Lecture 13: Memory Management and Address Translation

13.0 Main Points

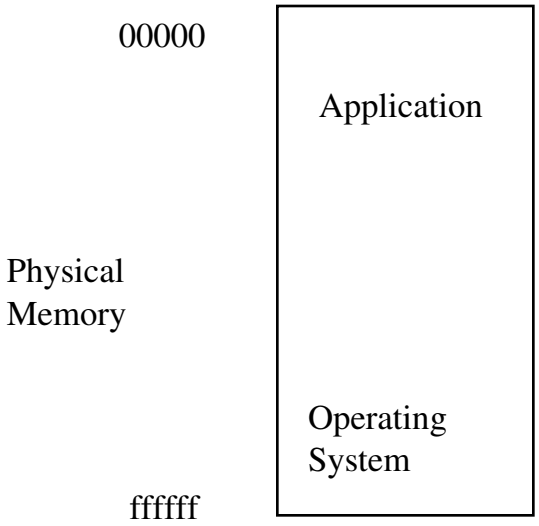
Options for managing memory: paging, segmentation, both combined

Comparison among options

13.1 No hardware translation

13.1.1 Uniprogramming

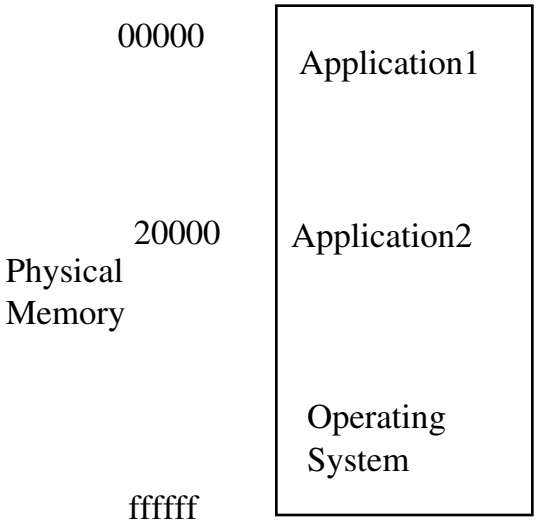
Personal computer operating systems: application always runs at the same place in physical memory, because each application runs one at a time (application given illusion of dedicated machine, by giving it reality of a dedicated machine). For example, load application into low memory, operating system into high memory. Application can address any physical memory location.



13.1.2 Multiprogramming: Linker-loader

Can multiple programs share physical memory, without hardware translation? Yes: when copy program into memory,

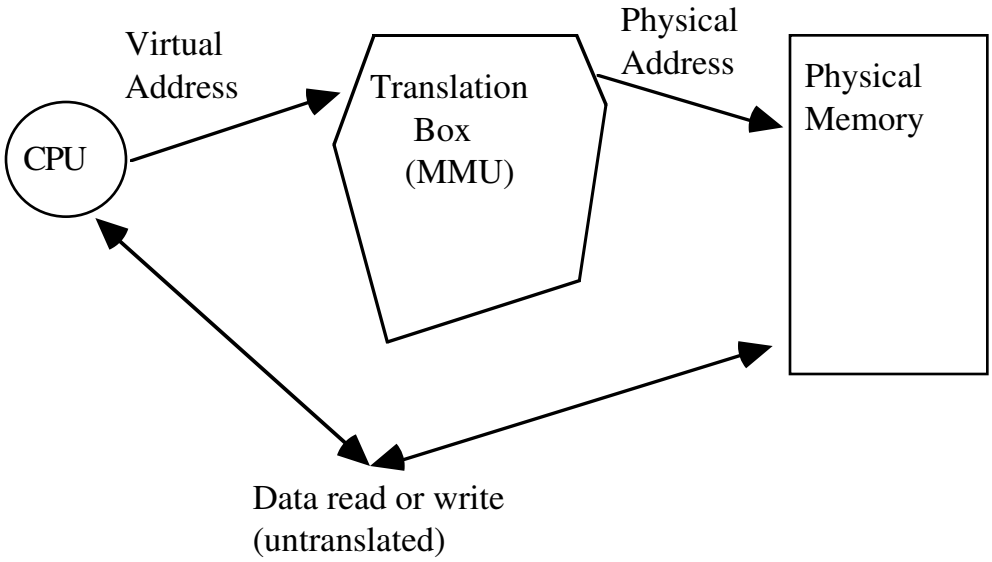
change its addresses (loads, stores, jumps) to use the addresses of where program lands in memory. This is called a **linker-loader**. Used to be very common.



