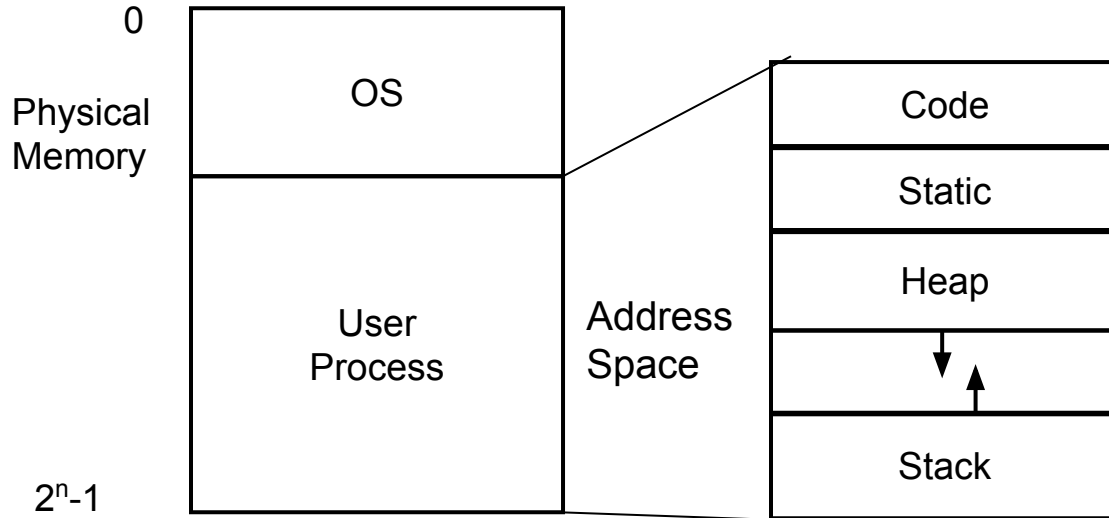


Virtual Addressing

- Address Space - abstraction where VA map to PA
- Physical Address Space - DRAM
- Virtual Address Space - abstraction - illusion of private memory

Uniprogramming - Old School

Uniprogramming: One program/process in memory runs at a time




Disadvantages:

- Only one process runs at a time
- Process can destroy OS

Multiprogramming Goals

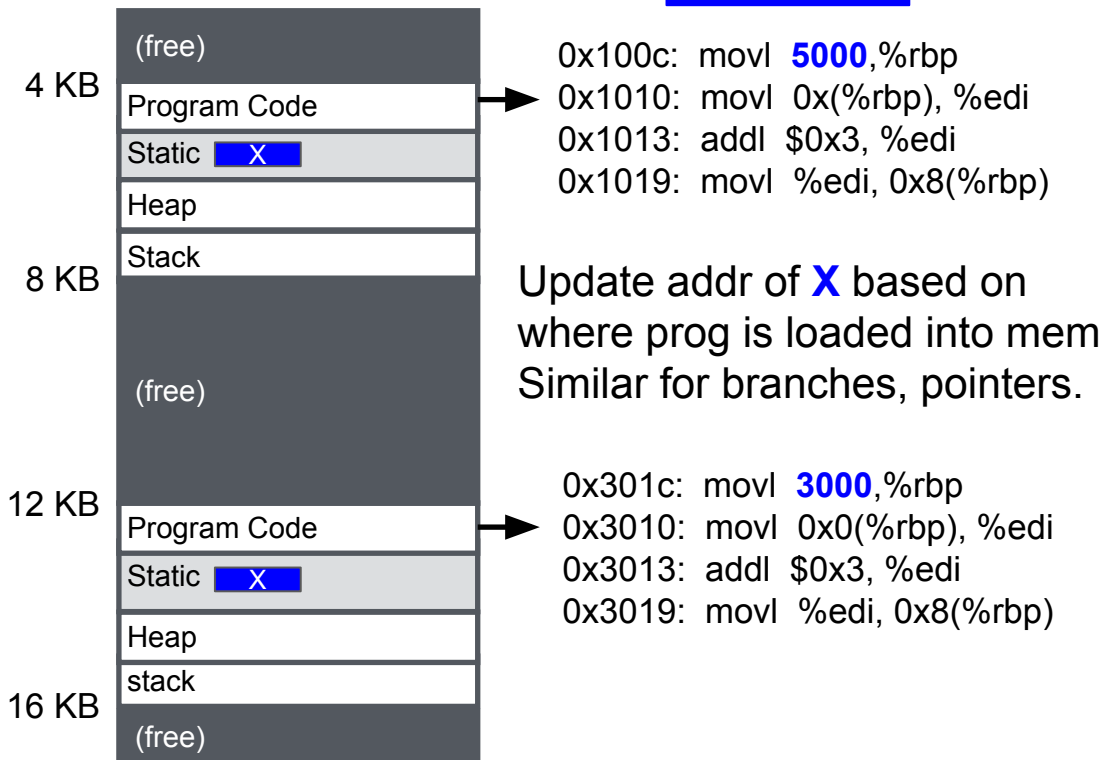
- Transparency
 - Processes are not aware that memory is shared
 - Many processes at various locations
 - Each thinks that have all of the memory
- Protection
 - Cannot corrupt OS or other processes
- Privacy
 - Cannot read data of other processes
- Efficiency
 - Do not waste memory resources - minimize fragmentation - holes of unusable memory
- Sharing
 - Cooperating processes can share portions of address space

