Operating System Labs

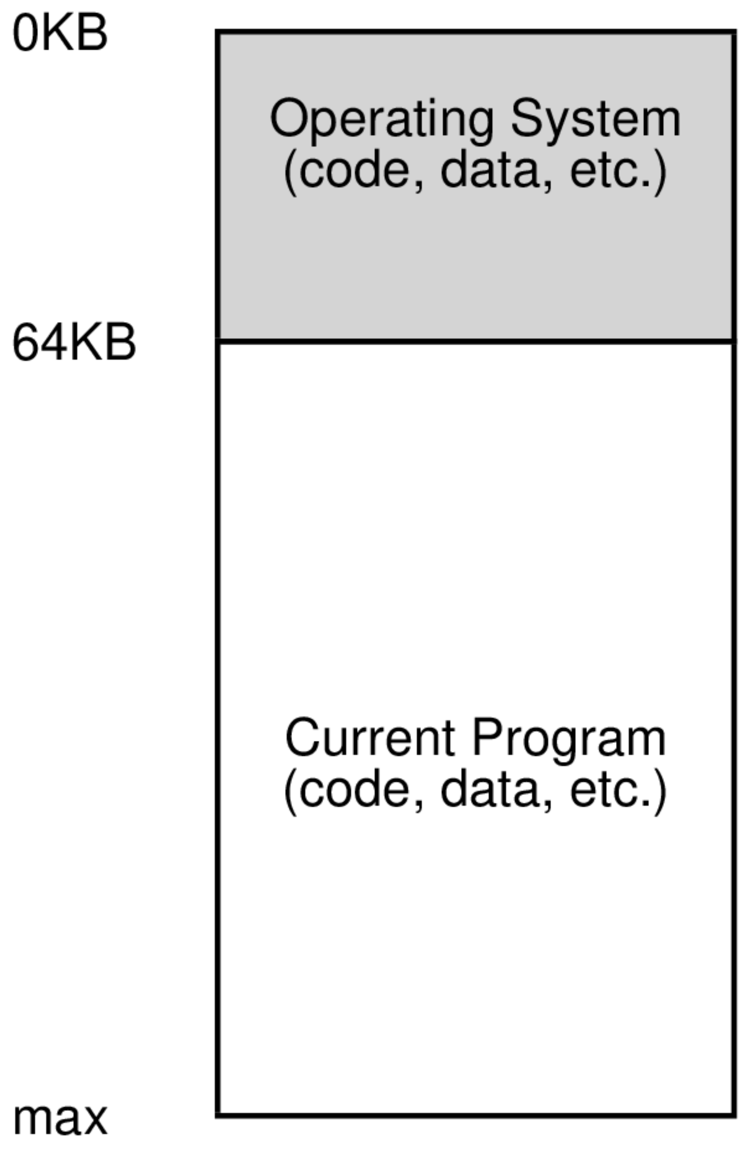
Yuanbin Wu
cs@ecnu

Operating System Labs

- Review of Memory Management

Memory Management

- Early days

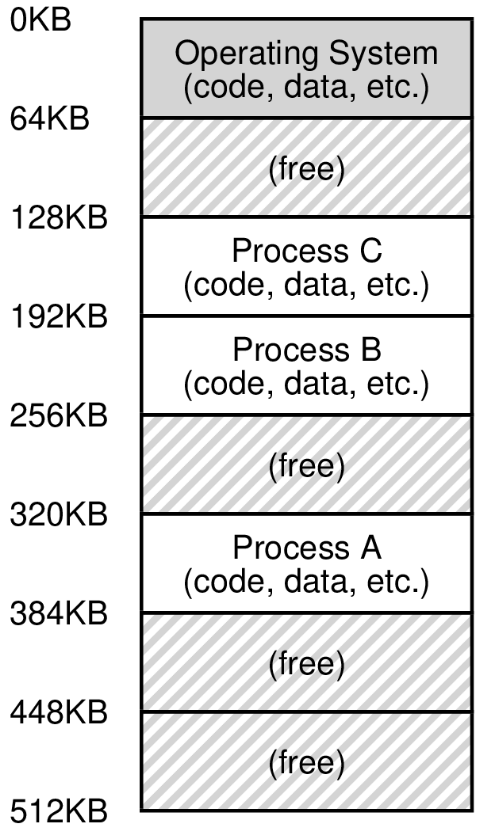

0KB
Operating System (code, data, etc.)
64KB
Current Program (code, data, etc.)
max

Memory Management

- Multiprogramming
  - multiple processes could be ready to run at a given time
  - the OS would switch between them
- Time sharing
  - many users might be concurrently using a machine

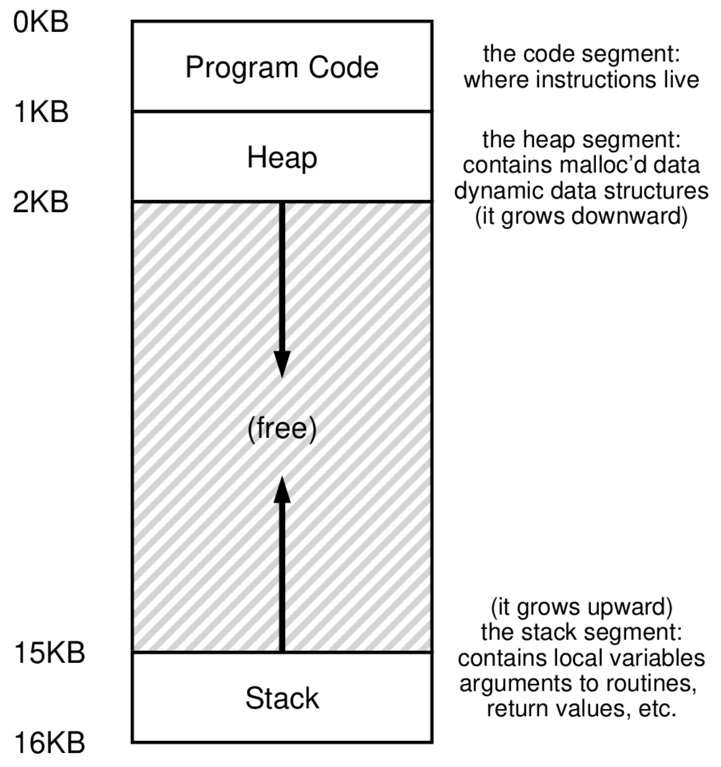
Memory Management

- Multiprogramming and Time Sharing
  - Multiple processes live in memory simultaneously


0KB
64KB
128KB
192KB
256KB
320KB
384KB
448KB
512KB
Operating System (code, data, etc.)
(free)
Process C (code, data, etc.)
Process B (code, data, etc.)
(free)
Process A (code, data, etc.)
(free)
(free)

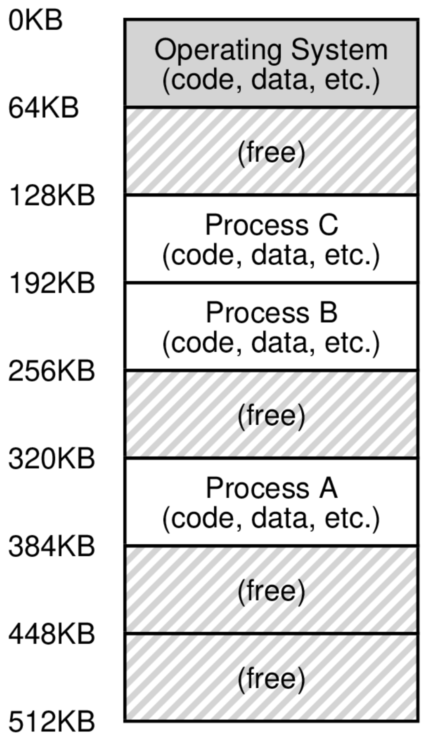
Memory Management

- Multiprogramming requires easy-to-use virtualization of memory
  - A concept called "address space"

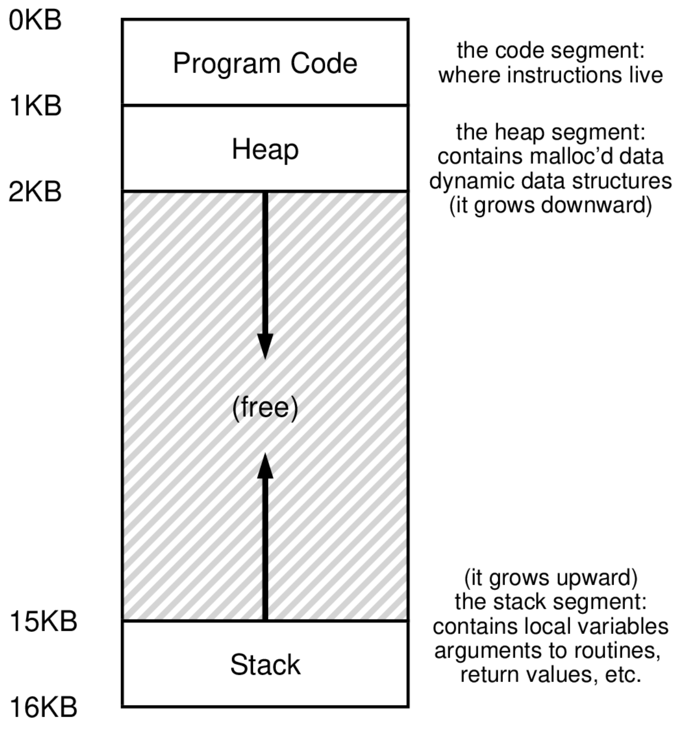

0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows downward)
(it grows upward) the stack segment: contains local variables arguments to routines, return values, etc.

