CS330 - Operating Systems

Virtual Memory

25-08-2025

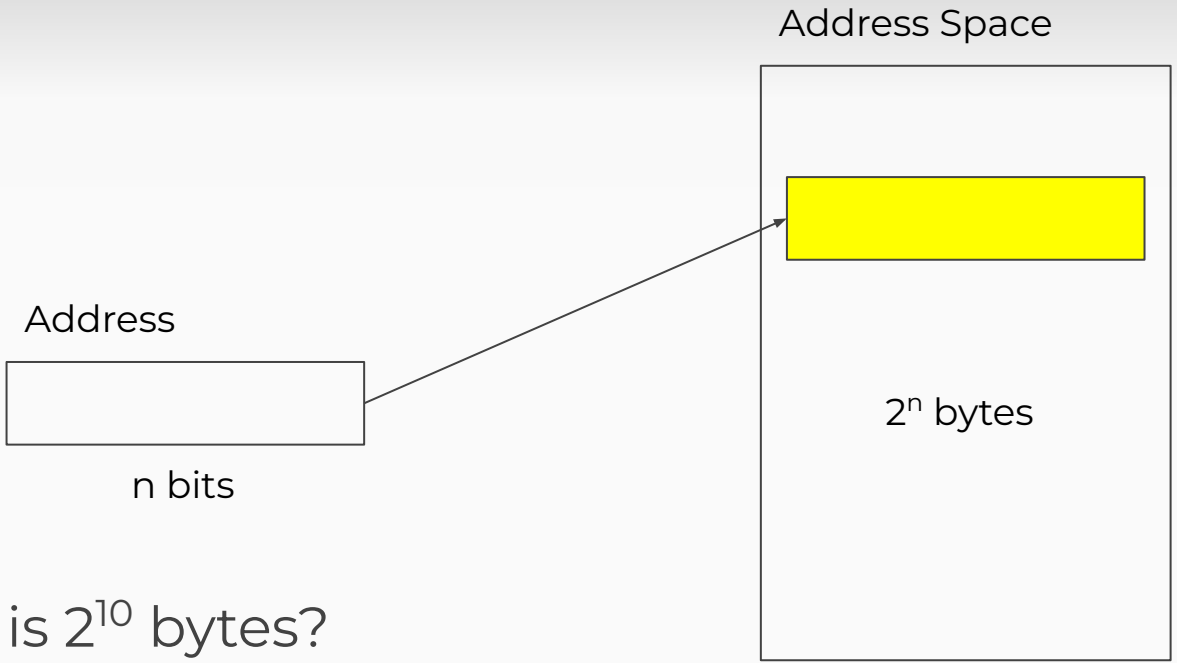
Virtualizing Resources

- Physical Reality:
  Different processes/threads share the same hardware
  - Need to multiplex CPU (synchronization and scheduling)
  - Need to multiplex use of Memory (**Address Spaces**)
  - Need to multiplex disk and devices (...)
- Why worry about memory sharing?
  - working state of a process is defined by its data in memory
- Address Spaces
  - the running program's view of memory in the system

Goals of Virtualizing Memory

- Transparency
  - Physical memory is invisible to user program
  - Program thinks it has own private large memory
- Efficiency
  - Not taking very long
  - Not taking too much space
- Protection/Isolation
  - Protect processes from each other

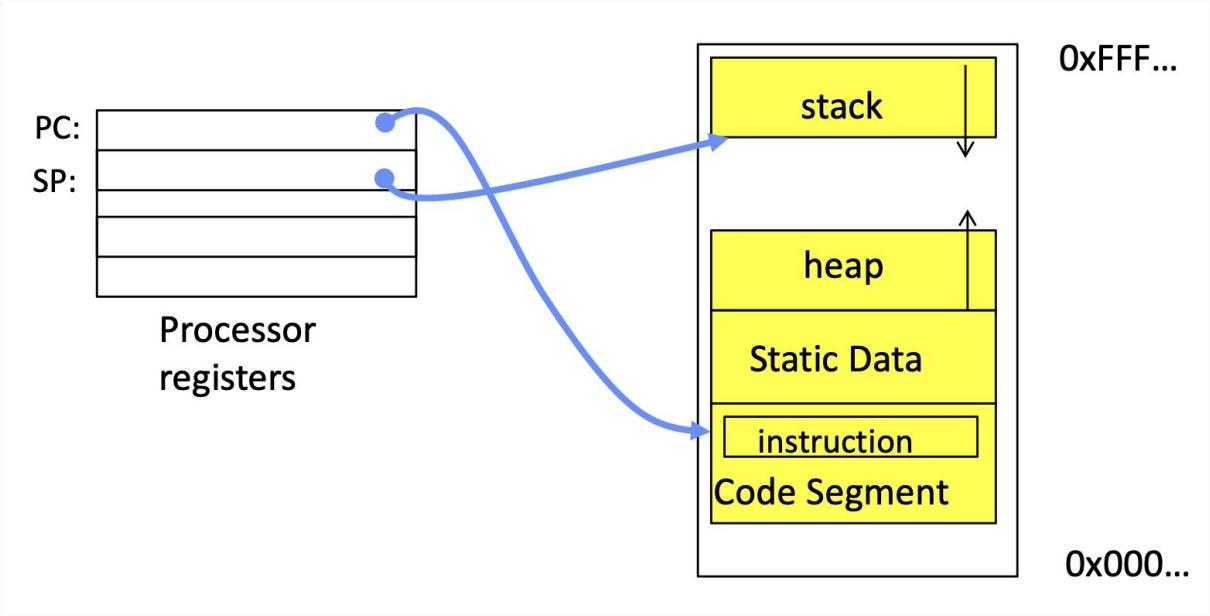
Address Spaces and Addresses



- What is 2^{10} bytes?
- How many bits to address each byte of 4KB memory?