UNIX **ld** works this way: compiler generates each .o file with code that starts at location 0. How do you create an executable from this? Scan through each .o, changing addresses to point to where each module goes in larger program (requires help from compiler to say where all the relocatable addresses are stored).

With linker-loader, no protection: one program's bugs can cause other programs to crash.

13.2 Hardware Translation Overview

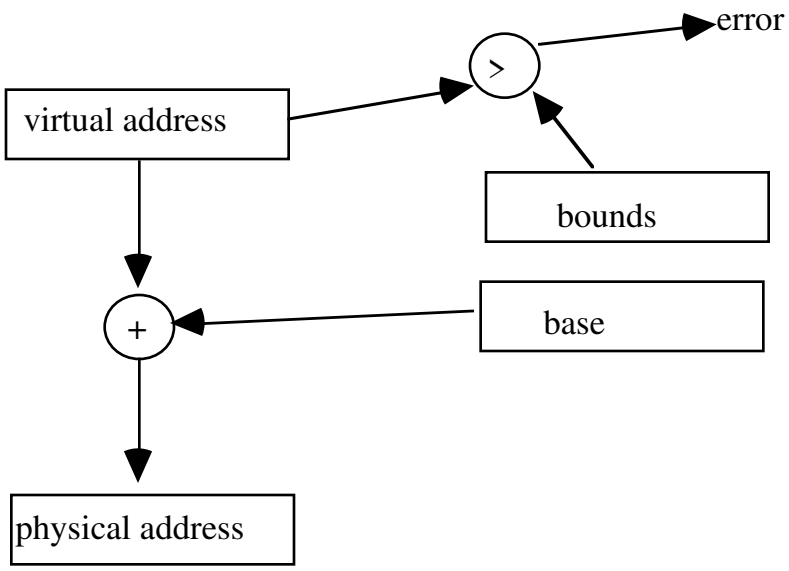


Think of memory in two ways:
view from the CPU -- what program sees, virtual memory
view from memory -- physical memory

Translation implemented in hardware; controlled in software. Various kinds of hardware translation schemes. Start with the simplest!