Memory Management

- Two views on memory
  - From processes: different processes have different address spaces
  - From OS: limited physical memory cells



0KB
Operating System (code, data, etc.)
64KB
(free)
128KB
Process C (code, data, etc.)
192KB
Process B (code, data, etc.)
256KB
(free)
320KB
Process A (code, data, etc.)
384KB
(free)
448KB
(free)
512KB



0KB
Program Code
the code segment: where instructions live
1KB
Heap
the heap segment: contains malloc'd data dynamic data structures (it grows downward)
2KB
(free)
(it grows upward) the stack segment: contains local variables arguments to routines, return values, etc.
15KB
Stack
16KB

Memory Management

- Memory management
  - How OS provide such easy-to-use address spaces for processes?
  - **Virtualization** of memory
    - Remember we have a virtualization of CPU

Memory Management

- Goals of Virtualize Memory
  - Transparency
  - Efficiency
  - Protection
    - The OS should make sure to protect processes from one another

Memory Management

- Transparency
  - OS should implement virtual memory in a way that is invisible to the running program
  - From the programmer's point of view:
    - Every address is fraud
    - Only the OS knows the truth

Memory Management

- Virtualize Memory
  - Limited Direct Execute
  - Hardware:
    - transparency, efficiency, protection
  - OS:
    - configure hardware correctly
    - manage free memory
    - handle exception
  - Hardware-based address translation

Memory Management

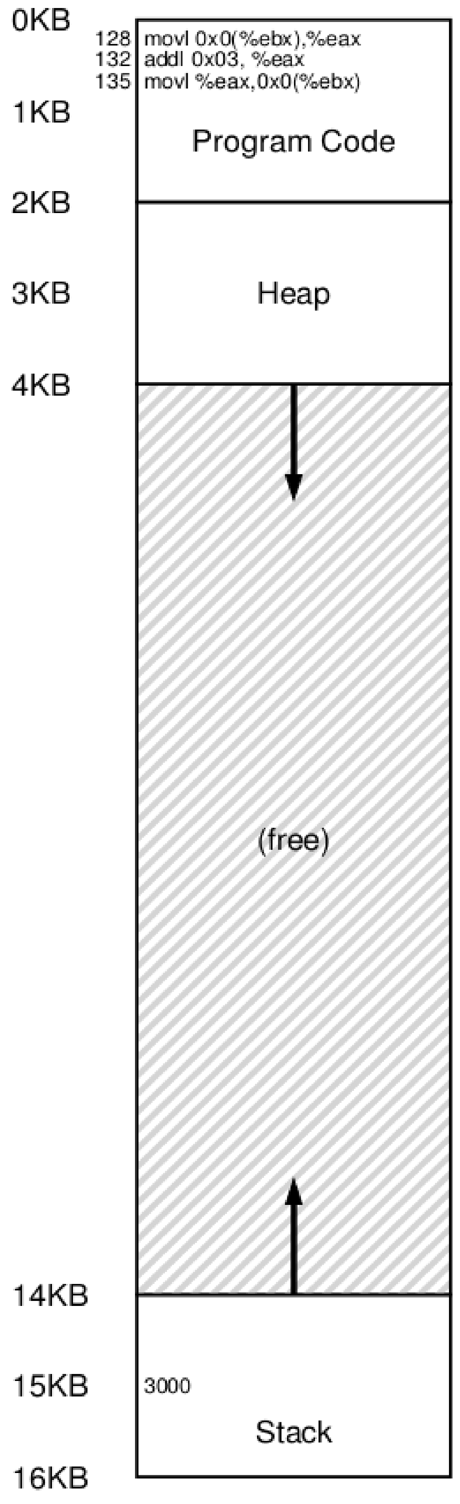
- Hardware: Transparency
  - We starts with a simple idea called
    - Base and bounds
    - Dynamical (hardware-based) allocation

An Example

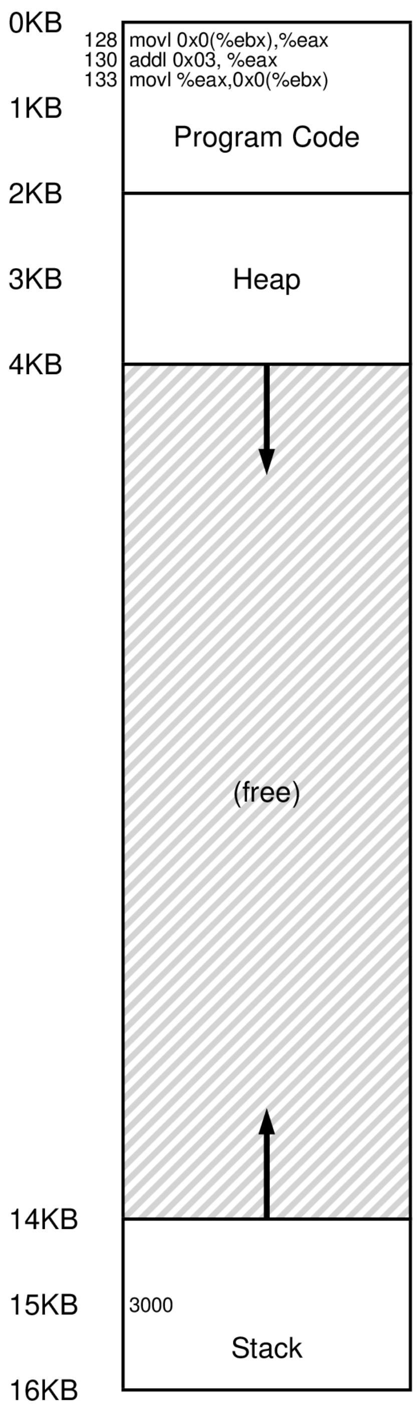
```
void func ()
{
    int x;
    x = x + 3;
}
```

```
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
```

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 space

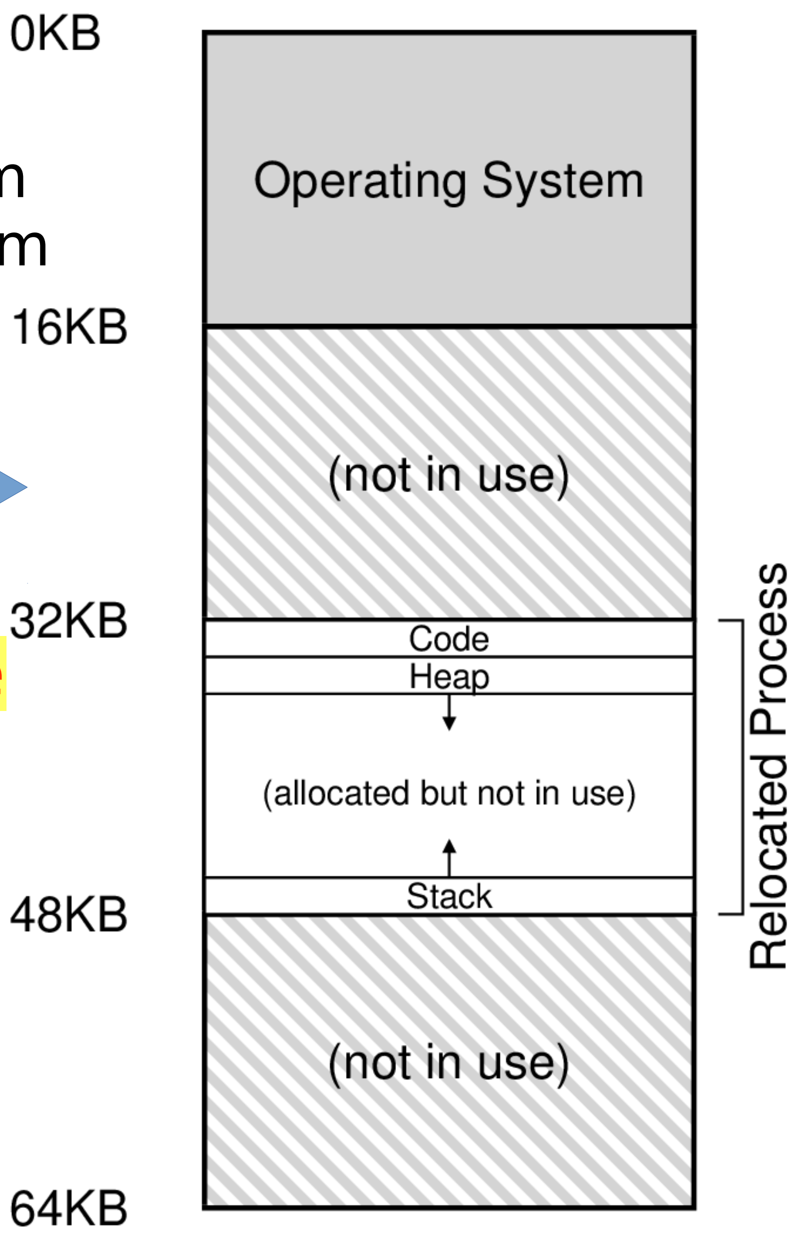


Physical Memory

Hardware:
- 2 registers in CPU
- Base: the start of phy mem
- Bound: the size of phy mem

