CS 423
Operating System Design: Memory Virtualization
02/07

Ram Alagappan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

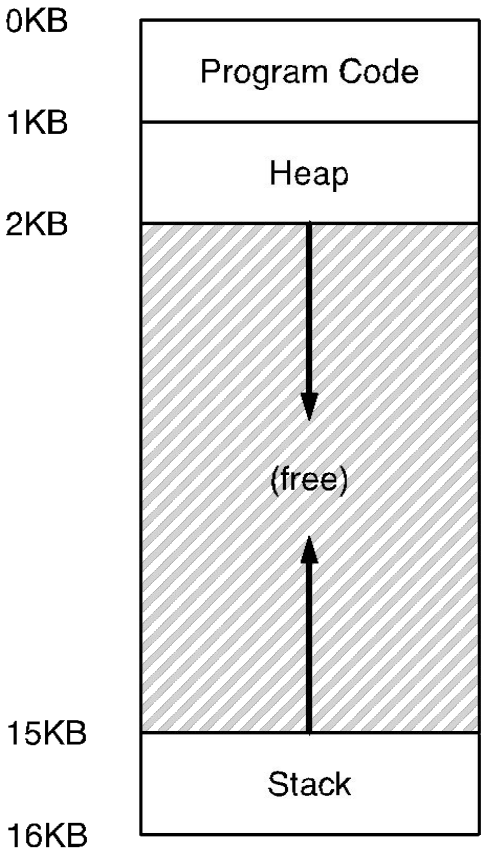
AGENDA / LEARNING OUTCOMES

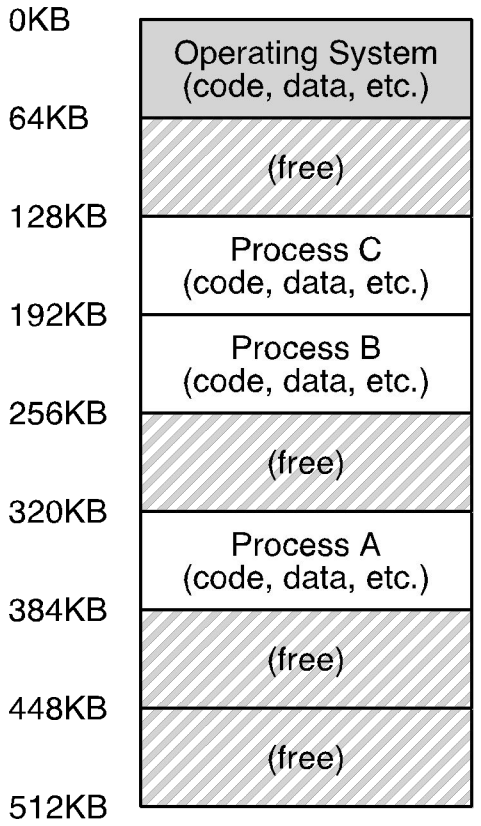
Memory virtualization

- How to virtualize memory?
- What are some techniques used by older systems?
- What are their pros and cons?
- Moving towards more modern systems… and will continue in the next lectures

RECAP

ABSTRACTION: ADDRESS SPACE





MEMORY ACCESS

Initial %rip = 0x10
%rbp = 0x200

```
0x10:   movl    0x8(%rbp), %edi
0x13:   addl    $0x3, %edi
0x19:   movl    %edi, 0x8(%rbp)
```

%rbp is the base pointer:
points to base of current stack frame

%rip is instruction pointer (or program counter)

Fetch instruction at addr 0x10
Exec:
load from addr 0x208

Fetch instruction at addr 0x13
Exec:
no memory access

Fetch instruction at addr 0x19
Exec:
store to addr 0x208

HOW TO VIRTUALIZE MEMORY

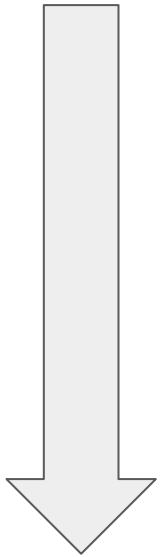
Problem: Addresses are “hardcoded” into process binaries

How to avoid collisions?

