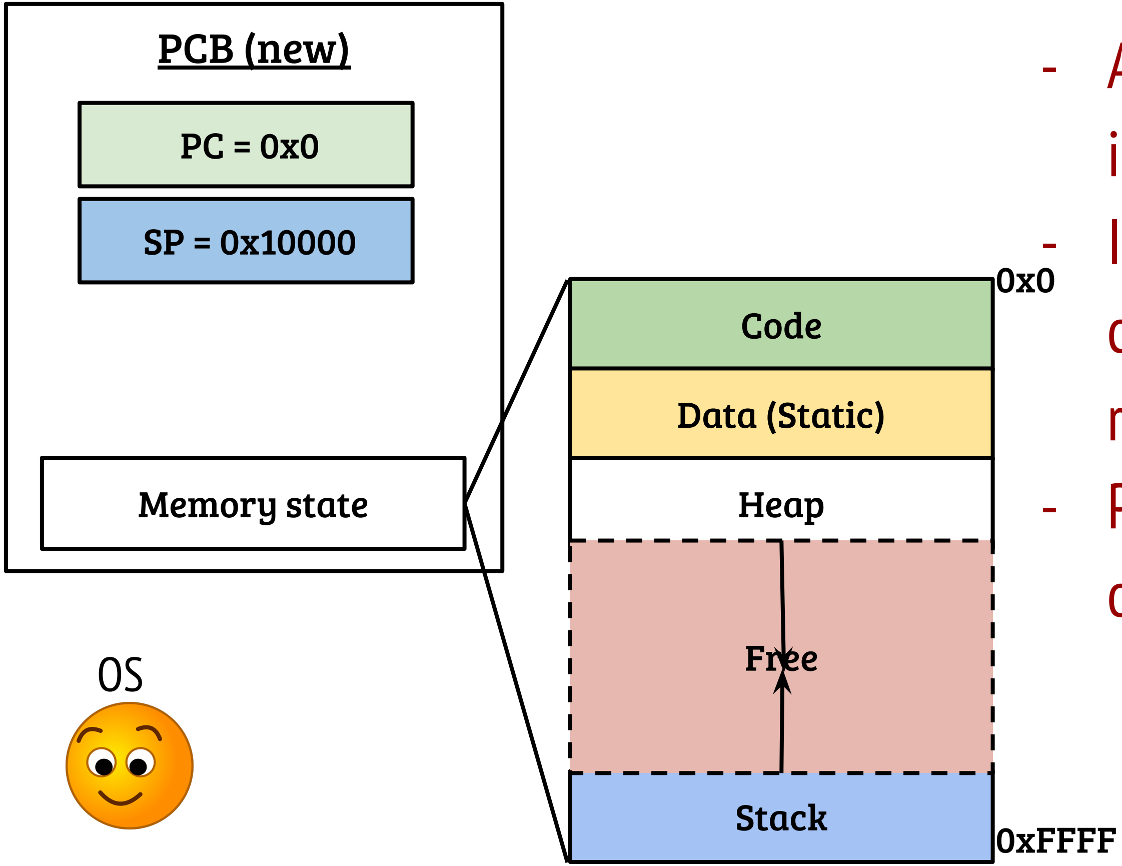
Exact layout can be decided by the OS, conventional layout is shown

The address space abstraction



- If all processes have same address space, how they map to actual memory?
- Architecture support used by OS techniques to perform memory virtualization i.e., translate virtual address to physical address (will revisit)
- What are the responsibilities of the OS during program load?
  - How CPU register state is changed?
- What is the OS role in dynamic memory allocation?

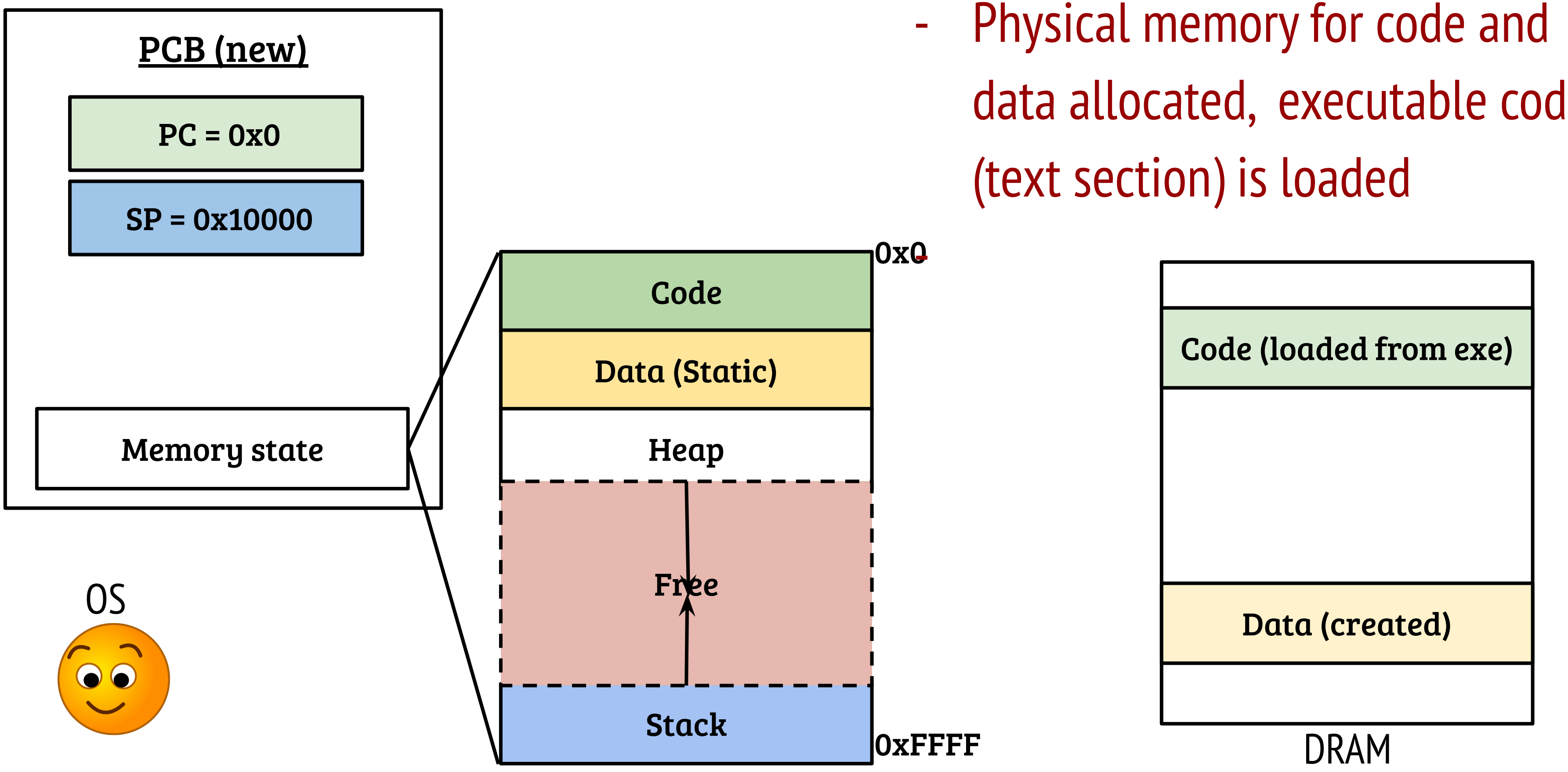
OS during program load (exec)



- A fresh address space is initialized
- In reality, parent address space copied at the time of fork( ) is reset
- PC and SP are set with addresses of code and stack, respectively

OS during program load (exec)

- Physical memory for code and data allocated, executable code (text section) is loaded



PCB (new)

PC = 0x0

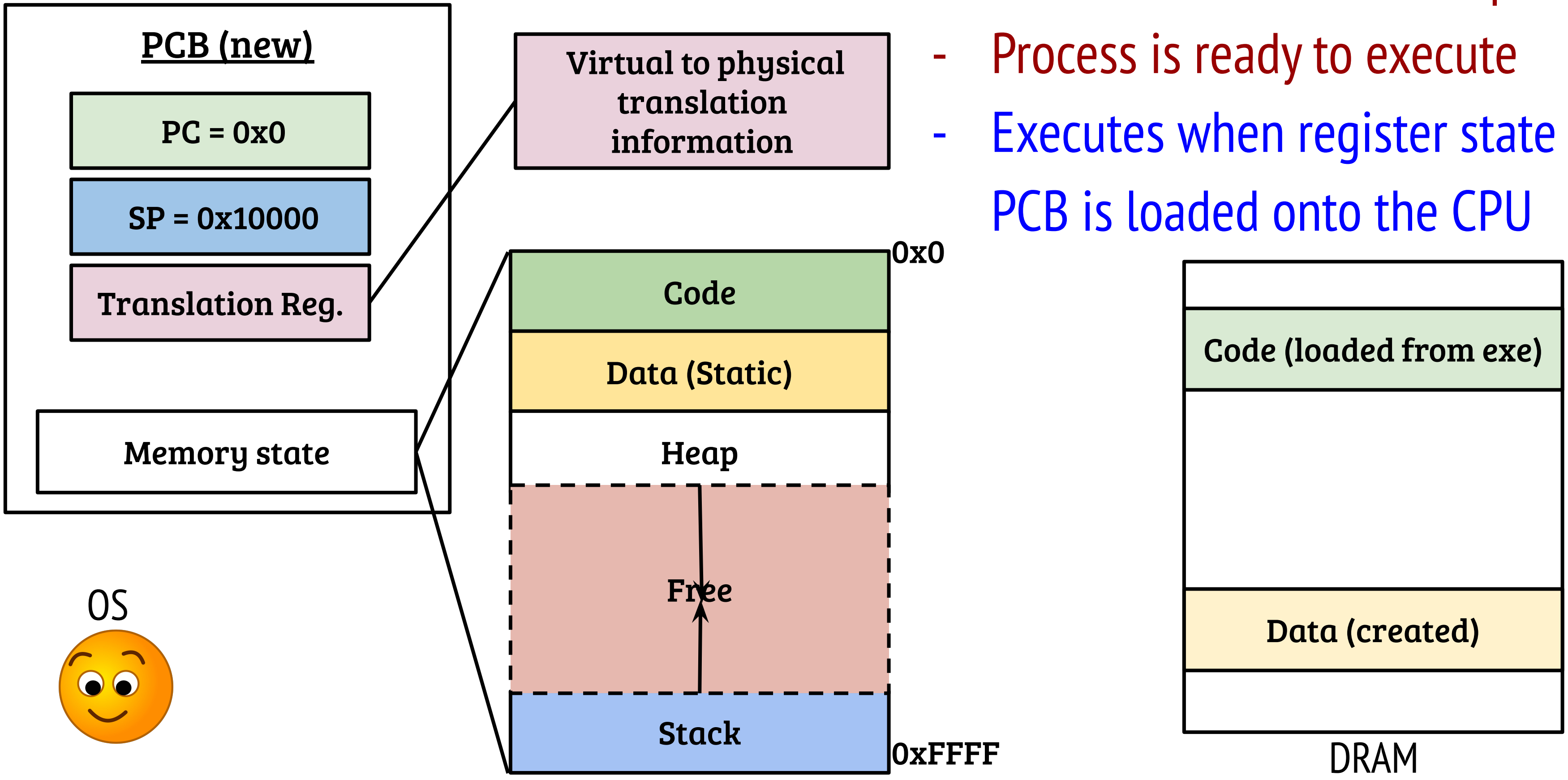
SP = 0x10000

Memory state

OS

OS during program load (exec)

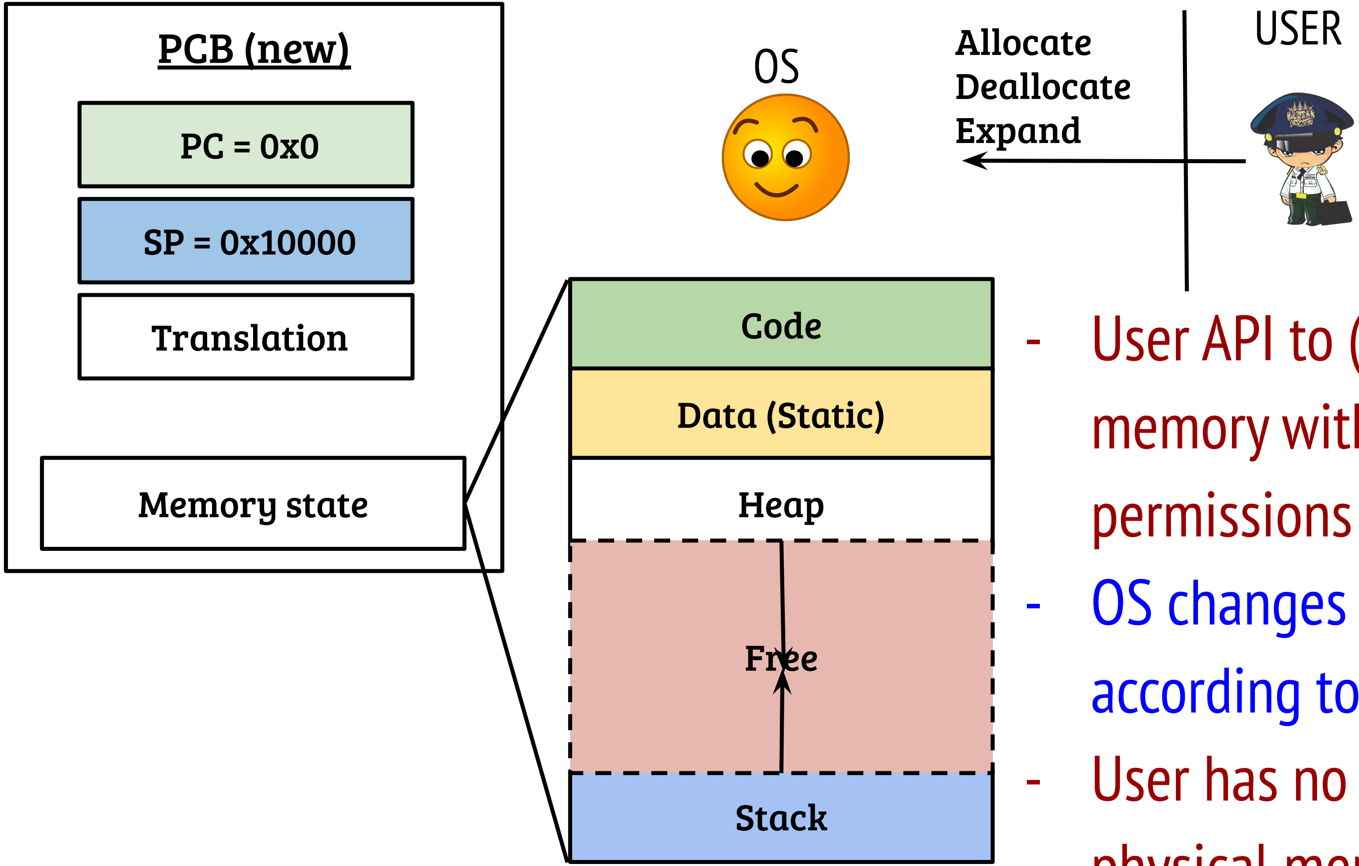
- Translation information updated
- Process is ready to execute
- Executes when register state in PCB is loaded onto the CPU



The address space abstraction

- If all processes have same address space, how they map to actual memory?
- Architecture support used by OS techniques to perform memory virtualization i.e., translate virtual address to physical address (will revisit)
- What are the responsibilities of the OS during program load?
  - How CPU register state is changed?
- Creating address space, loading binary, updating the PCB register state
- What is the OS role in dynamic memory allocation?

User API for memory management



PCB (new)

PC = 0x0

SP = 0x10000

Translation

Memory state

Code

Data (Static)

Heap

Free

Stack

OS

USER

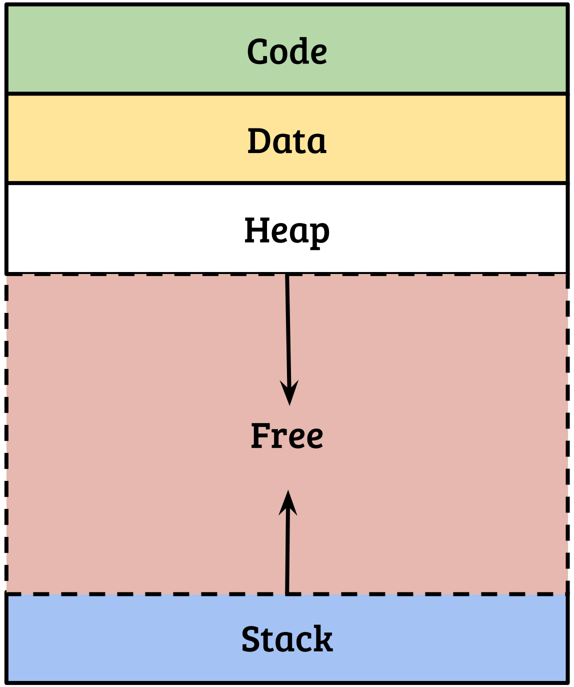
Allocate
Deallocate
Expand

- User API to (de)allocate heap memory with different access permissions
- OS changes the memory state according to the user request
- User has no direct control on physical memory

CS330: Operating Systems

Virtual memory: Memory API

Recap: Process address space

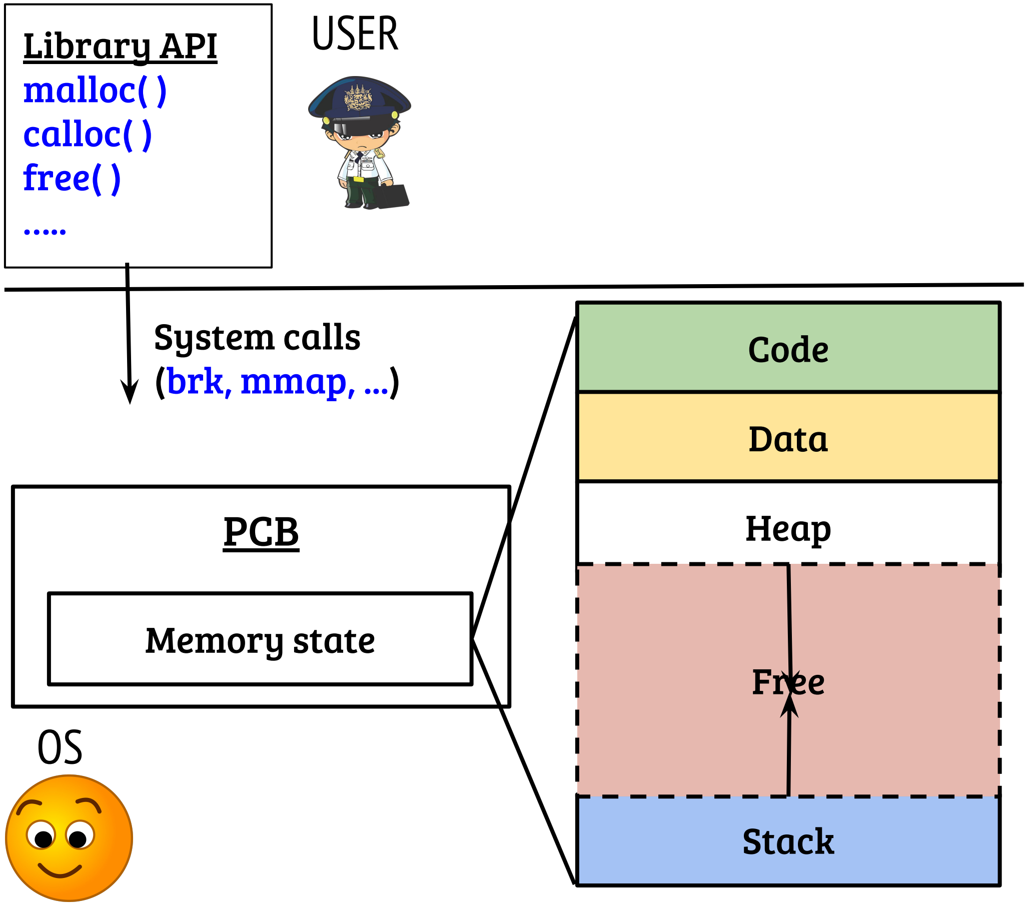


- Address space provides a unique view of memory to *all processes*
  - Address space is virtual
  - OS enables this virtual view

Recap: Process address space

- If all processes have same address space, how they map to actual memory?
- Architecture support used by OS techniques to perform memory virtualization i.e., translate virtual address to physical address (will revisit)
- What are the responsibilities of the OS during program load?
  - How CPU register state is changed?
- Creating address space, loading binary, updating the PCB register state
- What is the role of OS in dynamic memory allocation?

User API for memory management



Library API
malloc( )
calloc( )
free( )
.....

USER

System calls
(brk, mmap, ...)

PCB

Memory state

OS

Code

Data

Heap

Free

Stack

- Generally, user programs use library routines to allocate/deallocate memory
- OS provides some address space manipulation system calls (today's agenda)

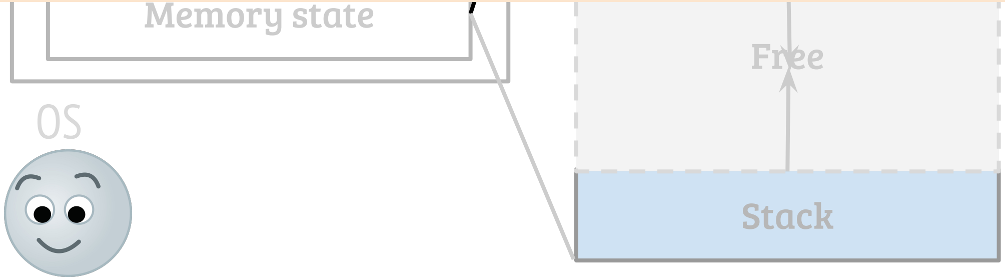
User API for memory management

Library API
malloc( )
calloc( )

USER

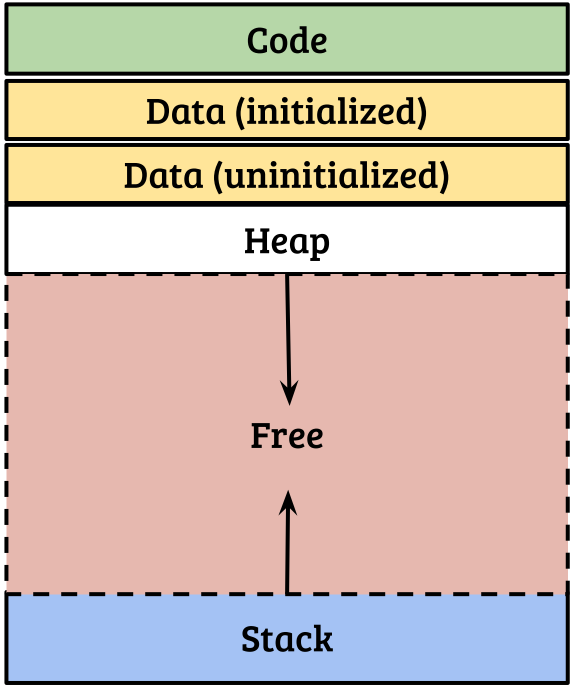
- Generally, user programs

- Can the size of segments change at runtime? If yes, which ones and how?
- How can we know about the segment layout at program load and runtime?
- How to allocate memory chunks with different permissions?
- What is the structure of PCB memory state?