Virtual Memory Approaches

- 1) Static Relocation - addresses are not changed by MMU
 - 2) Base
 - 3) Base + Bounds
 - 4) Segmentation
 - 5) Paging
 - Used by most systems today
 - Used by Xv6
 - We'll study paging the most
- Dynamic Relocation - addrs changed by MMU
- 
- A large curly bracket on the right side of the list groups items 2 through 5. A line extends from the middle of this bracket to the text 'Dynamic Relocation - addrs changed by MMU'. Additionally, a blue rounded rectangle encloses item 5 and its sub-points.

1) Static Relocation

X = X + 3;



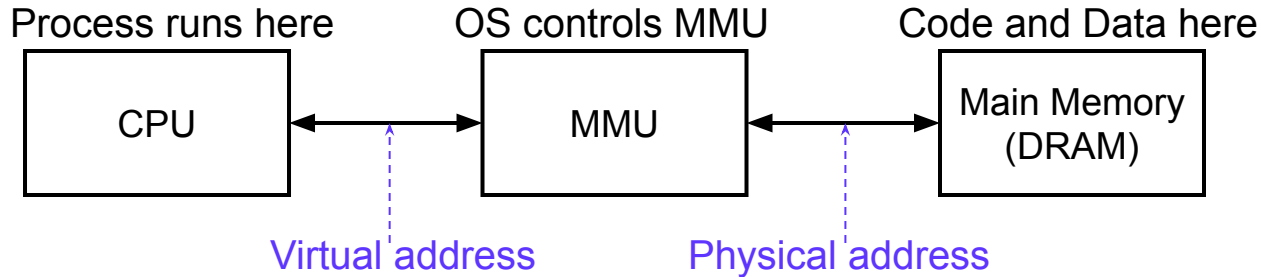
- Processes use **physical** address
- During load, OS updates program's code to have address based on where loaded
- Change instructions with address: jumps, pointers, static data

Disadvantages

- No protections
- Memory fragmentation - can't move process after loaded

Dynamic Relocation - Addresses changed by MMU

- We want to protect processes from each other
- We want all processes to have the same virtual address space
- We do not want to change program's code as it is loaded.
- Requires hardware support - Memory Management Unit (MMU)
- MMU dynamically changes address at every memory reference
 - Processes generate virtual addresses (in their address space)
 - MMU generates physical address, which is provided to memory