End of Recap

Mechanisms for Virtualization

1. Time Sharing
2. Static Relocation
3. Base
4. Base+Bounds
5. Segmentation

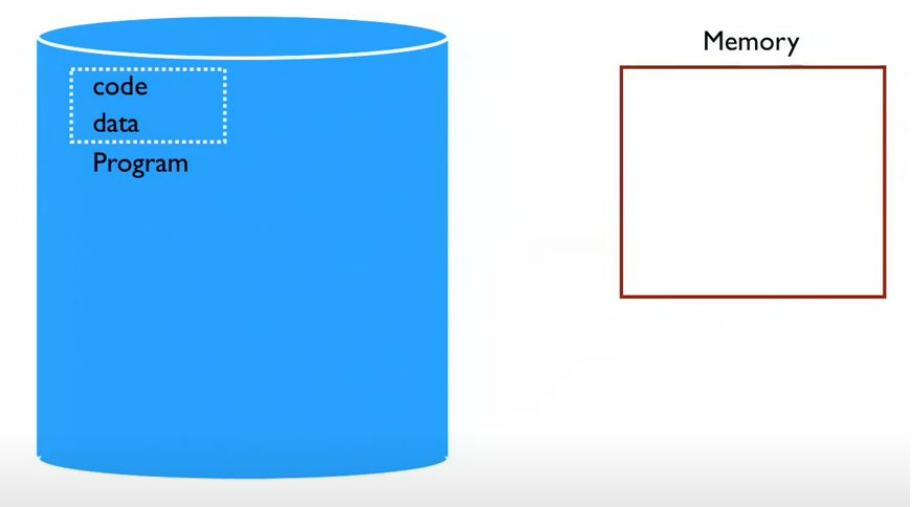


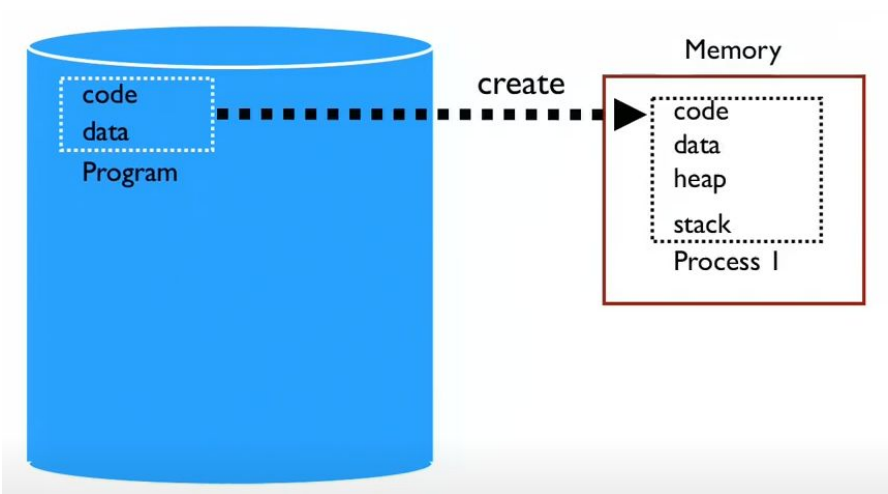
Limited practicality, has many problems

More practical, still has some problems

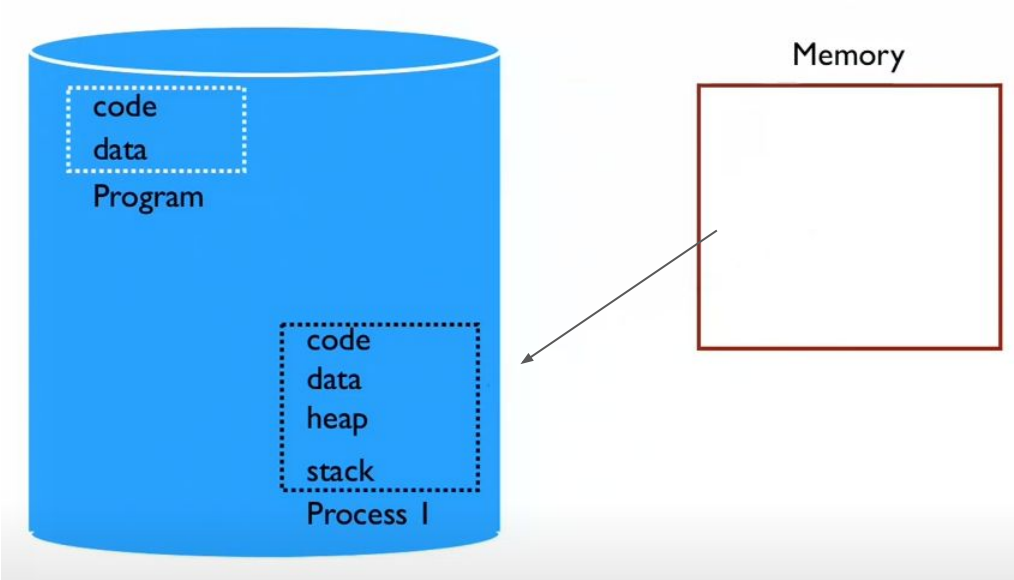
x86 and Linux Today: Paging, TLB, Efficient Page Tables

1. TIME SHARE MEMORY





Time Sharing Memory



PROBLEMS WITH TIME SHARING?

Ridiculously poor performance: must save memory snapshot to disk!

Better Alternative: space sharing!

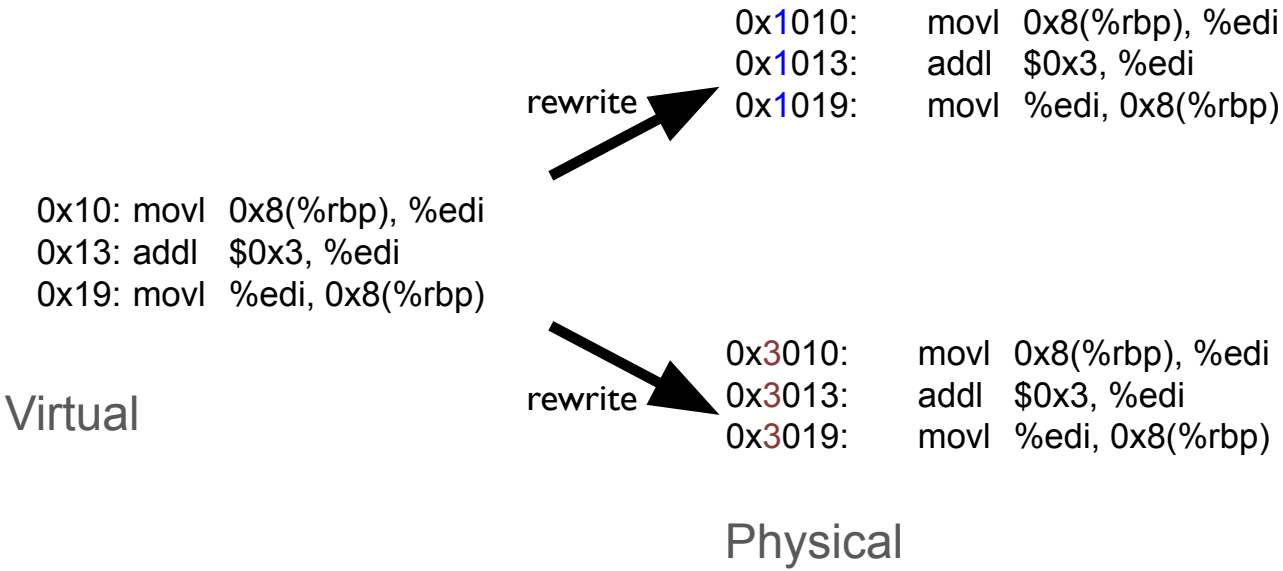
- At same time, memory space is divided across processes
- Remainder of solutions all use space sharing