physical = virtual + base

Base: 32K
Bound: 16K



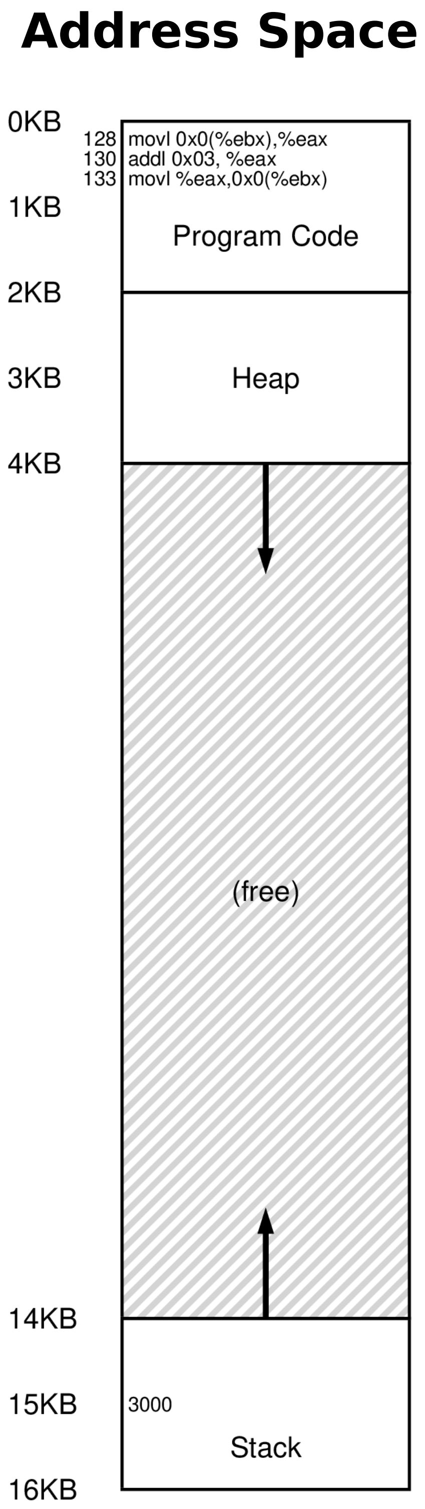
physical = virtual + base

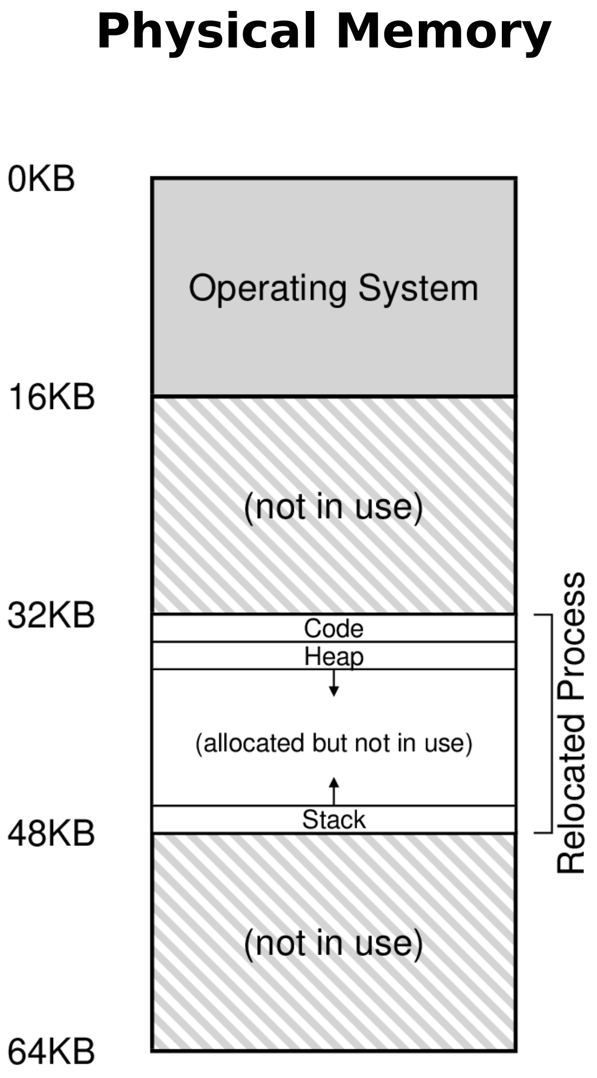
Fetch instruction at address 128
Execute (load from address 15 KB)
Fetch instruction at address 132
Execute (no memory reference)
Fetch the instruction at address 135
Execute (store to address 15 KB)

Visiting address 128

$$\begin{aligned} & 128 + 32K \\ = & 128 + 32768 \\ = & 32896 \end{aligned}$$

Base: 32K
Bound: 16K





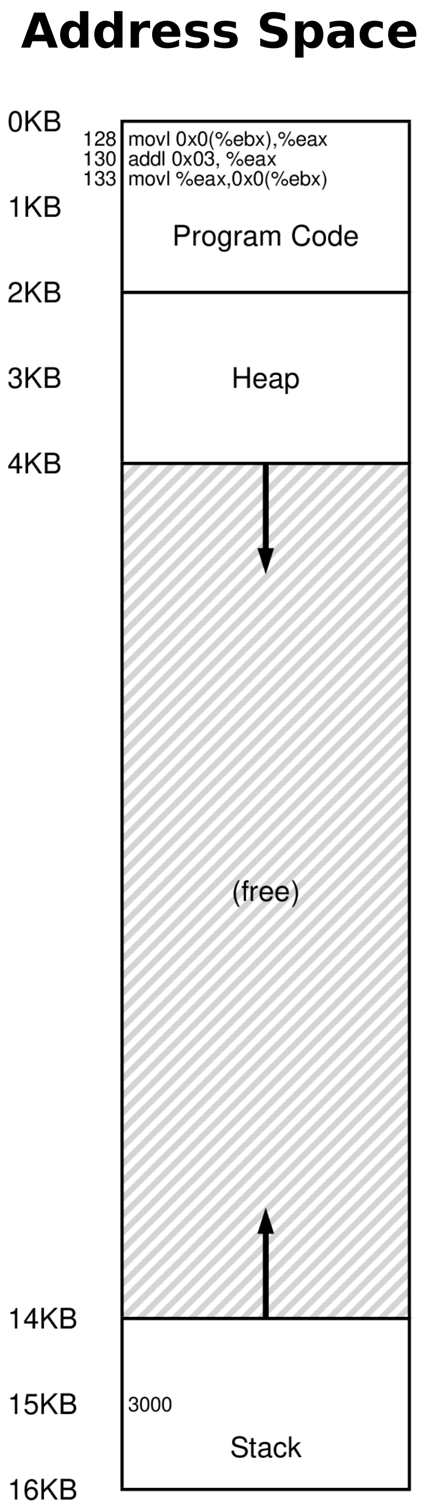
physical = virtual + base

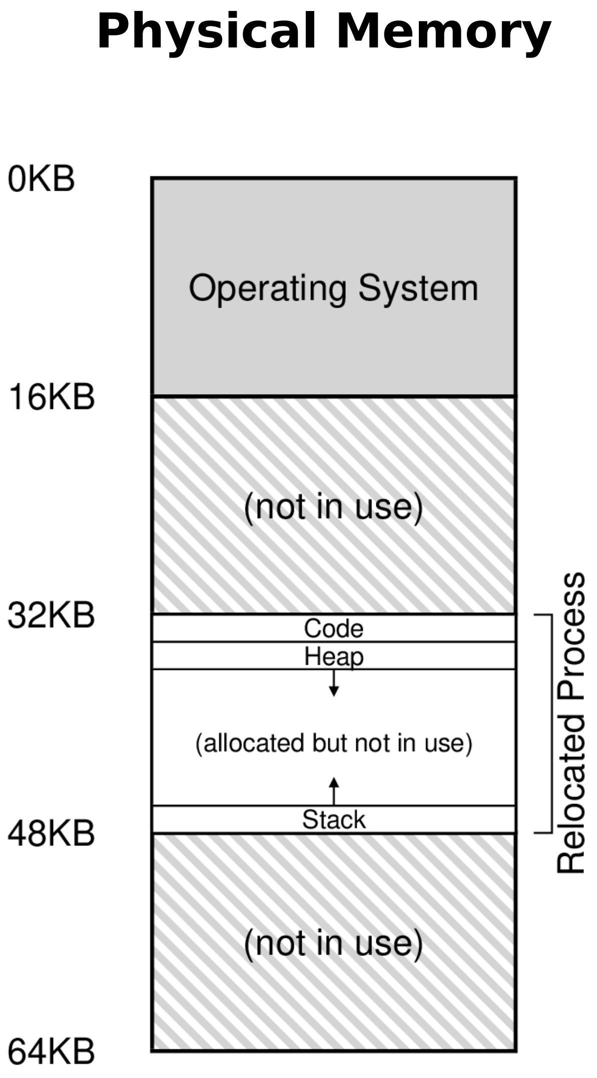
Fetch instruction at address 128
Execute (load from address 15 KB)
Fetch instruction at address 132
Execute (no memory reference)
Fetch the instruction at address 135
Execute (store to address 15 KB)

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

15K + 32K
= 47K

Base: 32K
Bound: 16K





Memory Management

- Hardware: Protection
  - Bounds reg
  - Raise an exception when the required address is illegal
  - Know how to do when exceptions are raised
  - E.g.

    Base: 0
    Bound: 4K

    - Then address 4400 is illegal according to the Bound

Memory Management

- Hardware: Efficiency
  - The registers are in CPU chip
  - The part of CPU related to address translation is called: MMU (memory management unit)

Memory Management

- Hardware requirements summary
  - Privileged mode
  - Base/bounds registers
  - Ability to translate virtual addresses and check if within bounds
  - Privileged instruction(s) to update base/bounds
  - Privileged instruction(s) to register exception handlers
  - Ability to raise exceptions

Memory Management

- OS:
  - Maintain a data structure: free list
    - Find place in physical memory for a process when creating it
    - Collect the space when a process terminate
  - Context switch
    - Correctly configure base / bound register
  - Handle exception

OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember addresses of...
	system call handler
	timer handler
	illegal mem-access handler
	illegal instruction handler
start interrupt timer	
	start timer; interrupt after X ms
initialize process table	
initialize free list	