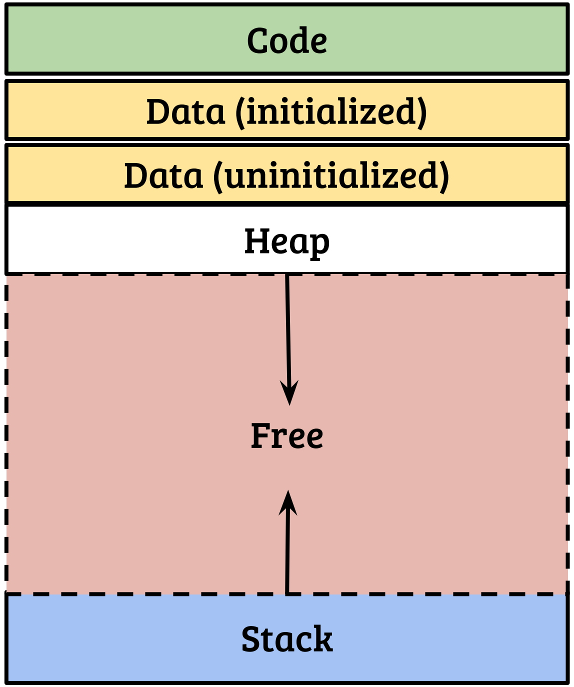
calls (today's agenda)

Dynamically sizing the segments (UNIX)



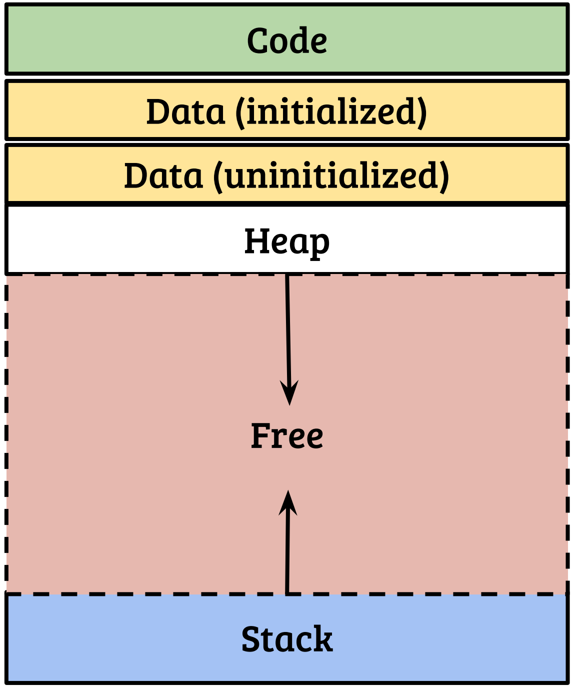
- Code segment size and initialized data segment size is fixed (at exe load)

Dynamically sizing the segments (UNIX)



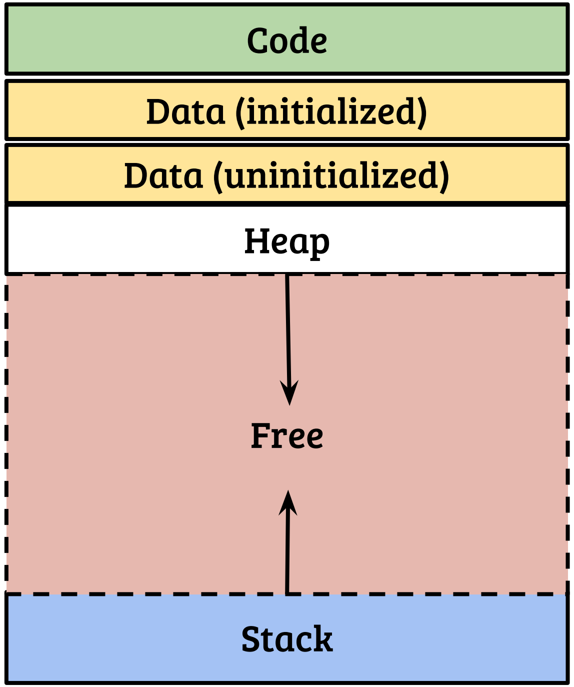
- Code segment size and initialized data segment size is fixed (at exe load)
- End of uninitialized data segment (a.k.a. BSS) can be adjusted dynamically

Dynamically sizing the segments (UNIX)



- Code segment size and initialized data segment size is fixed (at exe load)
- End of uninitialized data segment (a.k.a. BSS) can be adjusted dynamically
- Heap allocation can be discontinuous, special system calls like mmap( ) provide the facility

Dynamically sizing the segments (UNIX)



- Code segment size and initialized data segment size is fixed (at exe load)
- End of uninitialized data segment (a.k.a. BSS) can be adjusted dynamically
- Heap allocation can be discontinuous, special system calls like mmap( ) provide the facility
- Stack grows automatically based on the run-time requirements, no explicit system calls

Sliding the BSS (brk, sbrk)

int brk(void *address);

- If possible, set the end of uninitialized data segment (BSS) at address (address = heap start address)
- Can be used by C library to allocate/free memory dynamically

void * sbrk (long size);

- Increments the program's data space by size bytes and returns the old value of the end of bss
- sbrk(0) returns the current location of BSS

Finding the segments

- etext, edata and end variables mark the end of text segment, initialized data segment and the BSS, respectively (At program load)
- sbrk(0) can be used to find the start of heap segment
- Printing the address of functions and variables
- Linux provides the information in /proc/pid/maps

User API for memory management

Library API

USER

- Can the size of segments change at runtime? If yes, which ones and how?
- Heap and data segments can be adjusted using brk and sbrk
- How can we know about the segment layout at program load and runtime?
- Using predefined variables, sbrk, proc file system (Linux)
- How to allocate memory chunks with different permissions?
- What is the structure of PCB memory state?

Discontiguous allocation (mmap)

- mmap( ) is a powerful and multipurpose system call to perform dynamic and discontiguous allocation (explicit OS support)
- Allows to allocate address space
    - with different protections (READ/WRITE/EXECUTE)
    - at a particular address provided by the user
- Example: Allocate 4096 bytes with READ+WRITE permission

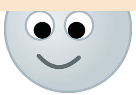
```
ptr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_ANONYMOUS
|MAP_PRIVATE, -1, 0);  // See the man page for details
```

User API for memory management

Library API

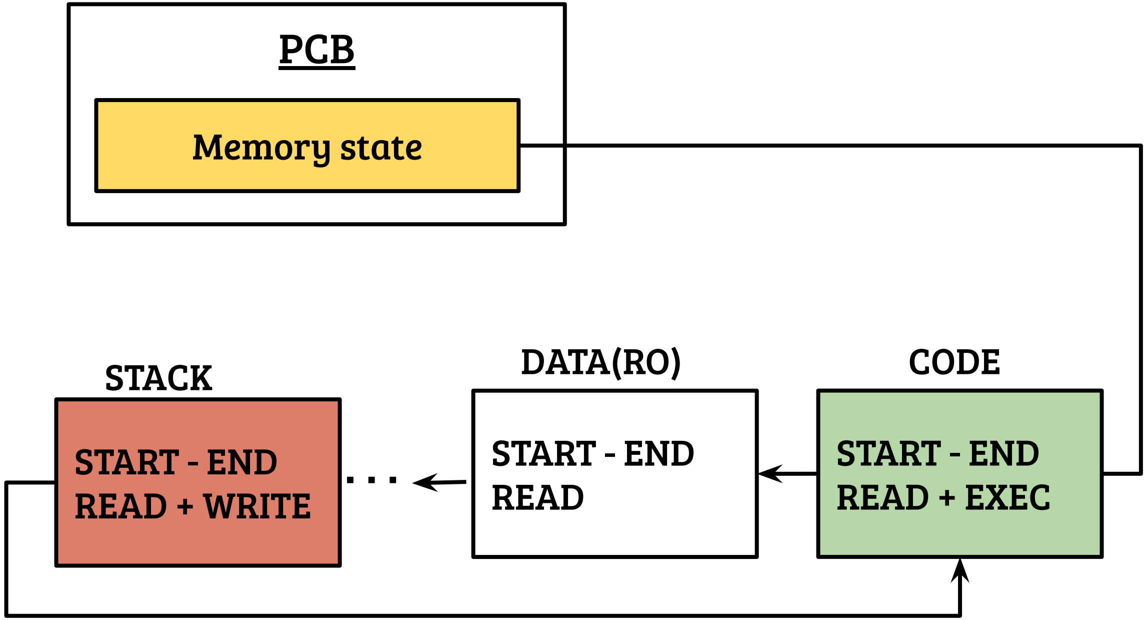
USER

- Can the size of segments change at runtime? If yes, which ones and how?
- Heap and data segments can be adjusted using brk and sbrk
- How can we know about the segment layout at program load and runtime?
- Using predefined variables, sbrk, proc file system (Linux)
- How to allocate memory chunks with different permissions?
- mmap( ) supports discontinuous allocation with different permissions
- What is the structure of PCB memory state?



Stack

Memory state of PCB (example)



- Maintained as a sorted circular list accessible from PCB
- START and END never overlap between two segment areas
- Can merge/extend areas if permissions match

User API for memory management

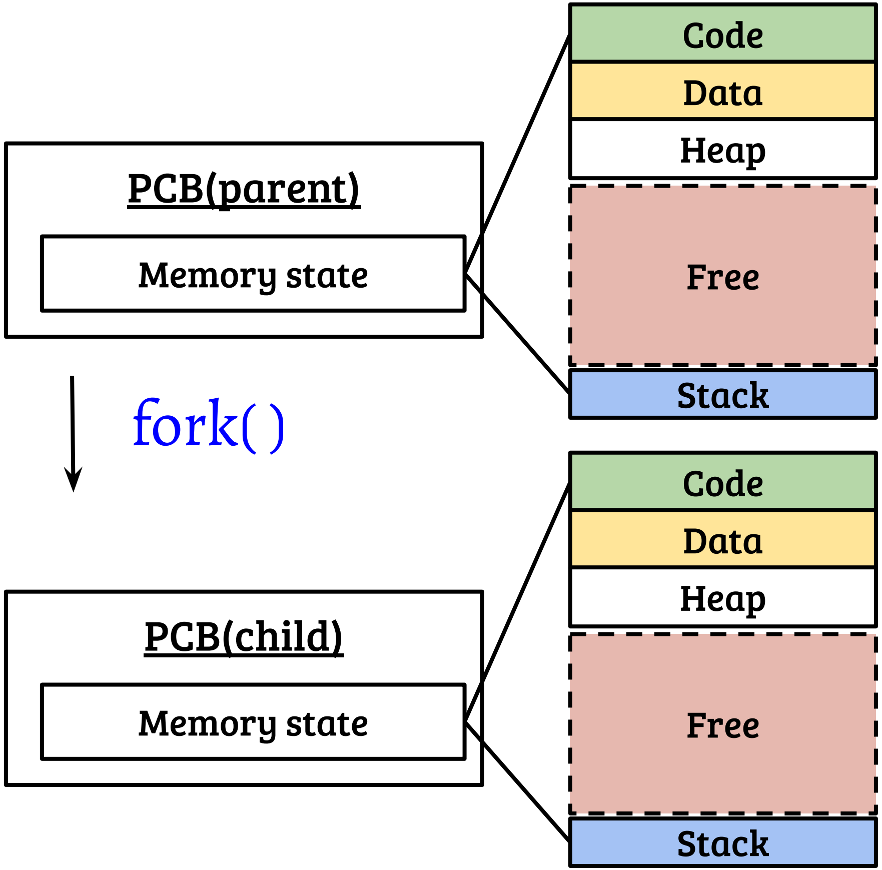
Library API

USER

- Can the size of segments change at runtime? If yes, which ones and how?
- Heap and data segments can be adjusted using brk and sbrk
- How can we know about the segment layout at program load and runtime?
- Using predefined variables, sbrk, proc file system (Linux)
- How to allocate memory chunks with different permissions?
- mmap( ) supports discontinuous allocation with different permissions
- What is the structure of PCB memory state?
- A sorted data structure of allocated areas can be used

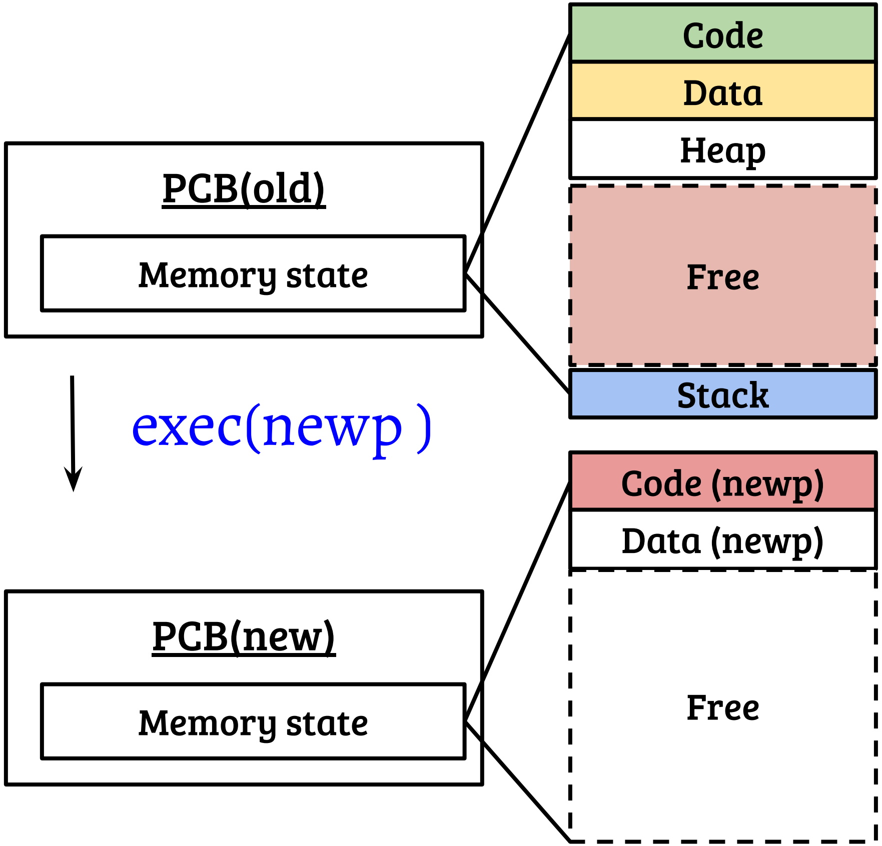
Stack

Inheriting address space through fork( )



- Child inherits the memory state of the parent
  - The memory state data structures are copied into the child PCB
- Any change through mmap( ) or brk( ) is per-process

Overriding address space through exec( )

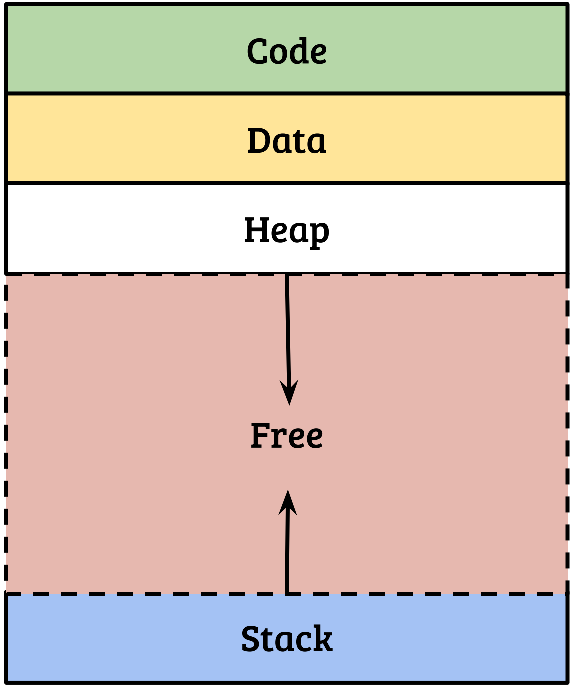


- The address space is reinitialized using the new executable
- Changes to newly created address space depends on the logic of new process

CS330: Operating Systems

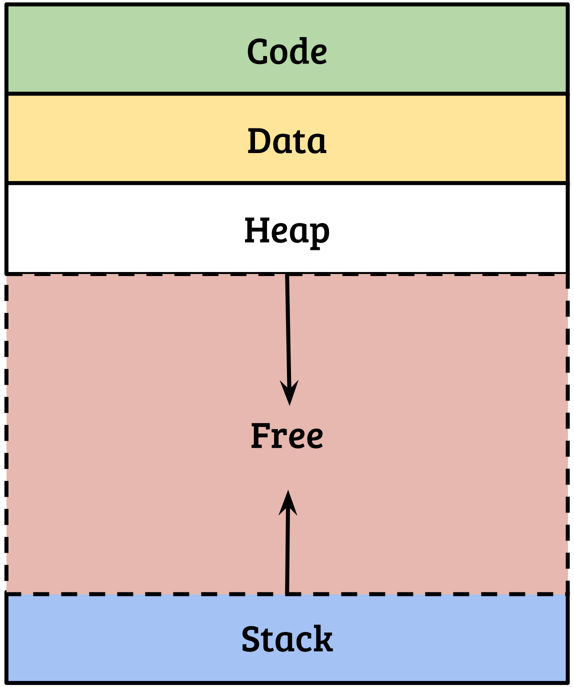
Virtual memory: Address translation

Recap: Process address space



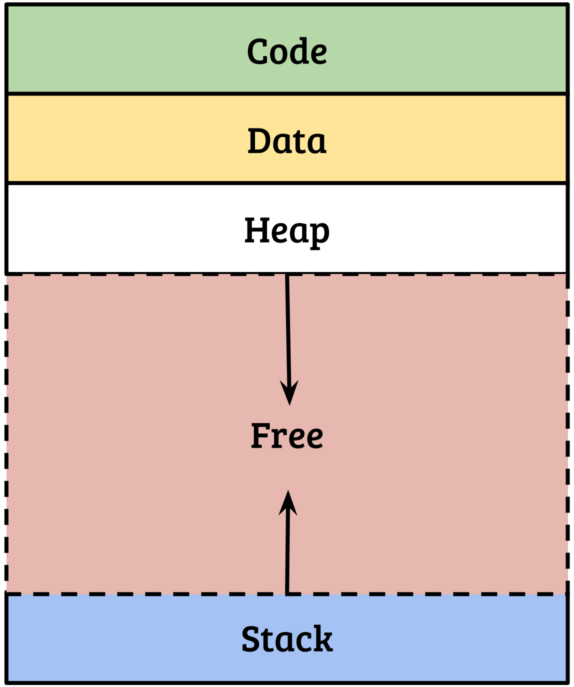
- Address space provides a unique view of memory to *all processes*
    - Address space is virtual
    - OS enables this virtual view

Recap: Process address space



- Address space provides a unique view of memory to *all processes*
  - Address space is virtual
  - OS enables this virtual view
- User can organize/manage virtual memory using OS APIs
  - No control on physical memory!

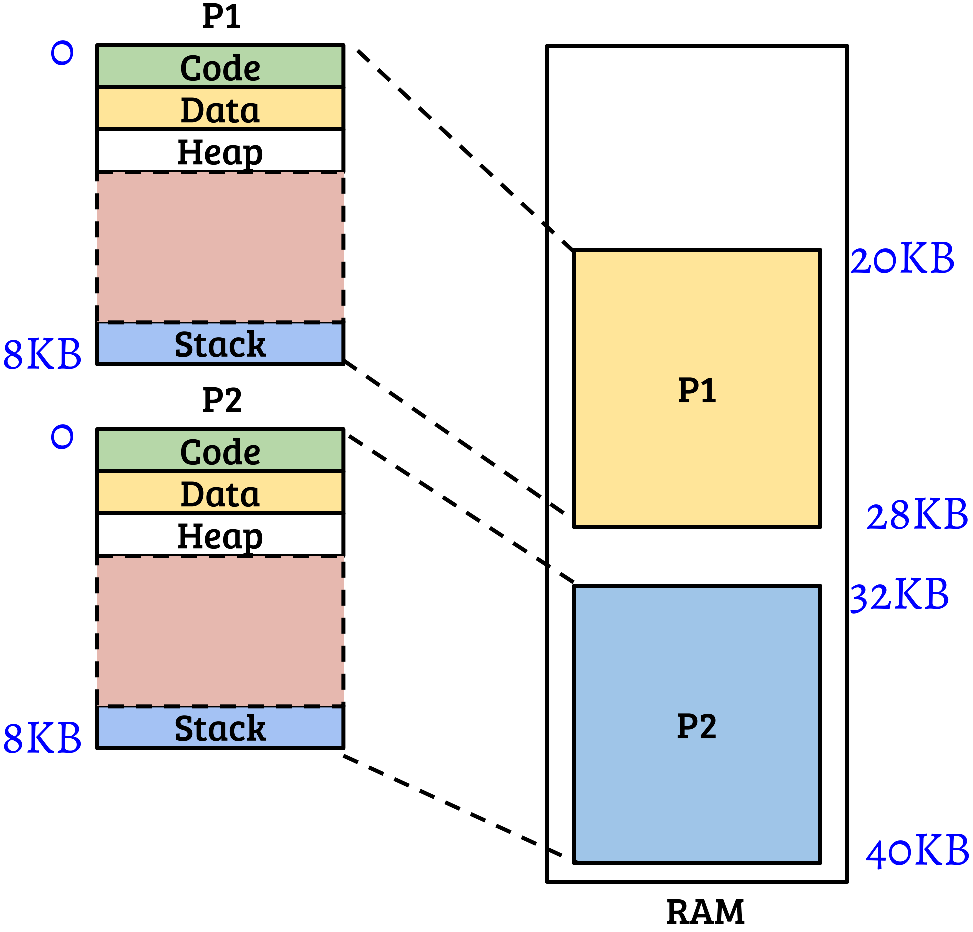
Recap: Process address space



- Address space provides a unique view of memory to *all processes*
  - Address space is virtual
  - OS enables this virtual view
- User can organize/manage virtual memory using OS APIs
  - No control on physical memory!

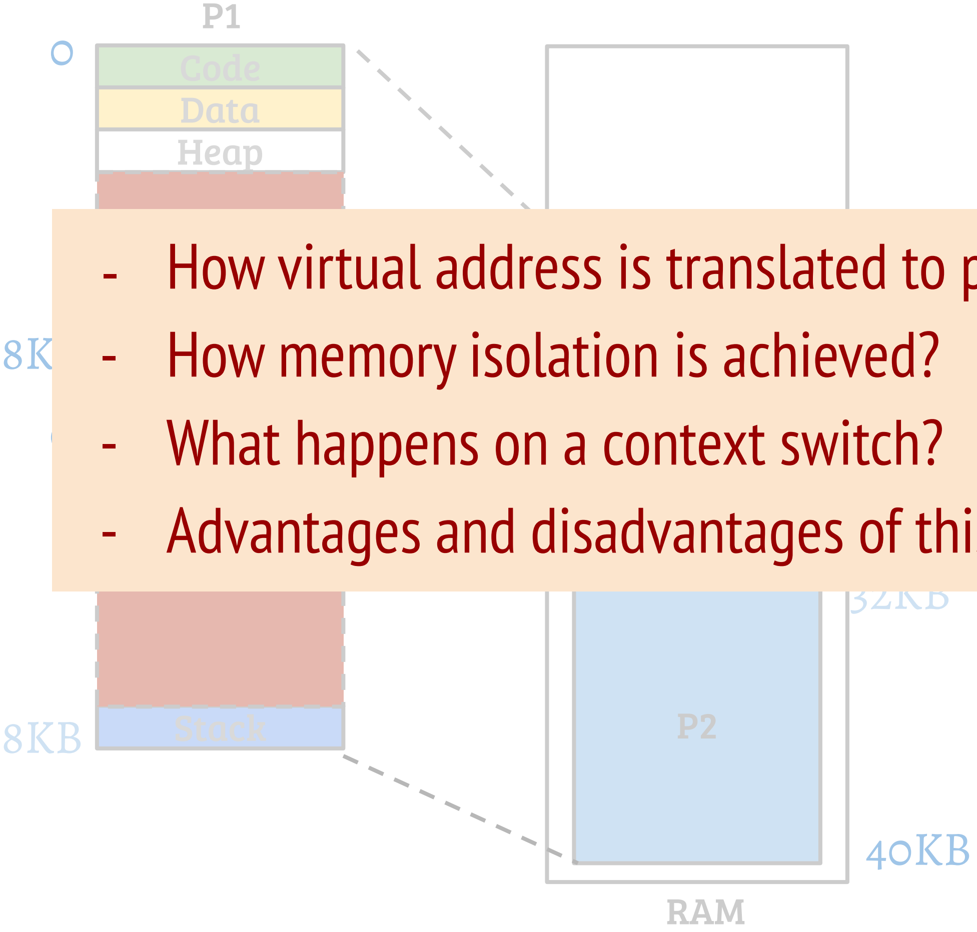
Today’s agenda: Virtual to physical address translation

Translation at address space granularity



- Physical memory of same size as the address space is allocated to each process
- Physical memory for a process can be at any address, but should be contiguous

Translation at address space granularity



- Physical memory of same size as

n

contiguous

- How virtual address is translated to physical address?
- How memory isolation is achieved?
- What happens on a context switch?
- Advantages and disadvantages of this scheme

Role of the compiler

Simple function

```
func( )
{
  int a = 100;
  a+ = 10;
}
```

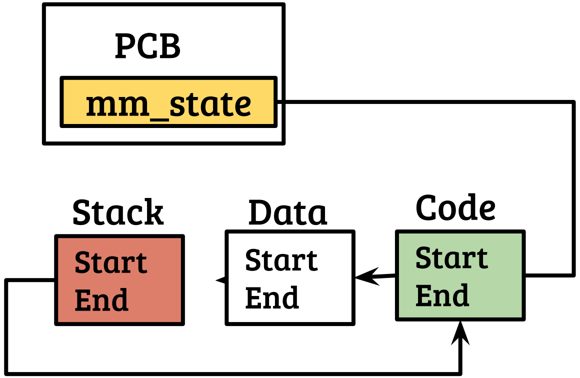
Compiled assembly

```
............
func:
10:    push %rbp;
12:    mov %rsp, %rbp;
15:    mov (%rbp), %rax
18:    add $10, %rax
21:    mov %rax, (%rbp)
24:    pop %rbp;
26:    ret;
.............
```

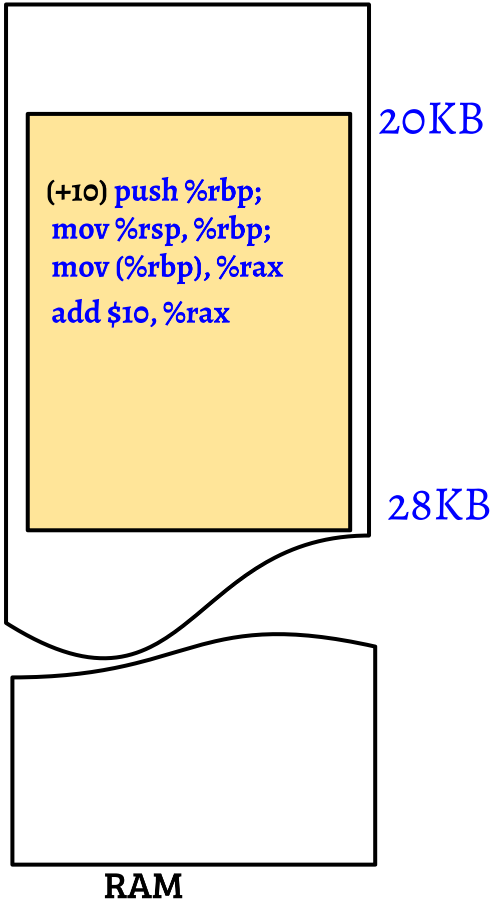
- Compiler generates the code with starting address zero
- Compiler does not know the stack address, blindly uses the registers (rbp, rsp)!