Address Space

- Set of accessible addresses + State associated with them



Address Space

- All vCPU's share non-CPU resources
  - Memory, I/O devices
  - Read/write memory

Address Space : Protection

- Simple multiplexing does not protect memory
- OS must protect user programs from one another
  - Prevent threads owned by one user from impacting threads owned by another user

Address Space : Protection

- Simple multiplexing does not protect memory
- OS must protect user programs from one another
  - Prevent threads owned by one user from impacting threads owned by another user
- OS must protect itself from user programs
  - Reliability (compromising OS generally causes it to crash)
  - Security (limit the scope of what threads can do)
  - Privacy (limit each thread to the data it is permitted to access)
  - Fairness (each thread should be limited to its appropriate share of system resources like CPU and memory)
- Can hardware help the OS protect itself?

Memory Access

- malloc
- free

```
char *src = "hello";
char *dst;
strcpy(dst, src);
```

Address Translation

- Hardware transforms each memory access
  - Virtual address to physical address
- OS sets up the hardware for translations
- Programs share memory
  - CPUs switch between programs

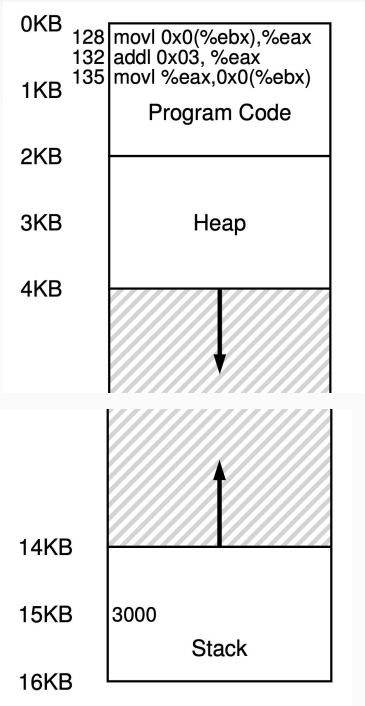
Address Translation

```
void func() {
    int x = 3000;
    x=x+3;
    …

128: movl 0x0(%ebx), %eax ;load 0+ebx into eax
132: addl $0x03, %eax     ;add 3 to eax register
135: movl %eax, 0x0(%ebx) ;store eax back to mem
```

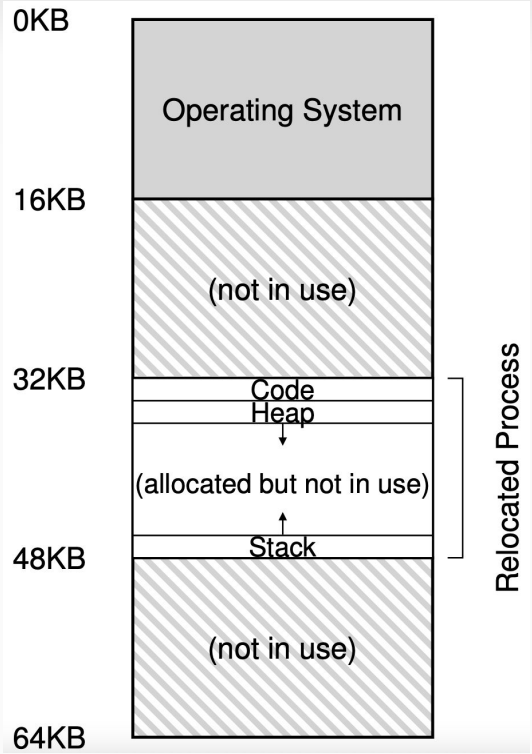
Address Translation

- Fetch instruction at address 128
- Execute this instruction
  (load from address 15 KB)
- Fetch instruction at address 132
- Execute this instruction
  (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction
  (store to address 15 KB)