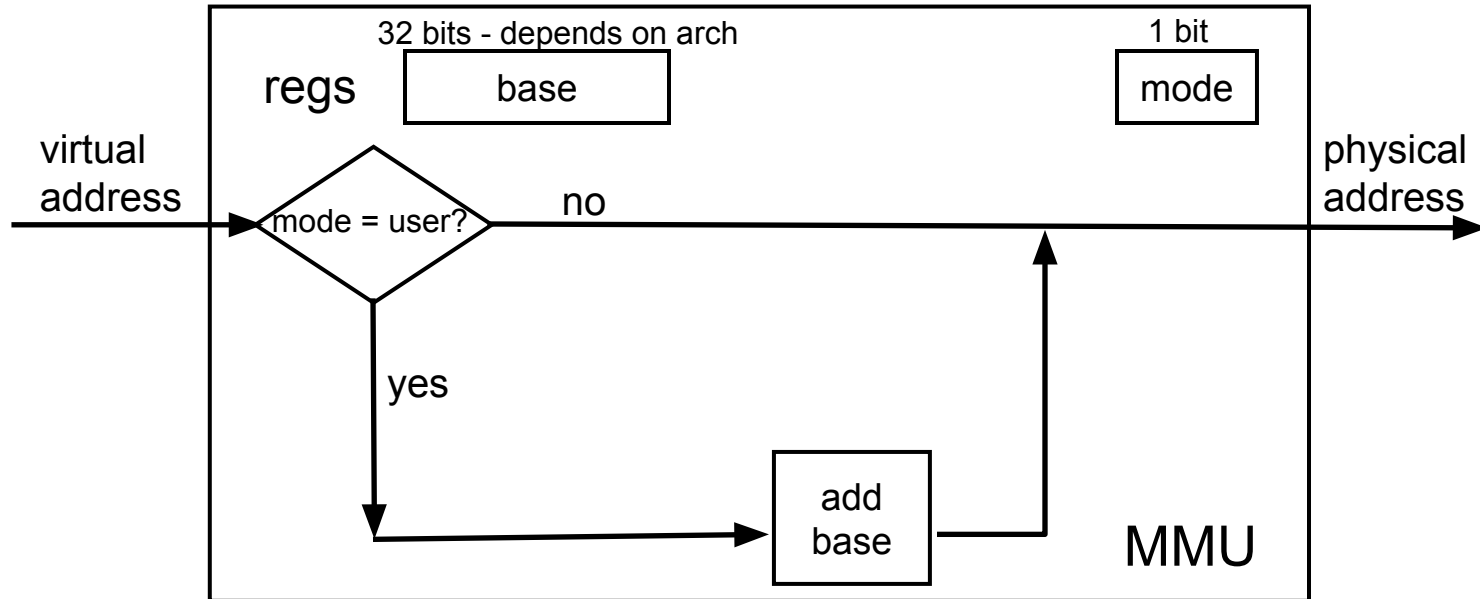


Dynamic Relocation Hardware Support

- CPU has two modes of operation
 - Kernel mode - Privileged, protected - OS runs in kernel mode
 - Enter kernel mode on traps (e.g. system calls, device interrupts)
 - Allows privileged instructions to be executed - **Can update MMU**
 - **Allows OS to access all of physical memory**
 - User mode: User processes run in user mode
 - **MMU translates virtual address to physical address**
- Minimal MMU contains **base register** for translation
 - base register contains start location for address space

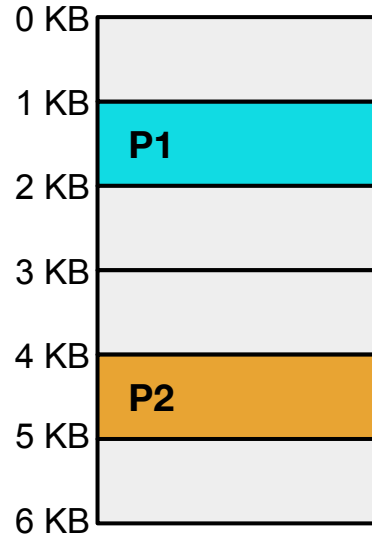
2) Dynamic Relocation: Base Register

- Translate virtual addresses on memory access of user process
 - MMU adds base register to virtual address to form physical address
 - Each process has different value in base register



P1: Base Reg has 1024

P2: Base Reg has 4096



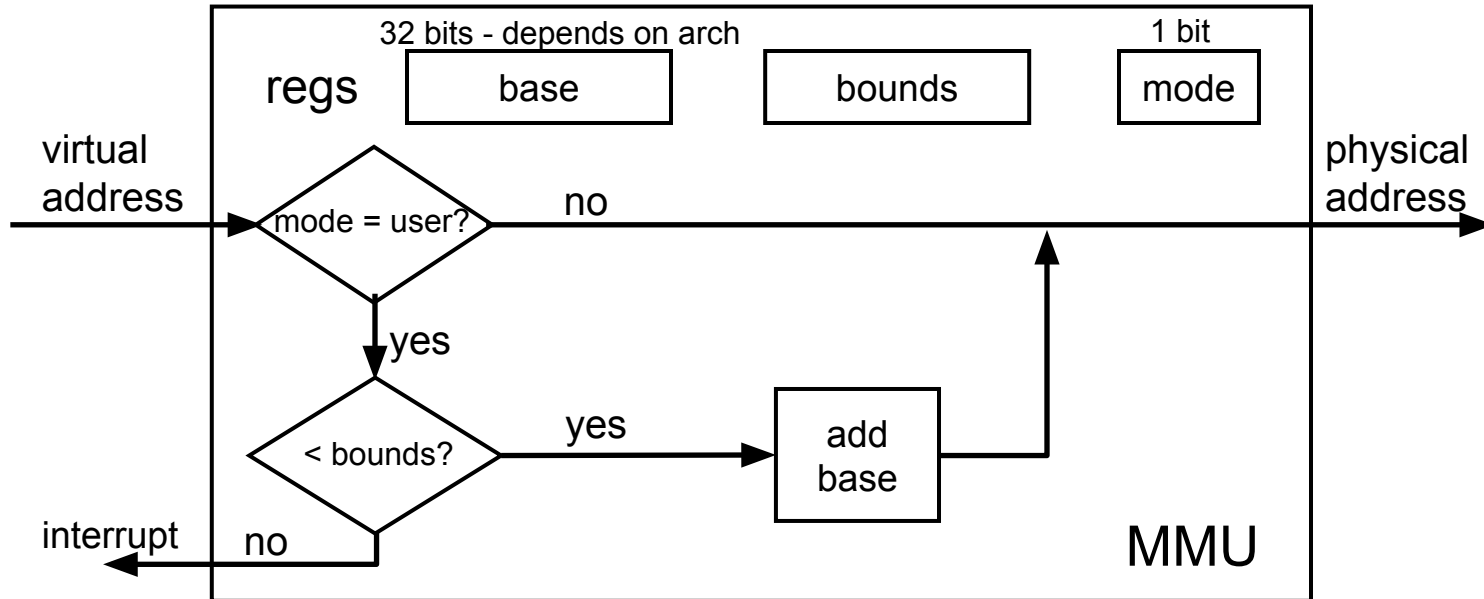
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 1000, R1	load 2024, R1