OS during binary load (simplified exec)

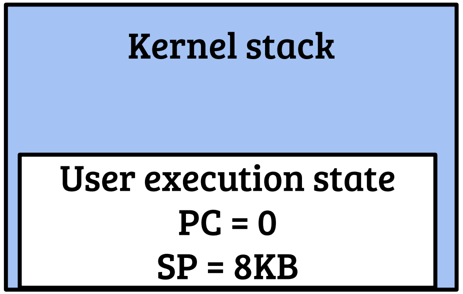
```
load_new_executable( PCB *current, File *exe)
{
  reinit_address_space(current → mm_state);
  allocate_phys_mem(current);
  load_exe_to_physmem(current, exe);
  set_user_sp(current → mm_state → stack_start);
  set_user_pc(current → mm_state → code_start);
  return_to_user;
}
```



Process state after exec( )

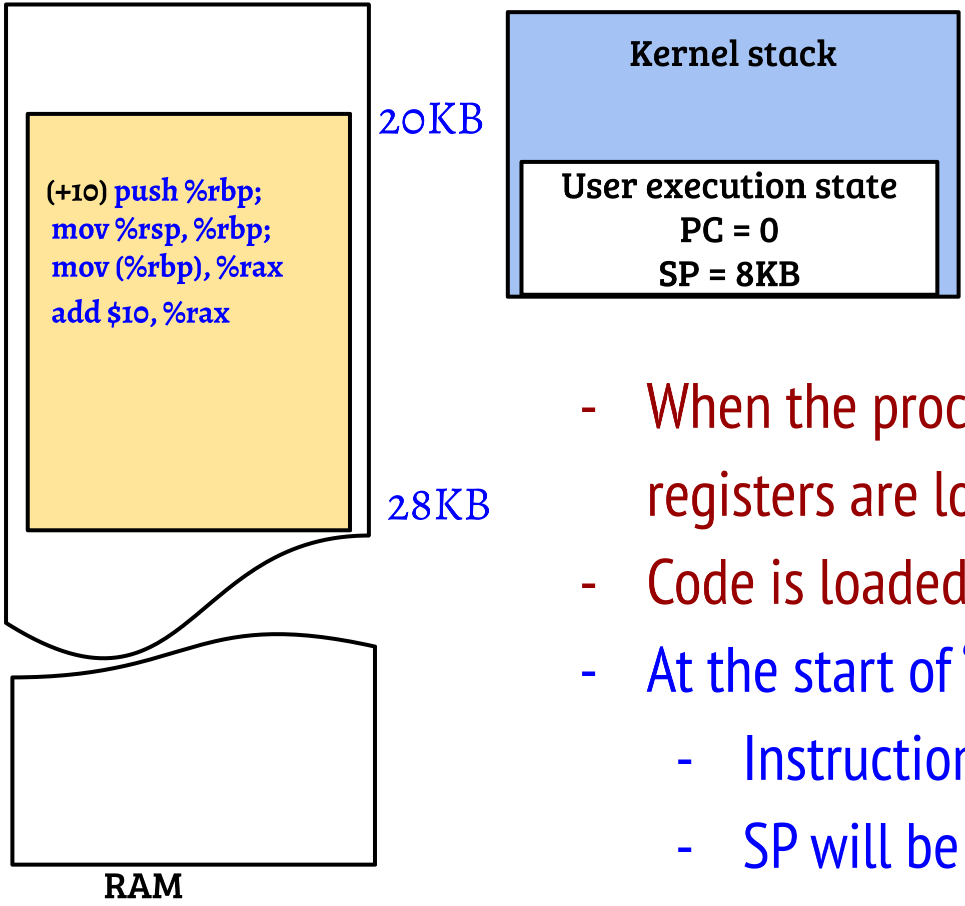


```
(+10) push %rbp;
 mov %rsp, %rbp;
 mov (%rbp), %rax
 add $10, %rax
```



- When the process returns to user space, the registers are loaded with virtual addresses
- Code is loaded into physical memory (@20KB)

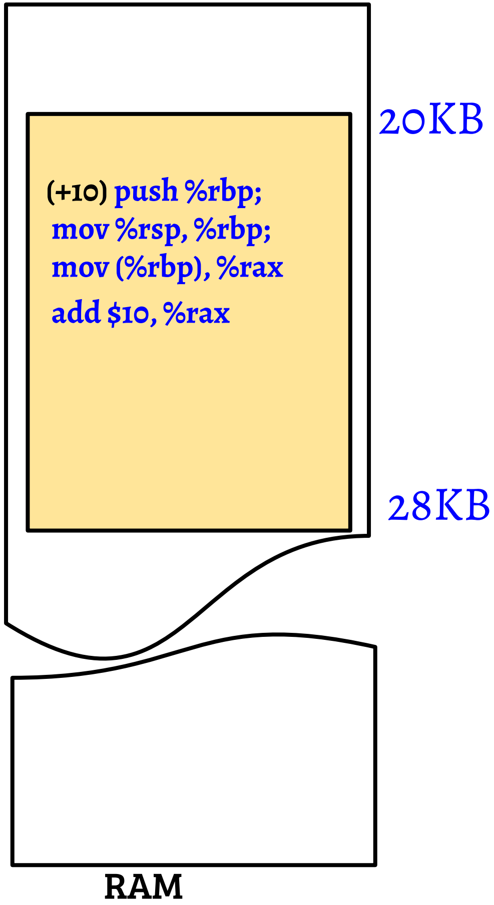
Process state after exec( )



```
(+10) push %rbp;
 mov %rsp, %rbp;
 mov (%rbp), %rax
 add $10, %rax
```

- When the process returns to user space, the registers are loaded with virtual addresses
- Code is loaded into physical memory (@20KB)
- At the start of “func” execution
  - Instruction fetch address is 10 (PC = 10)
  - SP will be around 8KB

Process state after exec( )



```
(+10) push %rbp;
 mov %rsp, %rbp;
 mov (%rbp), %rax
 add $10, %rax
```

Kernel stack

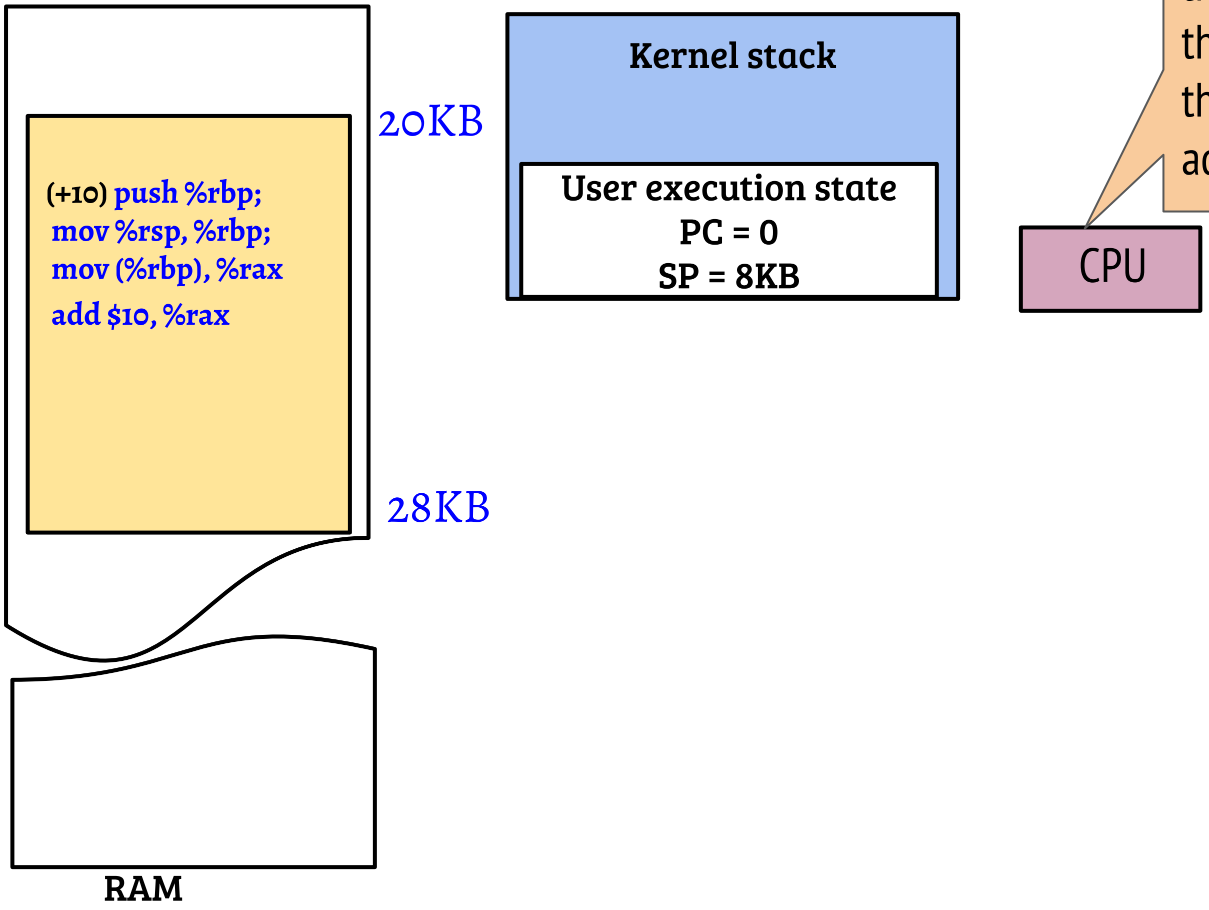
User execution state
PC = 0
SP = 8KB

OS

Dear HW! I have done my part. Help me with the translation, please!

- When the process returns to user space, the registers are loaded with virtual addresses
- Code is loaded into physical memory (@20KB)
- At the start of “func” execution
  - Instruction fetch address is 10
  - RSP will be around 8KB

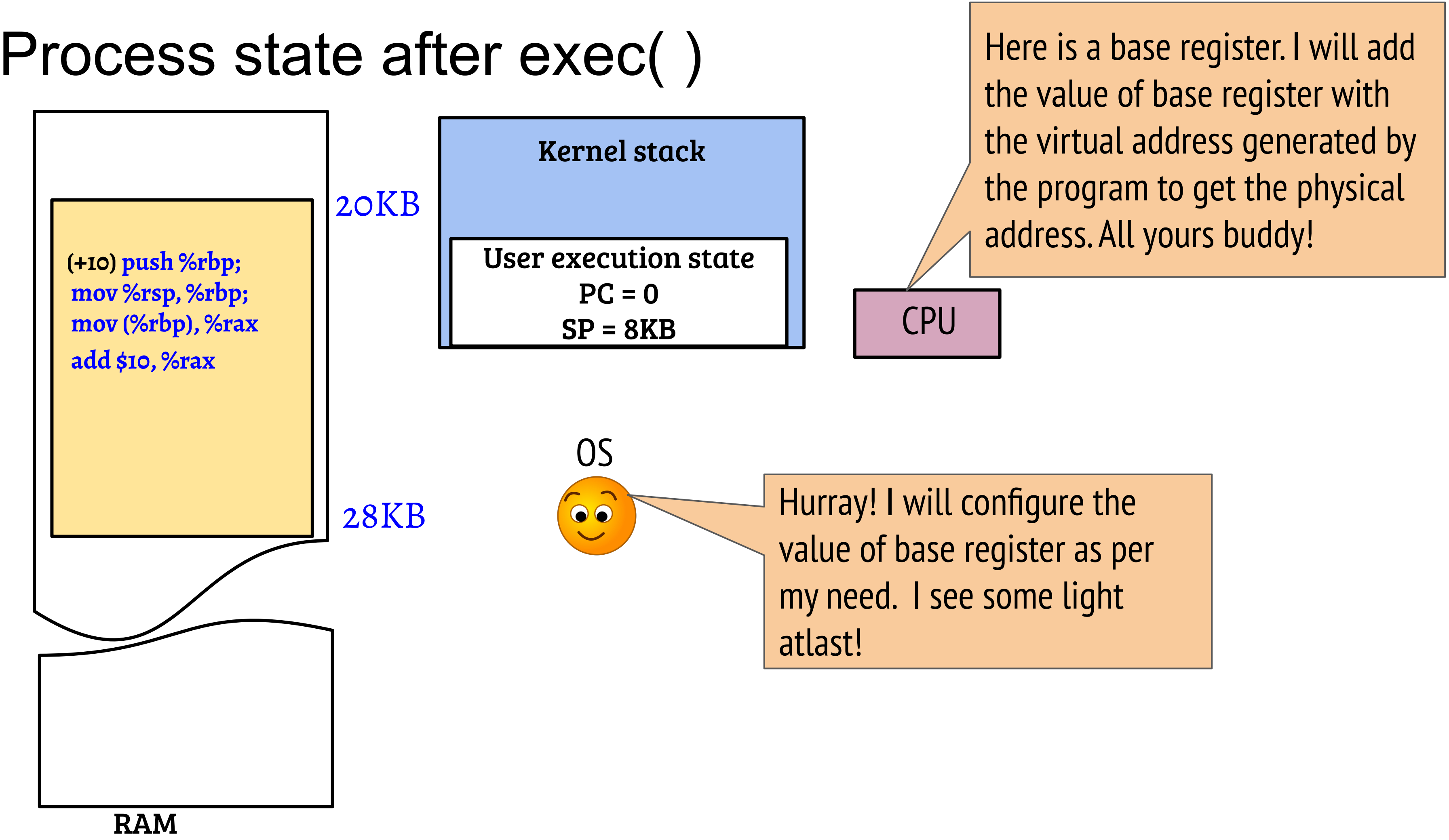
Process state after exec( )



```
(+10) push %rbp;
 mov %rsp, %rbp;
 mov (%rbp), %rax
 add $10, %rax
```

Here is a base register. I will add the value of base register with the virtual address generated by the program to get the physical address. All yours buddy!

Process state after exec( )

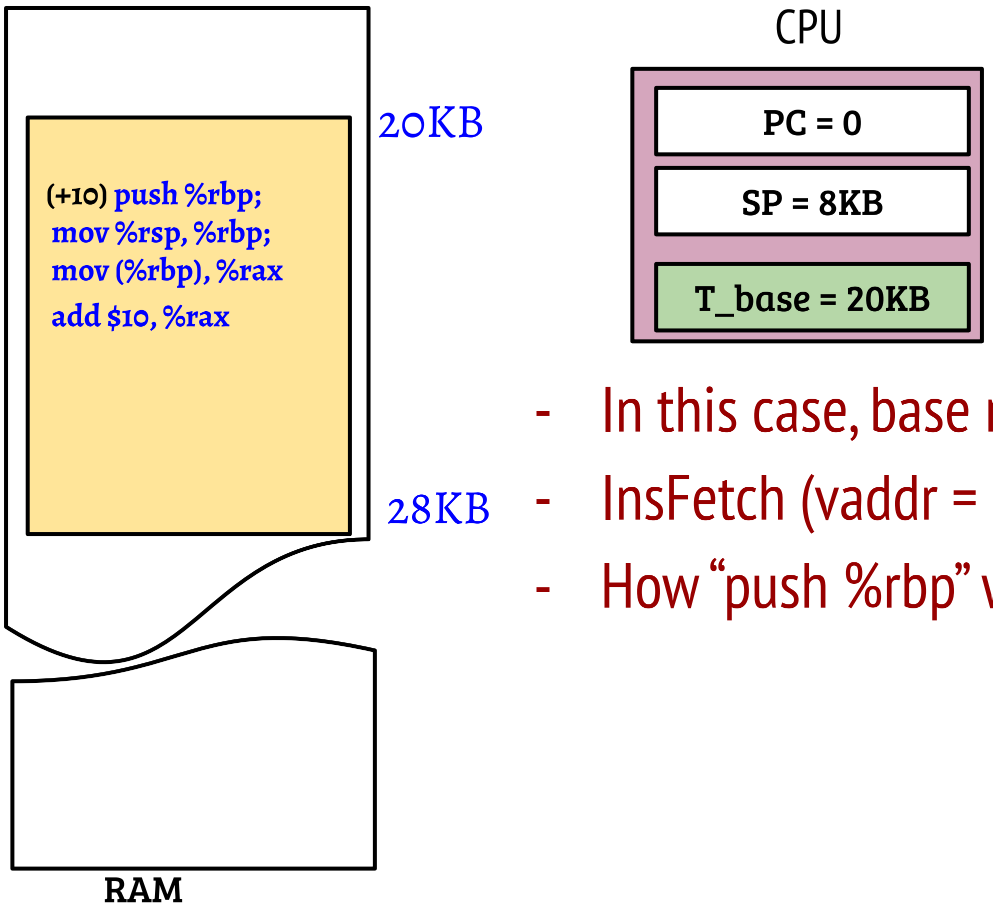


```
(+10) push %rbp;
 mov %rsp, %rbp;
 mov (%rbp), %rax
 add $10, %rax
```

Here is a base register. I will add the value of base register with the virtual address generated by the program to get the physical address. All yours buddy!

Hurray! I will configure the value of base register as per my need. I see some light atlast!

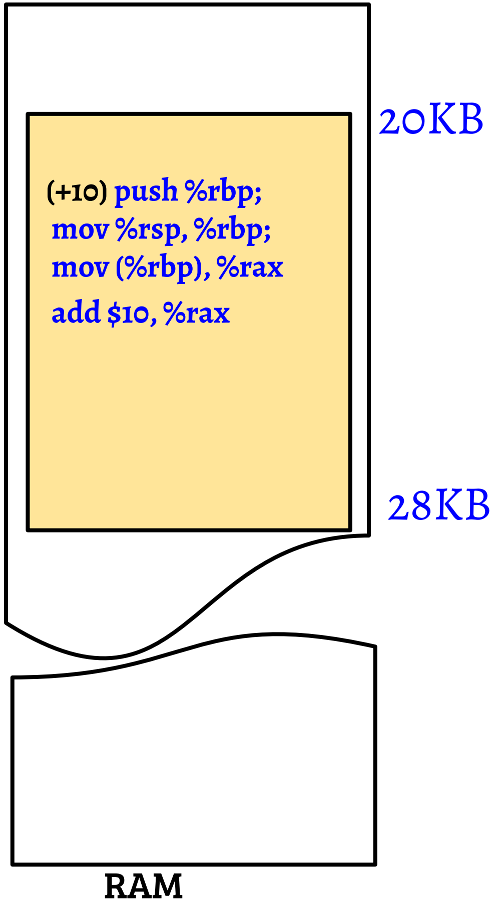
Translation

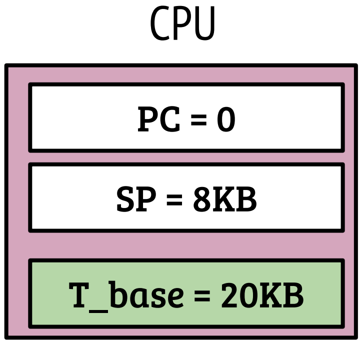


```
(+10) push %rbp;
 mov %rsp, %rbp;
 mov (%rbp), %rax
 add $10, %rax
```

- In this case, base register value should be 20KB
- InsFetch (vaddr = 10) ⇒ InsFetch (paddr = 20KB +10)
- How “push %rbp” works?