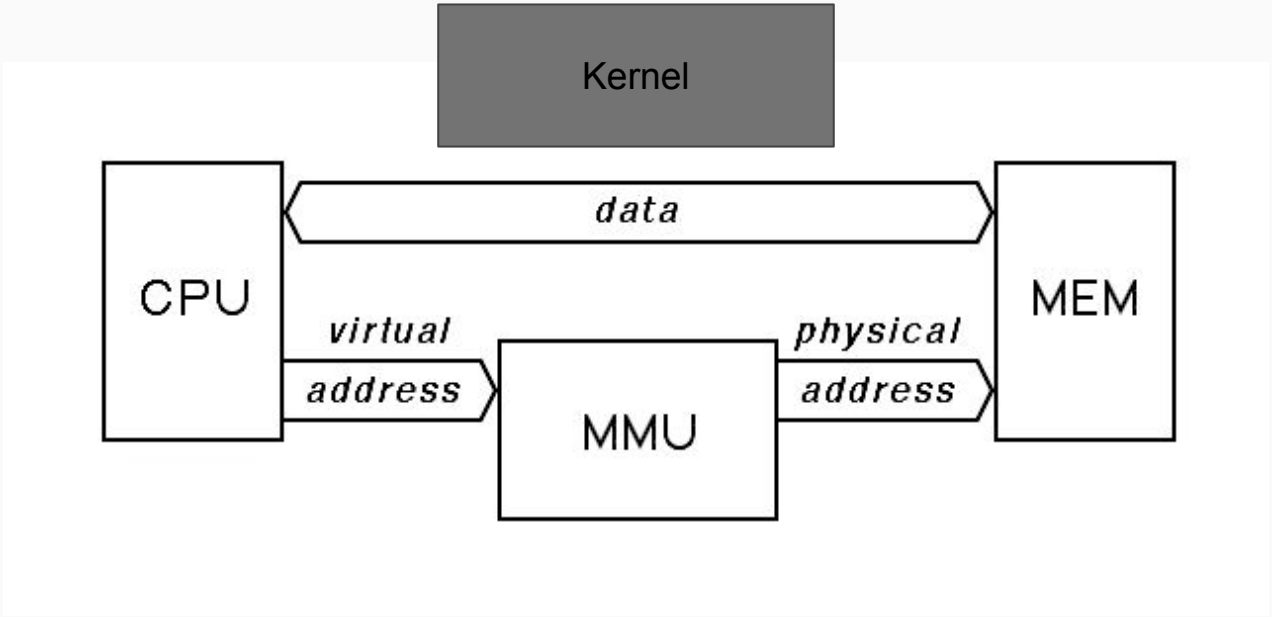


Address Translation

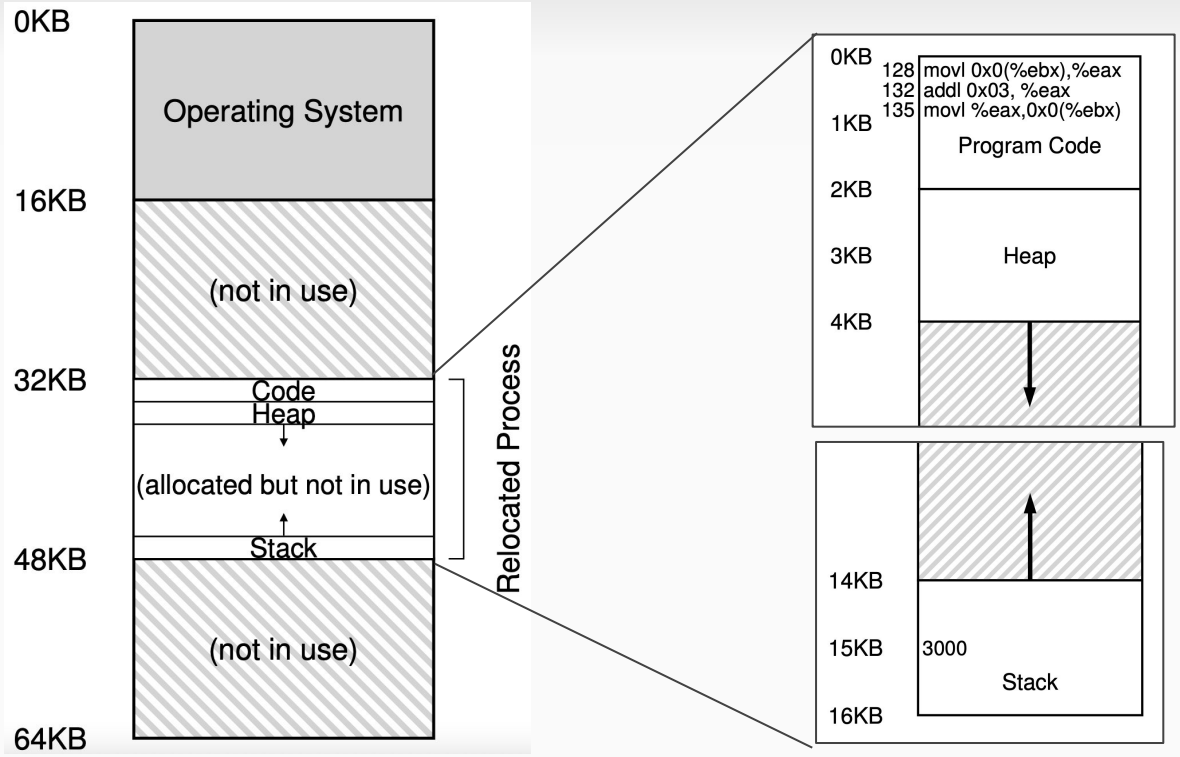
- OS itself is the first program mostly
- Program actually located at 32KB
- May need to relocate (when it blocks)