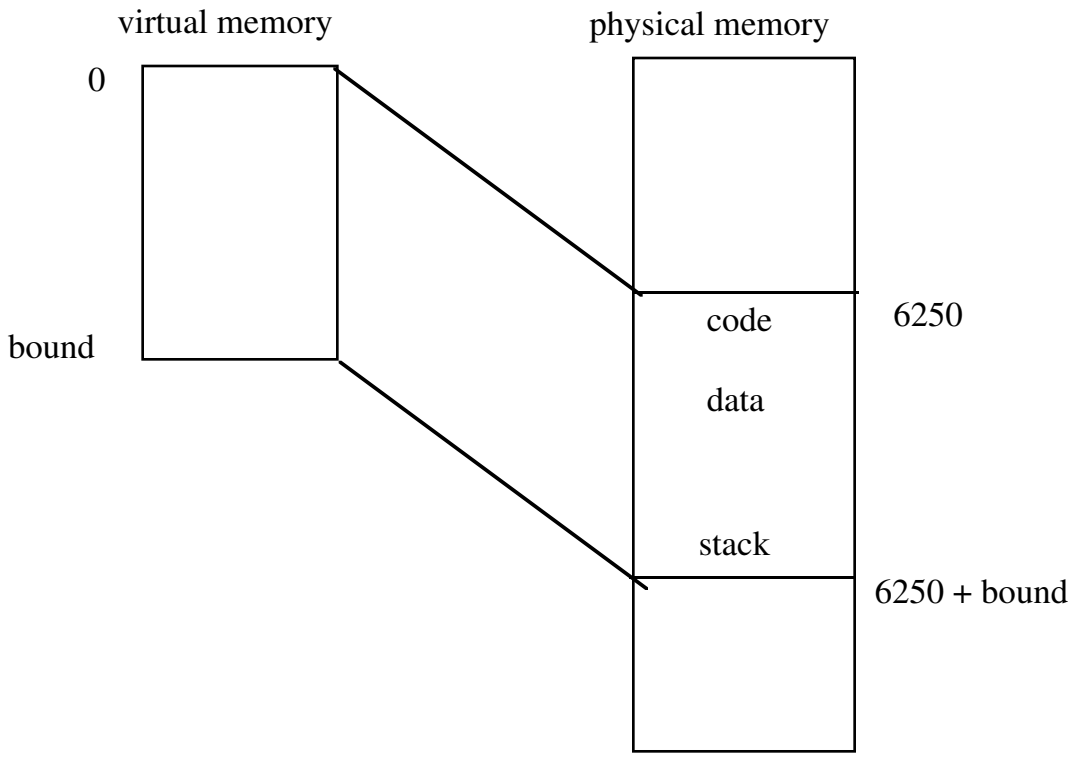
13.3 Base and Bounds

Base and bounds: Each program loaded into contiguous regions of physical memory, but with protection between programs. First built in the Cray-1.



Hardware Implementation of Base and Bounds Translation

Program has illusion it is running on its own dedicated machine, with memory starting at 0 and going up to size = bounds. Like linker-loader, program gets contiguous region of memory. But unlike linker-loader, protection: program can only touch locations in physical memory between base and base + bounds.



Virtual and Physical Memory Views in Base and Bounds System

Provides level of indirection: OS can move bits around behind the program's back, for instance, if program needs to grow beyond its bounds, or if need to coalesce fragments of memory. Stop program, copy bits, change base and bounds registers, restart.

Only the OS gets to change the base and bounds! Clearly, user program can't, or else lose protection.

Hardware cost:

- 2 registers
- adder, comparator

Plus, slows down hardware because need to take time to do add/compare on every memory reference.

Base and bounds, pros:
+ simple, fast

Cons:
1. hard to share between programs

For example, suppose two copies of "vi"
Want to share code
Want data and stack to be different
Can't do this with base and bounds!
2. Hard to grow address space. Want stack and heap to grow into each other (have to allocate maximum future needs.

3. complex memory allocation
First fit, best fit, buddy system

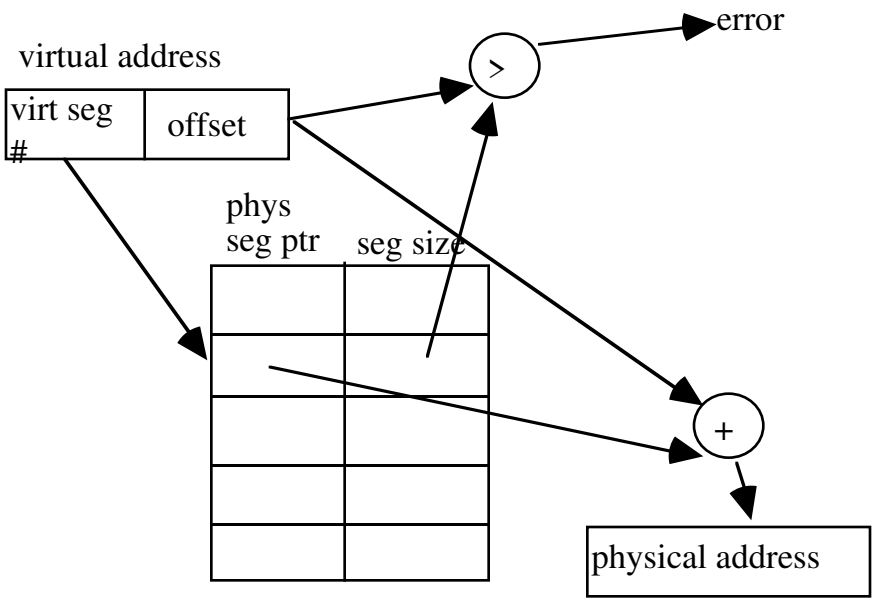
In worst case, have to shuffle large chunks of memory to fit new
program.