Visual Example of Base Register Dynamic Relocation

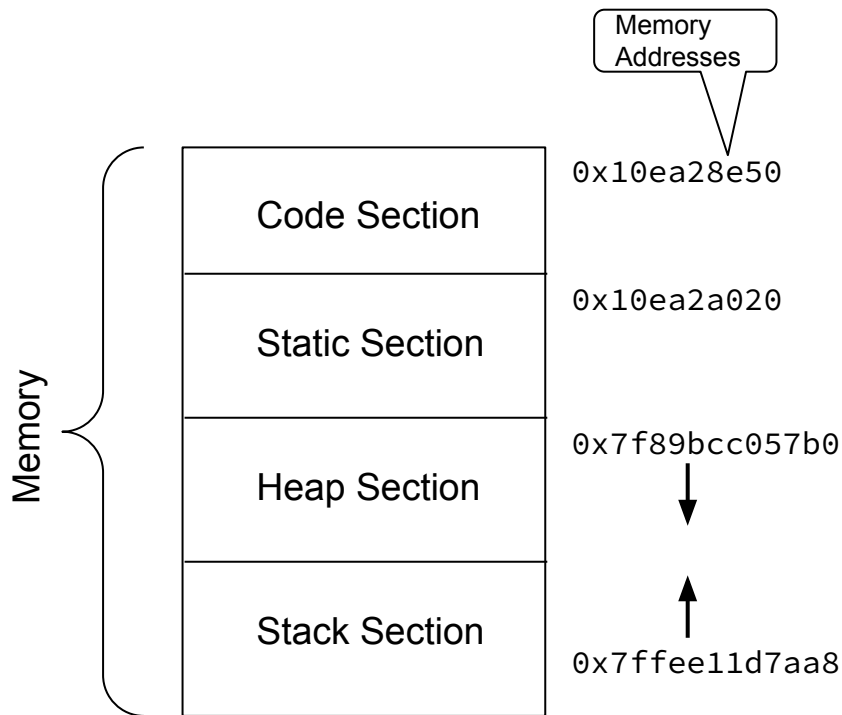
3) Implementation of Base + Bounds

Translation on every memory access of user process

- MMU compares logical address to bounds register
 - if logical address is greater, then generate interrupt
 - OS processes interrupt and terminates the process
- MMU adds base register to logical address to form physical address



Program / Process Sections

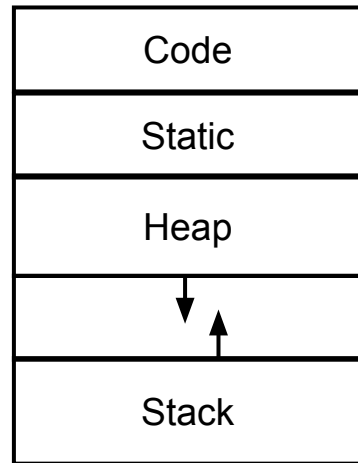


```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
int global = 0;
int main(int argc, char *argv[]) {
    int *p = malloc(sizeof(int));
    assert(p != NULL);
    printf("code:    %15p\n", main);
    printf("static:  %15p\n", &global);
    printf("heap:     %15p\n", p);
    printf("stack:    %15p\n", &p);
    return 0;
}
```

```
% gcc lecture2.c
% ./a.out
code:      0x10ea28e50
static:    0x10ea2a020 (0x11d0 byte gap)
heap:      0x7f89bcc057b0
stack:     0x7ffee11d7aa8 (0x75245d22f8 gap)
```

4) Segmented Addressing

- Divide address space into segments
 - Each segment is an entity in address space
 - code, static data, stack, heap
- The OS deals with each segment independently
 - Allocate separate blocks of physical memory
 - Segments grow and shrink
 - Segment protected (separate read/write/execute protection bits)
- Process virtual address has segment and offset within segment
 - Top bits of virtual address select segment
 - Low bits of virtual address select offset within segment



4) Segmented Addressing - Translation

- MMU contains Segment Table (per process)
 - Each segment has base and bounds, protection bits
 - Example: 14 bit virtual address, 4 segments; Two bits for segment and 12 bits for the offset.

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

Translate logical addresses (in hex) to physical addresses

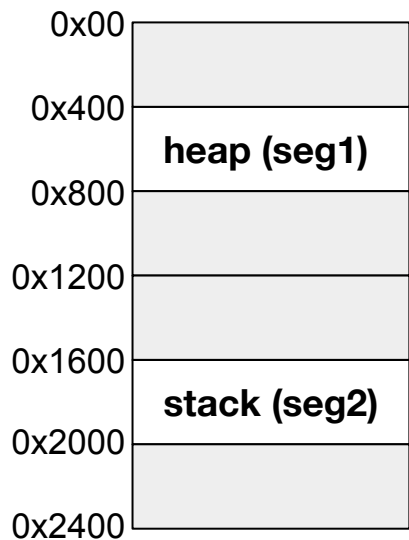
0x0240:

0x1108:

0x265c:

0x3002:

4) Segmented Addressing - Visual



Virtual	Physical
load 0x3010, R1	$0x1600 + 0x010 = 0x1610$
load 0x2010, R1	$0x400 + 0x010 = 0x410$
load 0x2100, R1	$0x400 + 0x100 = 0x500$

Segment numbers:

0: code






1: data

2: heap Base: 0x400

3: stack Base: 0x1600

MMU has 4 pairs of base, bounds regs.
One pair for each segment

Segmented - Multiprogramming Goals

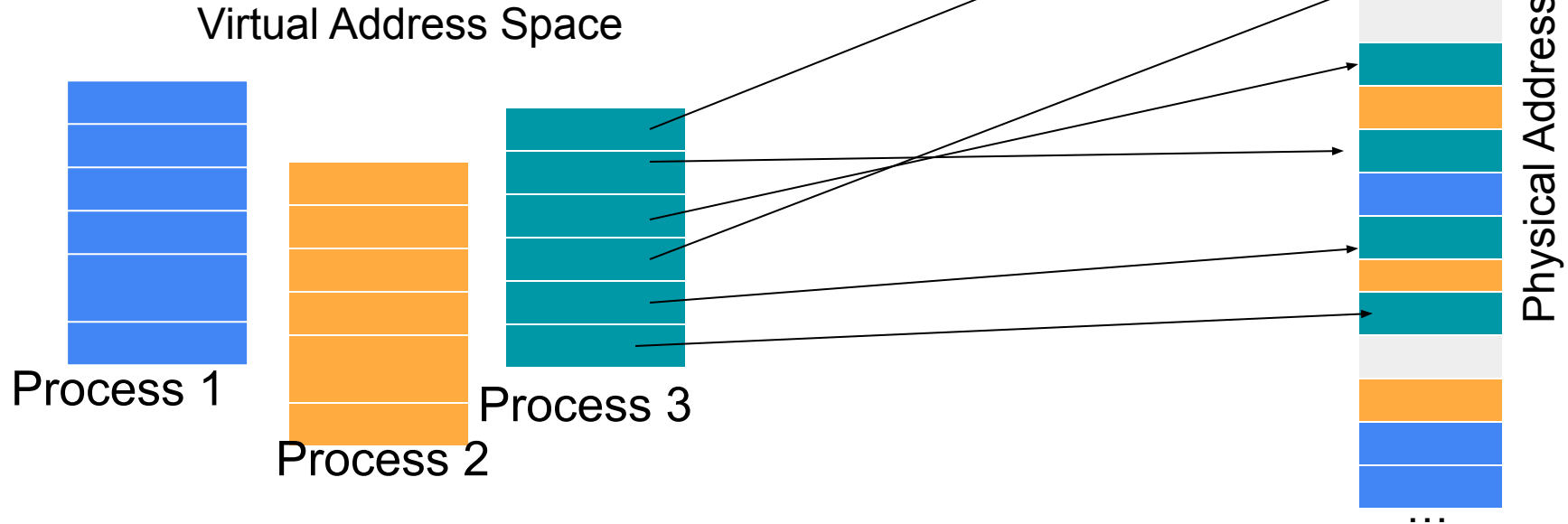
- Transparency 
 - Processes are not aware that memory is shared
 - Many processes at various locations
 - Each thinks that have all of the memory
- Protection 
 - Cannot corrupt OS or other processes
- Privacy 
 - Cannot read data of other processes
- Efficiency 
 - Do not waste memory resources - minimize **fragmentation** - holes of unusable memory
- Sharing 
 - Cooperating processes can share portions of address space

Who Controls the Base Register?

- Who translates addresses with base register?
(1) process, (2) OS, or (3) MMU
- Who modifies the base register?
(1) process, (2) OS, or (3) MMU
- How does the OS know what to put in the base register?

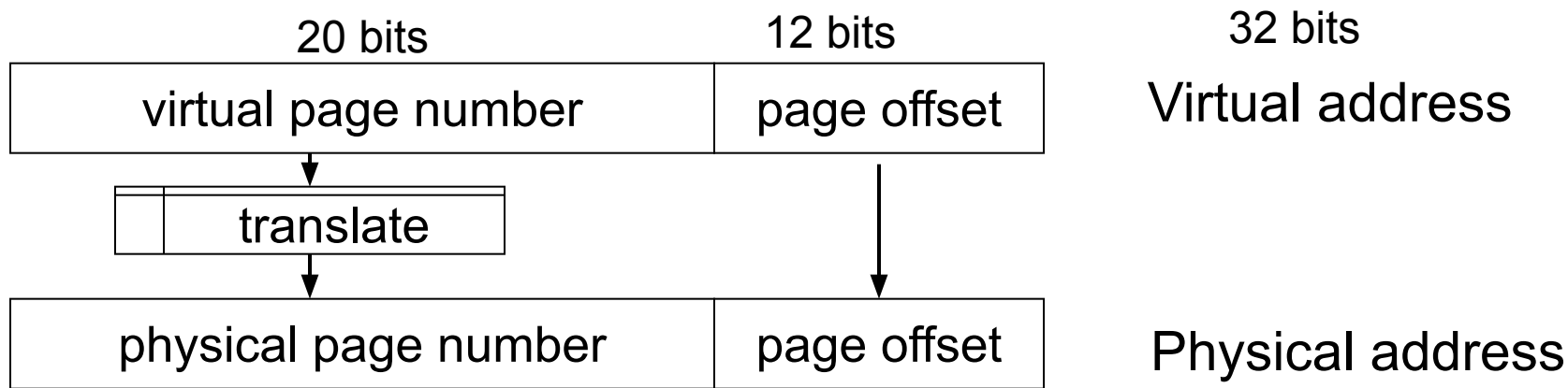
Paging - Fixed Sized Memory Allocation

- Divide address spaces into fixed-sized pages
e.g., 4096 bytes - popular size, used by Xv6
- Eliminate requirement that physical address space is contiguous