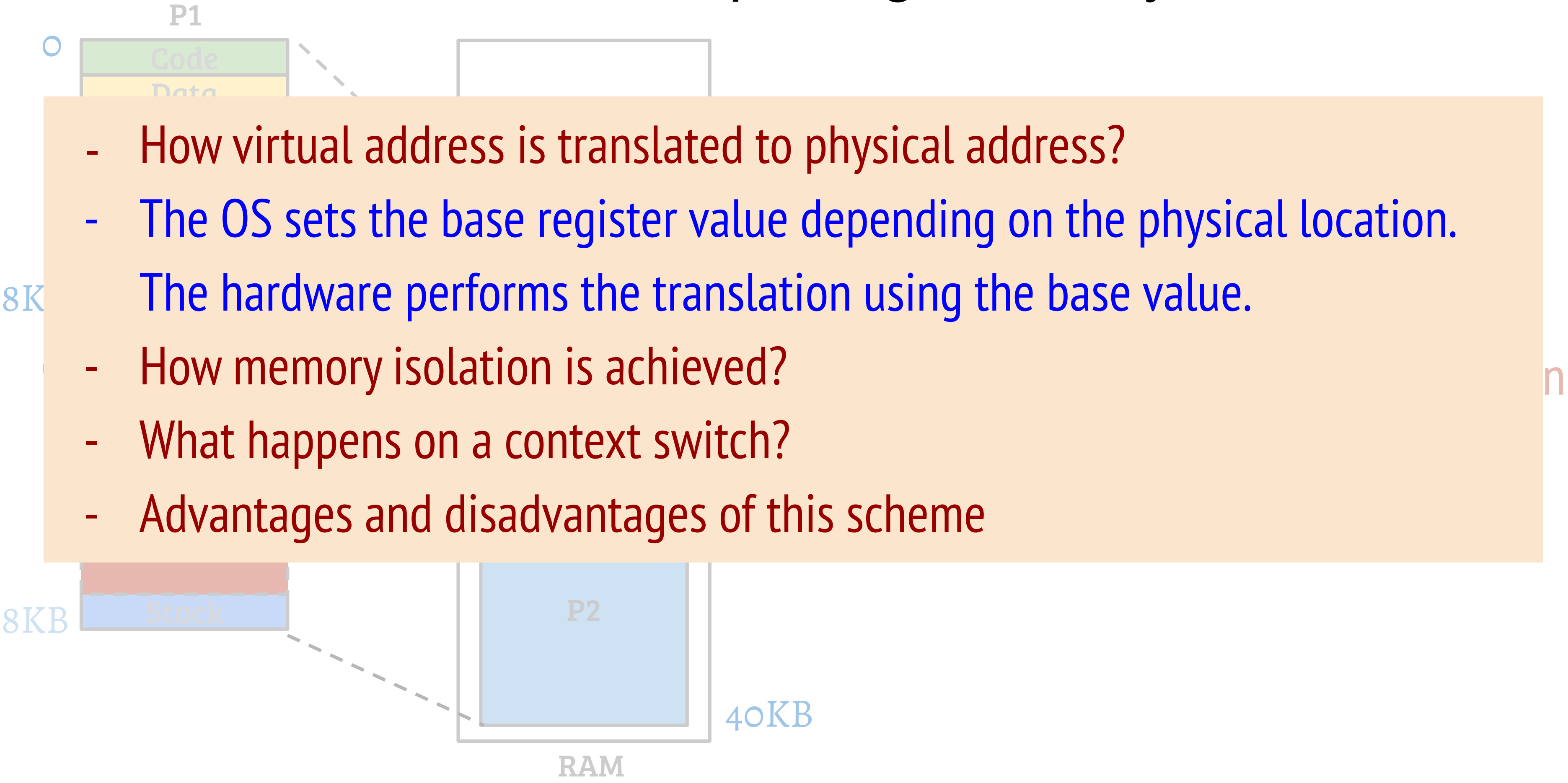
Translation





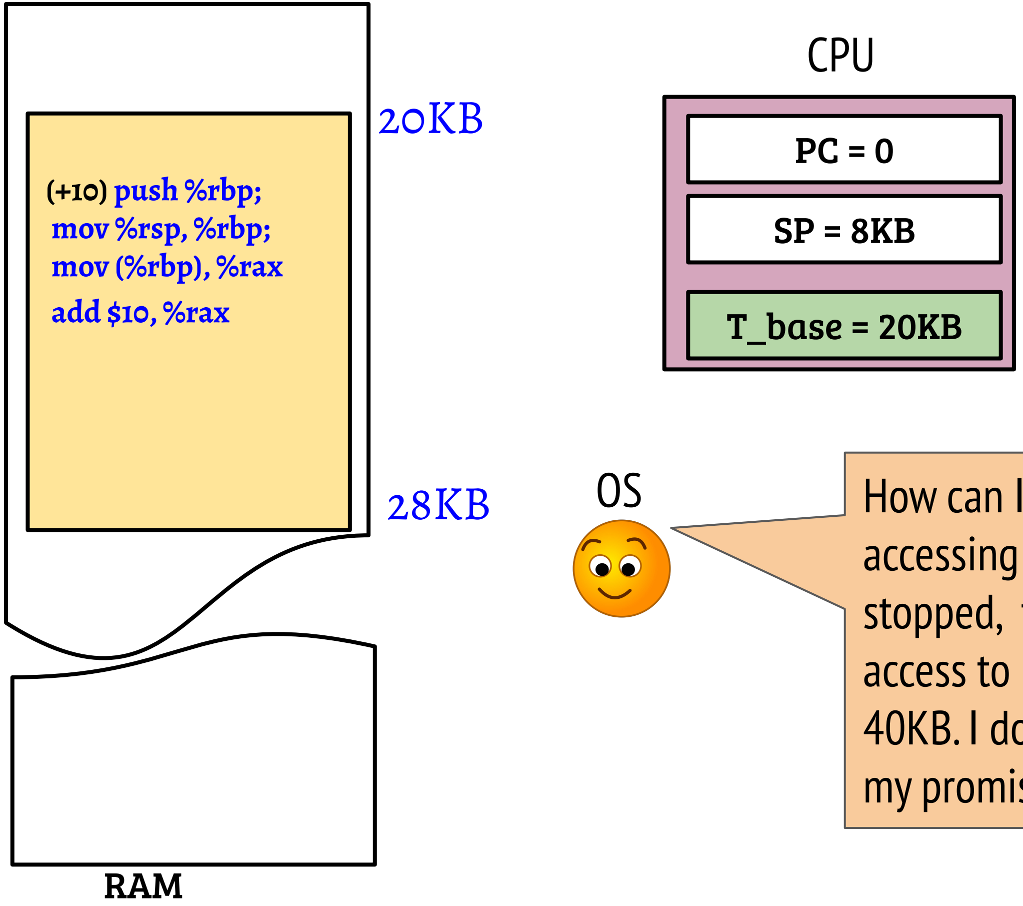
- In this case, base register value should be 20KB
- InsFetch (vaddr = 10) ⇒ InsFetch (paddr = 20KB +10)
- How “push %rbp” works?
- Assuming RSP = 8KB, “push %rbp” results in a memory store at address (8KB - 8)
    - CPU translates the address to (28KB - 8)

Translation at address space granularity



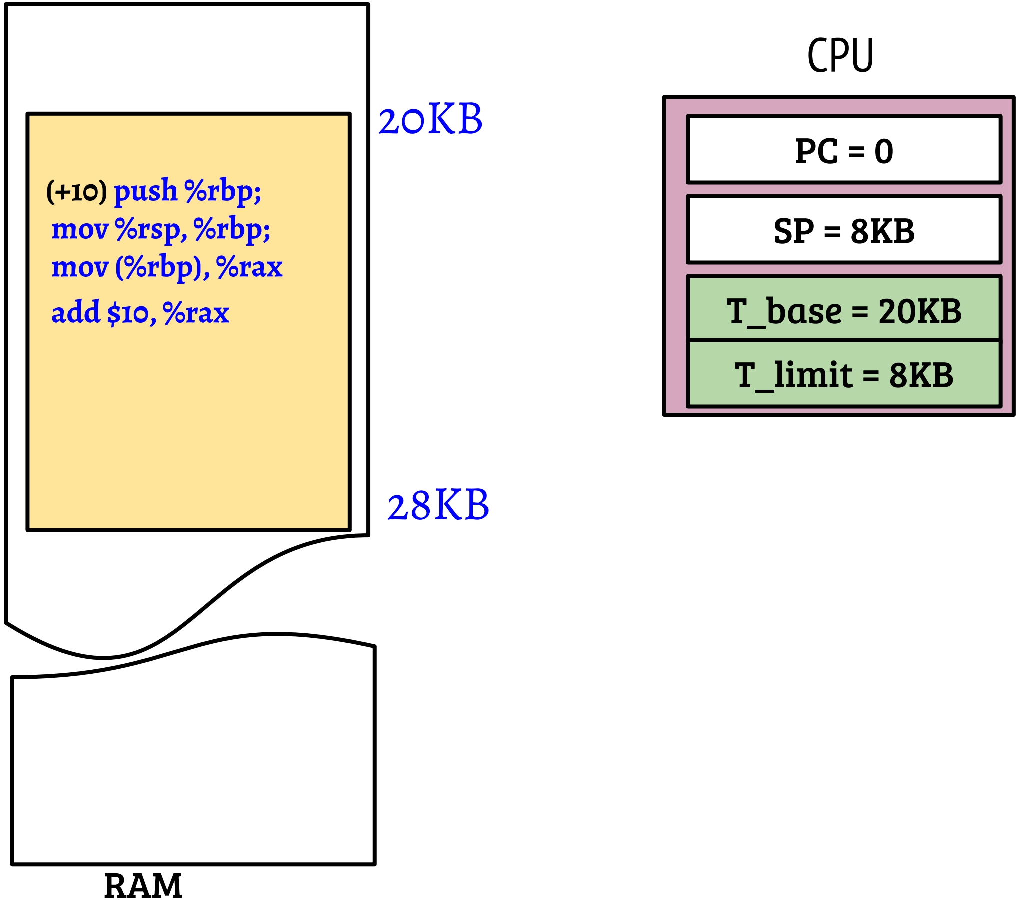
- How virtual address is translated to physical address?
- The OS sets the base register value depending on the physical location. The hardware performs the translation using the base value.
- How memory isolation is achieved?
- What happens on a context switch?
- Advantages and disadvantages of this scheme

Isolation: How to stop illegal access?



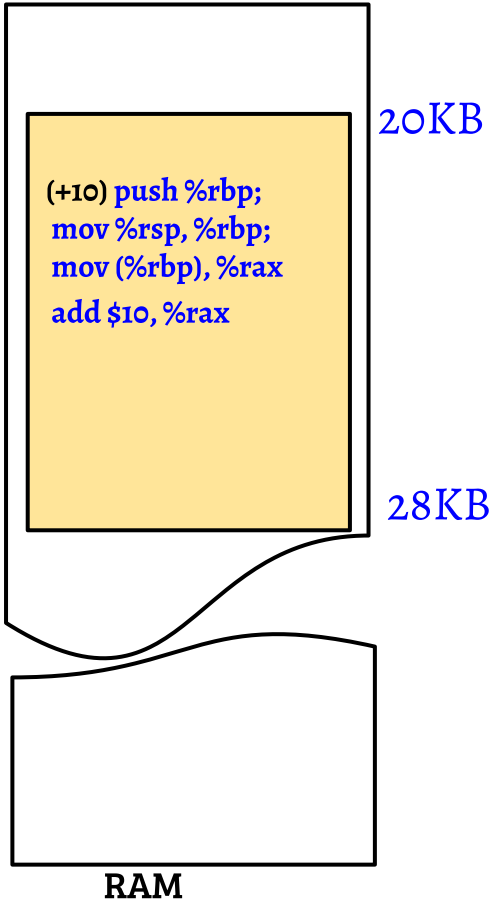
How can I stop a program from accessing VA = 20KB? If not stopped, the program gets access to physical address 40KB. I do not want to break my promise of isolation.

Isolation: How to stop illegal accesses?

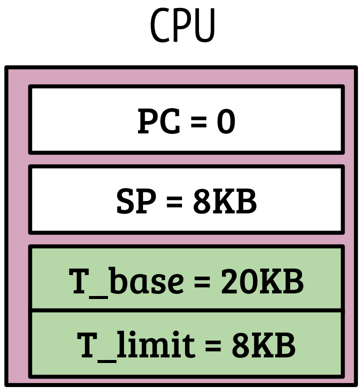


Once a cry baby always a cry baby! I also provide a limit register to enforce the limit during translation. Before you ask, these registers can only be changed from privileged mode

Isolation: How to stop illegal accesses?



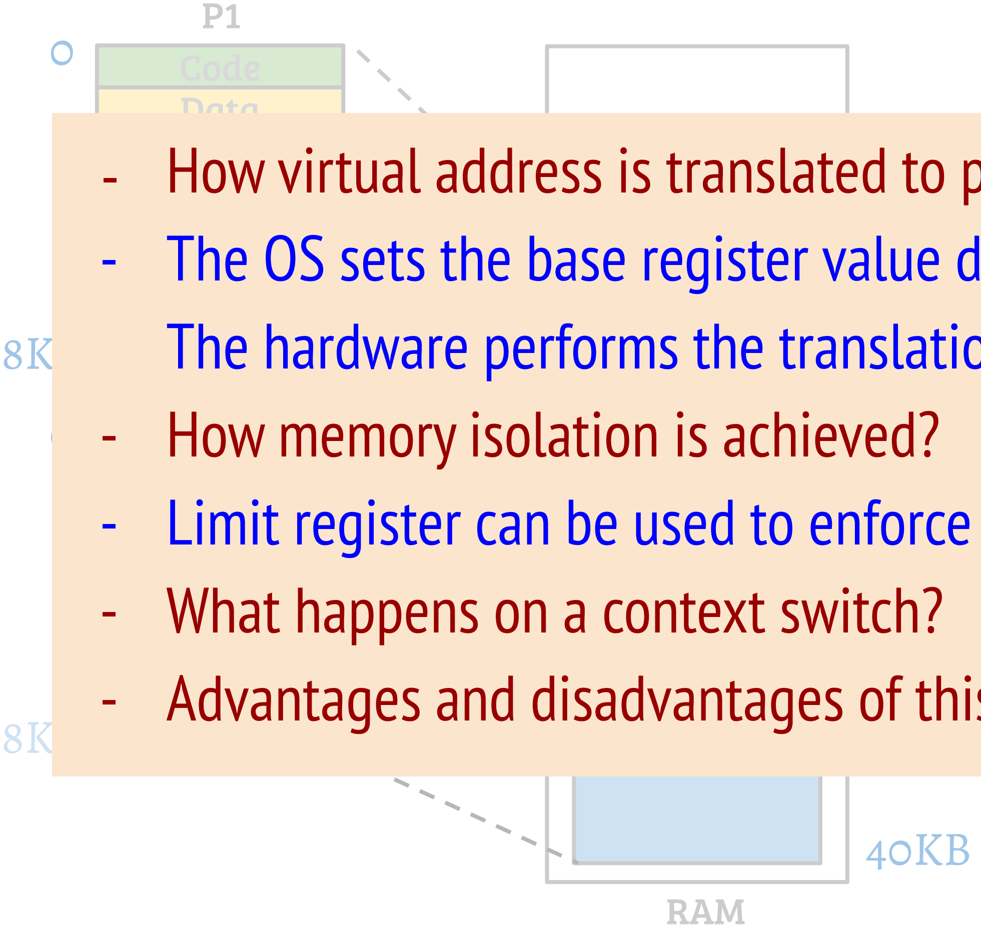
```
(+10) push %rbp;
 mov %rsp, %rbp;
 mov (%rbp), %rax
 add $10, %rax
```



Once a cry baby always a cry baby! I also provide a limit register to enforce the limit during translation. Before you ask, these registers can only be changed from privileged mode

- The hardware raises a fault if some program violates the limit.
- The OS fault handler may kill the process

Translation at address space granularity



- How virtual address is translated to physical address?
- The OS sets the base register value depending on the physical location.
  The hardware performs the translation using the base value.
- How memory isolation is achieved?
- Limit register can be used to enforce memory isolation
- What happens on a context switch?
- Advantages and disadvantages of this scheme

Context switch and translation information

- The base and limit register values can be saved in the outgoing process PCB during context switch
- Loaded from PCB to the CPU when a process is scheduled

Translation at address space granularity

- How virtual address is translated to physical address?
- The OS sets the base register value depending on the physical location.
  The hardware performs the translation using the base value.
- How memory isolation is achieved?
- Limit register can be used to enforce memory isolation
- What happens on a context switch?
- Save and restore limit and base registers
- Advantages and disadvantages of this scheme

Translation at address space granularity: Issues

- Physical memory must be greater than address space size
  - Unrealistic, against the philosophy of address space abstraction
  - Small address space size ⇒ Unhappy user

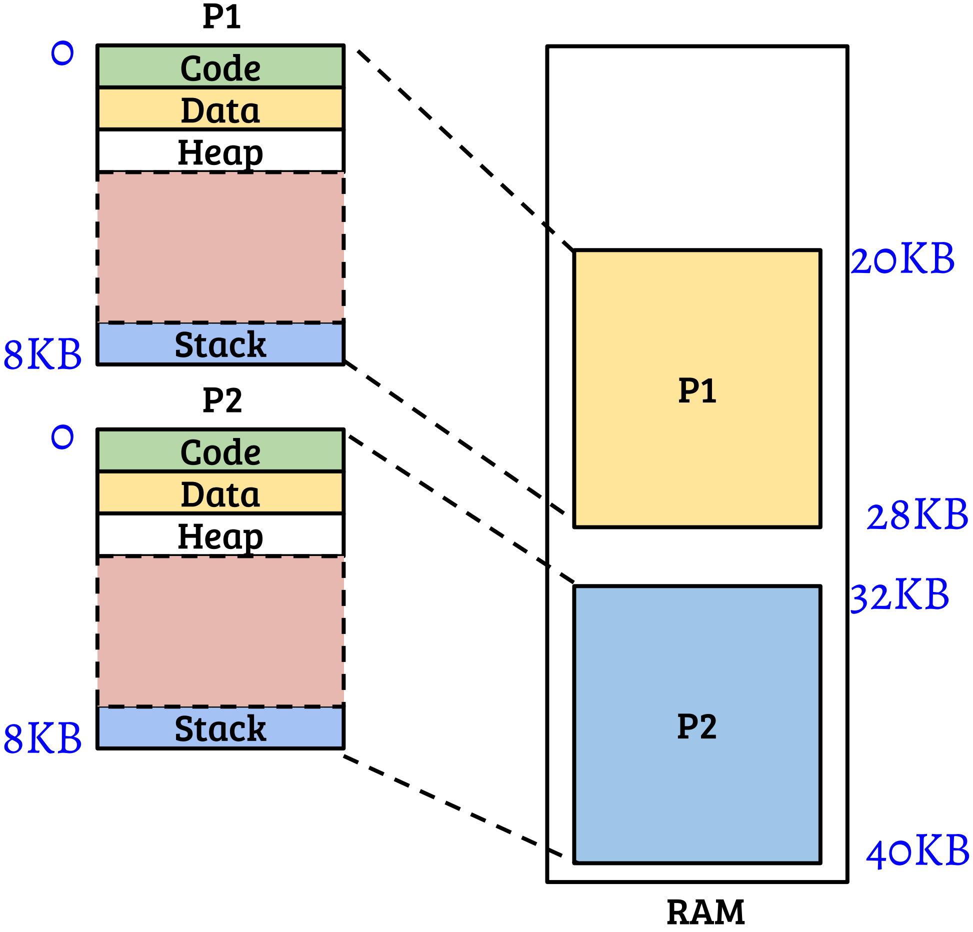
Translation at address space granularity: Issues

- Physical memory must be greater than address space size
    - Unrealistic, against the philosophy of address space abstraction
    - Small address space size ⇒ Unhappy user
- Memory inefficient
    - Physical memory size is same as address space size irrespective of actual usage ⇒ Memory wastage
    - Degree of multiprogramming is very less

CS330: Operating Systems

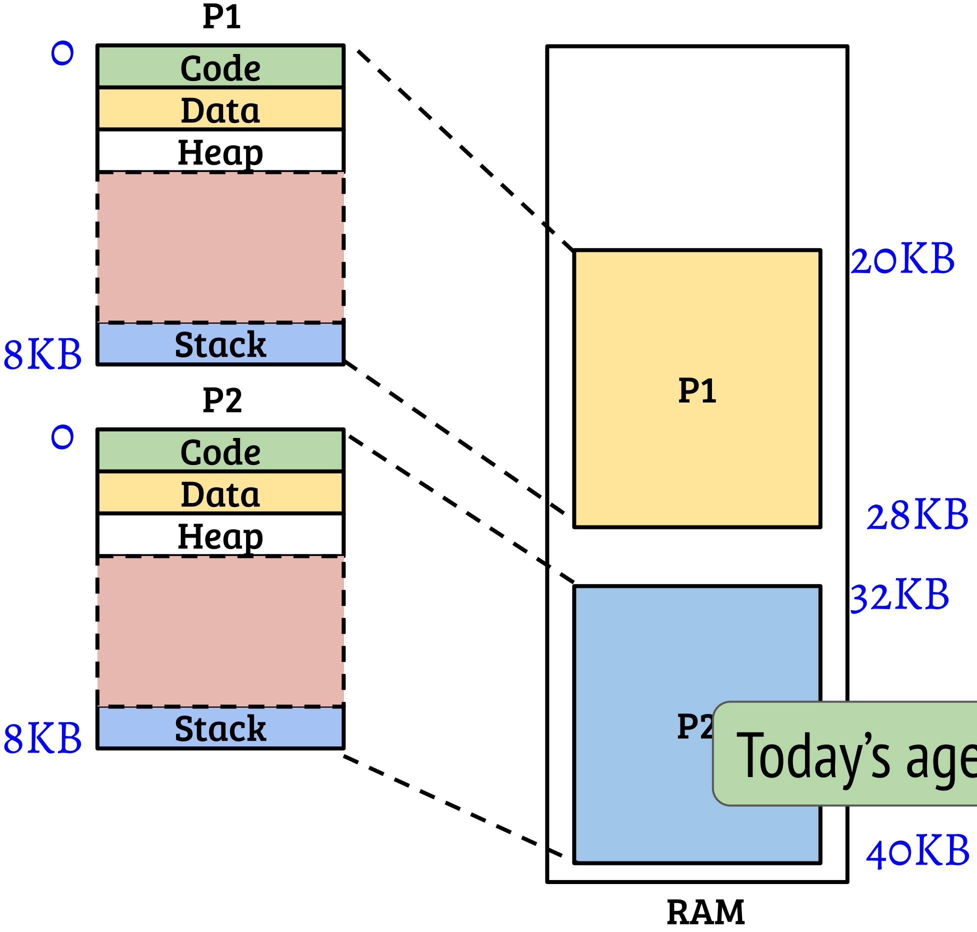
Virtual memory: Segmentation

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

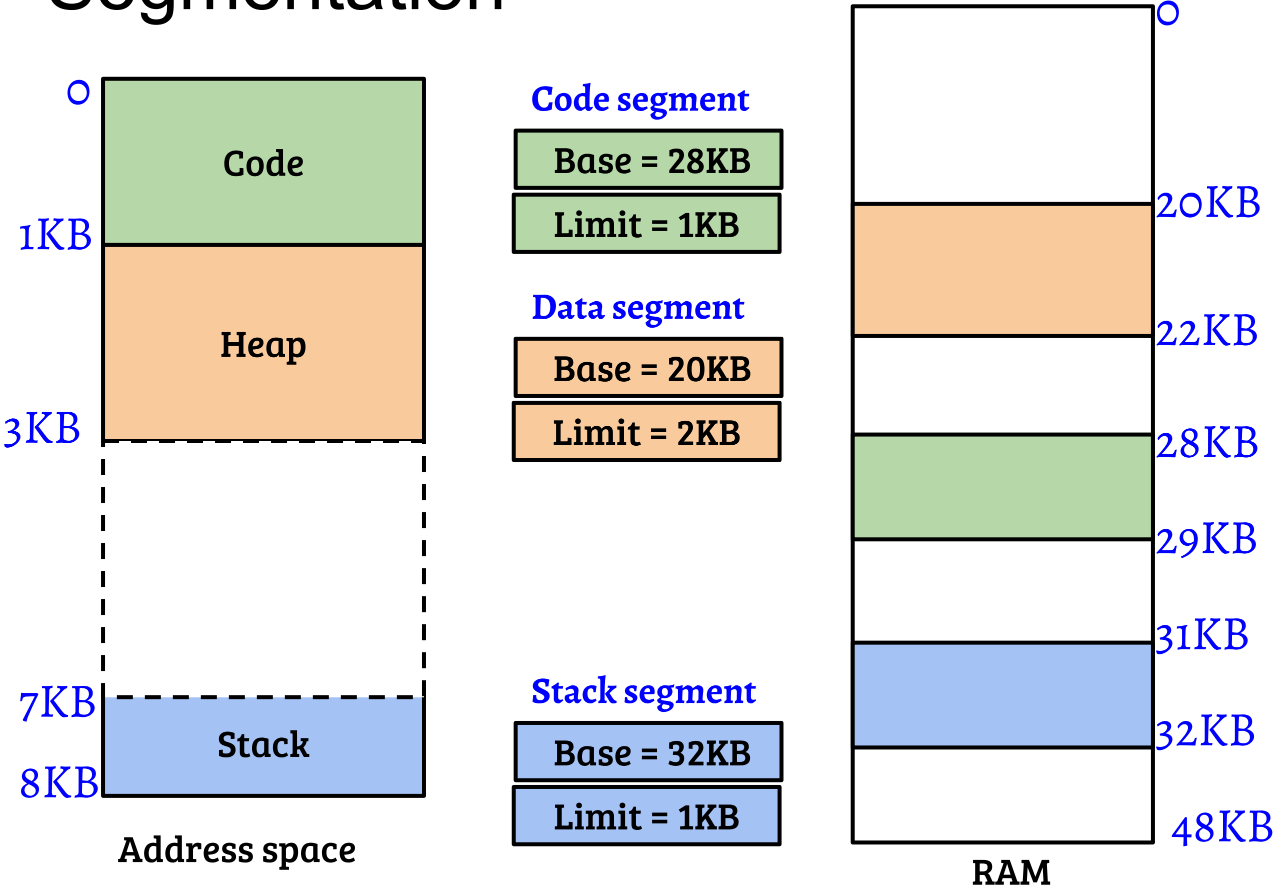
Recap: Translation at address space granularity