First, fix the first two (segmentation), then the first and third (paging), then all three at the same time (segmentation plus paging)!

13.4 Segmentation

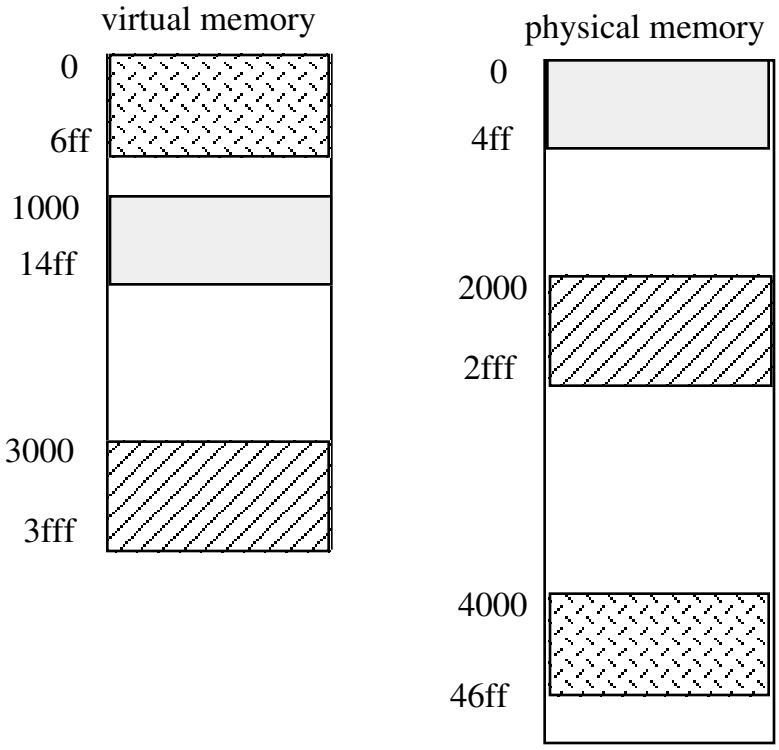
A **segment** is a region of logically contiguous memory.

Idea is to generalize base and bounds, by allowing a **table** of base&bound pairs.



For example, what does it look like with this segment table, in virtual memory and physical memory? Assume 2 bit segment ID, and 12 bit segment offset.

virtual segment #	physical segment start	segment size
code	0x4000	0x700
data	0	0x500
-	0	0
stack	0x2000	0x1000



This should seem a bit strange: the virtual address space has gaps in it! Each segment gets mapped to contiguous locations in physical memory, but may be gaps between segments.

But a correct program will never address gaps; if it does, trap to kerenl. Minor exception: stack, heap can grow. In UNIX, sbrk() increases size of heap segment. For stack, just take fault, system automatically increases size of stack.