2) STATIC RELOCATION

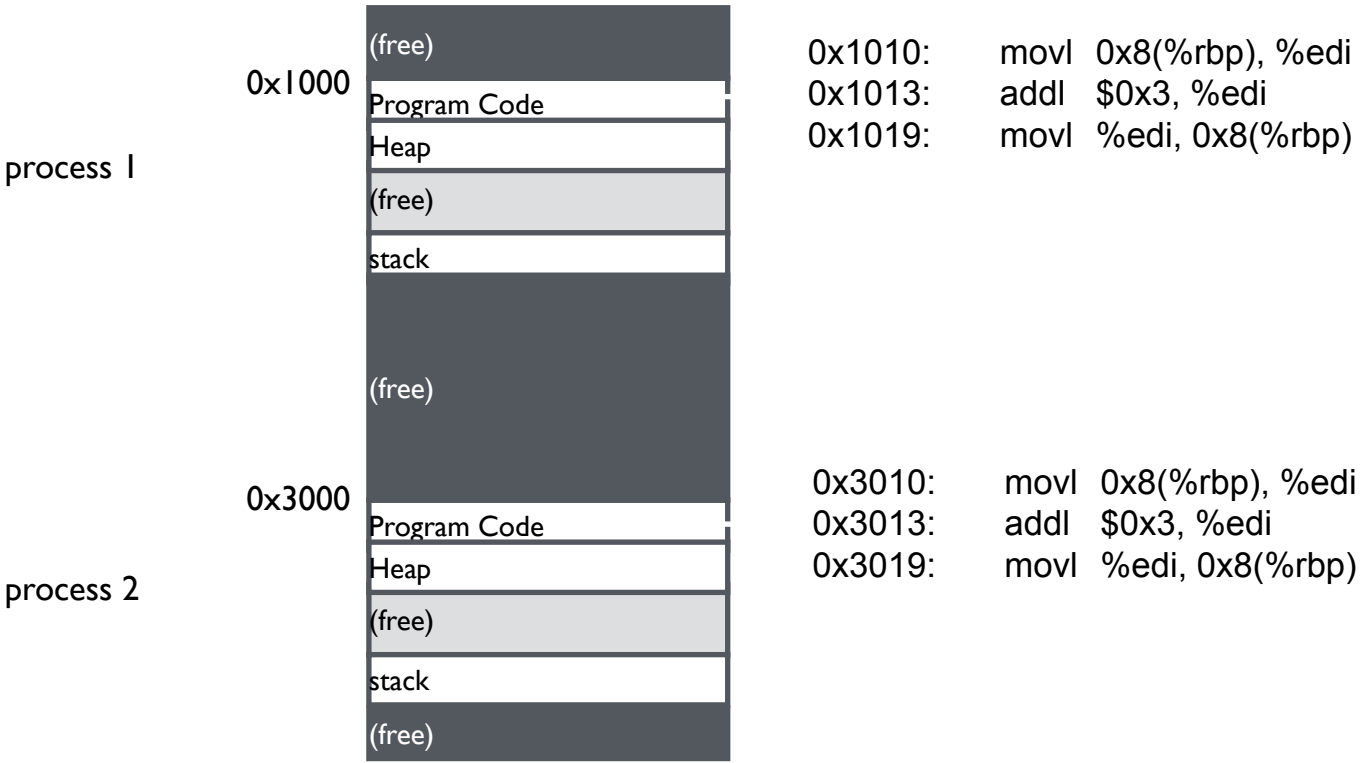
Idea: rewrite each program before loading it as a process in memory

Each rewrite for different process uses different addresses and pointers, change jumps, loads of static data

Done by OS or relocation loader

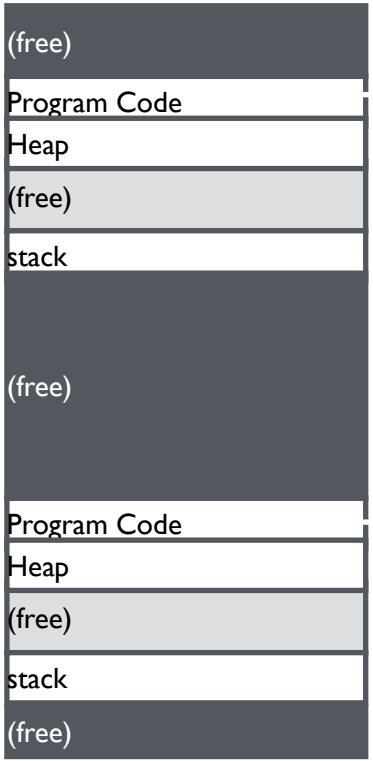


Static: Layout in Memory



Static Relocation: Disadvantages

Chat for a minute with neighbors…



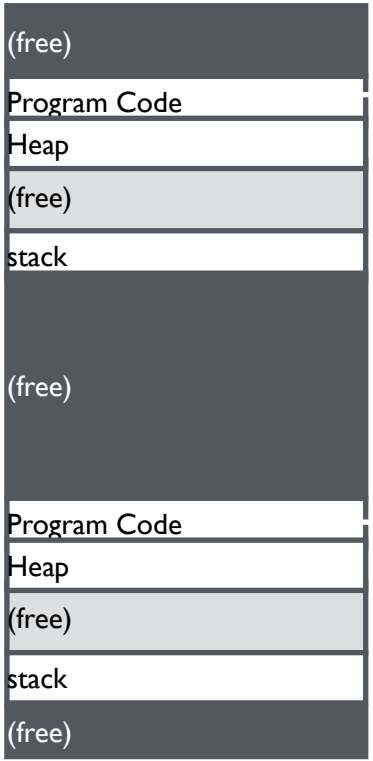
Static Relocation: Disadvantages

Cannot move address space after it has been placed

No protection

- Process can destroy OS or other processes (corrupt)
- No privacy (can read other process's memory)

Example: movl 0x2100, %eax → movl 0x3100, %eax



3) Dynamic Relocation

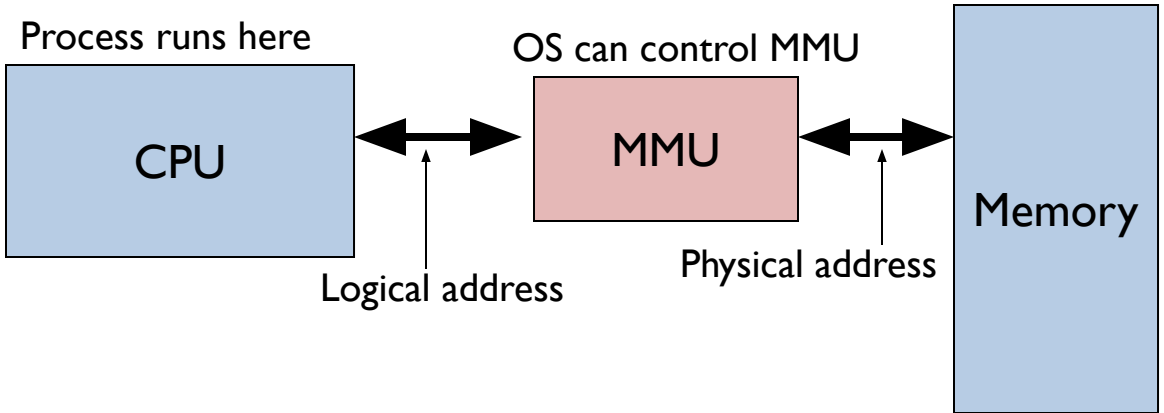
Goal: Dynamically relocate to different places, and protect processes from one another

Requires hardware support (so far OS was trying to virtualize alone!)