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

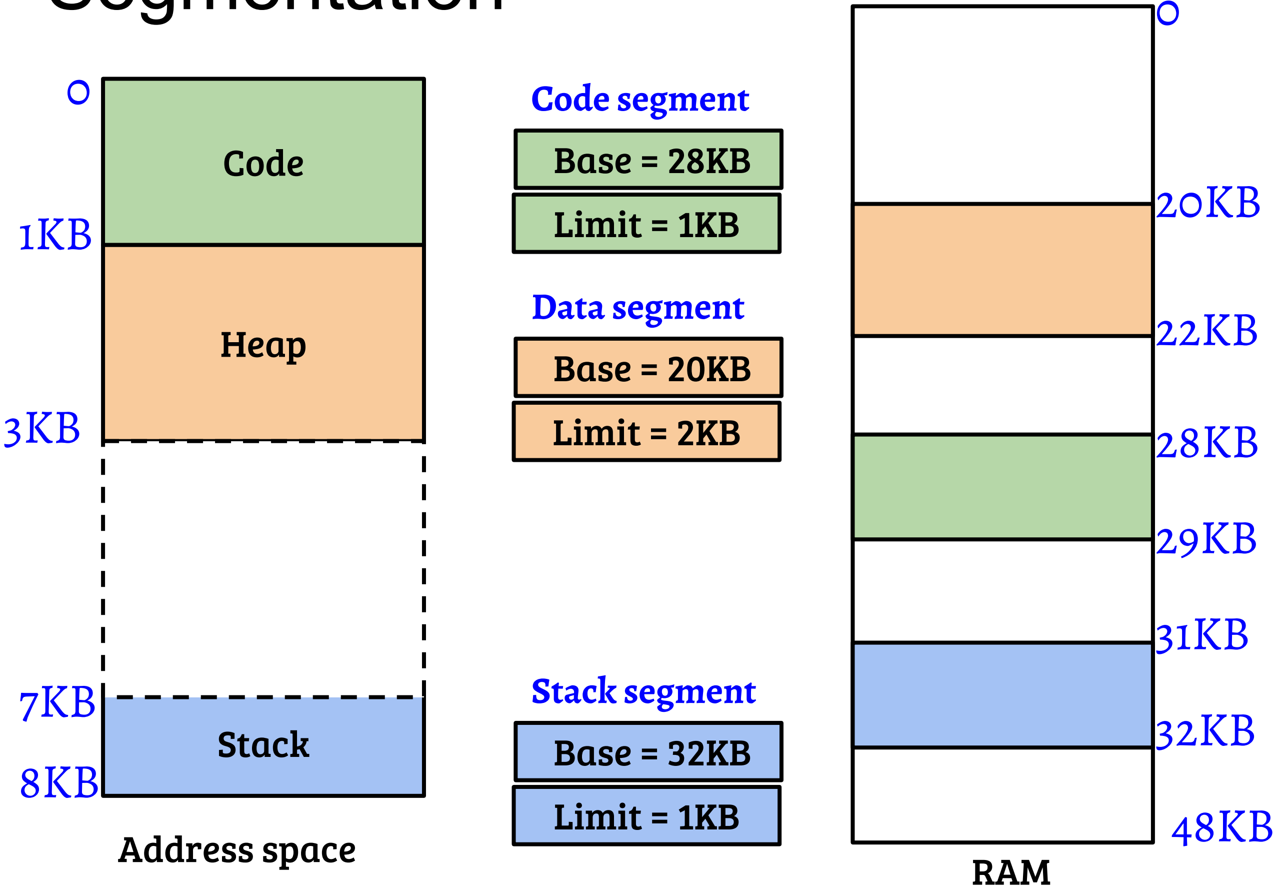
Today's agenda: Translation at segment granularity

Segmentation



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

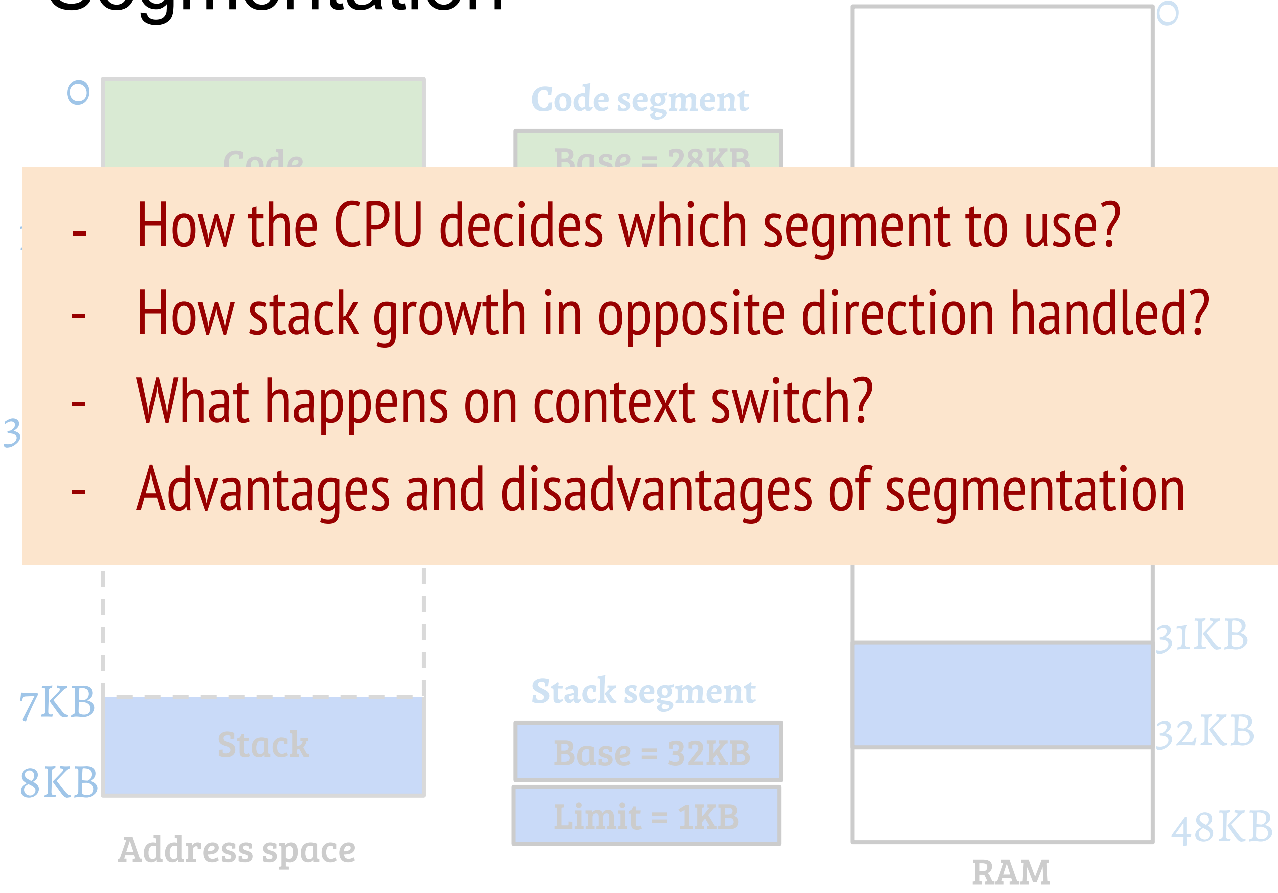
Segmentation



- 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

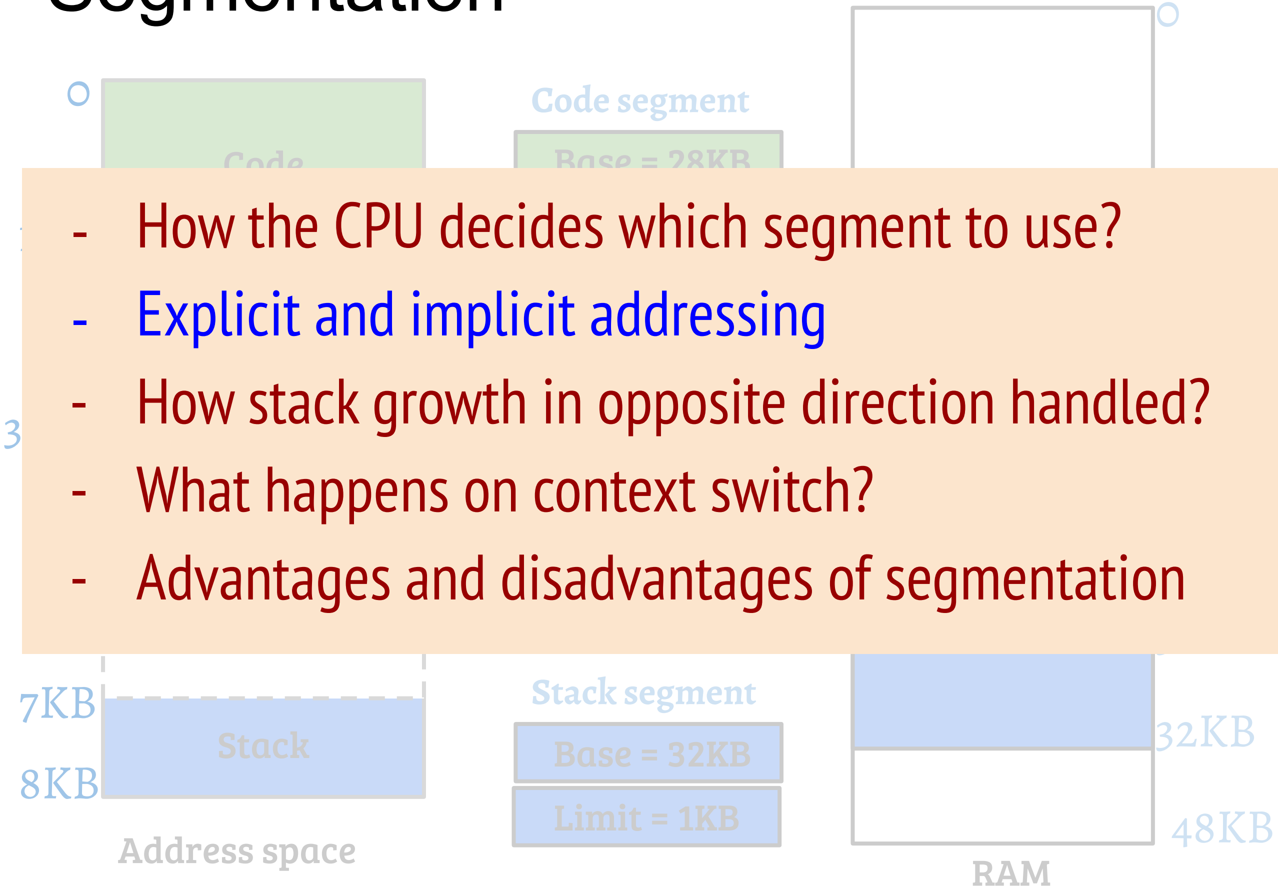
Segmentation: Implicit addressing

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

Segmentation: Implicit addressing

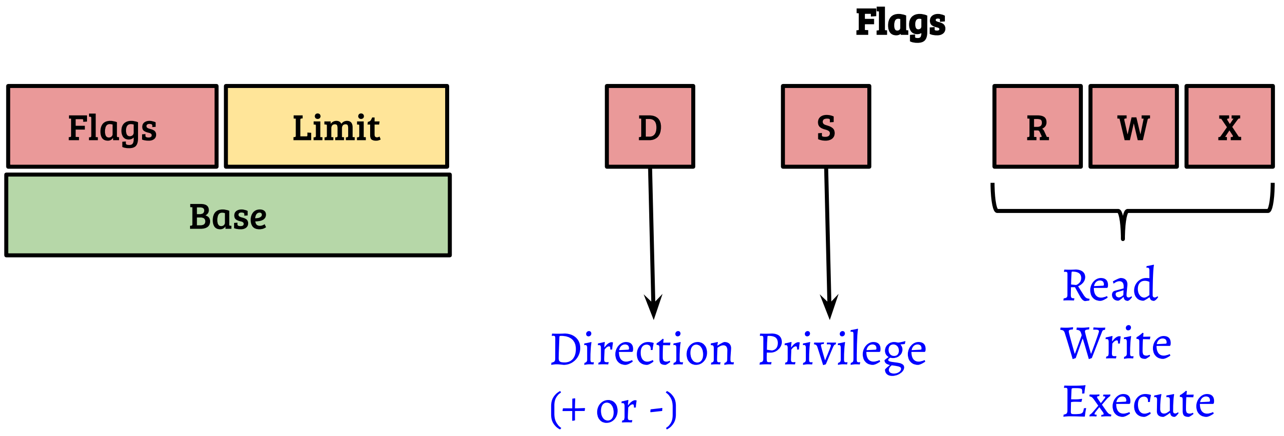
- The hardware selects the segment register based on the operation
- Code segment for instruction access
  - Fetch address, jump target, call address
- Stack segment for stack operations
  - Arguments for push and pop, indirect addressing with SP, BP
- Data segment for other addresses

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

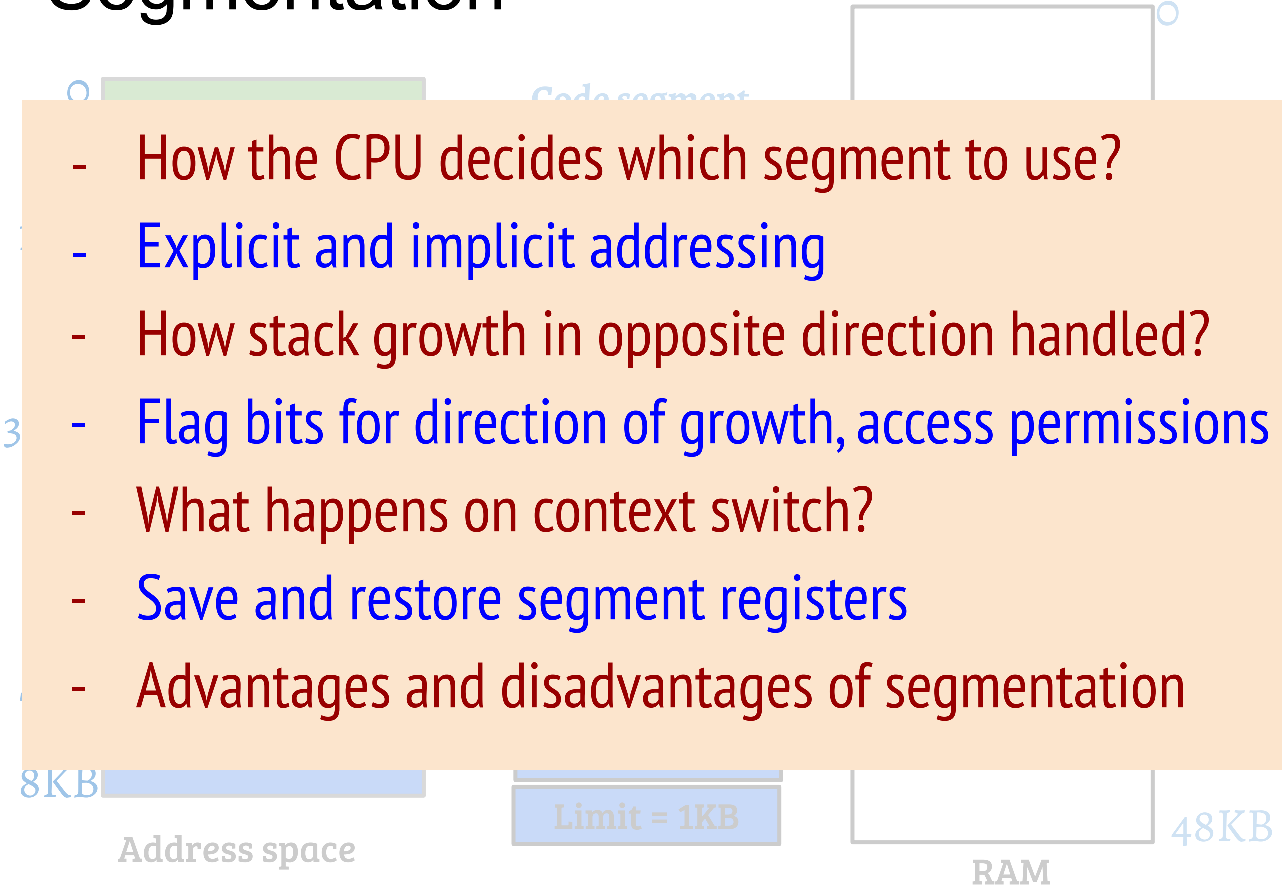
Segmentation (protection and direction)



- 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

- 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



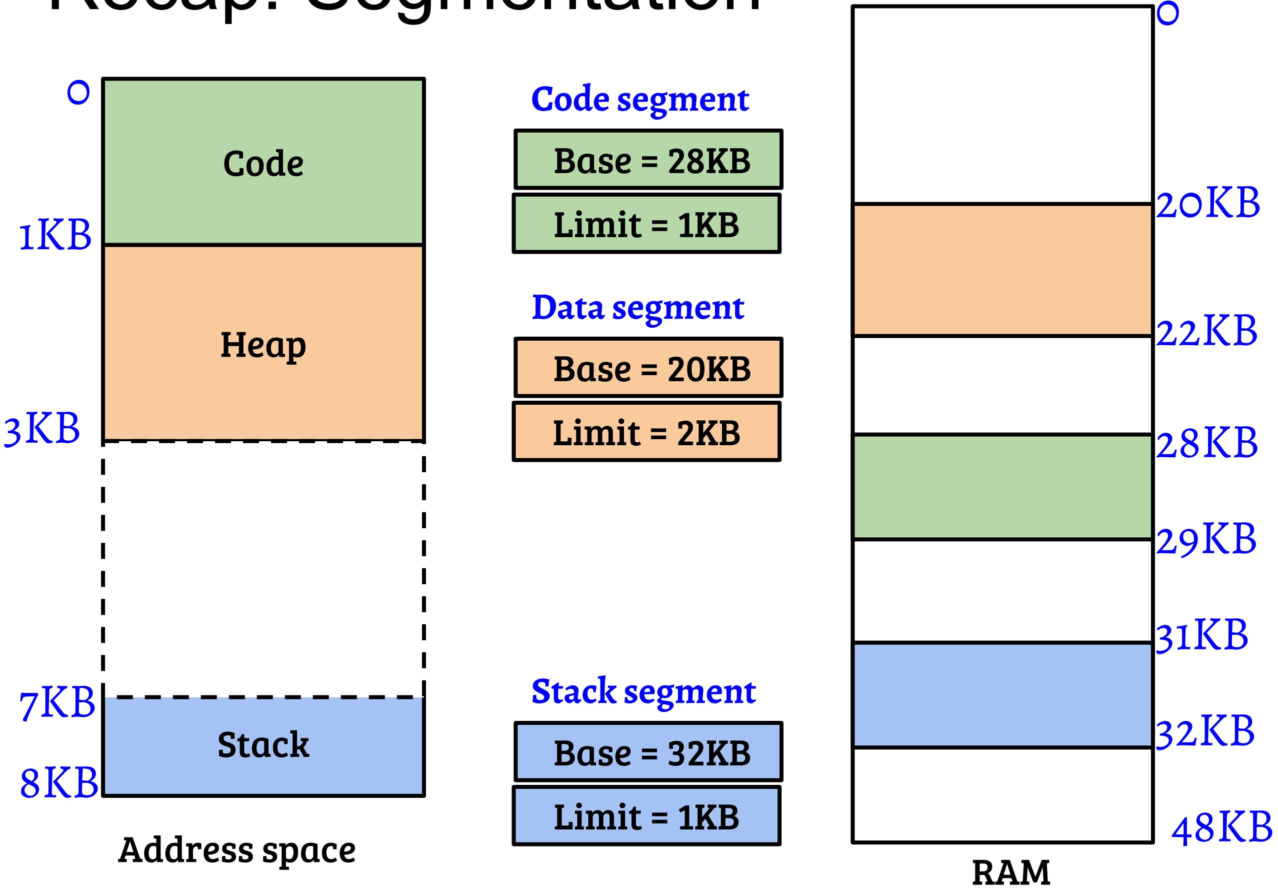
Advantages and disadvantages of segmentation

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

CS330: Operating Systems

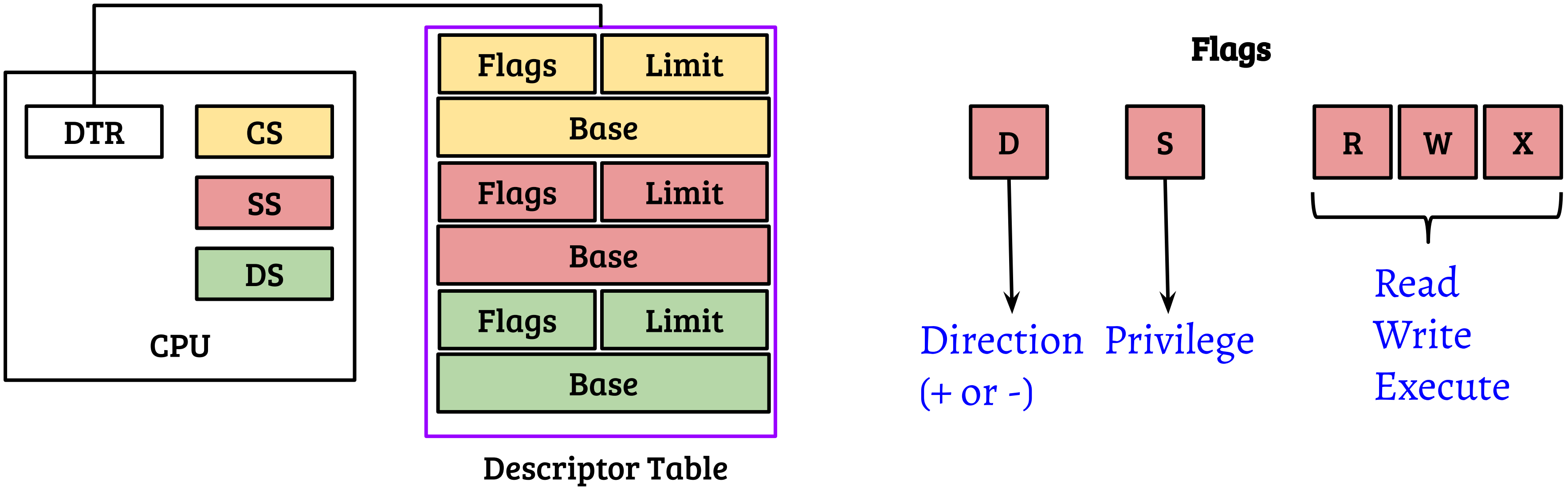
Virtual memory: Paging

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

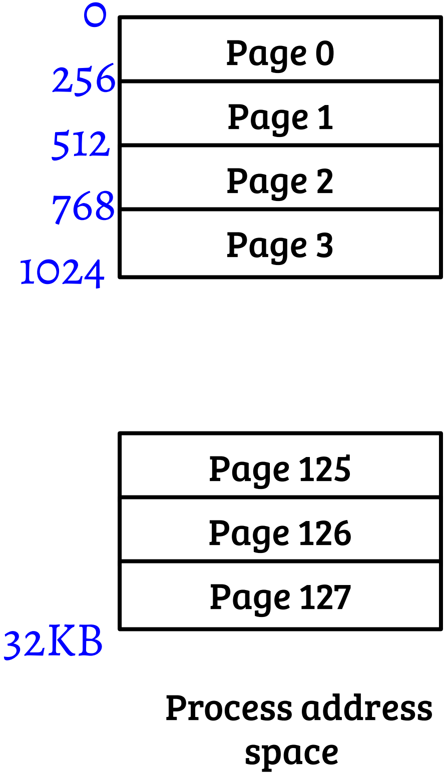
Paging

- 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

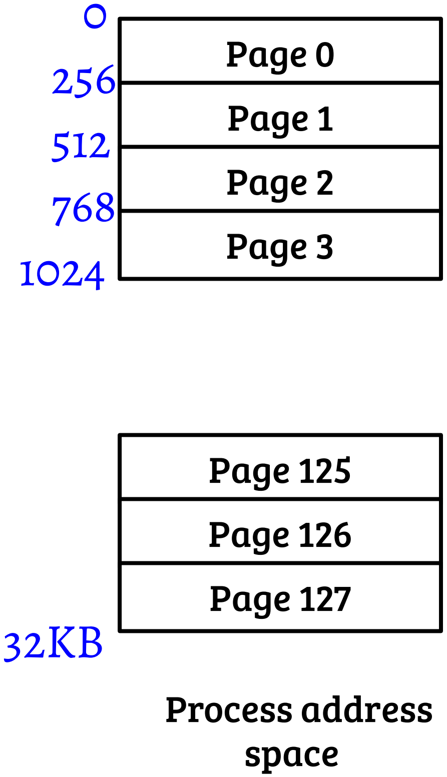
- Paging addresses the following issues with segmentation
  - External fragmentation caused due to variable sized segments
  - No support for discontinuous/sparse mapping
- The idea of paging
  - Partition the address space into fixed sized blocks (call it pages)
  - Physical memory partitioned in a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