Example: What happens with the above segment table, with the following as virtual memory contents? Code does: sin(x);

Virtual memory

Main: 240	sub 8, sp
244	store 1108, 4(sp)
248	store pc +8, (sp)
24c	jump 360

250	
...	
Sin: 360	load 4(sp), r2
364	load (r2), r3
...	
420	jump (sp)
...	
x: 1108	666
...	
3658	
365c	

Initially, pc = 240 and sp = 3660.

Physical Memory

x: 108	666
...	
2658	
265c	
...	
Main: 4240	sub 8, sp
4244	store 1108, 4(sp)
4248	store pc +8, (sp)
424c	jump 360
4250	
...	
Sin: 4360	load 4(sp), r2
4364	load (r2), r3
...	
420	jump (sp)

Segmentation pros & cons:
+ efficient for sparse address spaces

+ easy to share whole segments (for example, code segment)
Detail: Need to add protection mode in segmentation table. For example, code segment would be read-only (only execution and loads are allowed). Data and stack segment would be read-write (stores allowed).

- complex memory allocation
Still need first fit, best fit, etc., and re-shuffling to coalesce free fragments, if no single free space is big enough for a new segment.

How do we make memory allocation simple and easy?

13.5 Paging

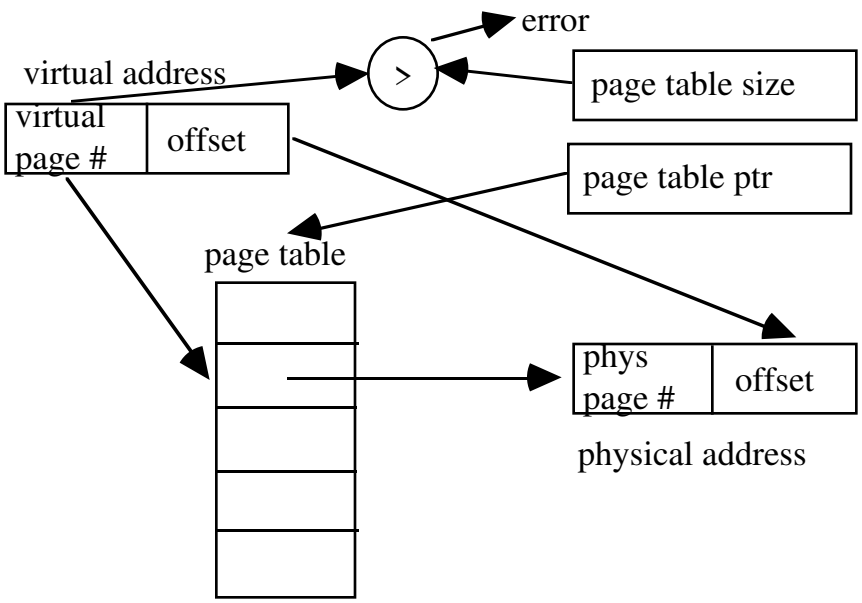
Allocate physical memory in terms of fixed size chunks of memory, or **pages**.

Simpler, because allows use of a bitmap. What's a bitmap?