- Memory Management Unit (MMU)

MMU dynamically changes process address at every memory reference

- Process generates logical or virtual addresses (in their address space)
- Memory hardware uses physical or real addresses



HW Support for Dynamic Relocation

Privileged (protected, kernel) mode: OS runs

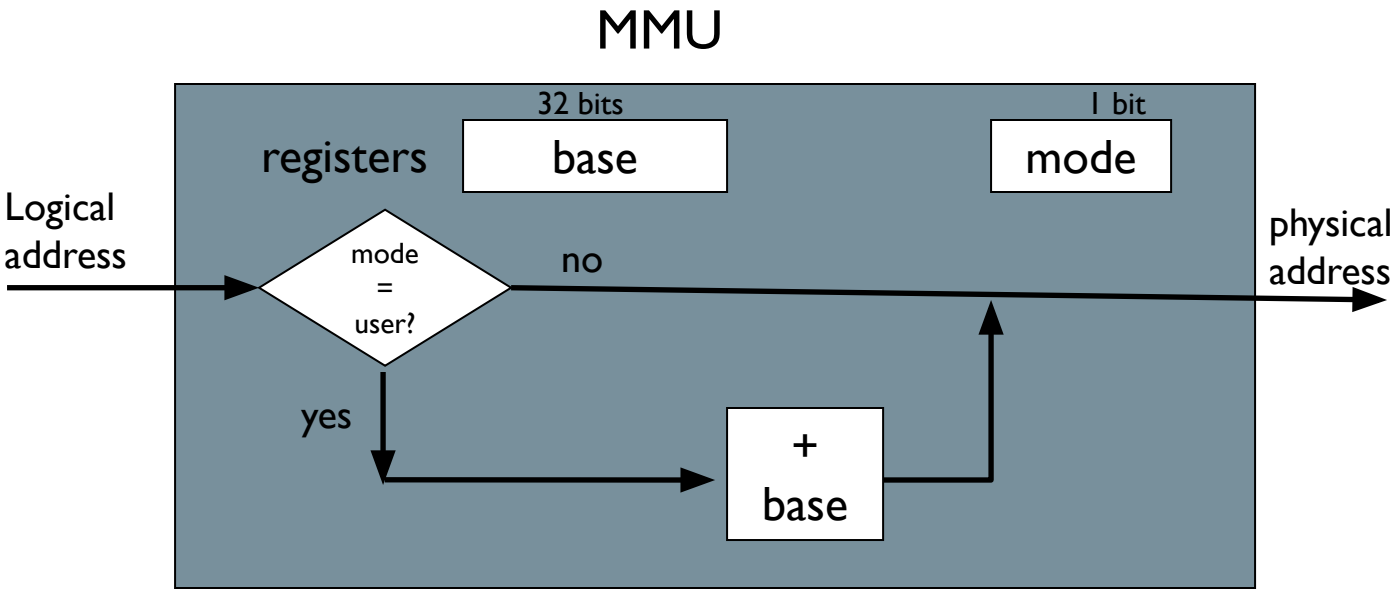
- When enter OS (trap, system calls, interrupts, exceptions)
- Allows certain instructions to be executed
  (Can manipulate contents of MMU)
- Allows OS to access all of physical memory

User mode: User processes run

- Perform translation of logical address to physical address

Implementation of Dynamic Relocation: BASE REG

Translation on every memory access of user process
MMU adds base register to logical address to form physical address



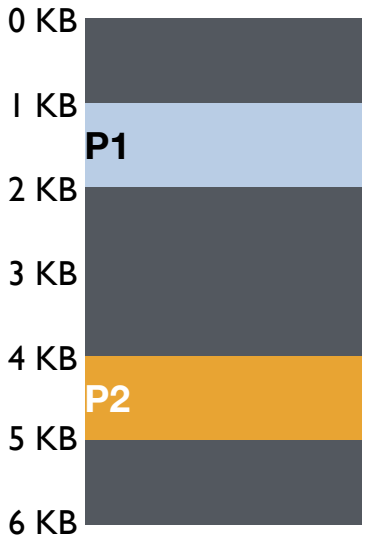
DYNAMIC RELOCATION WITH BASE REGISTER

Translate virtual addresses to physical by adding a fixed offset each time.

- Store offset in base register

Each process has different value in base register

- Dynamic relocation by changing value of base register



Virtual

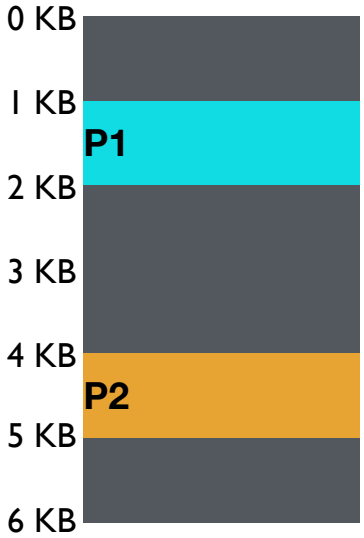
P1: load 100, R1

P2: load 100, R1

P2: load 1000, R1

P1: load 1000, R1

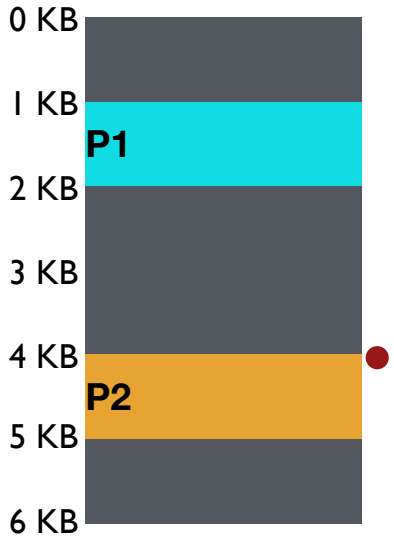
VISUAL Example of DYNAMIC RELOCATION: BASE REGISTER



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5096, R1
P1: load 1000, R1	load 2024, R1

Can P1 hurt P2?

How well does dynamic relocation do with base register for protection?



Virtual	Physical	
P1: load 100, R1	load 1124, R1	
P2: load 100, R1	load 4196, R1	
P2: load 1000, R1	load 5096, R1	
P1: load 100, R1	load 2024, R1	
P1: store 3072, R1	store 4096, R1	(3072 + 1024)

How well does dynamic relocation do with base register for protection?