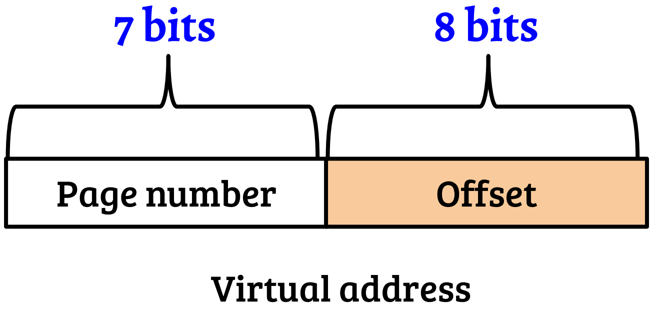


- Virtual address size = 32KB, Page size = 256 bytes
- Address length = 15 bits {0x0 - 0x7FFF}
- # of pages = 128

Paging example (pages)



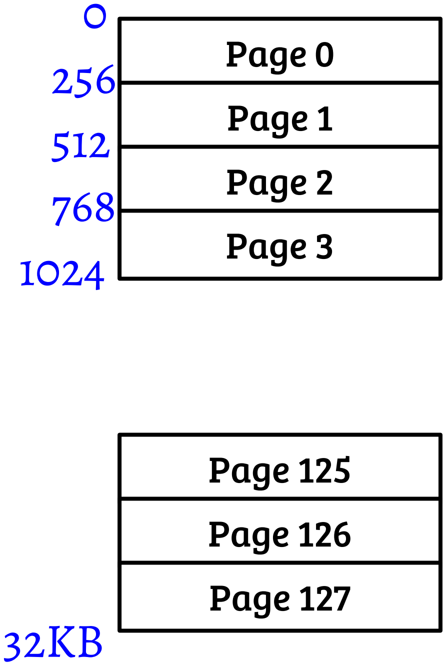
- Virtual address size = 32KB, Page size = 256 bytes
- Address length = 15 bits {0x0 - 0x7FFF}
- # of pages = 128



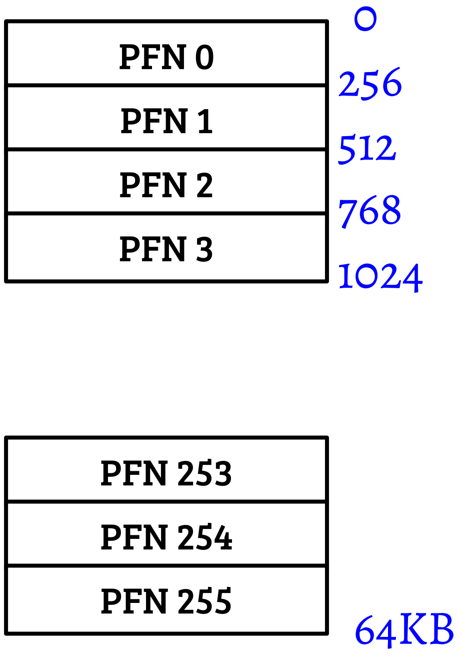
- Example: For Virtual address *0x0510*, Page number = 5, offset = 16

Paging example (page frames)

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



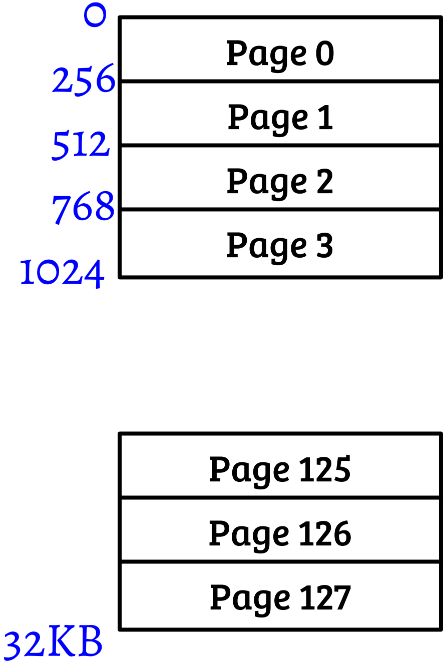
Process address space



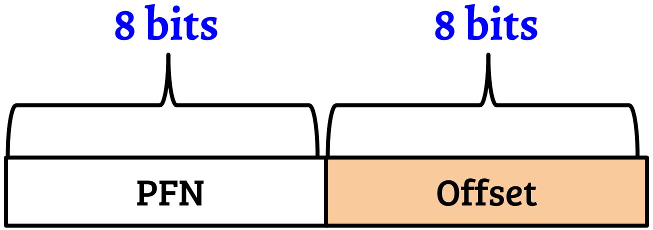
DRAM

Paging example (page frames)

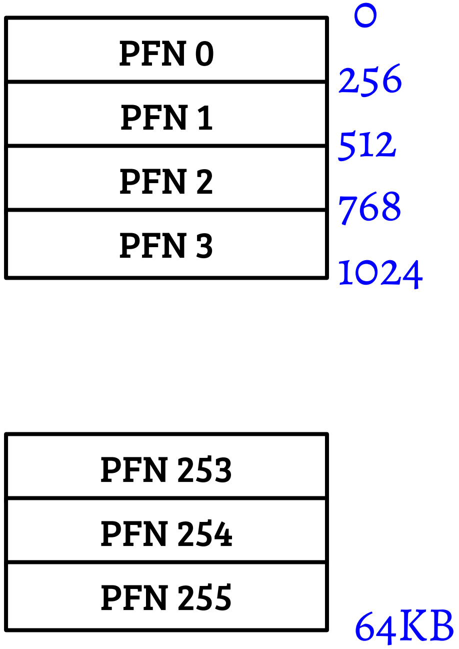
- Physical address size = 64KB
- Address length = 16 bits {0x0 - 0xFFFF}
- # of page frames = 256



Process address space



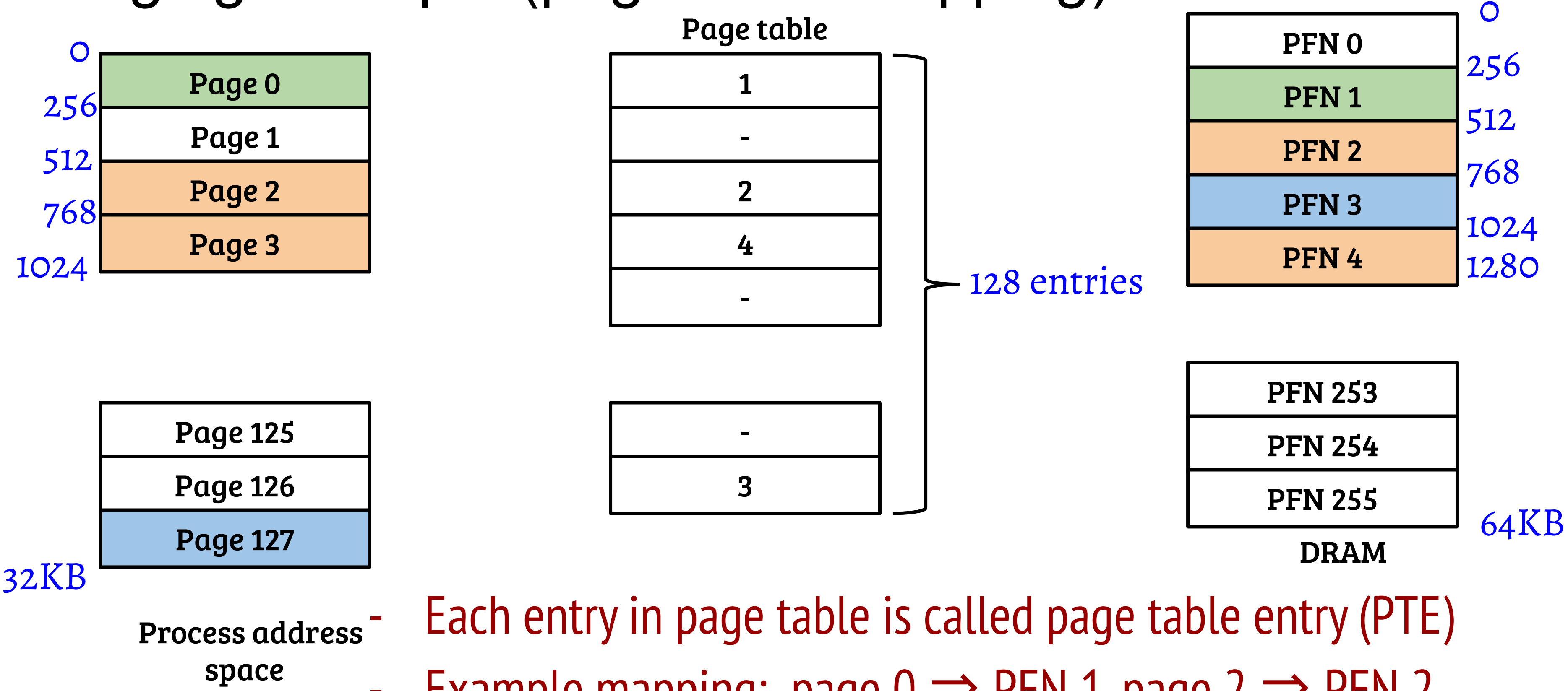
Physical address



DRAM

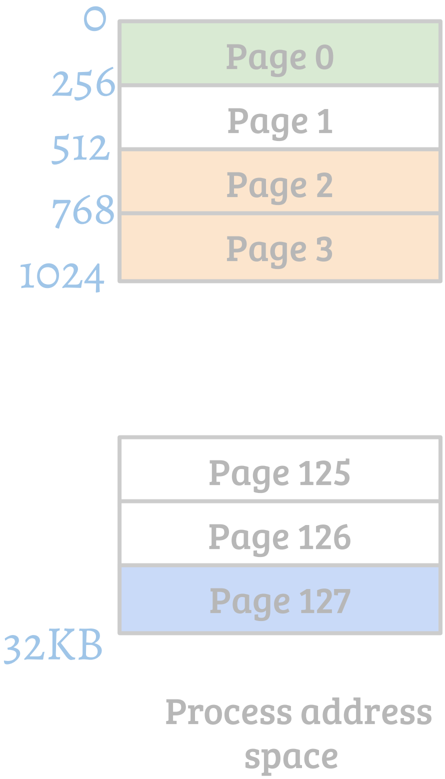
- Example: For physical address *0x1F51*, PFN = 31, offset = 81

Paging example (page table mapping)

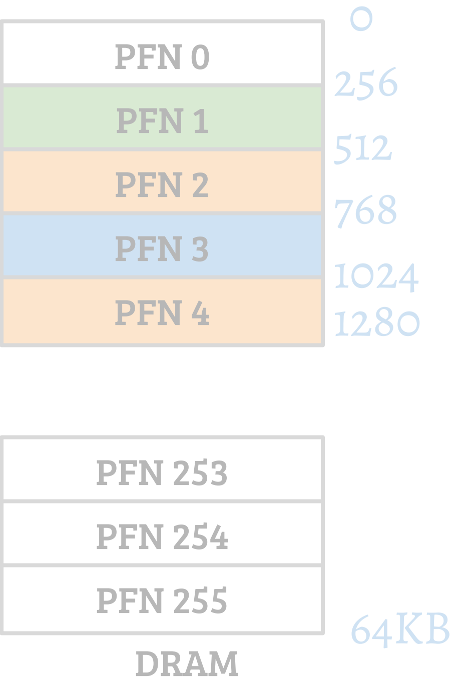


- Each entry in page table is called page table entry (PTE)
- Example mapping: page 0 ⇒ PFN 1, page 2 ⇒ PFN 2 and so on

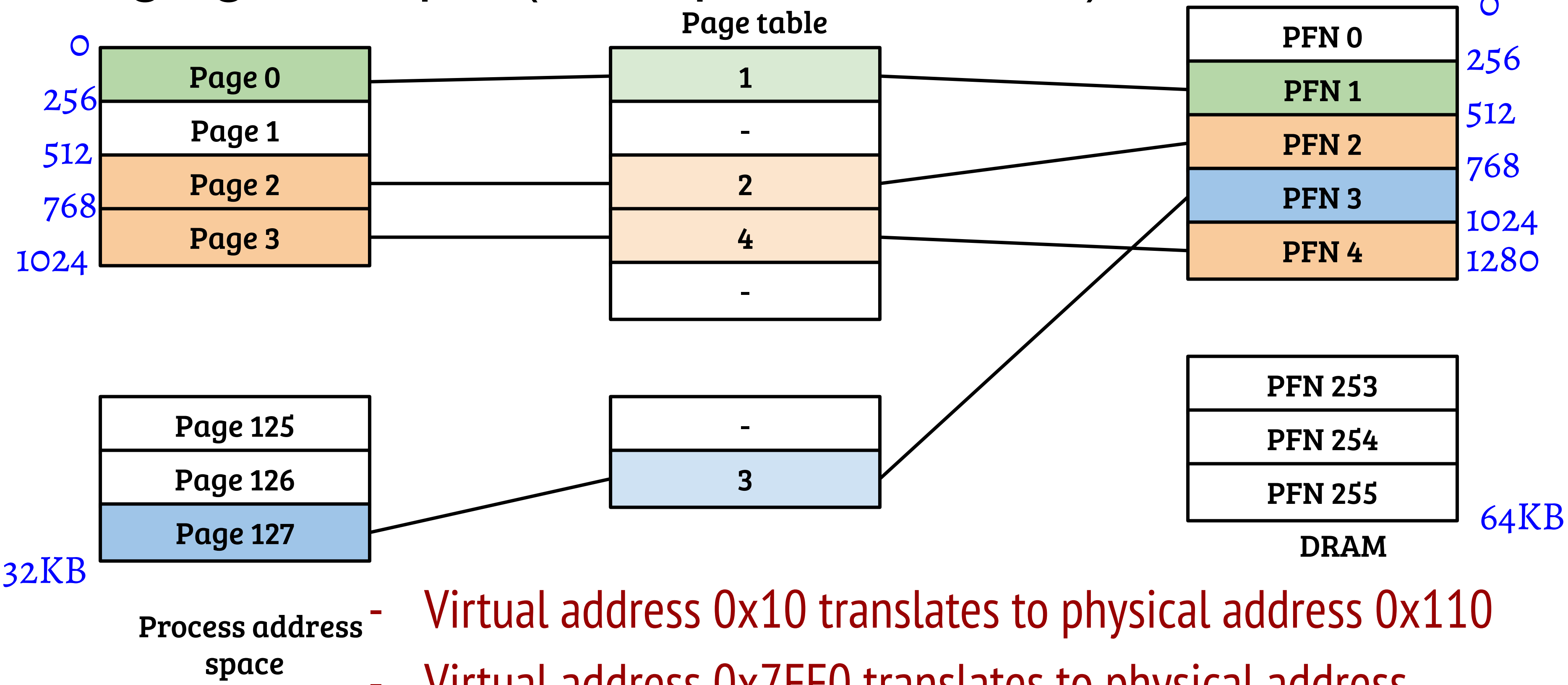
Paging example (page table walk)



```
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;
}
```

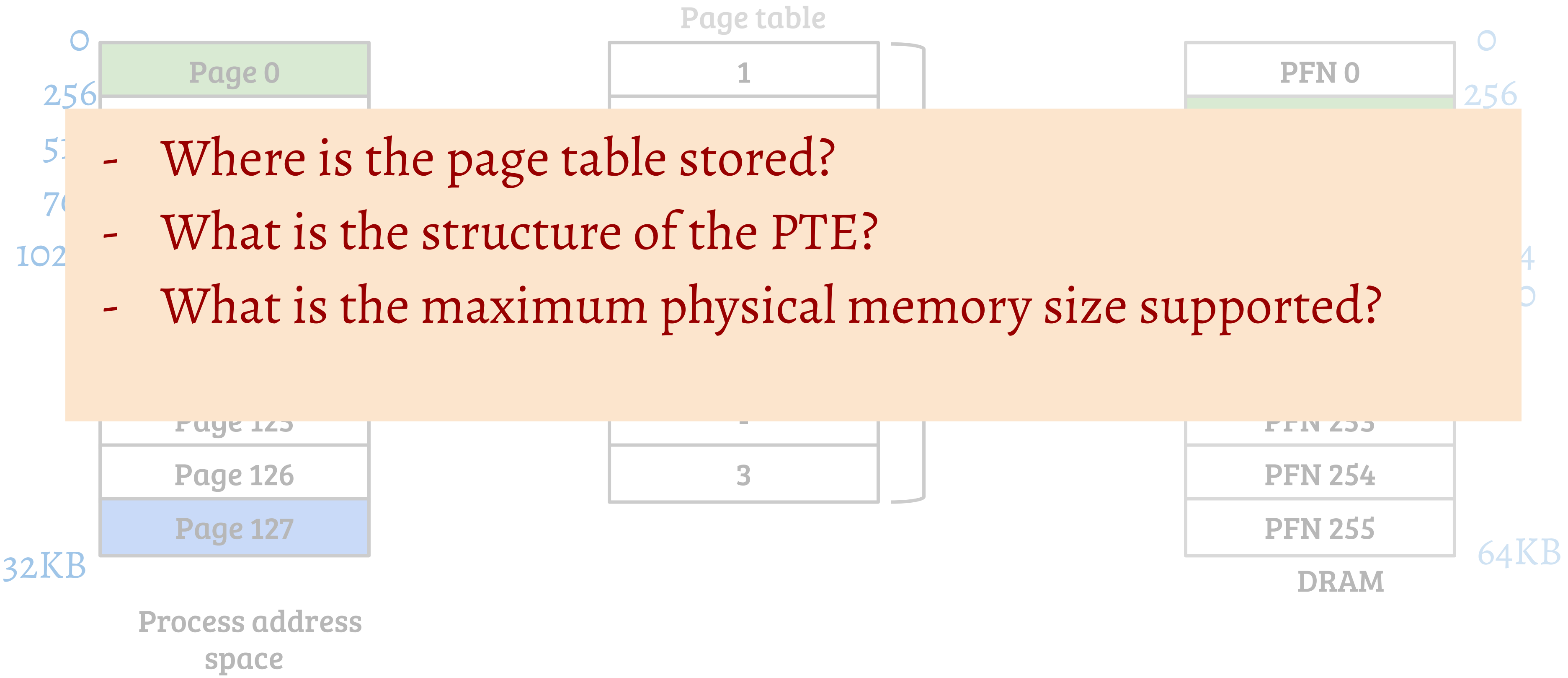


Paging example (example translation)



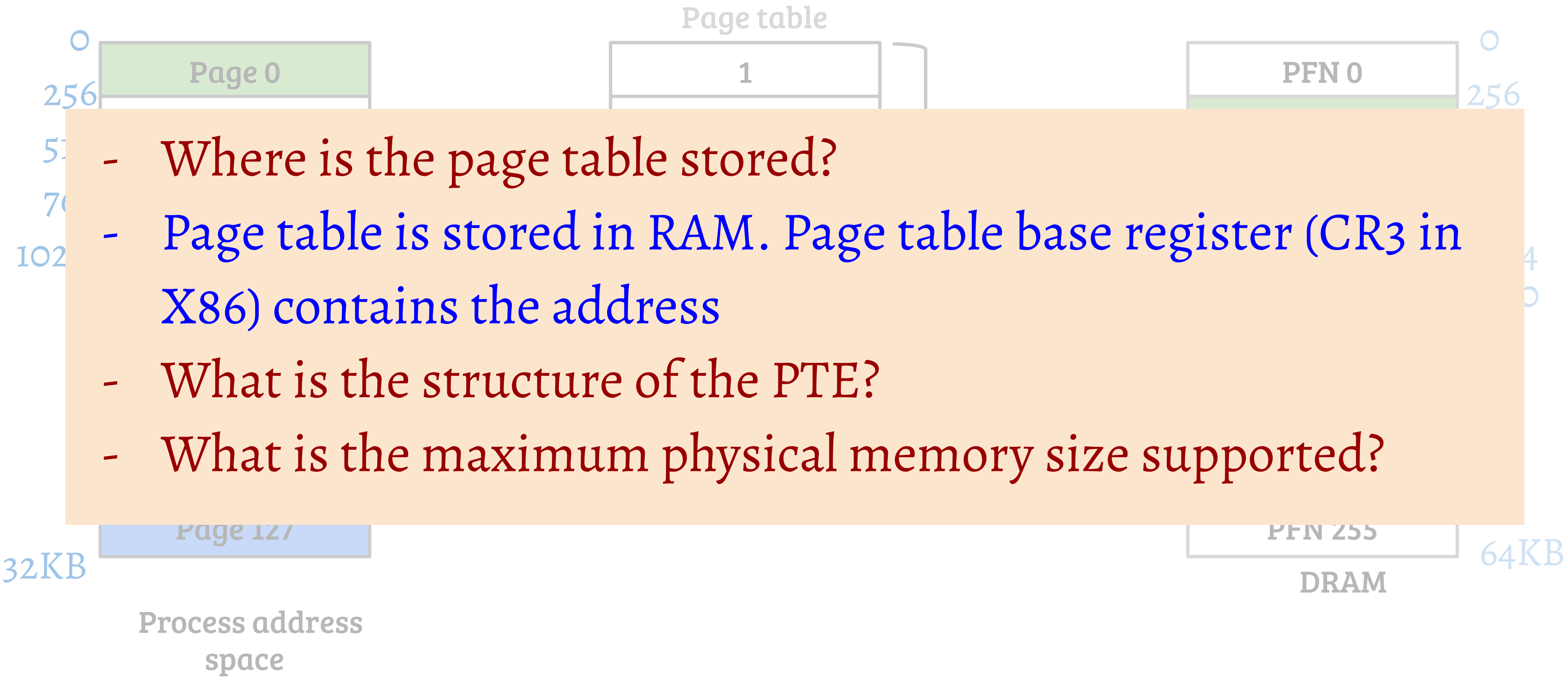
- Virtual address 0x10 translates to physical address 0x110
- Virtual address 0x7FF0 translates to physical address 0x3F0

Paging example (page table walk)



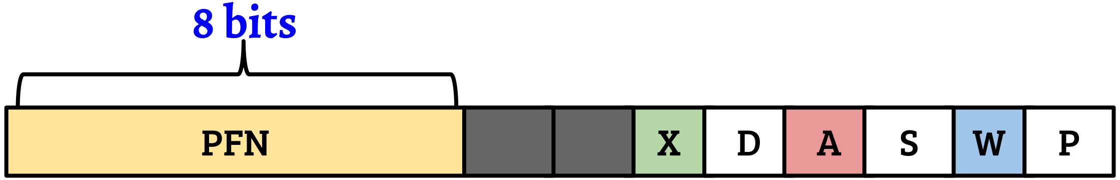
- Where is the page table stored?
- What is the structure of the PTE?
- What is the maximum physical memory size supported?

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?