0011111000000001100

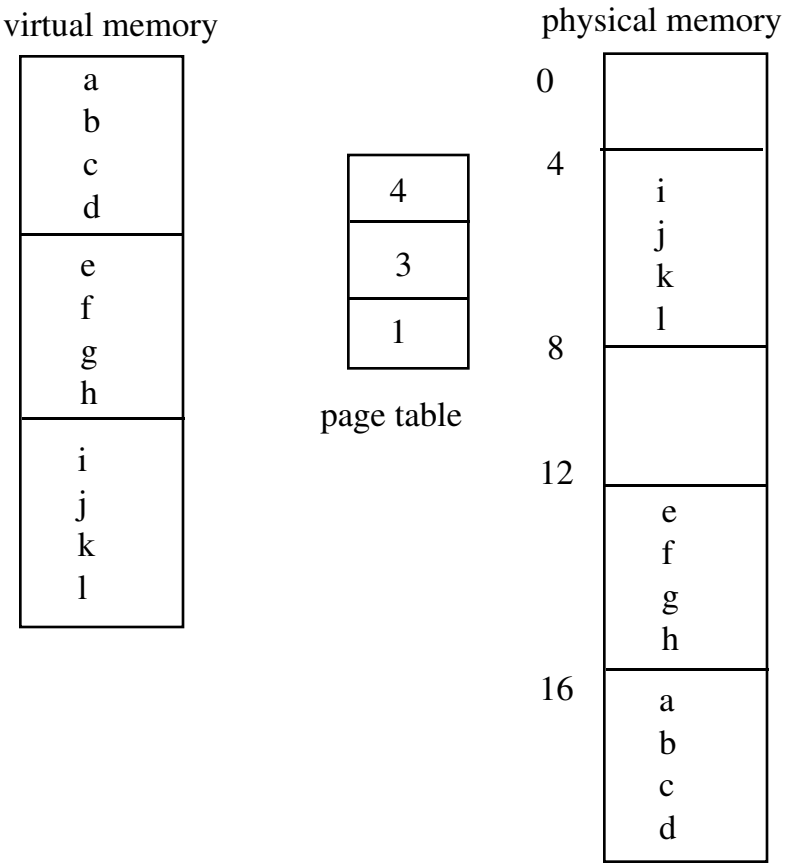
Each bit represents one page of physical memory -- 1 means allocated, 0 means unallocated. Lots simpler than base&bounds or segmentation

Operating system controls mapping: any page of virtual memory can go anywhere in physical memory.



Each address space has its own page table, in physical memory. Hardware needs two special registers -- pointer to physical location of page table, and page table size.

Example: suppose page size is 4 bytes.



Where is virtual address 6? 9?

Questions:

1. What if page size is very small? For example, VAX had a page size of 512 bytes. Means lots of space taken up with page table entries.

2. What if page size is really big? Why not have an infinite page size? Would waste unused space inside of page. Example of internal fragmentation.

Fragmentation: wasted space

external -- free gaps between allocated chunks
internal -- free gaps because don't need all of allocated chunk

With segmentation need to re-shuffle segments to avoid external fragmentation. Paging suffers from internal fragmentation.

3. What if address space is sparse? For example: on UNIX, code starts at 0, stack starts at 2^31 - 1. With 1KB pages, 2 million page table entries -- because have to have table that maps entire virtual address space.

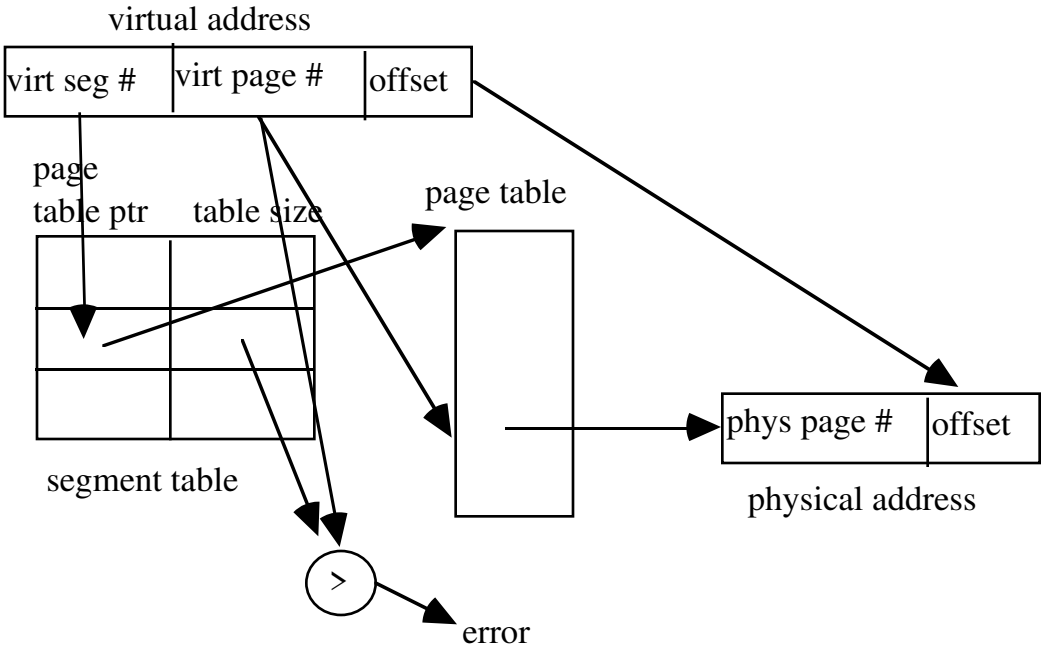
Paging pros&cons:
+ simple memory allocation
+ easy to share
- big page tables if sparse address space