4) DYNAMIC WITH BASE+BOUNDS

Idea: limit the address space with a bounds register

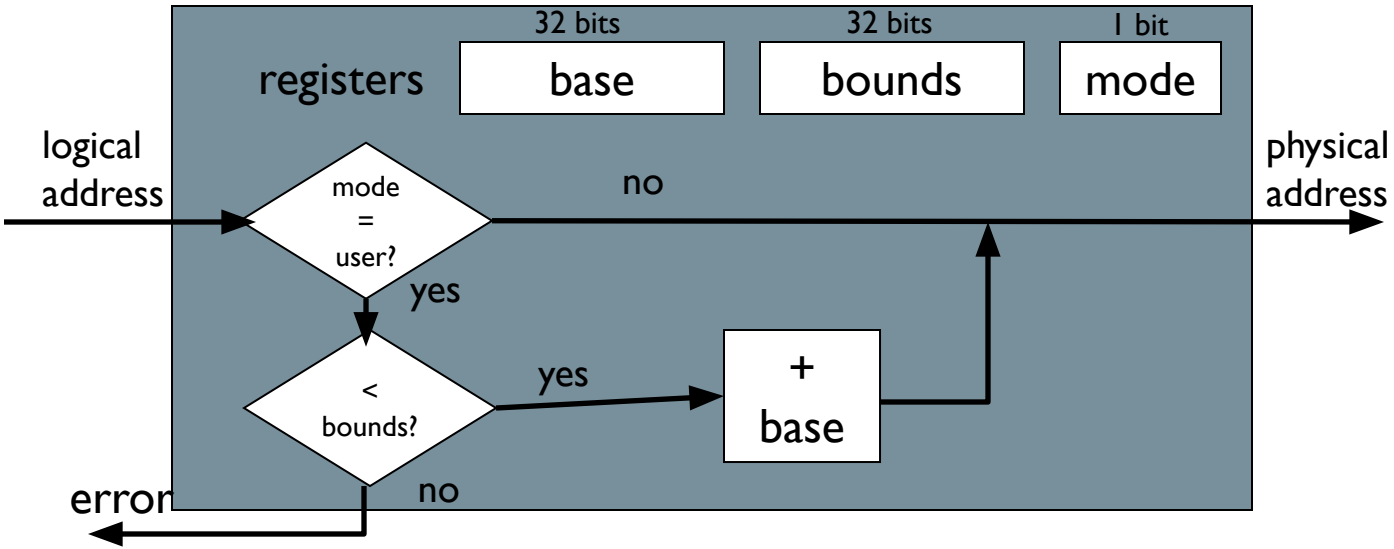
Base register: smallest physical addr (or starting location)
Bounds register: size of this process's virtual address space

OS kills process if process loads/stores beyond bounds

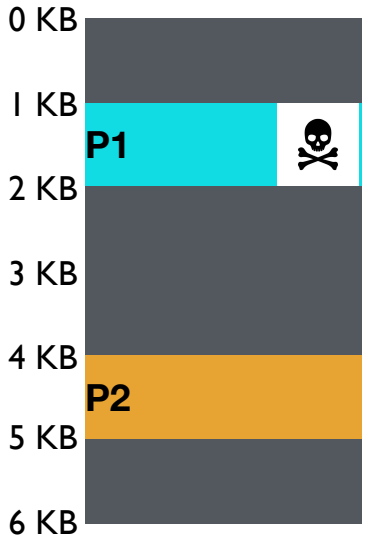
Implementation of BASE+BOUNDS

Translation on every memory access of user process

- MMU compares logical address to bounds register
  if logical address is greater, then generate error
- MMU adds base register to logical address to form physical address



Can P1 hurt P2?
Not in the Base and Bounds approach



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	Interrupt OS!

Managing Processes with Base and Bounds

Context-switch: Add base and bounds registers to PCB
Steps

- Change to privileged mode
- Save base and bounds registers of old process
- Load base and bounds registers of new process
- Change to user mode and jump to new process

Protection requirement

- User process cannot change base and bounds registers
- User process cannot change to privileged mode

Base and Bounds Advantages

Provides protection (both read and write) across address spaces
Supports dynamic relocation
    Can place process at different locations initially and also move address spaces

Simple, inexpensive implementation: Few registers, little logic in MMU
Fast: Add and compare in parallel

Base and Bounds DISADVANTAGES

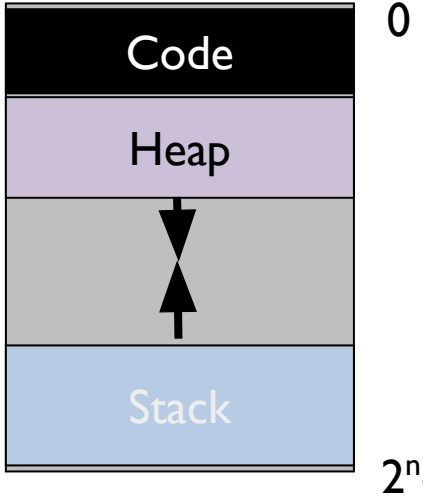
Chat with neighbors for 1 minute…

Base and Bounds DISADVANTAGES

Disadvantages

- Entire process must be allocated contiguously in physical memory
  Must allocate memory that may not be used by process

- No partial sharing: Cannot share parts of address space
- For example: two processes running the same code



Chat with Neighbors for 3 mins

1. Does the base register contain physical or virtual address?
2. Who converts VA to PA based on base register? Process? OS? HW?
3. Can the OS do the address translation? What is the downside?
4. Who modifies contents of base register?
5. What happens on a context switch to the base register?
6. Are there cases where the base register is not changed on switch?

5) Segmentation

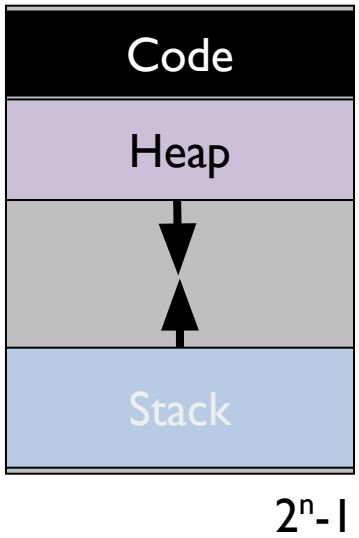
Divide address space into logical segments

- Each segment corresponds to logical entity in address space (code, stack, heap)

Each segment has separate base + bounds register

Each segment can independently:

1. Be placed in physical memory
2. Grow and shrink
3. Be protected (read/write/exec)



Segmented Addressing

Process now specifies segment and offset within segment

How does process designate a particular segment?

- Use part of logical address
  - Top bits of logical address select segment
  - Low bits of logical address select offset within segment

What if small address space, not enough bits?

Special registers

Segmentation Implementation

MMU contains Segment Table (per process)

- Each segment has own base and bounds, protection bits
- Example: 14 bit logical address, 4 segments;

How many bits for segment?

How many bits for offset?

Segment	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xfff	1 1
3	0x0000	0x000	0 0

Example Translations

Segment	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xfff	1 1
3	0x0000	0x000	0 0

Remember:
1 hex digit □ 4 bits

Translate logical (in hex) to physical

0x1108:
Segment number: 1
Physical addr: 0x0000 + 108 = 0x0108

0x3002: ?