Address Translation



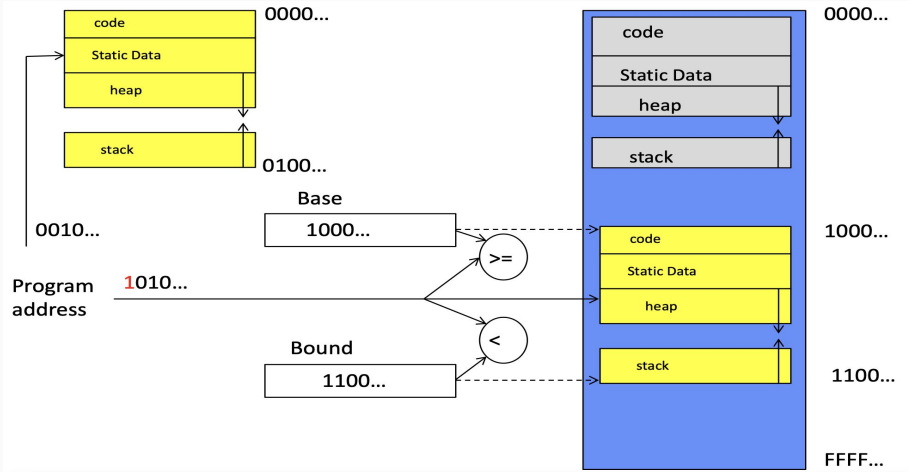
Address Translation



Dynamic Relocation - Base & Bound

- Process is assigned contiguous segment of memory
- Loads physical address into *base* and *bound* (limit - may also be size depending on the type of architecture)

```
physical address = virtual address + base
```



Base & Bound

```
128: movl 0x0(%ebx), %eax
```

Base : 32 KB

Bounds : 64 KB

Base & Bound

- Allows address space to be anywhere in memory
- Ensures process accesses its own space
- OS decides where in physical address to load the address space
- Relocation happens at runtime - can change even after creation

Base & Bound - OS and H/W Requirements

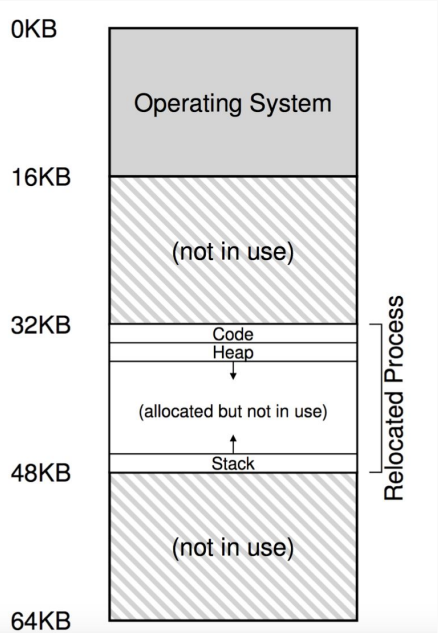
- H/W
  - Privileged Mode
    - Instructions to update base and bound
    - Register exception handlers
  - Base & bounds registers
  - Translate VA and check bounds
  - Allow raising exceptions
- OS
  - Manage memory
  - Manage registers
  - Handle exceptions

Base & Bound - Cons

- Contiguous block of memory needed in physical memory
  - Internal Fragmentation

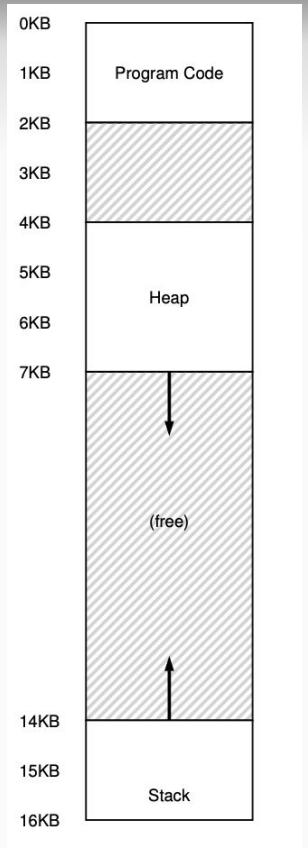
Segmentation

- Base & bounds pair per logical segment
- Segment is continuous memory portion
  - Code
  - Stack
  - Heap
- Each segment separate in memory



Segmentation

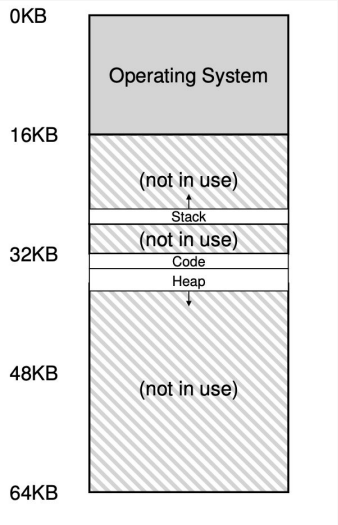
- Base & bounds pair per logical segment
- Segment is continuous memory portion
  - Code
  - Stack
  - Heap
- Each segment separate in memory
- Hardware structure in MMU
  - Set of three base & bound registers