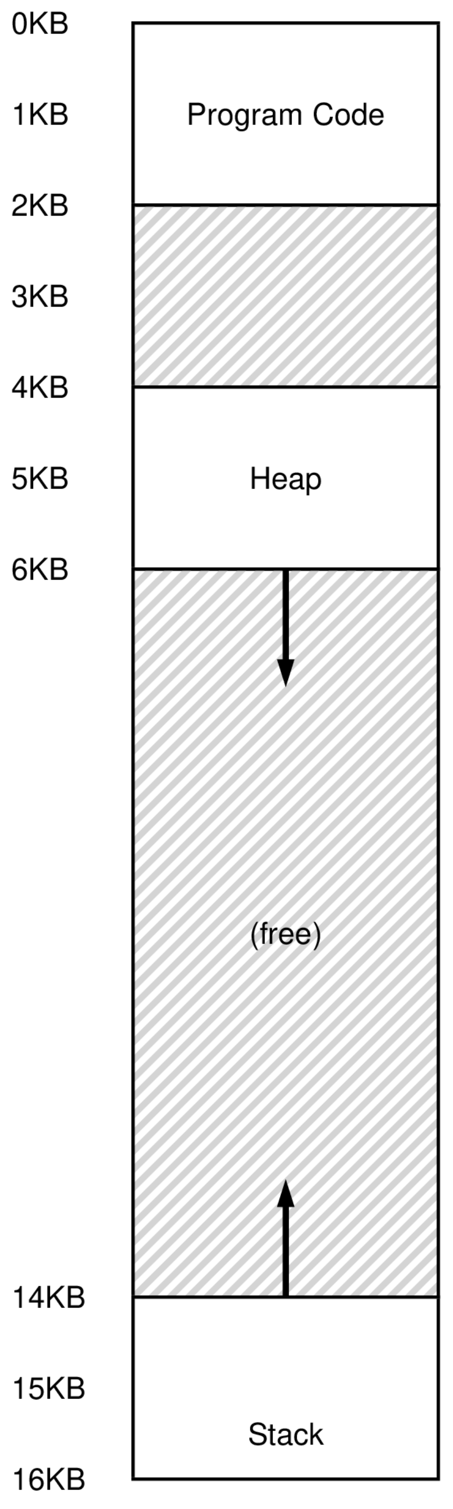
OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
To start process A:		
allocate entry in process table		
allocate memory for process		
set base/bounds registers		
return-from-trap (into A)		
	restore registers of A	
	move to **user mode**	
	jump to A's (initial) PC	
		Process A runs
		Fetch instruction
		...

Memory Management

- Two implementation of virtual memory
  - Segmentation
  - Paging

Segmentation

- The problem of Base and Bound
  - Load entire address space
  - The problem:
    - Wasteful
  - Motivation
    - How to support large address space



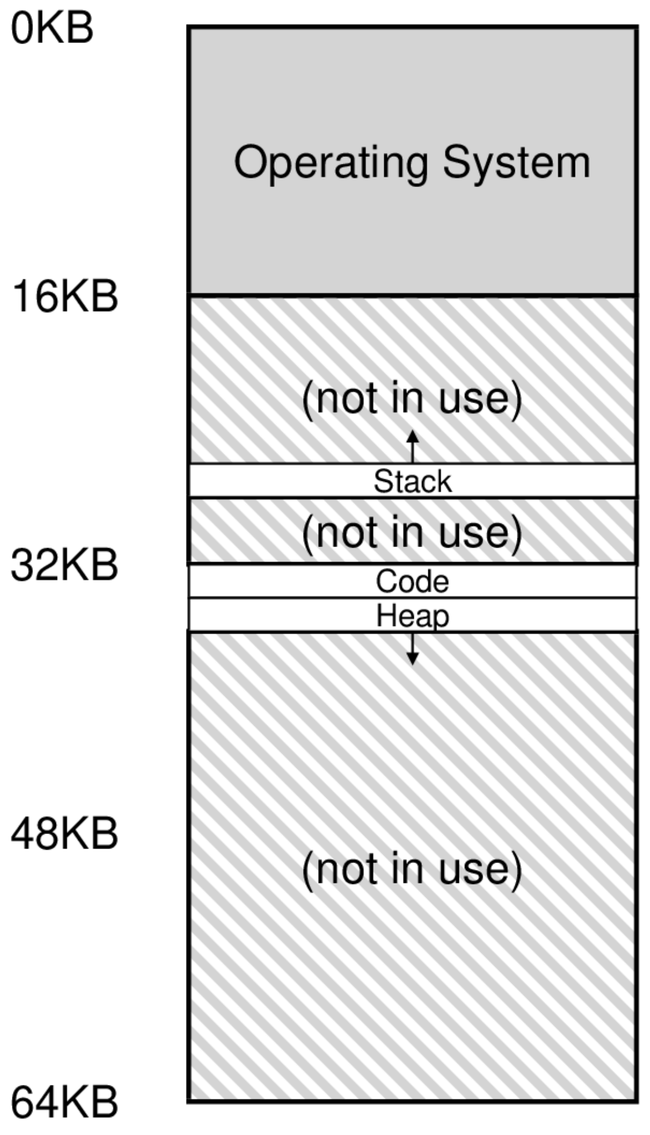
Segmentation

- Solution:
  - Multiple base/bound
  - 3 logical segmentations
    - Code
    - Stack
    - Heap
  - 3 groups of base/bound registers

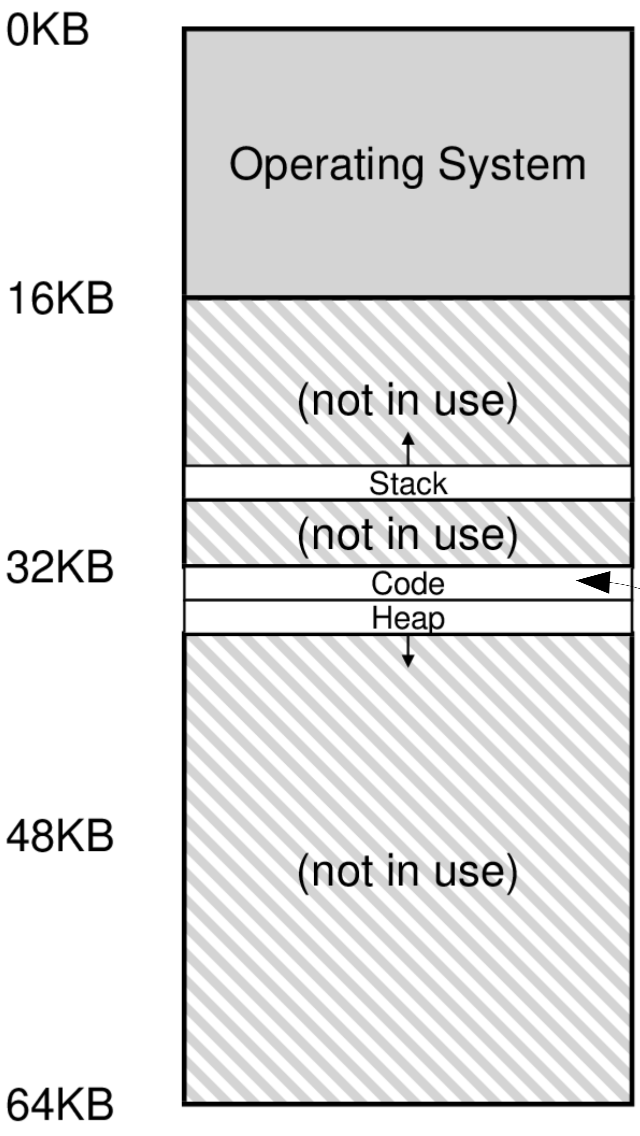
Segmentation

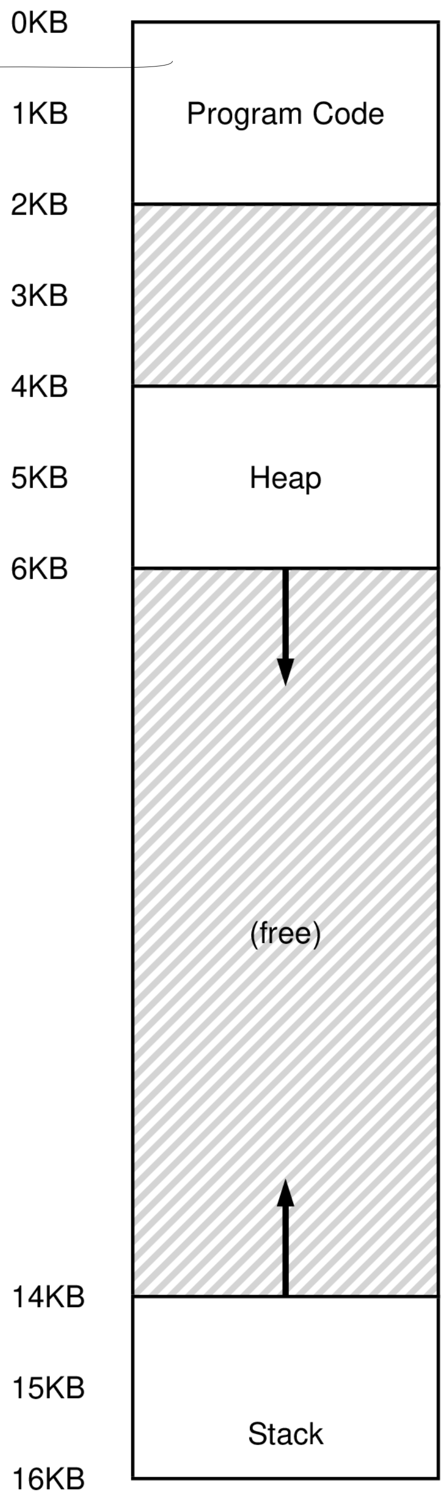
- Multiple base/bound
  - Physical memory