Virtual	Physical
load 0x1010, R1	0x400 + 0x010 = 0x410
load 0x1100, R1	0x400 + 0x100 = 0x500

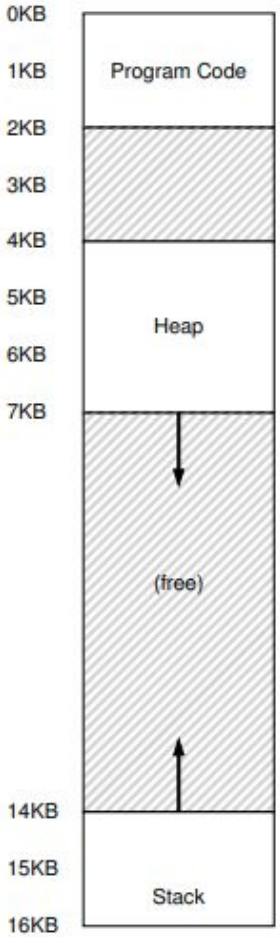
Segment numbers:

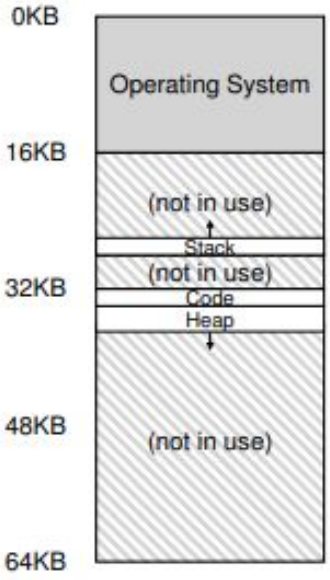
0: code+data

1: heap

2: stack

HOW DO STACKS GROW ?





Stack goes 16K □ 14K,
In physical memory it is 28K □ 26K

Virtual address 15KB (hex: 0x3C00)
top 2 bits (0x3) segment ref,
offset is 0xC00 = 3K
How do we make CPU translate that ?

Negative offset = subtract max segment from offset
= 3K – 4K = -1K
Add to base = 28K – 1K = 27K

HW needs to track one more bit of info for every segment (grows positively?)

Segmented Architectures

X86:

- Stack Segment (SS): Pointer to the stack
- Code Segment (CS): Pointer to the code
- Data Segment (DS): Pointer to the data
- Extra Segment (ES): Pointer to the extra data

Some earlier architectures:

- Supported a segment table with 1000s of segments
- Segment table in memory and hardware can lookup for translations

Advantages of Segmentation

Enables sparse allocation of address space

Stack and heap can grow dynamically

- Heap: If no data on free list, dynamic memory allocator requests more from OS (e.g., UNIX: malloc lib calls sbrk())
- Stack: OS recognizes reference outside legal segment, extends stack implicitly

Different protection for different segments

- Enables sharing of selected segments (two process with same code)
- Example: no write permissions for code segment

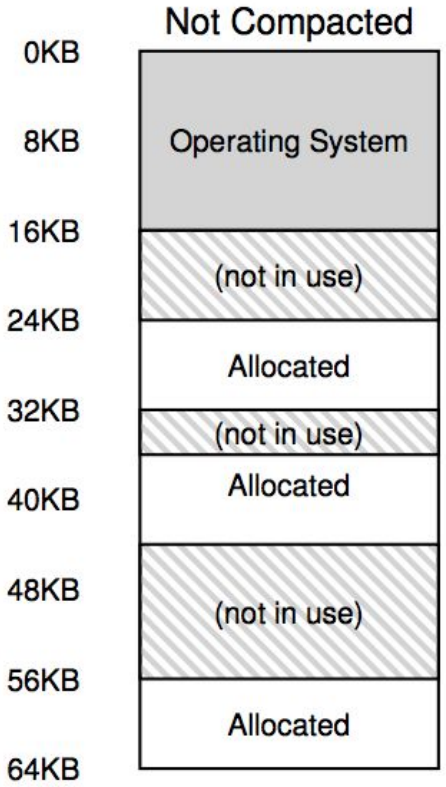
Supports dynamic relocation of each segment

Disadvantages of Segmentation

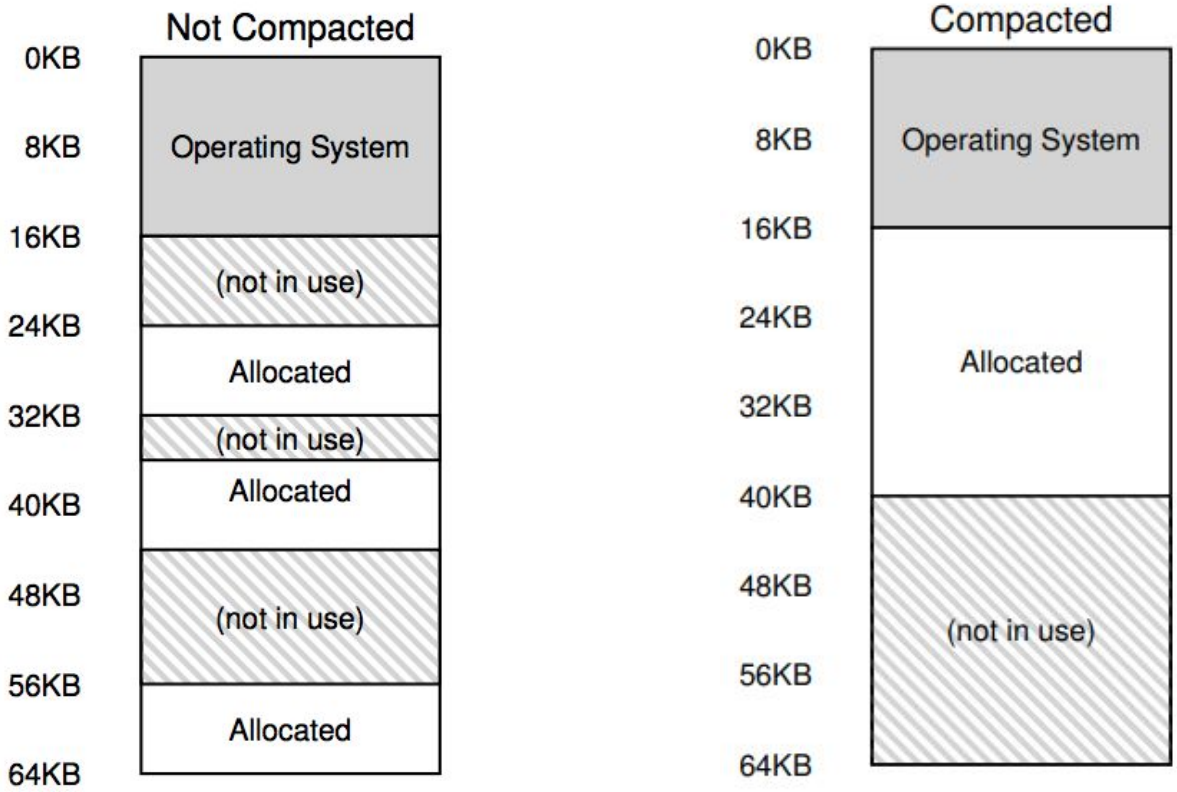
Each segment must be allocated contiguously

May not have sufficient physical memory for large segments?