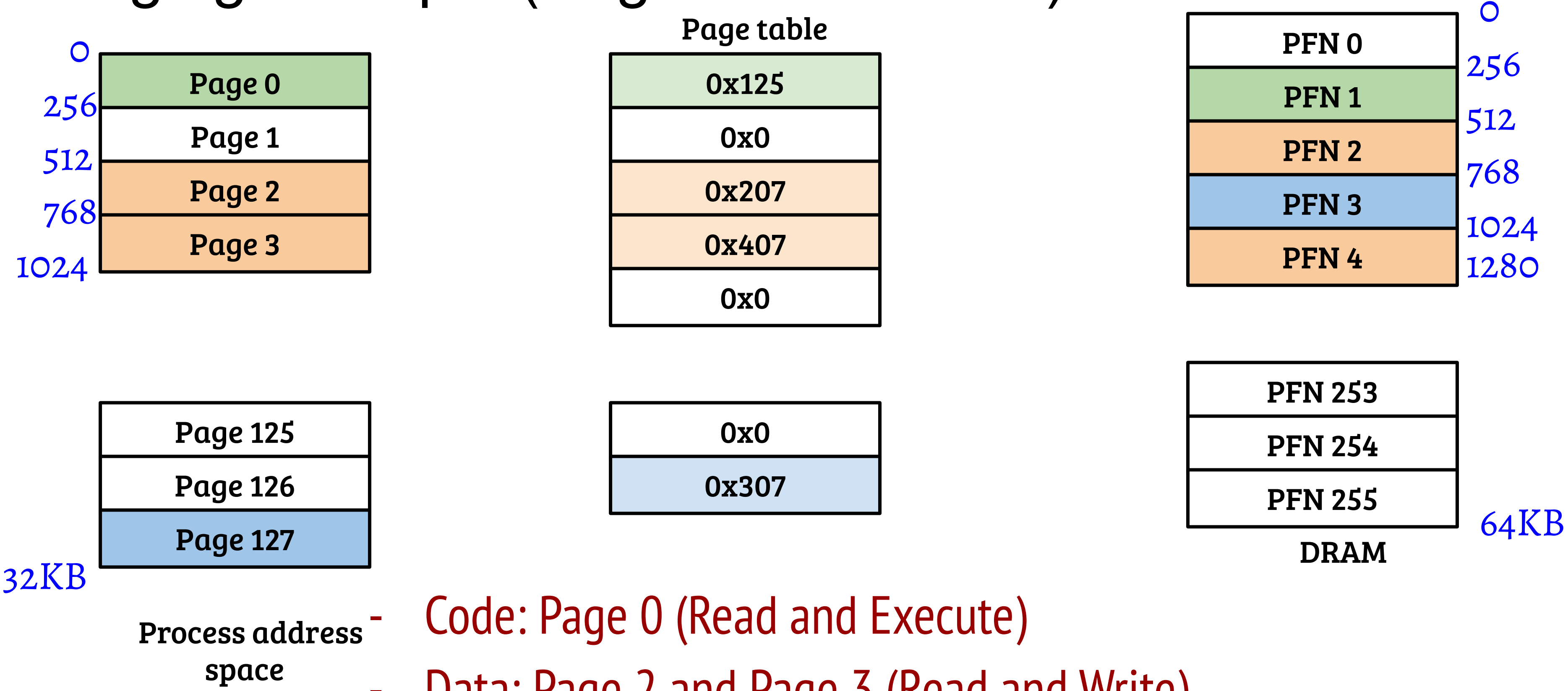
Paging example (structure of an example PTE)



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

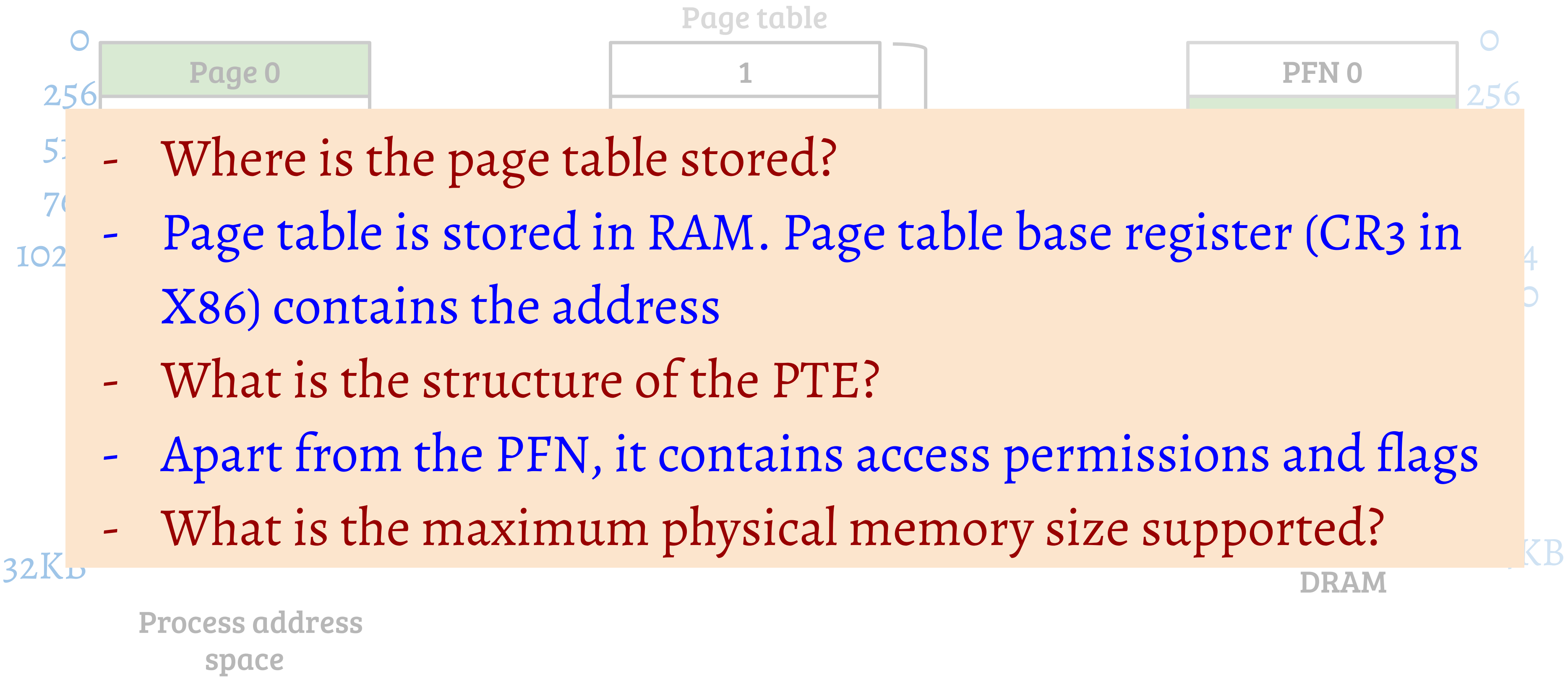
P	Present bit, 1 ⇒ entry is valid
W	Write bit, 1 ⇒ Write allowed
S	Privilege bit, 0 ⇒ only kernel mode access is allowed
A	Accessed bit, 1 ⇒ Address accessed (set by H/W during walk)
D	Dirty bit, 1 ⇒ Address written (set by H/W during walk)
X	Execute bit, 1 ⇒ Instruction fetch allowed for this page
■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

Paging example (Page table entries)



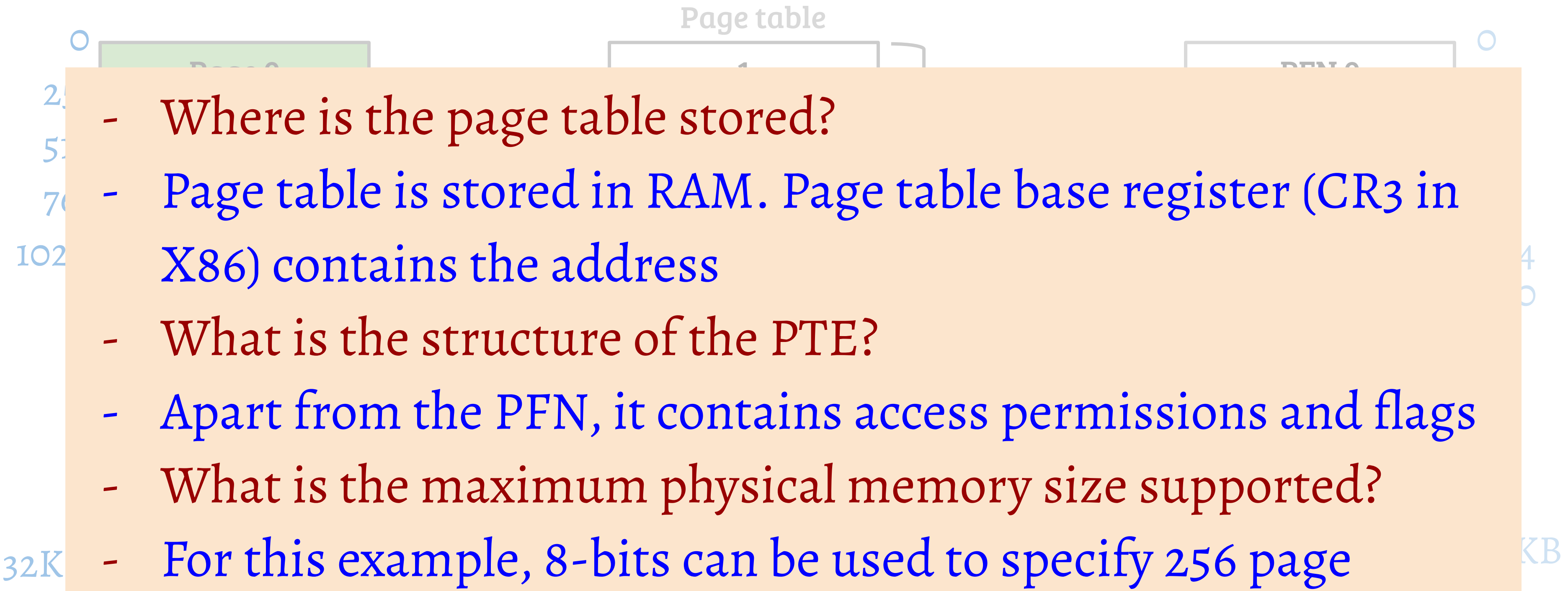
- Code: Page 0 (Read and Execute)
- Data: Page 2 and Page 3 (Read and Write)
- Stack: Page 127 (Read and Write)

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?

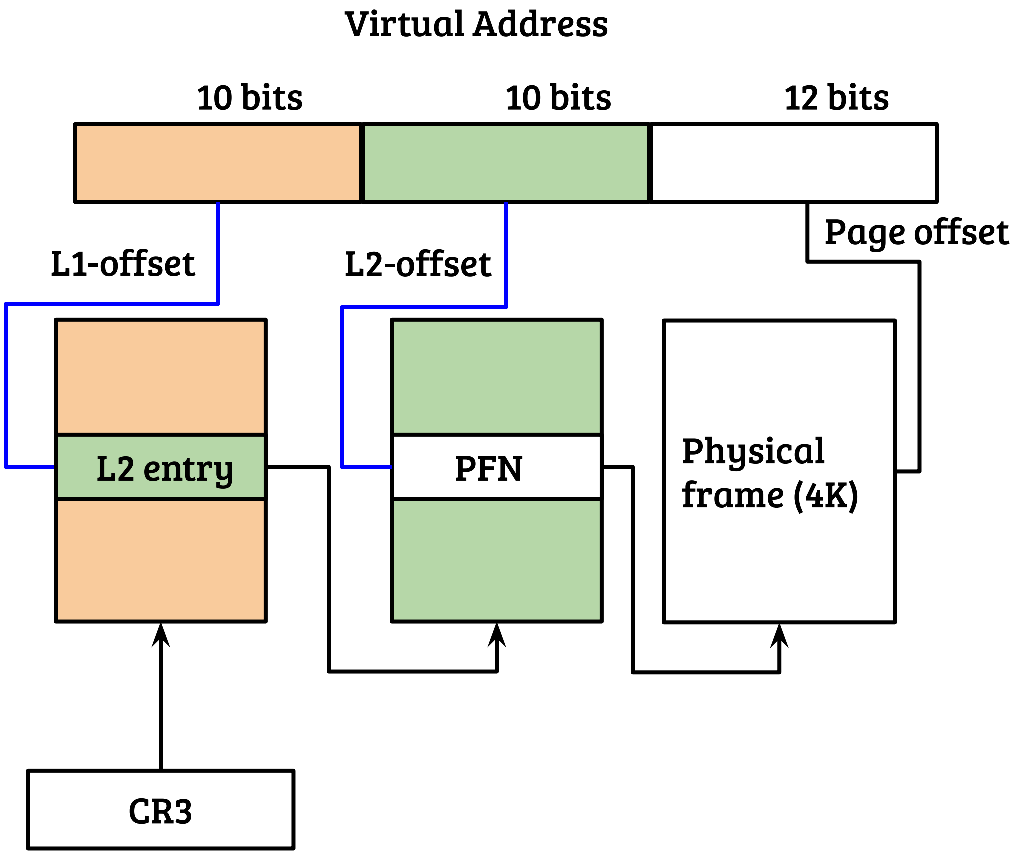
Paging: one level of page table may not be feasible!

- 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

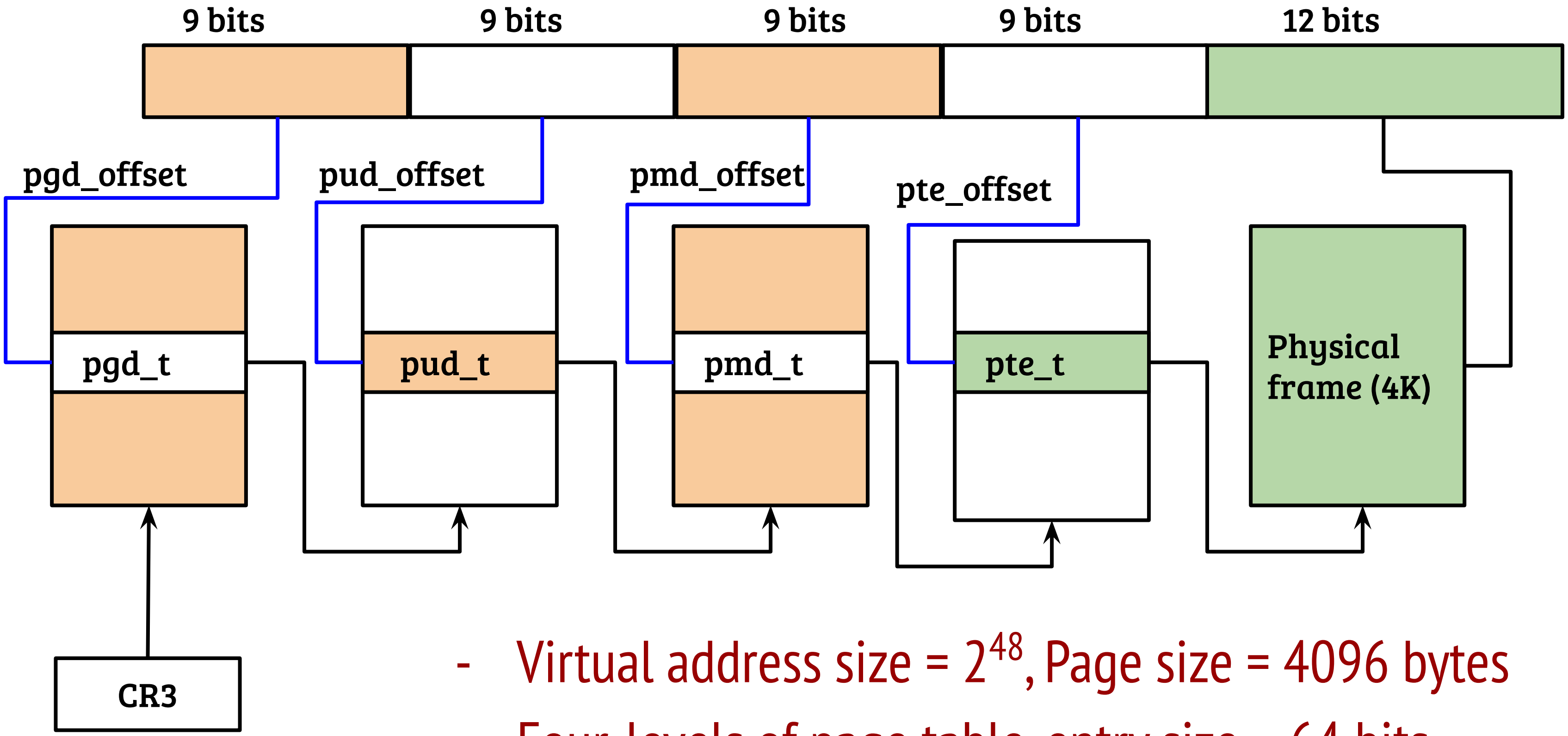
Virtual memory: Multilevel paging and TLB

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 increases internal fragmentation
  - Small pages may not be suitable to hold all mapping entries

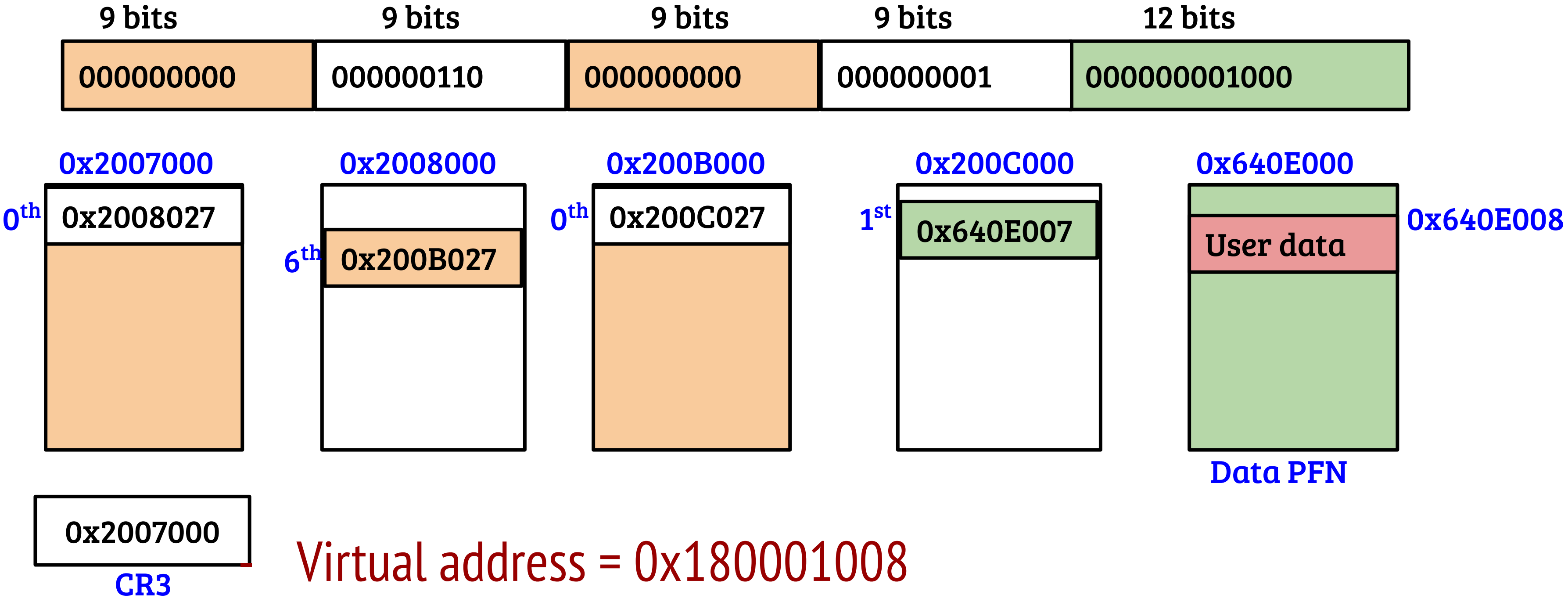
Today's agenda: Multi-level pages tables and implications

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



- Virtual address size = 2^{48}, Page size = 4096 bytes
- 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?

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

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

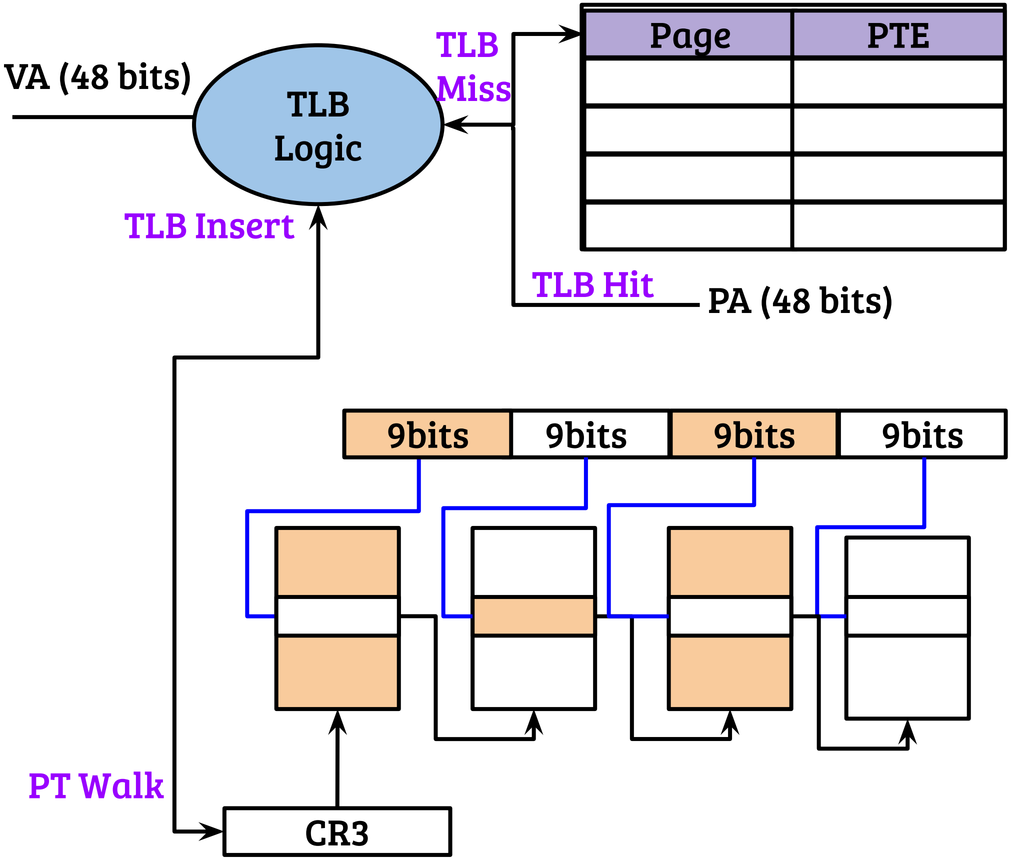
Paging: translation efficiency

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?
- One code page (0x20) and one stack page (0x7FFF). Caching these translations, will save a lot of memory accesses.

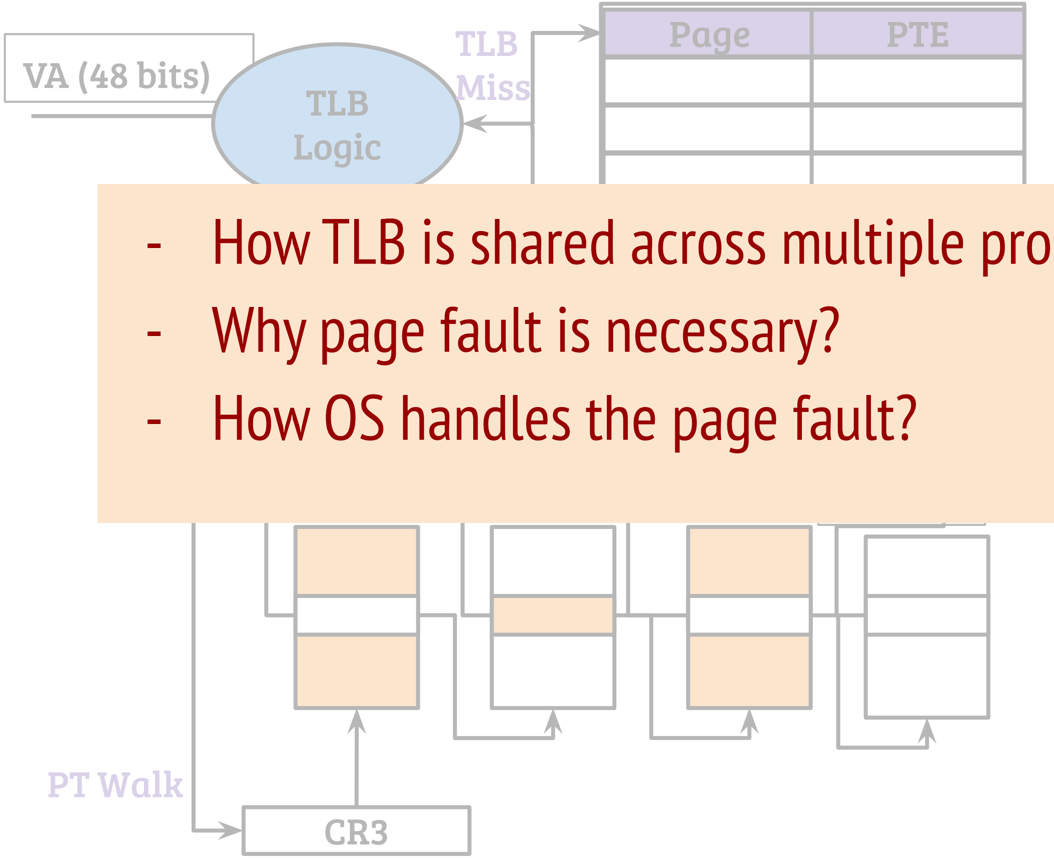
required (for translation) during the execution of the above code?