Segmentation	**Base**	**Size**
Code	32K	2K
Heap	34K	2K
Stack	28K	2K



Example: multiple base/bound





Visit virtual memory
100

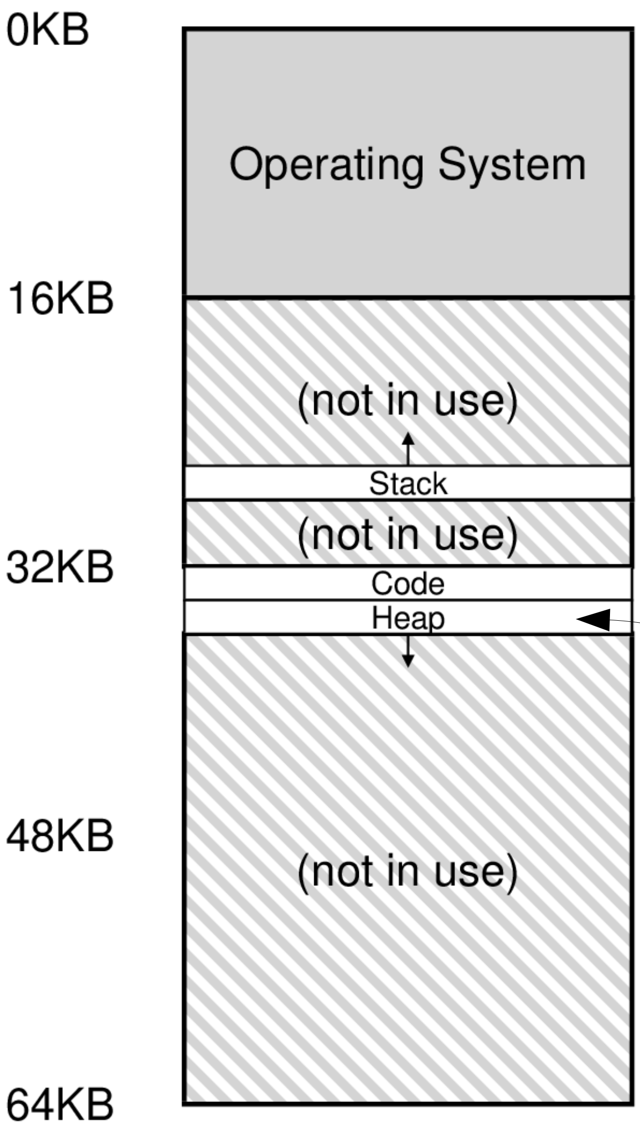
Address translation:
32K+100 = 32868

Address checking:
100 < 2K

Visit physical memory:
32868

Segmentation	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

Example: multiple base/bound



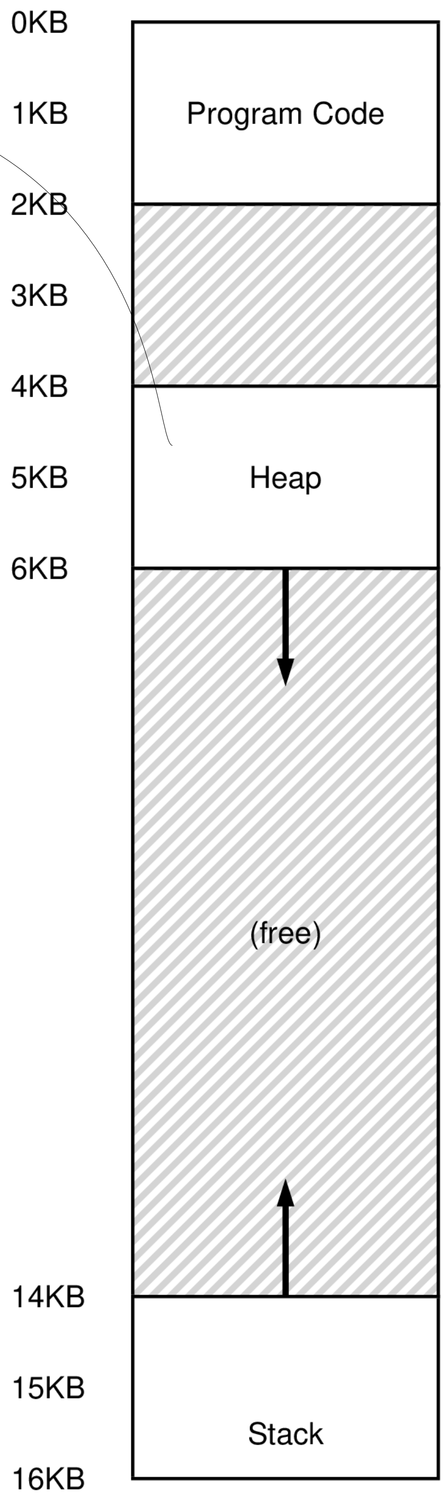
Visit virtual memory 4200

Address translation: **34K**+(4200-4K)=34920

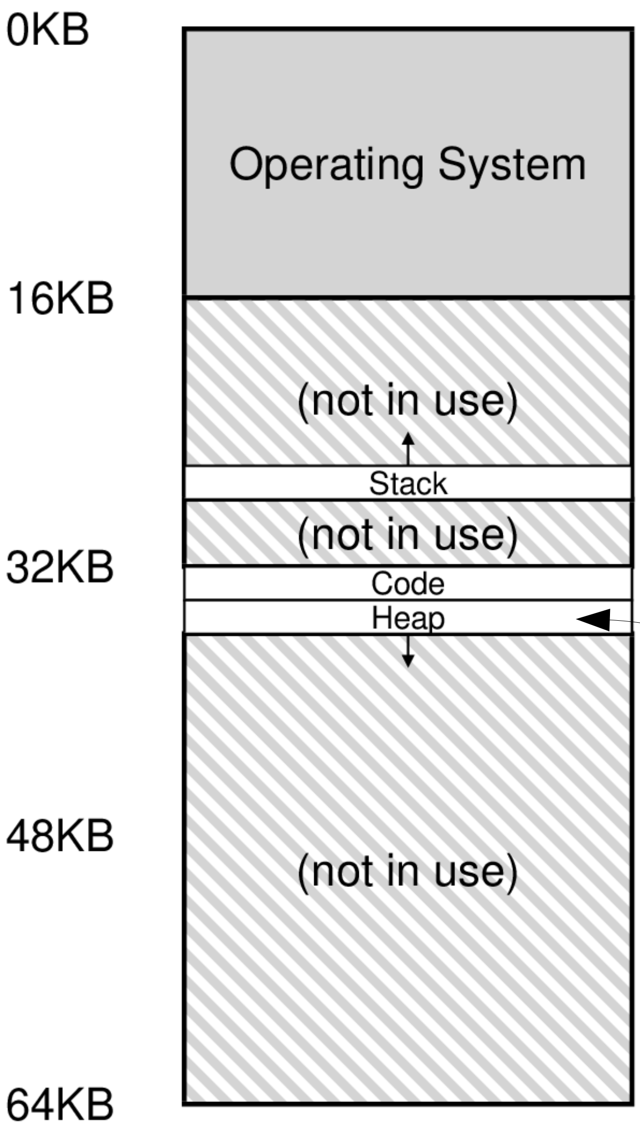
Address checking: 104 < **2K**

Visit physical memory: 34920

Segmentation	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K



Example: multiple base/bound

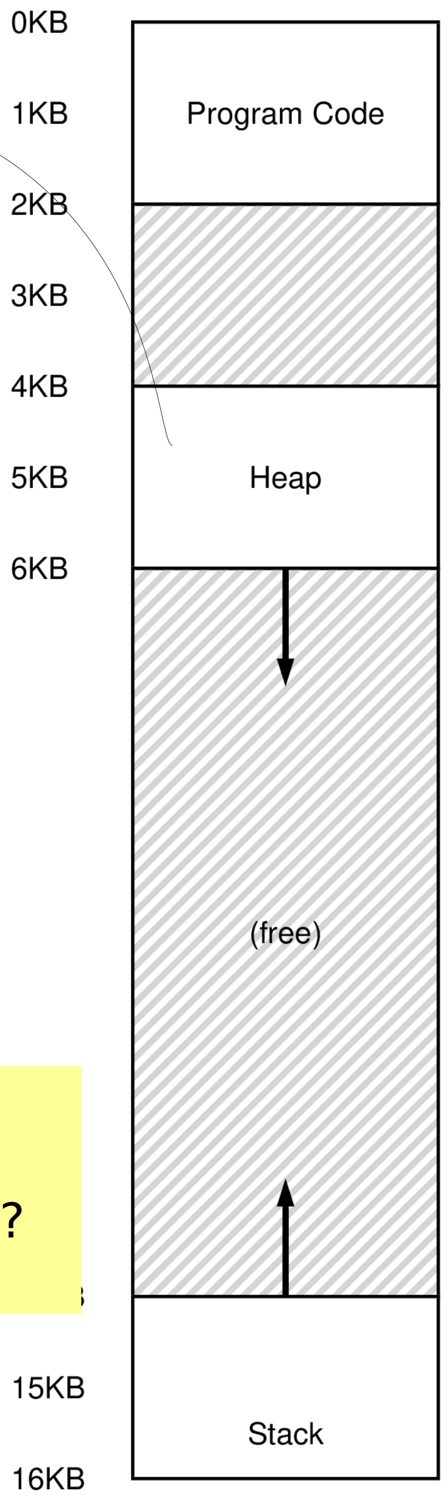


Visit virtual memory
4200

Address translation:
34K+(4200-4K)=34920

Address checking:
104 < **2K**

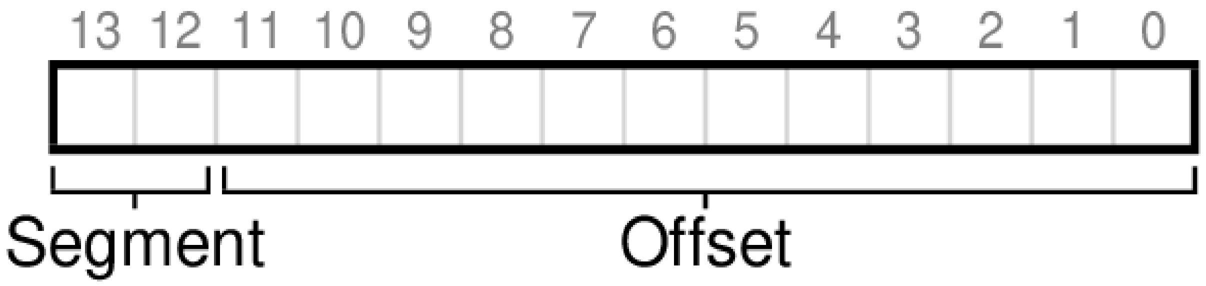
Visit physical memory:
34920



Problem:
How we know 4200 is at heap?
How to interpret an virtual address?