External Fragmentation

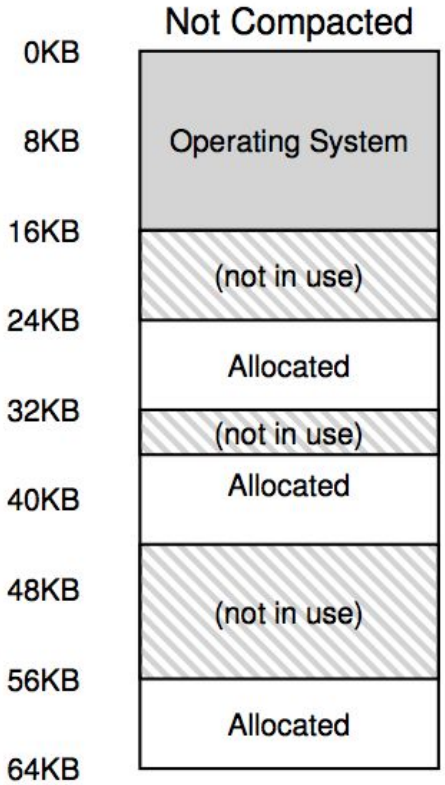


Compacting


Not Compacted
0KB
8KB
16KB
24KB
32KB
40KB
48KB
56KB
64KB
Operating System
(not in use)
Allocated
(not in use)
Allocated
(not in use)
Allocated
Compacted
0KB
8KB
16KB
24KB
32KB
40KB
48KB
56KB
64KB
Operating System
Allocated
(not in use)

PAGING
(more modern systems including Linux use this)

FRAGMENTATION

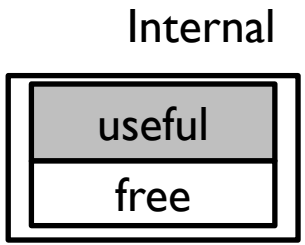


Definition: Free memory that can't be usefully allocated

Types of fragmentation

External: Visible to allocator (e.g., OS)

Internal: Visible to requester

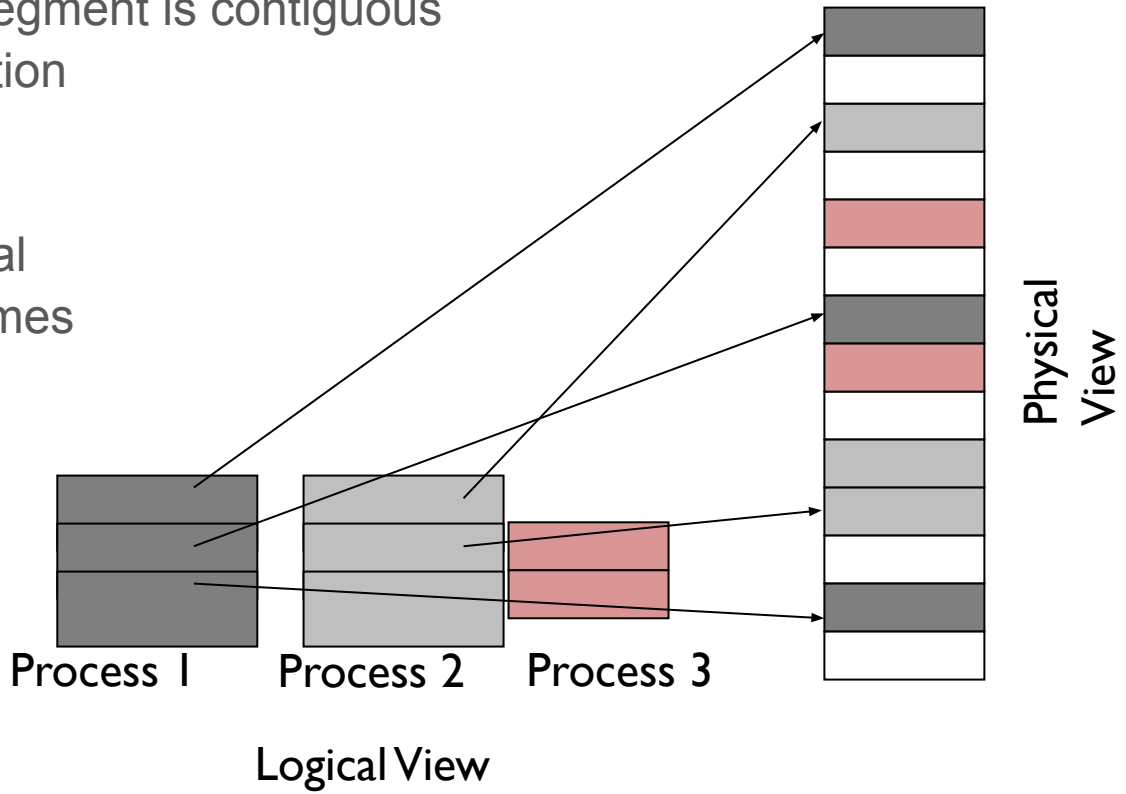


Paging

Goal: Eliminate requirement that segment is contiguous
Eliminate external fragmentation