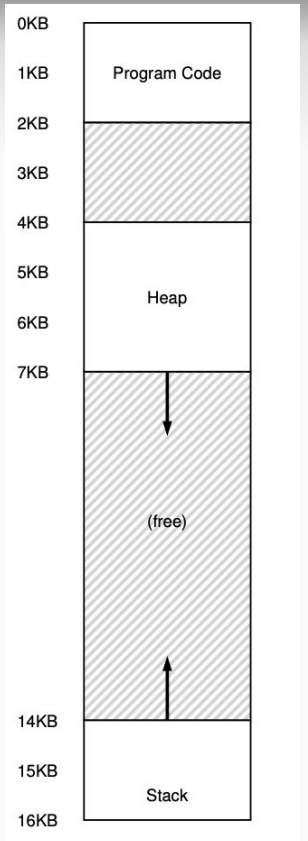


Segmentation - Example

Segment	Base	Bound
Code	32K	2K
Heap	34K	3K
Stack	28K	2K

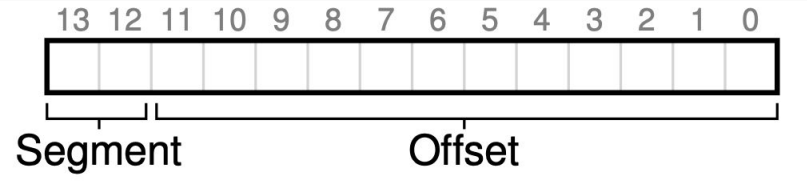
What is the size of the address in bits?