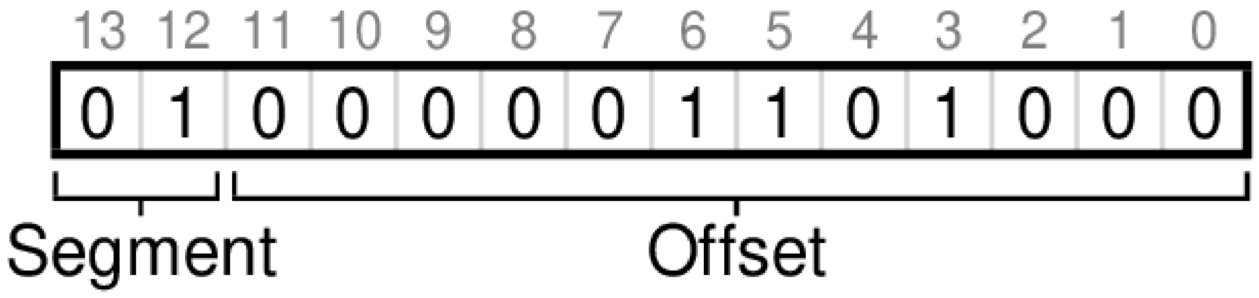
Segmentation

- Which segmentation are we referring to
  - Explicit approach
    - top few bits of the virtual address
  - Example:
    - 16K address space → 14 bit



Segmentation

- Which segmentation are we referring to
  - Example: 4200



Segmentation

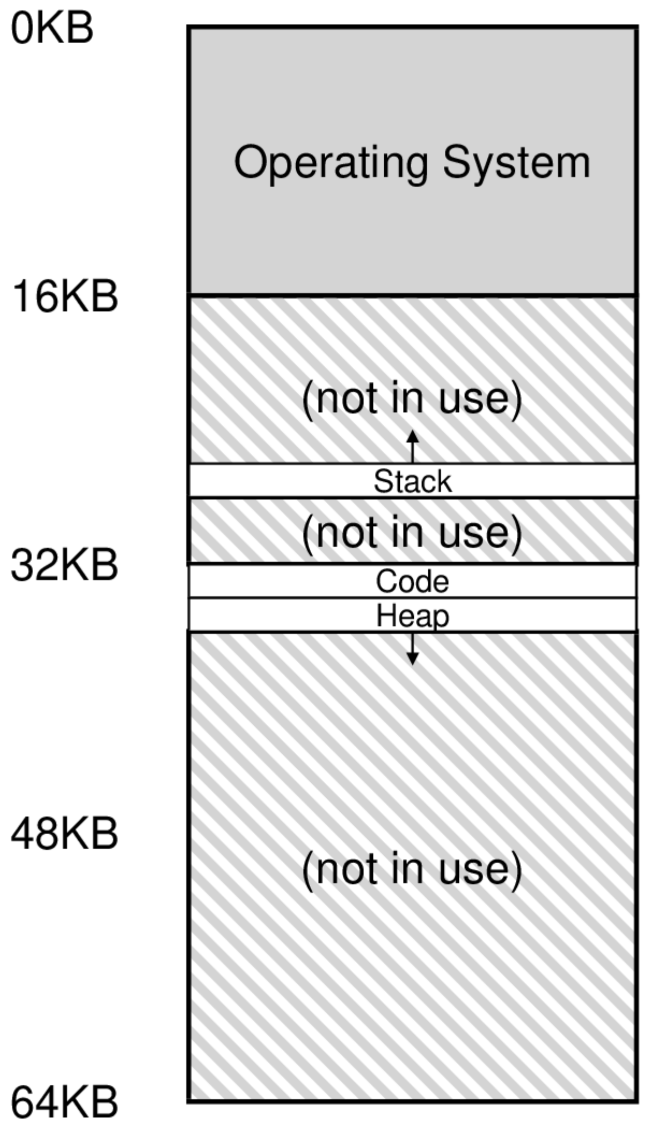
- Which segmentation are we referring to

```
// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT

// now get offset
Offset = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment])
    RaiseException(PROTECTION_FAULT)
else
    PhysAddr = Base[Segment] + Offset

Register = AccessMemory(PhysAddr)
```

Segmentation

- About the stack
  - Difference
    - growth backwards
    - 28K - 26K

Segmentation	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

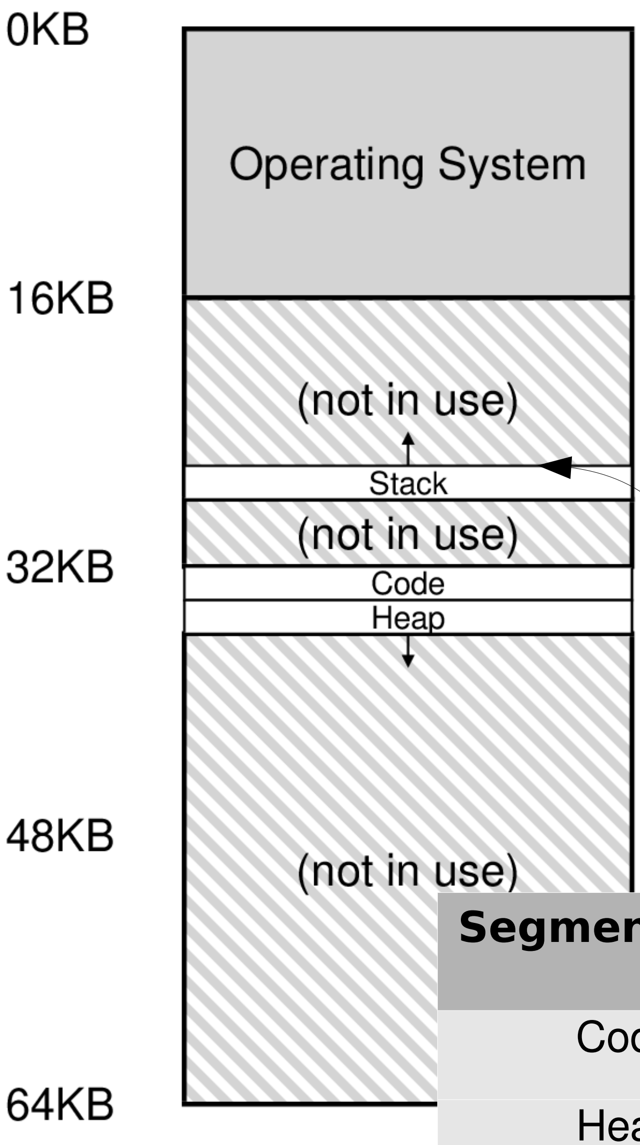


Segmentation

- About the stack
  - Solution: extra hardware support
  - one bit in MMU
    - 1: growth in positive direction
    - 0: growth in negative direction

Segmentati on	**Base**	**Size**	**Grows Postive**
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

Example: multiple base/bound



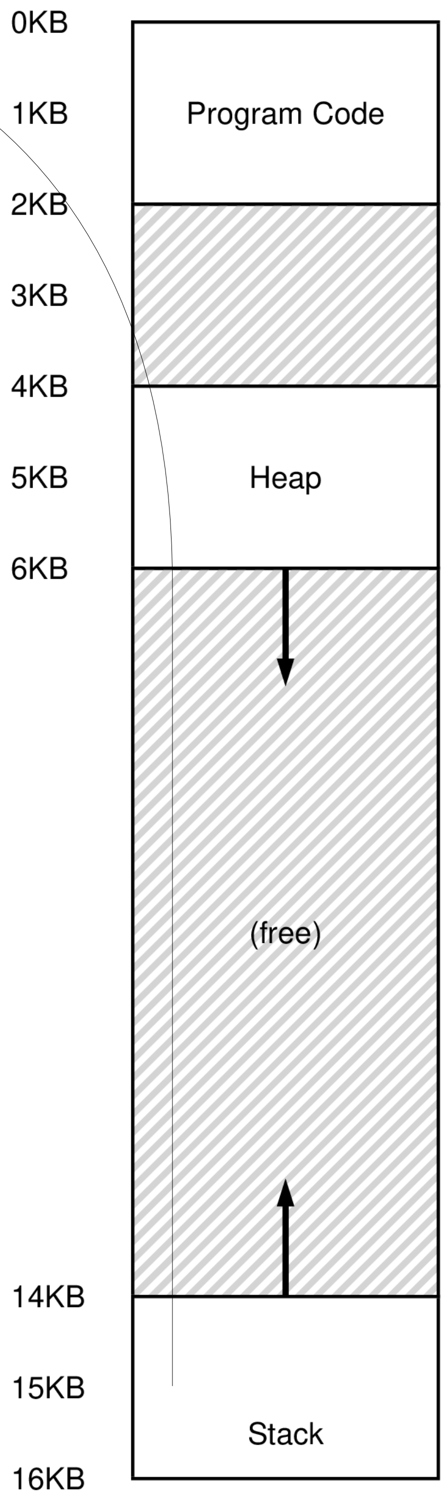
Visit virtual memory
15K 11 11 00 00 00 00 00

Address translation:
[1] segment = 11 → stack reg
[2] offset = 3K
[3] maximum segment = 4K
[4] 3K – 4K = -1K
[5] physical addr:
28K + (-1K)= 27K

Address checking:
|-1K| < 2K

Visit physical memory:
27K

Segmentation	Base	Size	Grows Postive
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0



Segmentation

- Support for Sharing
  - Protection bit

Segmentation	Base	Size	Grows Postive	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