Idea:
Divide address spaces and physical memory into fixed-sized pages/frames

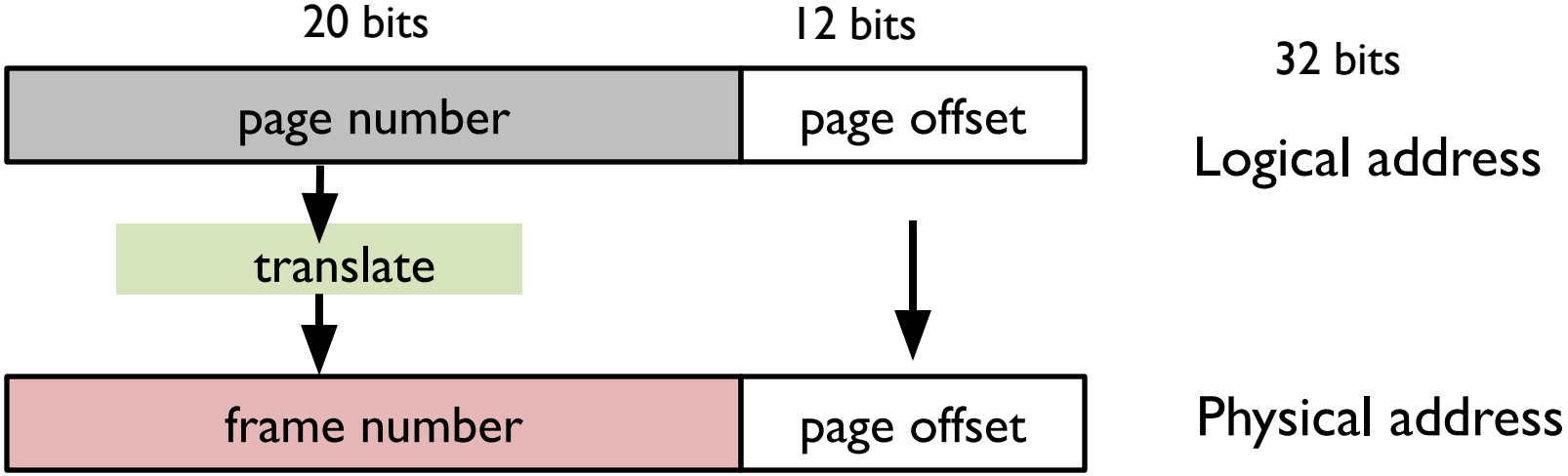
Size: 2^n, Example: 4KB



Translation of Page Addresses

How to translate logical address to physical address?

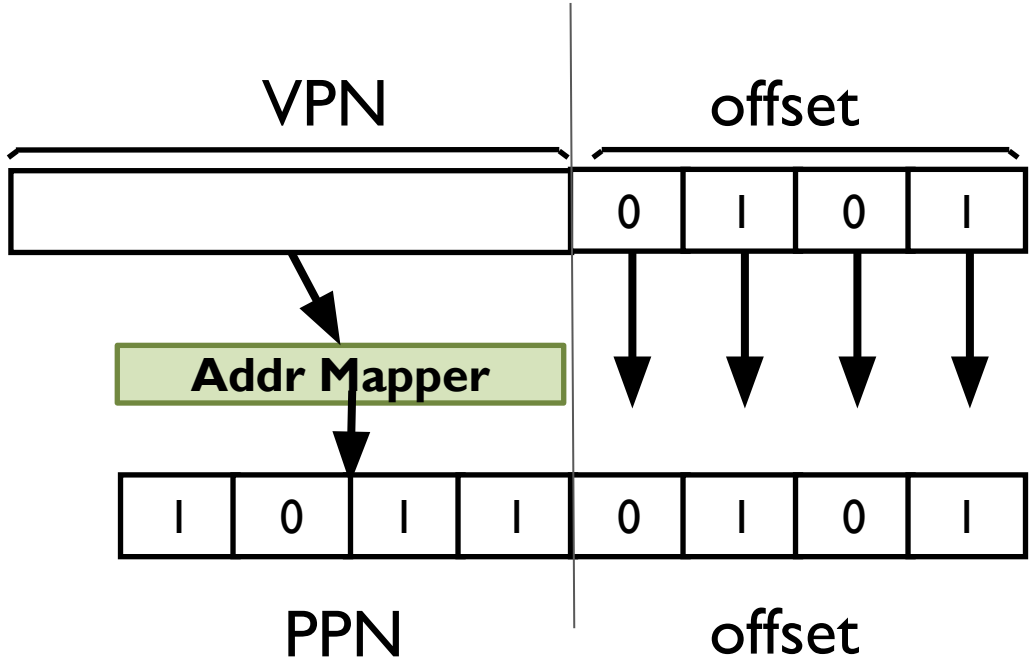
- High-order bits of address designate page number
- Low-order bits of address designate offset within page



ADDRESS FORMAT

Page Size	Low Bits (offset)	Virt Addr Bits	High Bits (vpn)	Virt Pages
16 bytes		10		
1 KB		20		
1 MB		32		
512 bytes		16		
4 KB		32		

VIRTUAL □ PHYSICAL PAGE MAPPING



Number of bits in virtual address

need not equal

number of bits in physical address

bits in VPN

> # bits in PPN?

What does this mean?

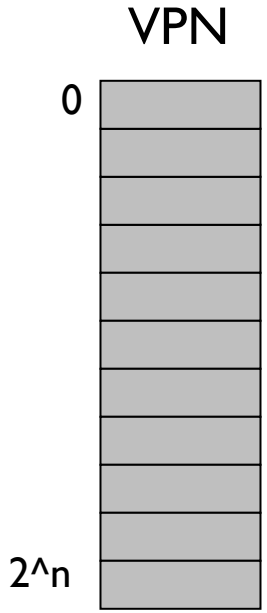
Is inverse possible?

How should OS translate VPN to PPN/PFN?

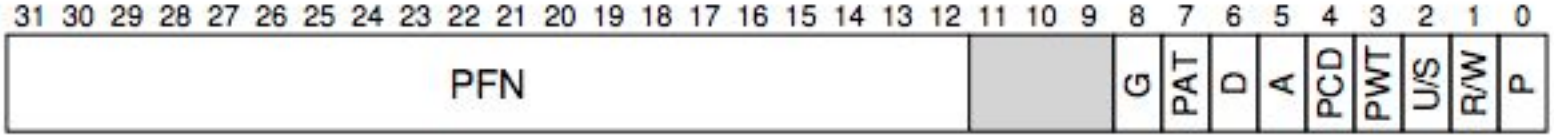
Linear PageTable

What is a good data structure ?

Simple solution: Linear page table aka *array*



A Single PTE:



Advantages of Paging

No external fragmentation
- Any page can be placed in any frame in physical memory

Fast to allocate and free
- Alloc: No searching for suitable free space
- Free: Doesn’t have to coalesce with adjacent free space

Simple to swap-out portions of memory to disk (later lecture)
- Page size matches disk block size
- Can run process when some pages are on disk
- Add “present” bit to PTE

How Big is a Page Table?

Assume 32-bit addresses
Assume 4K pages
Assume 4 byte page table entries (PTE)

How large is PT for each process?

Implications?

Implications

Storage for page tables may be substantial

- Simple page table: Requires PTE for all pages in address space
  Entry needed even if page not allocated ?

Additional memory reference to page table Very inefficient

- Page table must be stored in memory
- MMU stores only base address of page table

Internal fragmentation: Page size may not match size needed by process

- Wasted memory grows with larger pages

How to solve? Next lecture… (efficient page tables, TLBs)