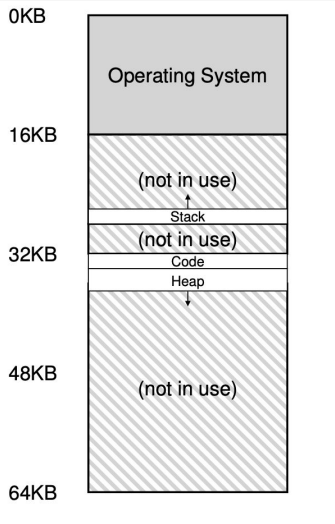


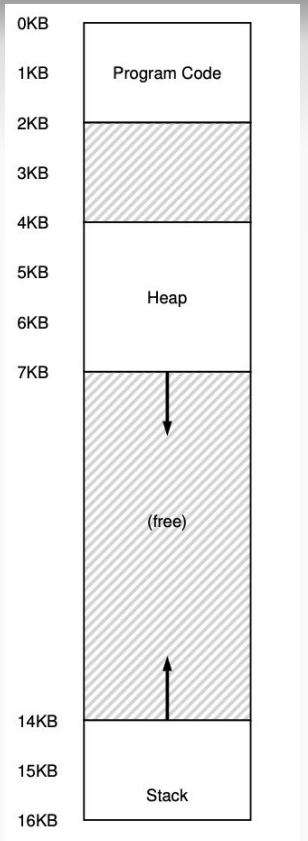


Segmentation - Segments and Offsets

Segment	Base	Bound
Code	32K	2K
Heap	34K	3K
Stack	28K	2K

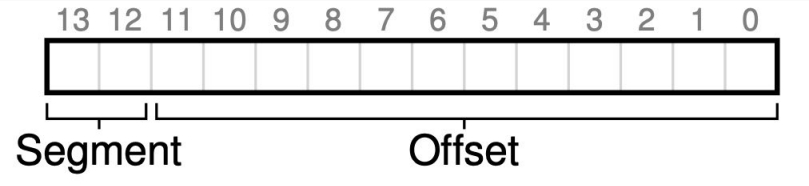


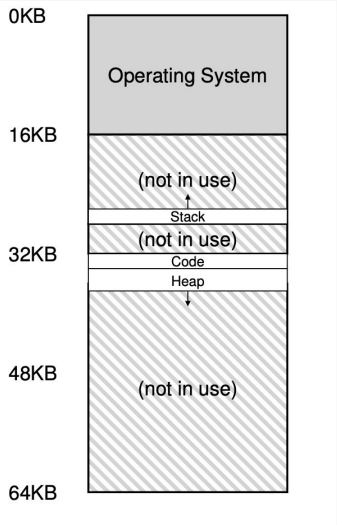


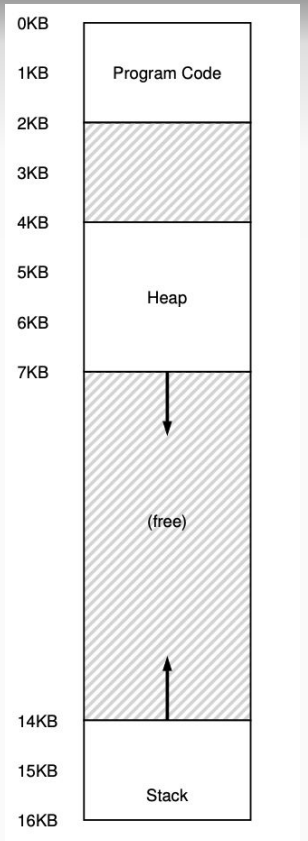


Segmentation - Example

Segment	Base	Bound	Grows +ve
Code	32K	2K	1
Heap	34K	3K	1
Stack	28K	2K	0



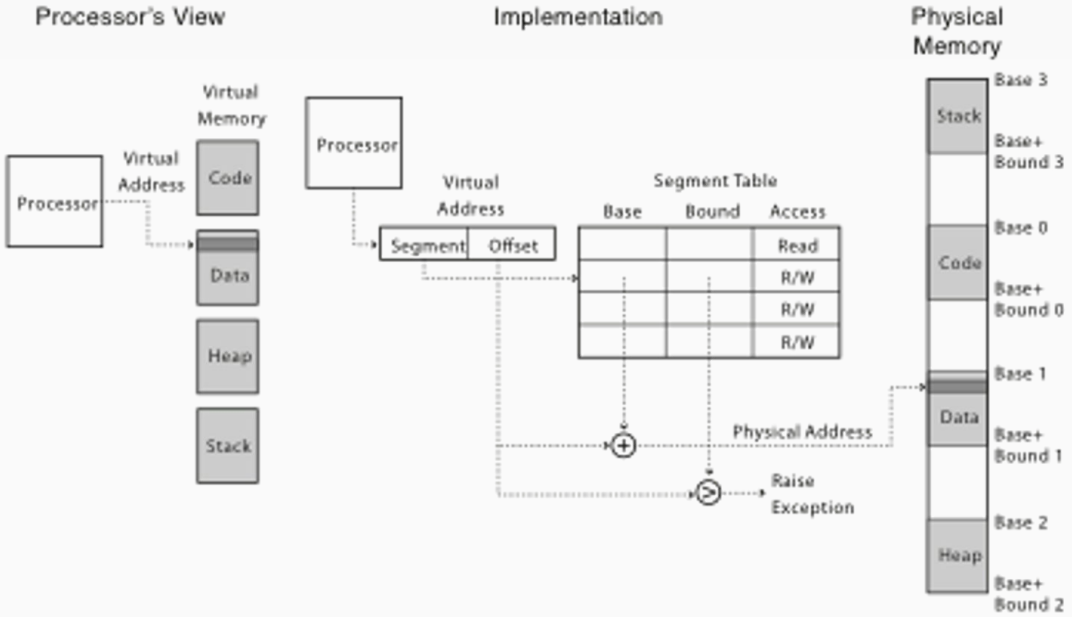




Segmentation

- Each segment has:
  - Start virtual address (VA)
  - Base physical address
  - Bound
- Virtual Address is OK if it inside some segment:
  Segment Start < V.A. < Segment Start + Segment Bound
- For the segment that contains this virtual address:
  Physical Address = (V.A. - Segment Start) + Base

Segmentation - Overview



OS Support for Segmentation

- What should the OS do on context switch?
- What should the OS do when segments grow?
- How does the OS manage free space?

OS Support for Segmentation

- What should the OS do on context switch?
  - Save segment table and register values
- What should the OS do when segments grow?
  - Add more memory if needed or throw an exception
- How does the OS manage free space?
  - Rearrange processes in physical memory
  - Manage free-list more effectively

Segmentation Summary

- Pros?
  - Can share code/data segments between processes
  - Can protect code segment from being overwritten
  - Can transparently grow stack/heap as needed
- Cons?
  - Complex memory management
    - Need to find chunk of a particular size
  - May need to rearrange memory from time to time to make room for new segment or growing segment
    - External fragmentation: wasted space between chunks