Segmentation

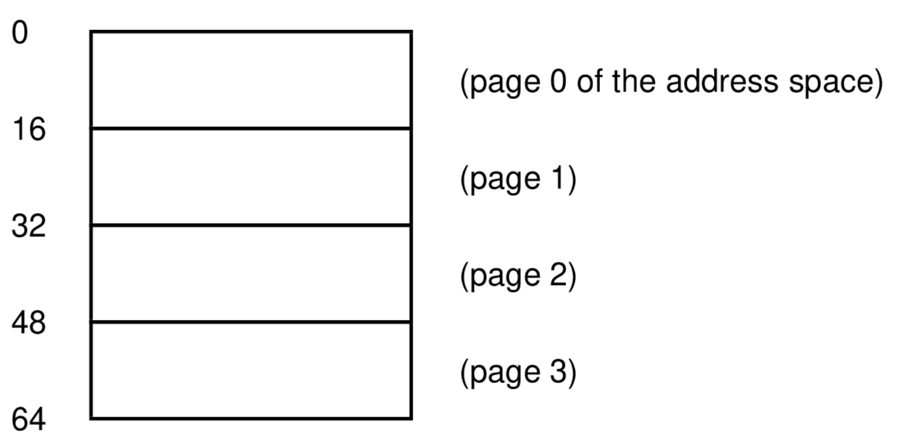
- Summary
  - Base/Bound registers in MMU
  - Multiple Base/Bound
  - Growth direction
  - Protection
- Problem
  - Where to place new address spaces
  - External fragmentation
  - Free memory management

Paging

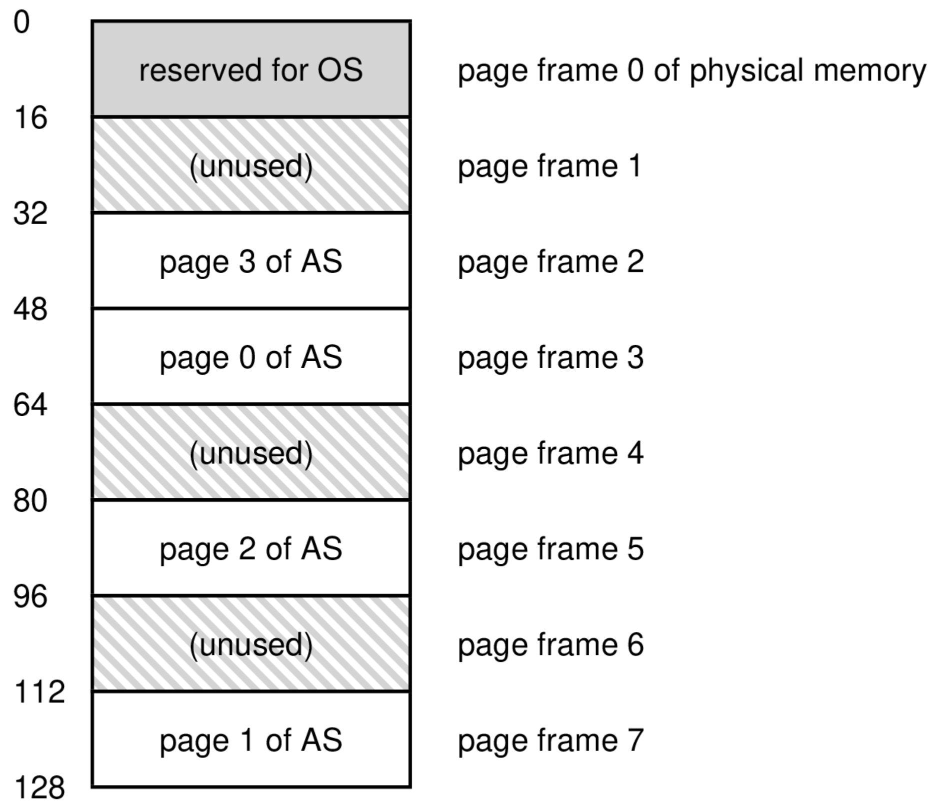
- Segmentation
  - Splitting address space with variable size logical segmentations
- Paging
  - Divide address space into fixed size units (pages)

Paging

- Example:
  - 64 Byte address space
  - 16 Byte page
  - 128 Byte physical memory



Pages of the virtual address space are placed at different locations throughout physical memory

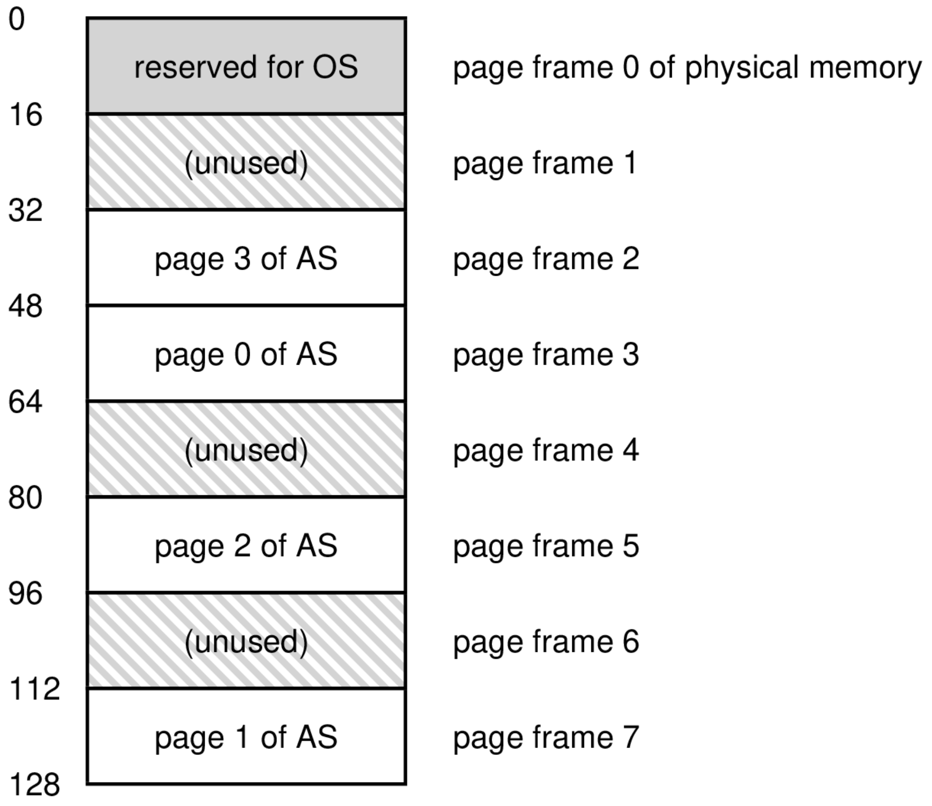


Paging

- Advantages
  - Flexible
    - make no assumptions about the direction the heap/stack grow, how they are used.
  - Simple
    - Simple free memory management
    - A free list of free pages

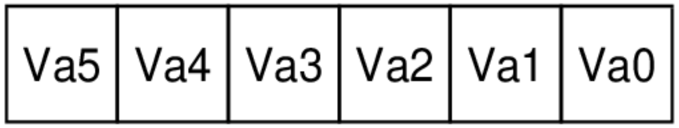
Paging

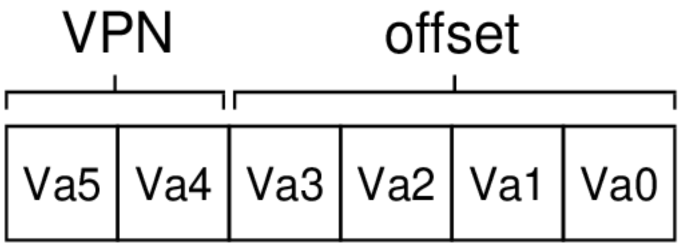
- Virtual page → physical frame
  - **Page Table**
  - A data structure
    - VP0 → PF3
    - VP1 → PF7
    - VP2 → PF5
    - VP3 → PF2
  - In each process



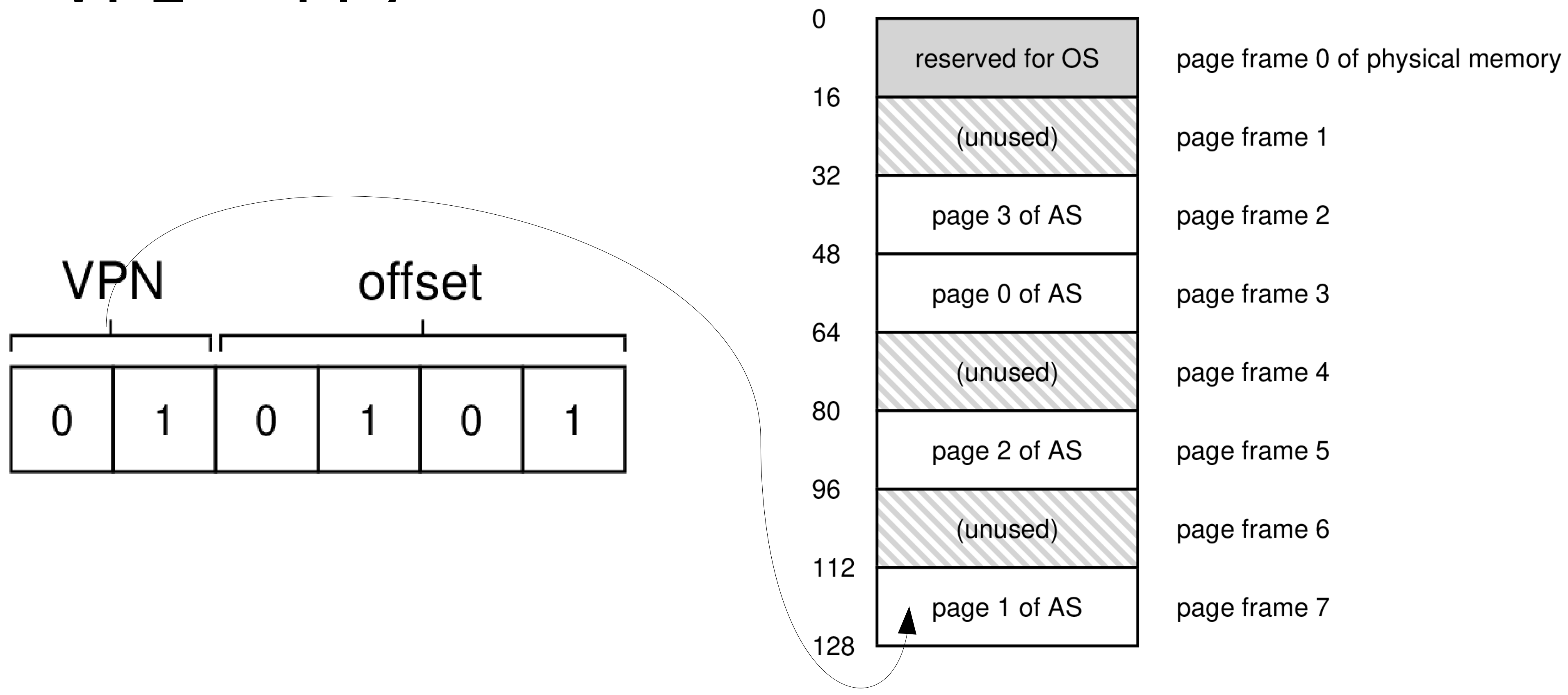
Paging

- Address translation
  - Virtual address:
    - **Virtual Page Num (VPN)**
    - **Offset**
  - Example
    - 64 Byte virtual address
    - 16 Byte page