Is there a solution that allows simple memory allocation, easy to share memory, **and** is efficient for sparse address spaces?

How about combining paging and segmentation?

13.6 Multi-level translation

Multi-level translation. Use tree of tables. Lowest level is page table, so that physical memory can be allocated using a bitmap. Higher levels are typically segmented. For example, 2-level translation:



Just like recursion -- could have any number of levels. Most architectures today do some flavor of this.

Questions:

Where are segment table/page tables stored? Segment tables are usually in special CPU registers, because they are small. Page tables, usually in main memory

How do we share memory? Can share entire segment, or a single page.

Example: Suppose virtual address has 4 bits of segment #, 8 bits of virtual page #, and 12 bits of offset.

Segment Table

page table ptr	page table size
2000	0x14
0	0

1000	0xd
0	0

Physical Memory

...	
1000	0x6
	0xb
	0x4
...	
2000	0x13
	0x2a
	0x3
....	

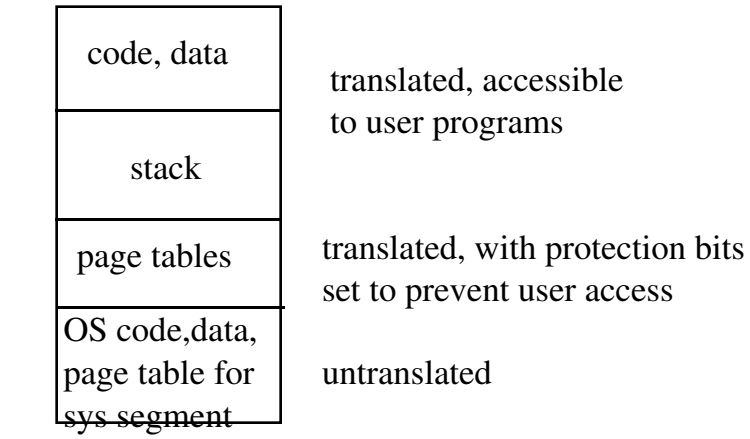
What do the following addresses translate to?
2070?
202016 ?
104c684 ?
210014 ?