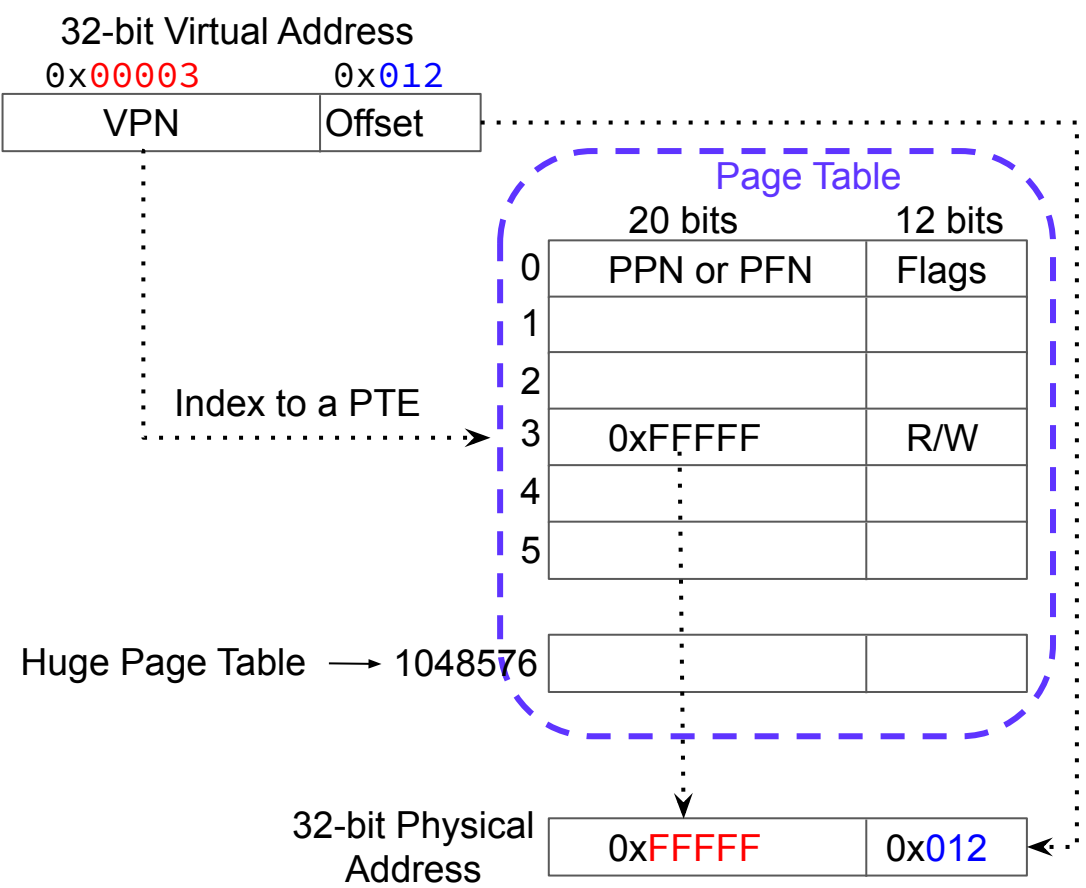


Paging - Virtual Addresses to Physical Addresses

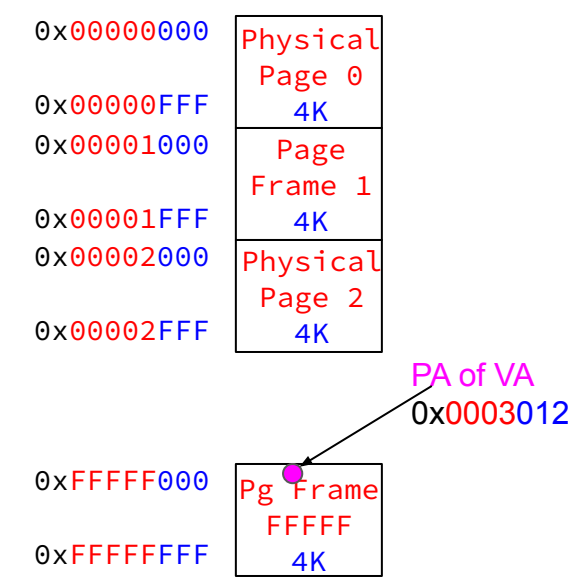
- Translate virtual address to physical address
 - High-order bits of address designate virtual page number
 - Low-order bits of address designate offset within page



- MMU uses page table to perform translation
- Page table is a data structure in memory
- OS creates a page table for each process



2**32 == 4,294,967,296
 2**48 == 281,474,976,710,656



Chrome browser, NetBeans, Geany, and MS Word all have the virtual address 0x00003012, but each one maps to a different physical address. Chrome may be 0xFFFFF012, NetBeans may be 0x10000012, Geany may be 0x00FFF012, and MS Word may be 0x00020012. Each has its own page table.