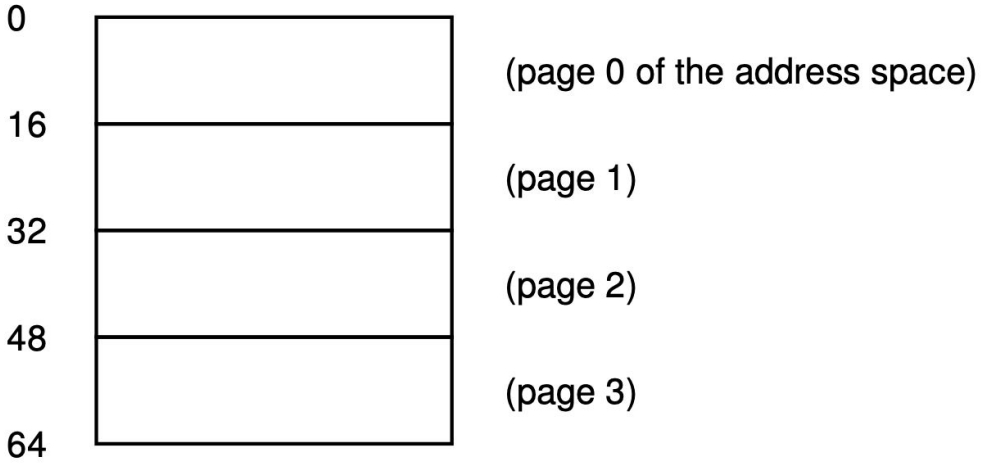
Paging

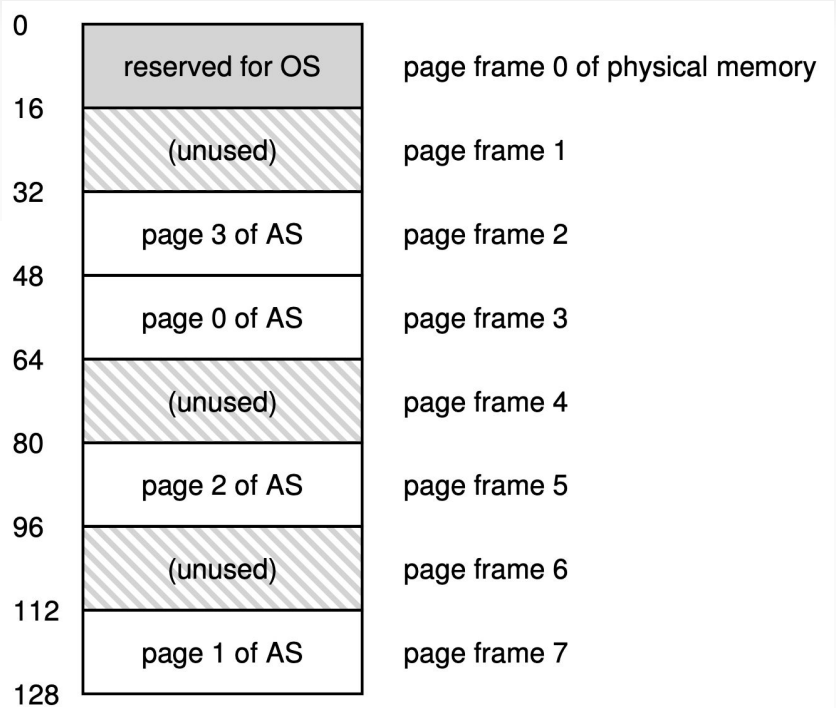
- Manage memory in fixed size units, or **pages**
  - In physical memory, these are called **frames**

Paging

- Manage memory in fixed size units, or **pages**
  - In physical memory, these are called **frames**
- Finding a free page is easy
  - Bitmap allocation: 0011111100000001100
  - Each bit represents one physical page frame
- Each process has its own page table
  - Stored in physical memory
  - Hardware registers
    - pointer to page table start
    - page table length

Paging - Example





Paged Translation

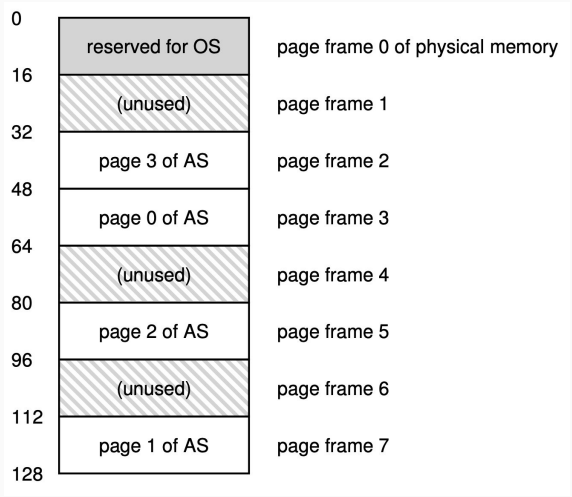
- Page table
  - Per-process data structure