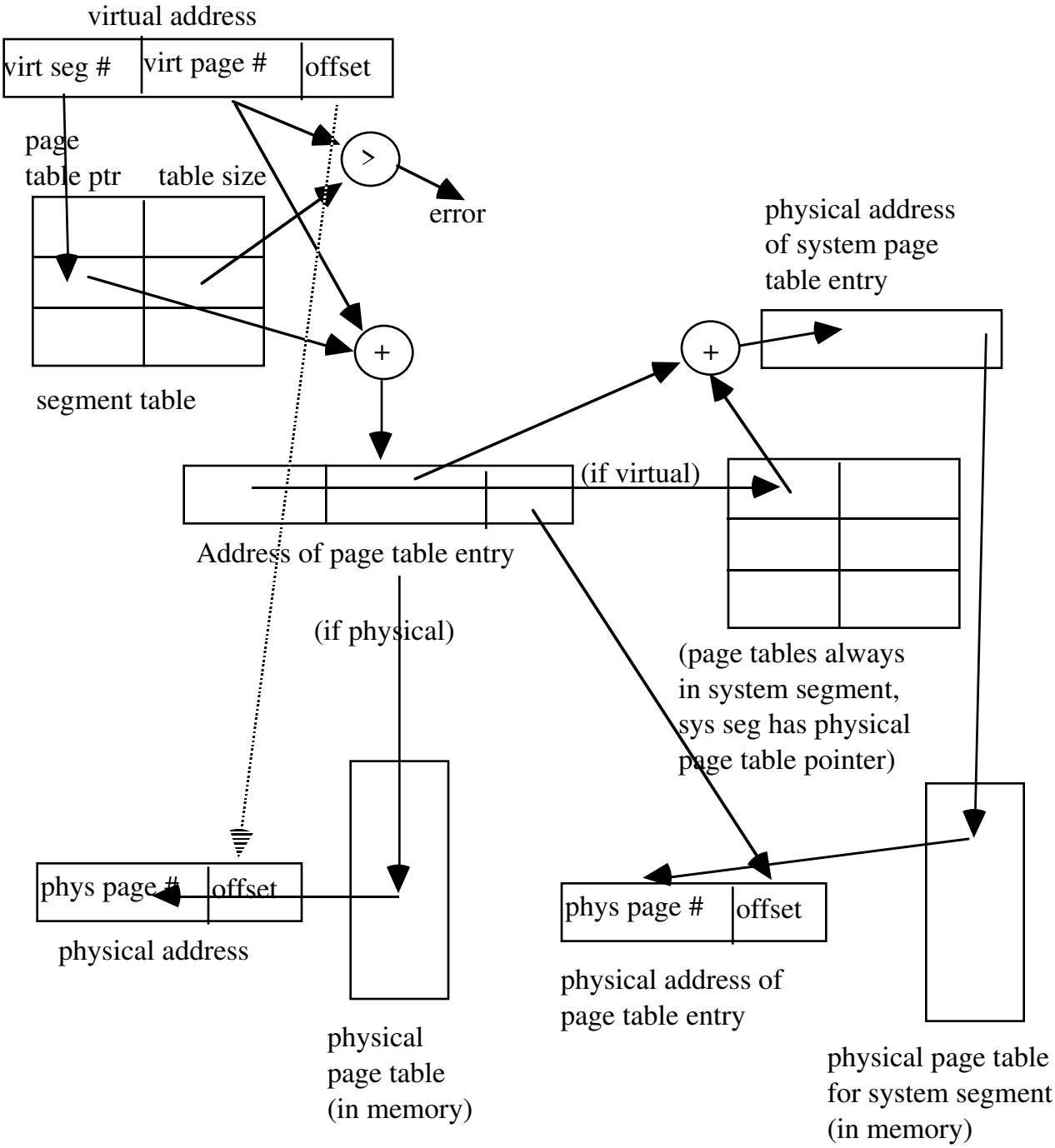
Multilevel translation:
+ only need to allocate as many page table entries as we need. In other words, sparse address spaces are easy.
+ easy memory allocation
+ share at seg or page level
- pointer per page (typically 4KB - 16KB pages today)
- page tables need to be contiguous
- two lookups per memory reference

13.7 Multilevel translation with paged page tables

Above, page tables have to be allocated linearly in memory. Can we page them? Yes! Idea is to put the page tables in a special segment that is translated, but inaccessible to user programs. To prevent recursion, the page table for this

segment is in physical memory. Thus, the segment table contains page table pointers that are virtual for some segments, and physical for some others. In fact, this is how the MIPS and the HP's work!





This means that potentially, each memory reference involves three memory references (one for the system page table, one for the user page table, and one for the real data).

How do we reduce the overhead of translation? TLB's!