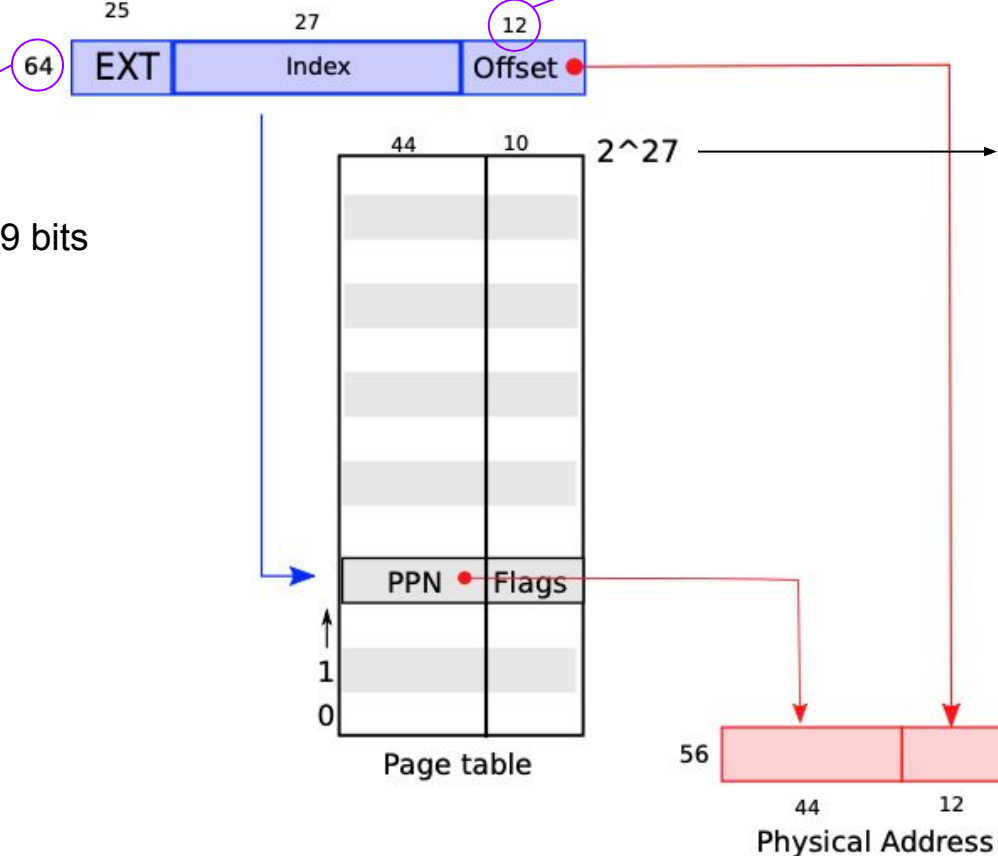
Processes have own virtual address space. All processes share same physical address space - there is only one main memory.

Xv6 Logical Page Table

- 64-bit VA
- Only use lower 29 bits
- 27-bit VPN
- 12-bit offset
- 4096 byte page

Virtual address

Page is $2^{12} = 4096$ bytes



- Huge Page Table
- 2^{27} entries
- 134,217,728 entries
- 8 bytes per entry
- 1,073,741,824 bytes
- 1 page table per proc
- sparsely populated
- 50K prog has 1G PT
- PT is 262144 pages
- 50K prog is 13 pages

56-bit PA
Max DRAM is 2^{56} bytes
72,057,594,037,927,936

Figure 3-1 of Xv6 Book

Xv6 3 Level Page Table

- 2^9 is 512
- 512 entries per PD
- 8 bytes per entry
- 4096 bytes per PD
- PD fits on one page!
- 50K prog has 3 or 4 pages for PD and 13 pages for code/data