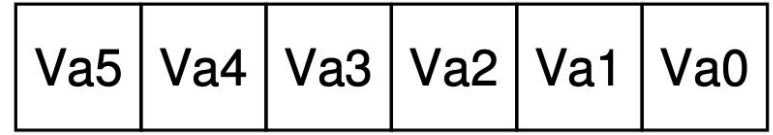
```
movl 21, %eax
```

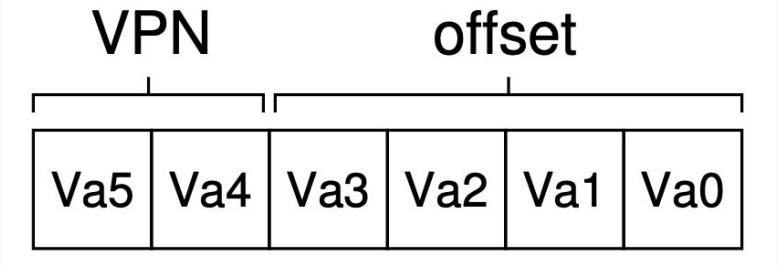
Paged Translation

- Page table
  - Per-process data structure

```
movl 21, %eax
```



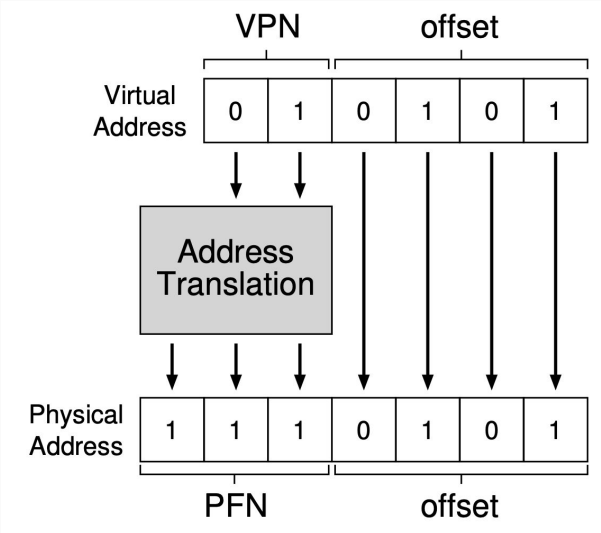


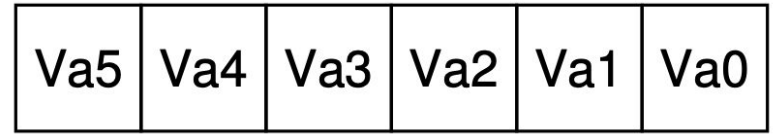


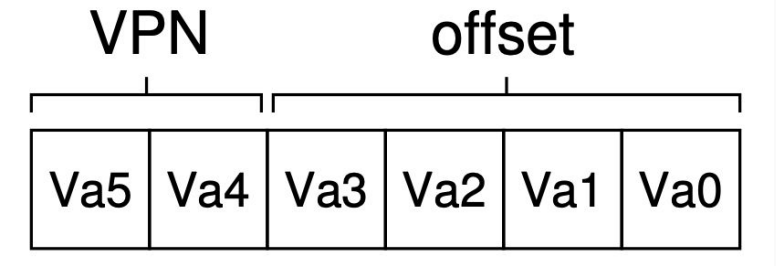
Paged Translation

- Page table
  - Per-process data structure

```
movl 21, %eax
```

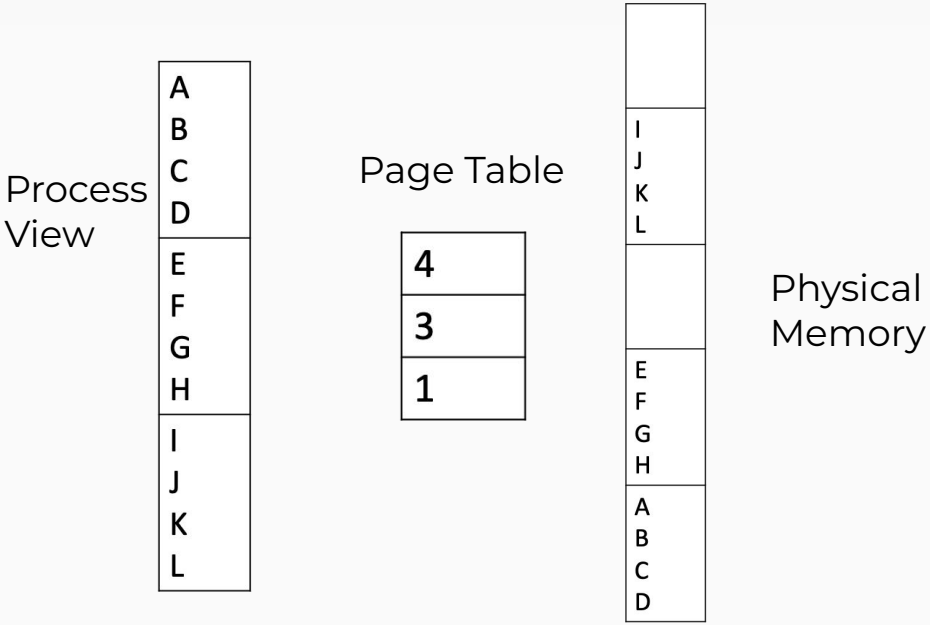






Paged Translation - Example

- Suppose page size is 4 bytes
- Where is VA 6?



Paging

- With paging, what is saved/restored on a process context switch?

Paging

- With paging, what is saved/restored on a process context switch?
  - Pointer to page table, size of page table
  - Page table itself is in main memory
- What if page size is very small?
- What if page size is very large?

Paging

- With paging, what is saved/restored on a process context switch?
  - Pointer to page table, size of page table
  - Page table itself is in main memory
- What if page size is very small?
- What if page size is very large?
  - Internal fragmentation: if we don't need all of the space inside a fixed size chunk

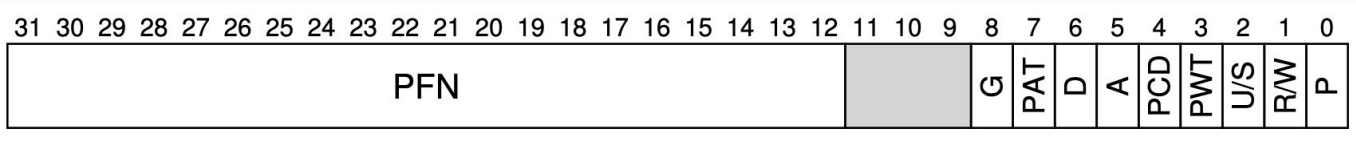
Page Tables

- Stores virtual-to-physical address translations
- One page table per process
- Where are page tables stored?

Page Tables

- Stores virtual-to-physical address translations
- One page table per process
- Where are page tables stored?
  - Can get large

- Page table entry



Slowness of Paging

```
movl 21, %eax
```

```
VPN     = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))

offset   = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
```

Summary of Paging

- Manage memory in fixed size units, or pages
- Each process has its own page table
  - Stored in physical memory
  - Hardware registers
    - pointer to page table start
    - page table length
- Cons
  - **Too slow**
  - Too much memory