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

d

into the TLB directly, it can flush entries

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

TLB: Sharing across applications

Process (A)

Process (B)

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

Process (A)

Process (B)

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

Process (A)

Process (B)

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

Process (A)

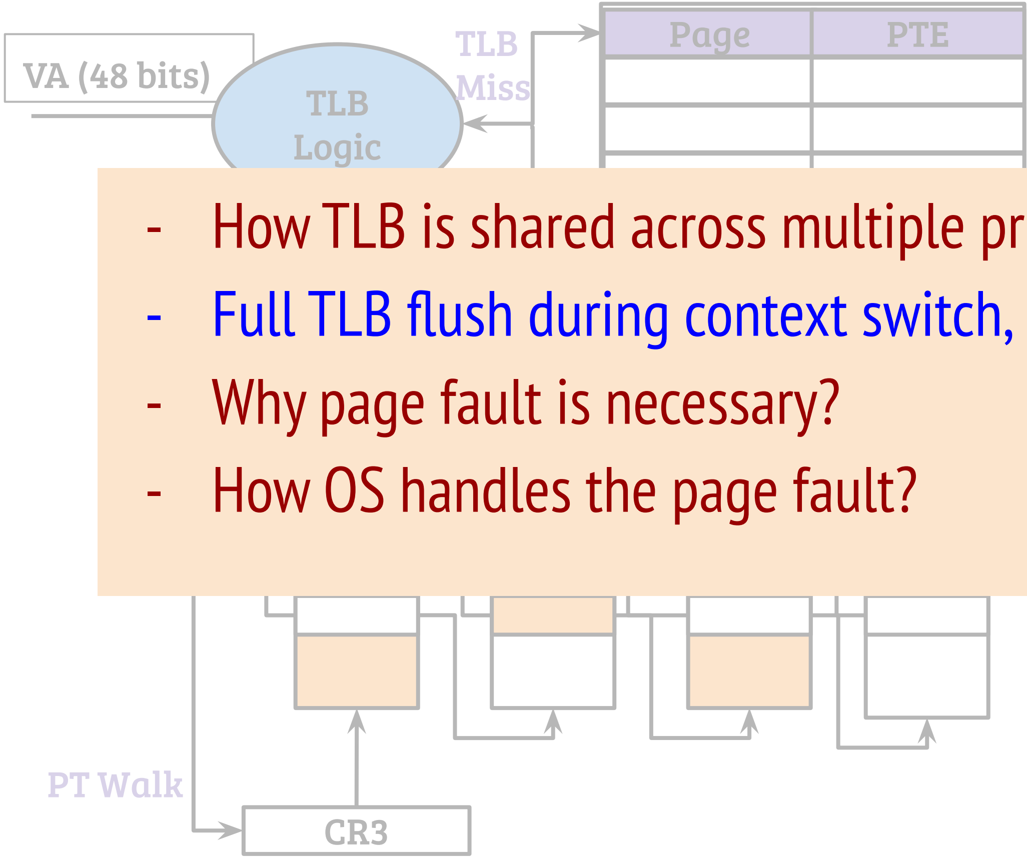
Process (B)

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)



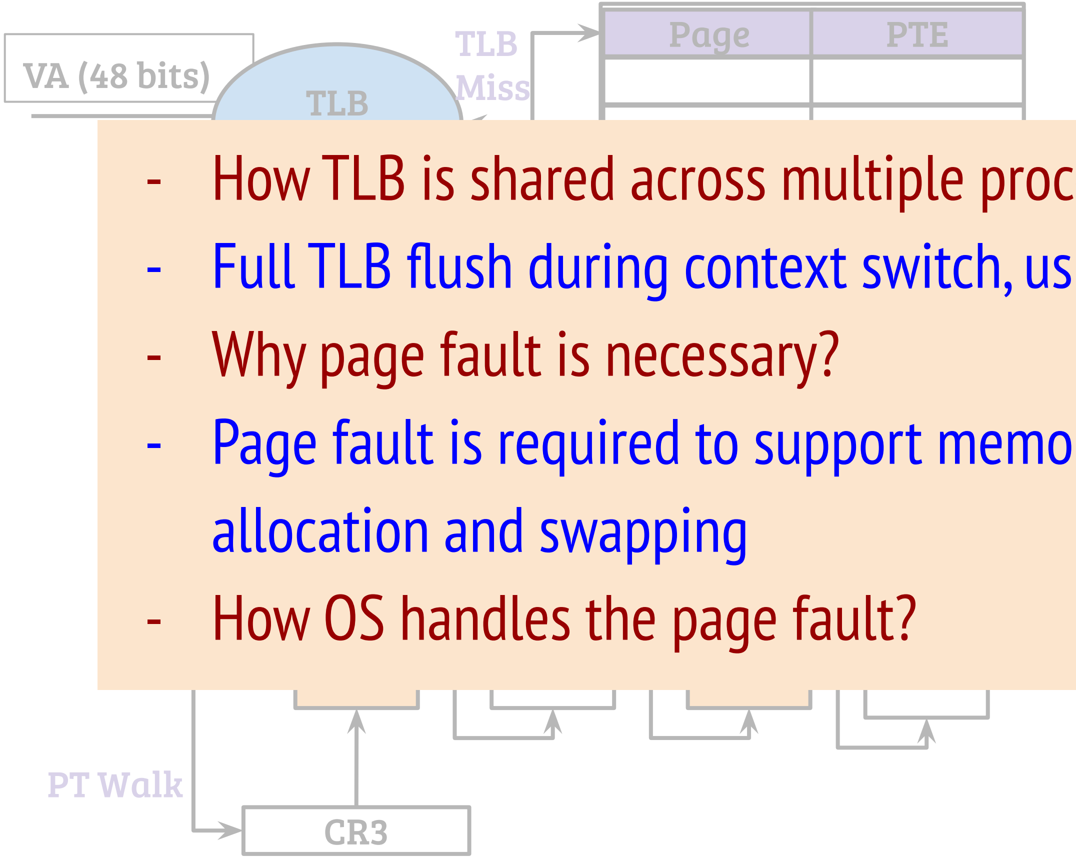
- TLB in the path of address

d

entries

- 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?

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
```

Error code

Other and unused	I	R	U	W	P

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

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);
}
```

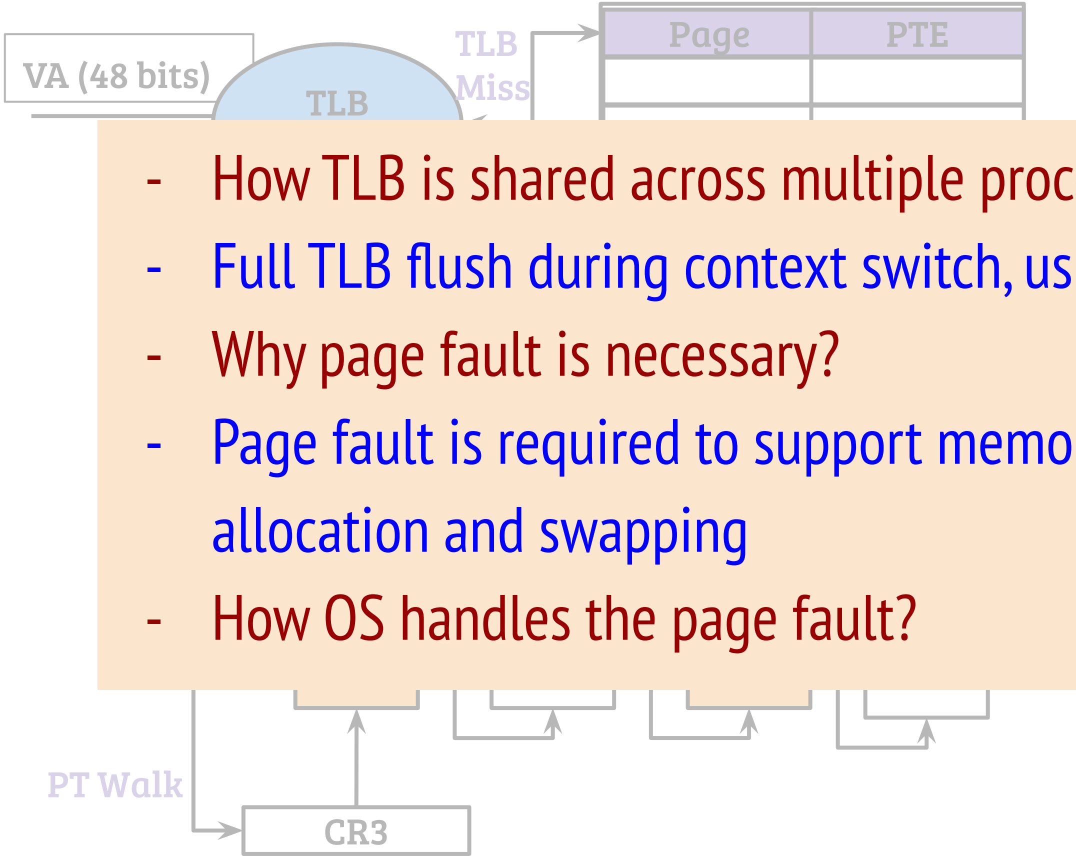
Address translation (TLB + PTW)

- 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

Recap: Address translation



- 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
```

Error code

Other and unused	I	R	U	W	P

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

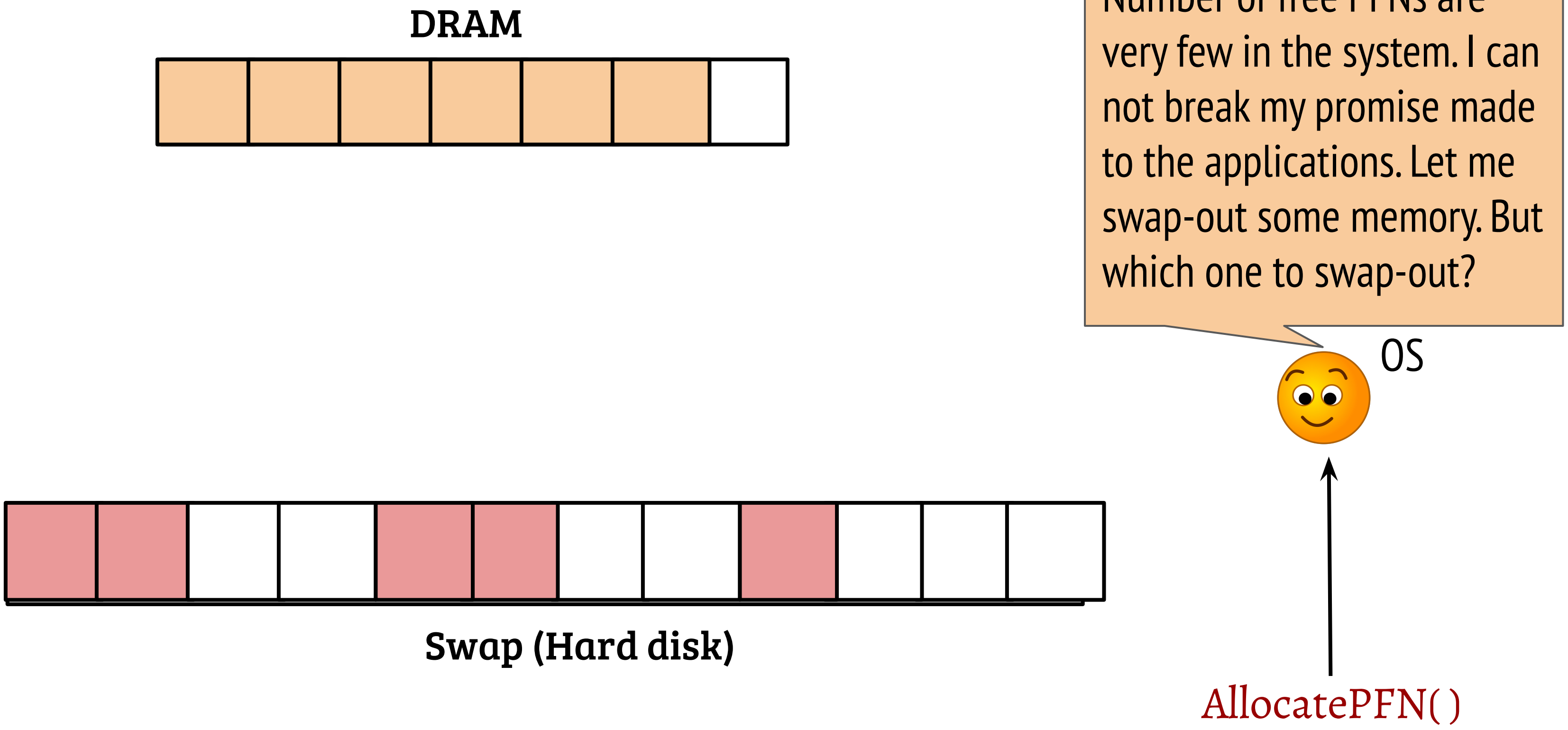
Page fault handling in X86: OS fault handler

```
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)

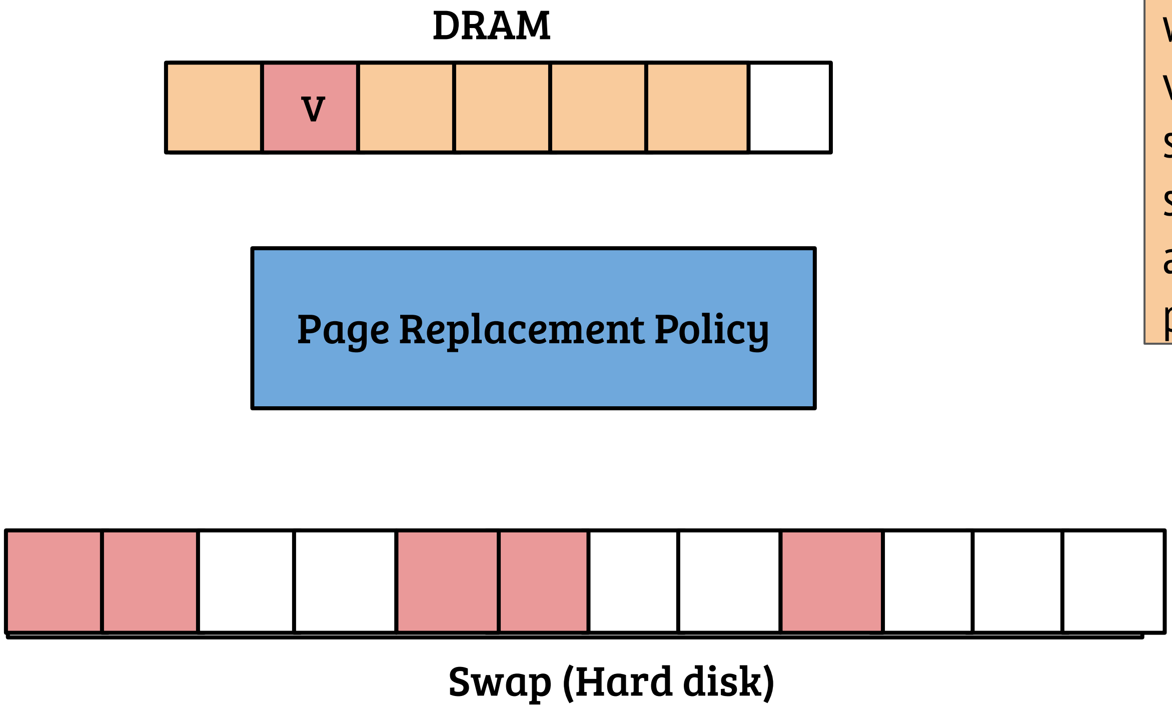
- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- Page fault is required to support memory over-commitment through lazy allocation and swapping
- How OS handles the page fault?
- The hardware invokes the page fault handler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

Swapping (swap-out)

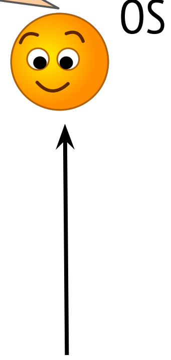


Number of free PFNs are very few in the system. I can not break my promise made to the applications. Let me swap-out some memory. But which one to swap-out?

Swapping (swap-out)

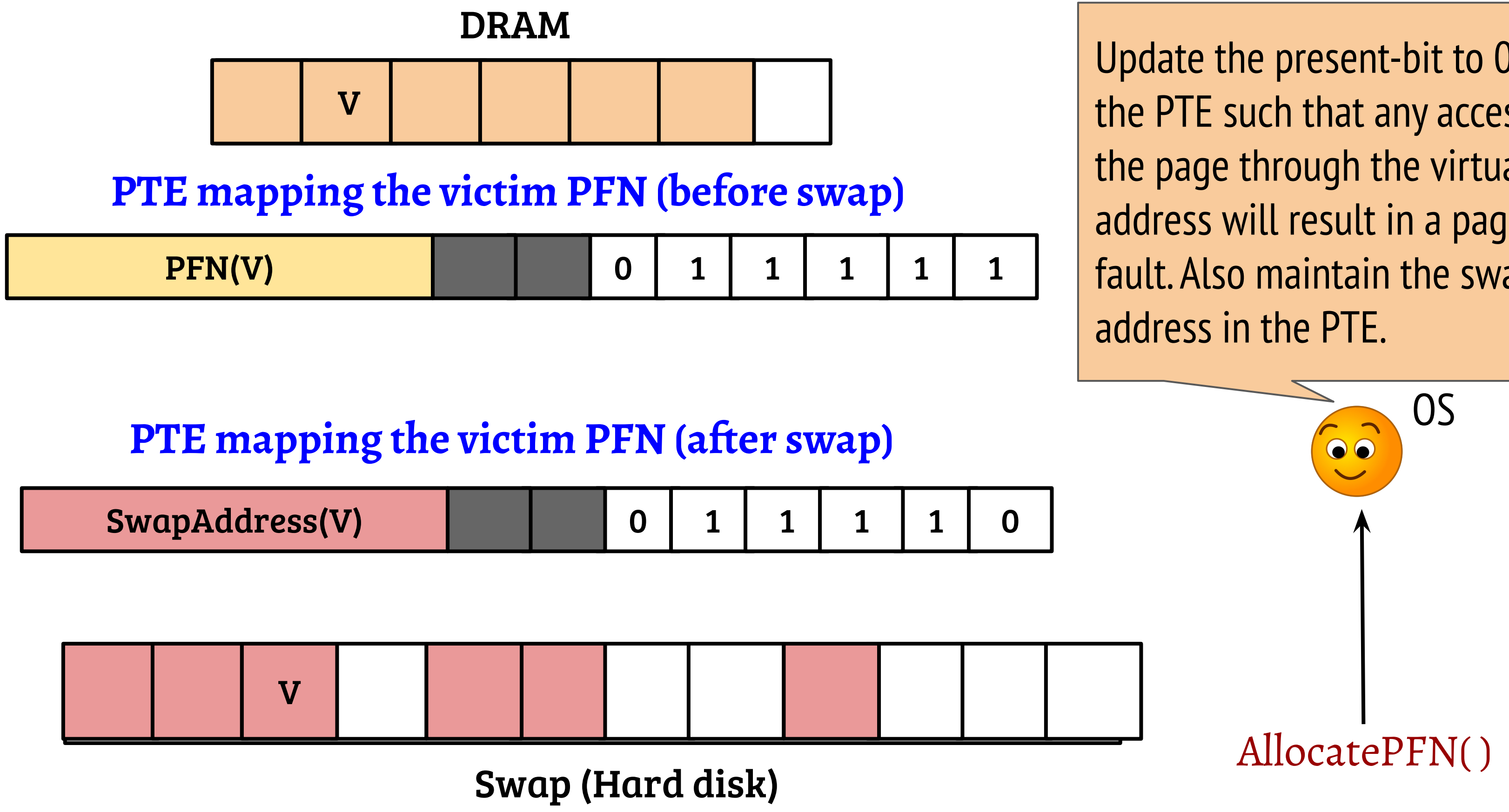


My page replacement policy will help me deciding the victims (V). Can I just swap-out? What if the swapped-out pages are accessed? I should be prepared for that too!



AllocatePFN( )

Swapping (swap-out)



PTE mapping the victim PFN (before swap)

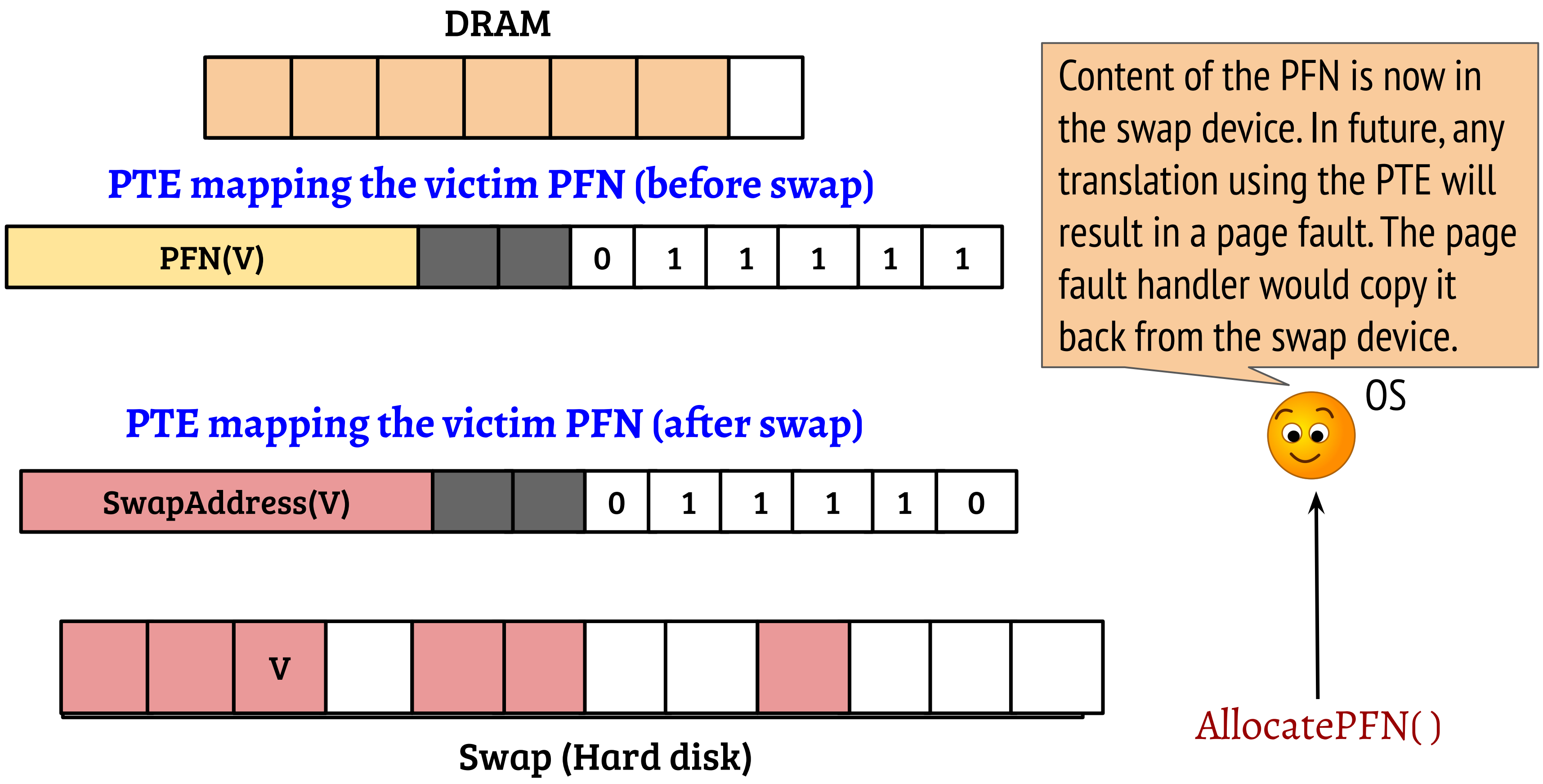
PFN(V)			0	1	1	1	1	1

PTE mapping the victim PFN (after swap)

SwapAddress(V)			0	1	1	1	1	0

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.

Swapping (swap-out)


DRAM
PTE mapping the victim PFN (before swap)
PFN(V) 0 1 1 1 1 1
PTE mapping the victim PFN (after swap)
SwapAddress(V) 0 1 1 1 1 0
V
Swap (Hard disk)
OS
AllocatePFN( )

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.

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