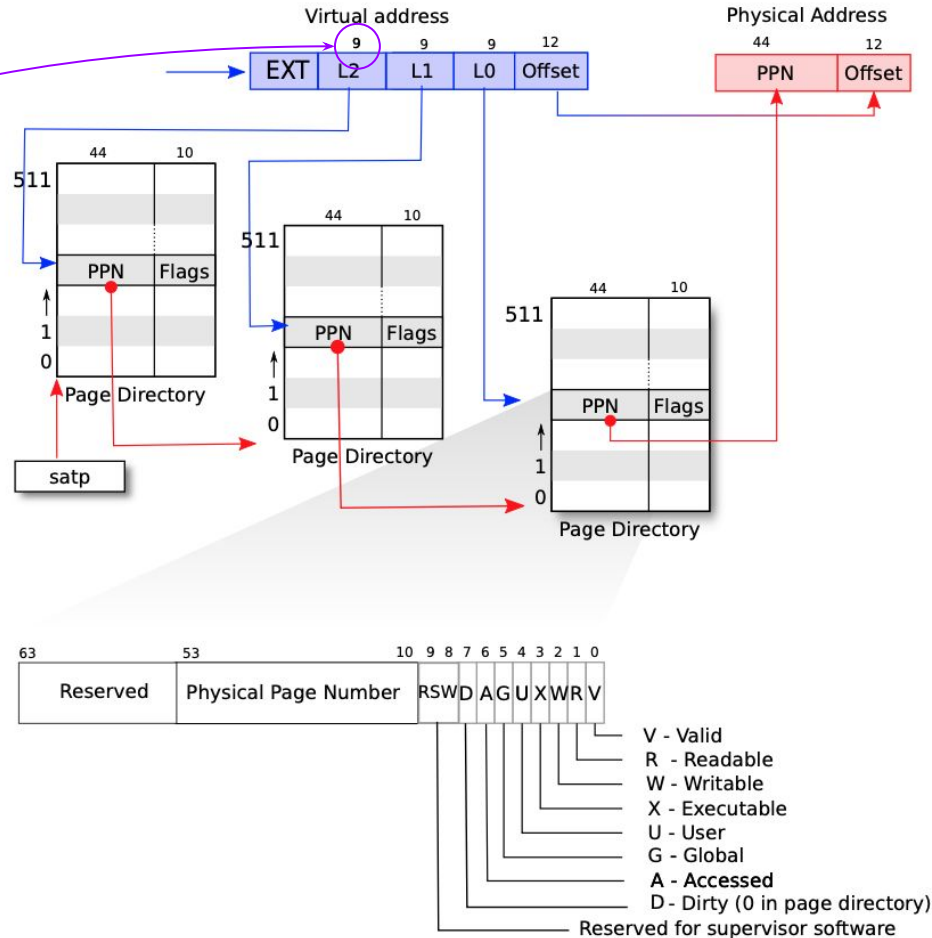


Figure 3-2 of Xv6 Book

Memory Accesses of Paging

- Three level page table requires 4 memory accesses for each load, store, fetch
- One access for each PD and one access for the actual data / instruction

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char argv[])
{
    int x
    x = x + 3;
}

s0 has 0x214

0x10: lw    a5,-20(s0) # get x from stack
0x14: addi  a5,a5,3
0x18: sw    a5,-20(s0) # put x on stack
```

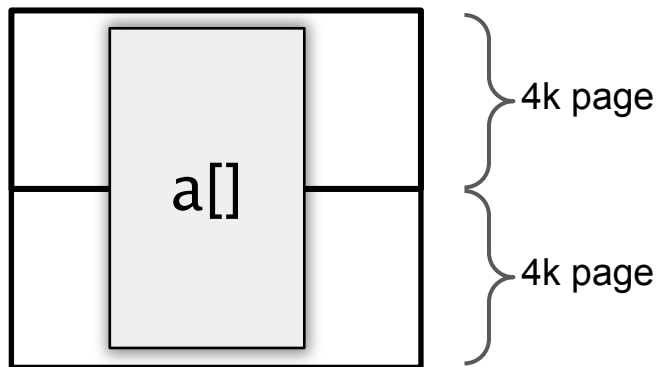
Memory Accesses	PA	Page
Fetch instruction at addr 0x10	1	4
Decode		
Exec: load from addr 0x200	2	8
Fetch instruction at addr 0x14	3	12
Decode		
Exec: no memory access		
Fetch instruction at addr 0x18	4	16
Decode		
Exec: store to addr 0x200	5	20

Translation Lookaside Buffer (TLB)

- OS allocates a page table for each process
- When a process is running the OS puts base address of PD1 in SATP register
- MMU uses SATP to walk the page table and translate VA to PA
- MMU has a TLB cache that stores recently used VAs and their translations
- If VA translation is in TLB, MMU does not walk the PT, uses TLB translation
- Processes share VA
- On a context switch, the new process cannot use existing TLB
- On a context switch, invalidate the existing TLB entries
 - `sfence.vma zero, zero # RISC-V instr` to flush TLB on context switch
- Alternatively, TLB includes an Address Space Identifier, similar to a PID

TLB Performance

```
int a[2048] // 8196 bytes
int sum = 0;
for (i=0; i<2048; i++) {
    sum += a[i];
}
```



Calculate miss rate of TLB for data:

TLB misses / # TLB lookups

TLB lookups:

= number of accesses to `a` = 2048

TLB misses:

= number of unique pages accessed

= $2048 / (\text{elements of 'a' per 4K page})$

= $2K / (4K / \text{sizeof(int)}) = 2K / 1K$

= 2

Miss rate - TLB must be updated:

$2/2048 = 0.1\%$

Hit rate: $(1 - \text{miss rate})$

99.9%

Summary

- Physical memory refers to storage cells in DRAM.
- A byte of physical memory has an address, called a physical address.
- Instructions use only virtual addresses, which the paging hardware translates to physical addresses, and then sends to the DRAM hardware to read or write storage.
- Unlike physical memory and virtual addresses, virtual memory isn't a physical object, but refers to the collection of abstractions and mechanisms the kernel provides to manage physical memory and virtual addresses.
- Address space is a collection of VAs that are mapped to PAs
- The kernel has an address space. Each process has an address space