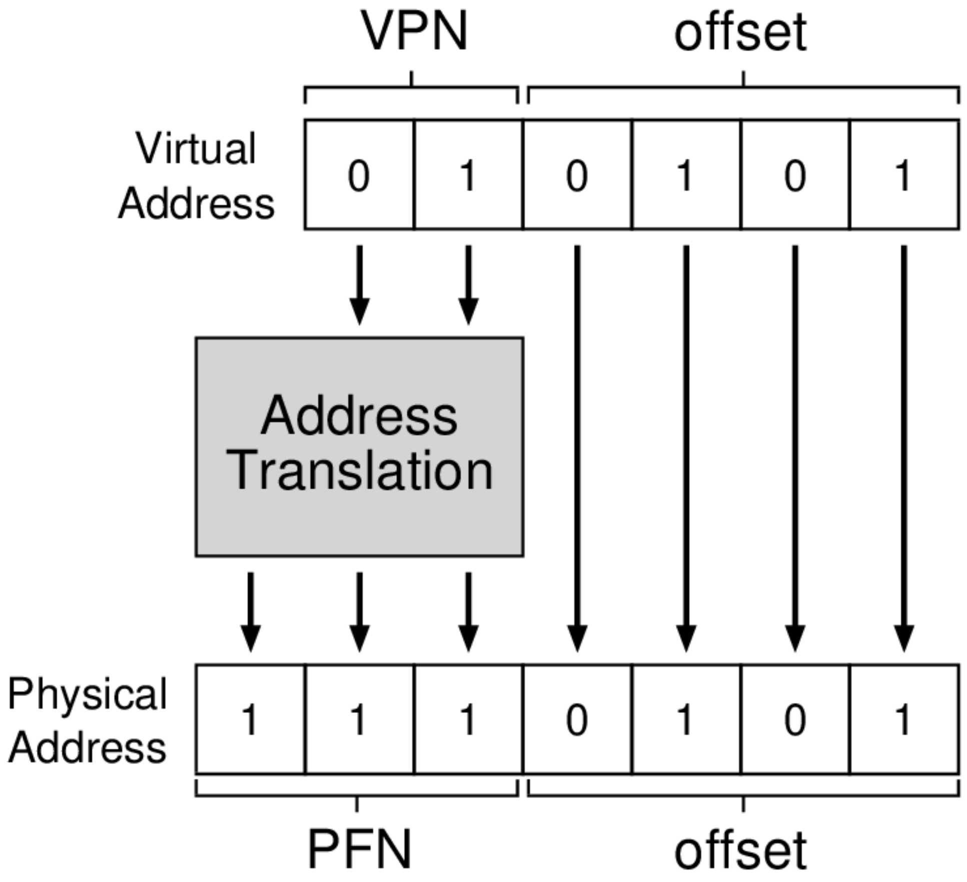


- Address translation
  - movl 21, %eax
  - Binary of 21: 010101
  - 5th byte (0101) of 1st virtual page (01)
- VP1 → FP7



Paging

- Address translation



Paging

- Questions
  - Where are page tables stored?
  - What are the typical contents of the page table?
  - How big are the tables?
  - Does paging make the system (too) slow?

Paging

- How big are the tables?
  - 32bit address space
  - 4K page size
  - 20bit VPN + 12bit offset
  - 2^{20} = 1M

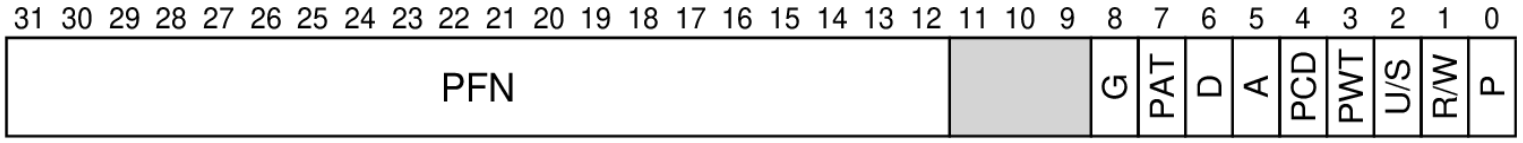
    translations that the OS would manage
  - For each process!
- Page Table Entry (PTE)
  - 4 Byte
- Page table size: $2^{20} * 4 = 4M$
- If we have 100 active processes: 400M
- How about 64bit systems?

Paging

- Where are page tables stored?
  - Not in MMU (so big)
  - In OS's memory
    - Physical memory managed by OS
    - Virtual memory of OS (can be swapped out)

Paging

- What's actually in a page table?
  - Page Table Entry (PTE)
  - An array (linear page table)
  - OS indexes the array with VPN
- PTE
  - PFN
  - Valid bit: whether the VPN is unused
  - Protection bit: read/write/execute
  - Present bit: whether the page on physical memory or on disk (swapped out)
  - Dirty bit: whether the page has been modified since it is brought into memory
  - Reference bit: whether a page has been accessed



Paging

- Too slow

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
```

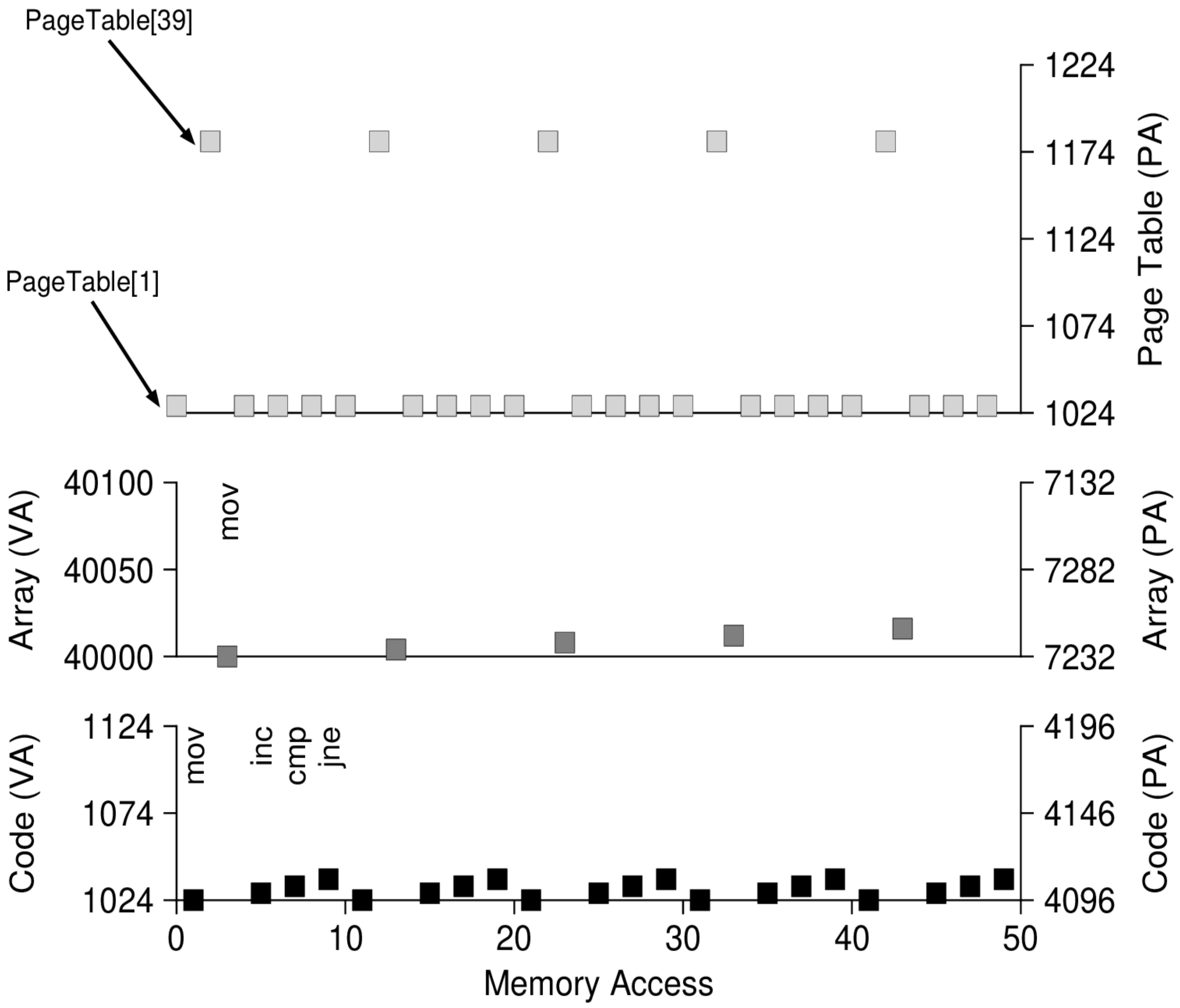
- Example

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

```
0x1024 movl  $0x0, (%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8, %eax
0x1030 jne 0x1024
```

Paging

- Too slow



Paging

- Faster translation
  - With the help of hardware (in MMU)
    - Translation Lookaside Buffer (TLB)
    - Cache
    - Temporal and spatial locality
- Smaller page table
  - Hybrid segmentation and paging
  - Multi-layer page table