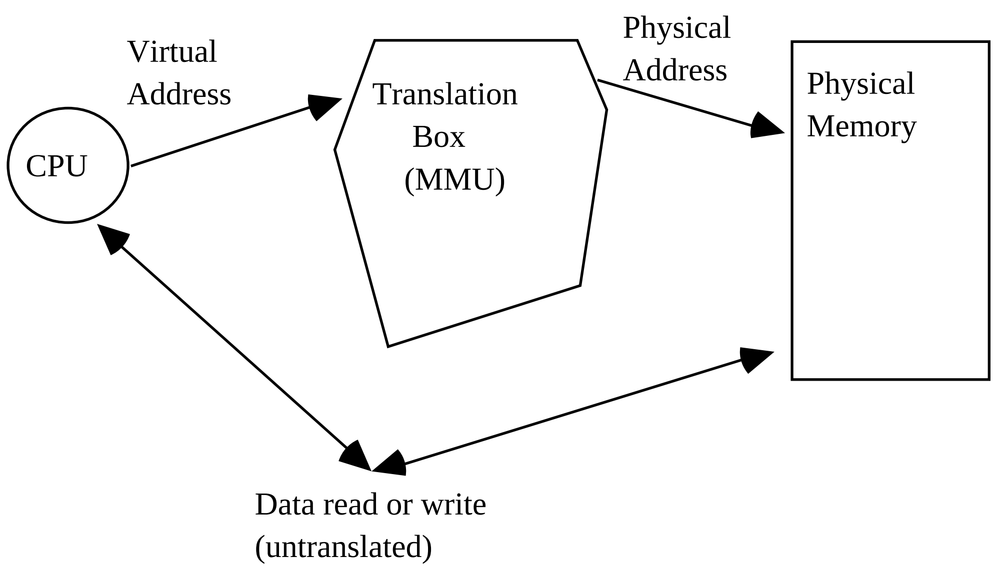
Lecture 13: Address Translation

13.1 Main Points

Memory management options: paging, segmentation, hybrids
Virtual vs. physical view of memory
Internal vs. external fragmentation

13.2 Hardware Translation Overview

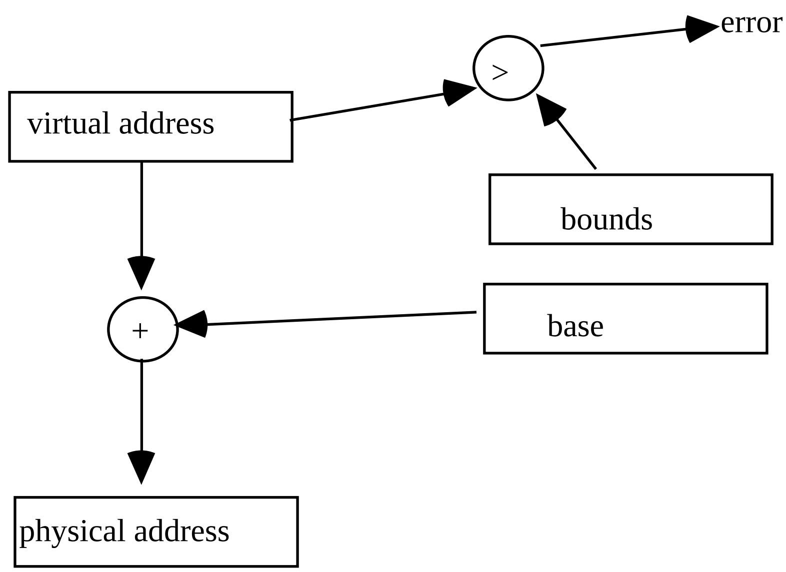


Think of memory in two ways:
view from the CPU -- what program sees, virtual memory
view from memory -- physical memory

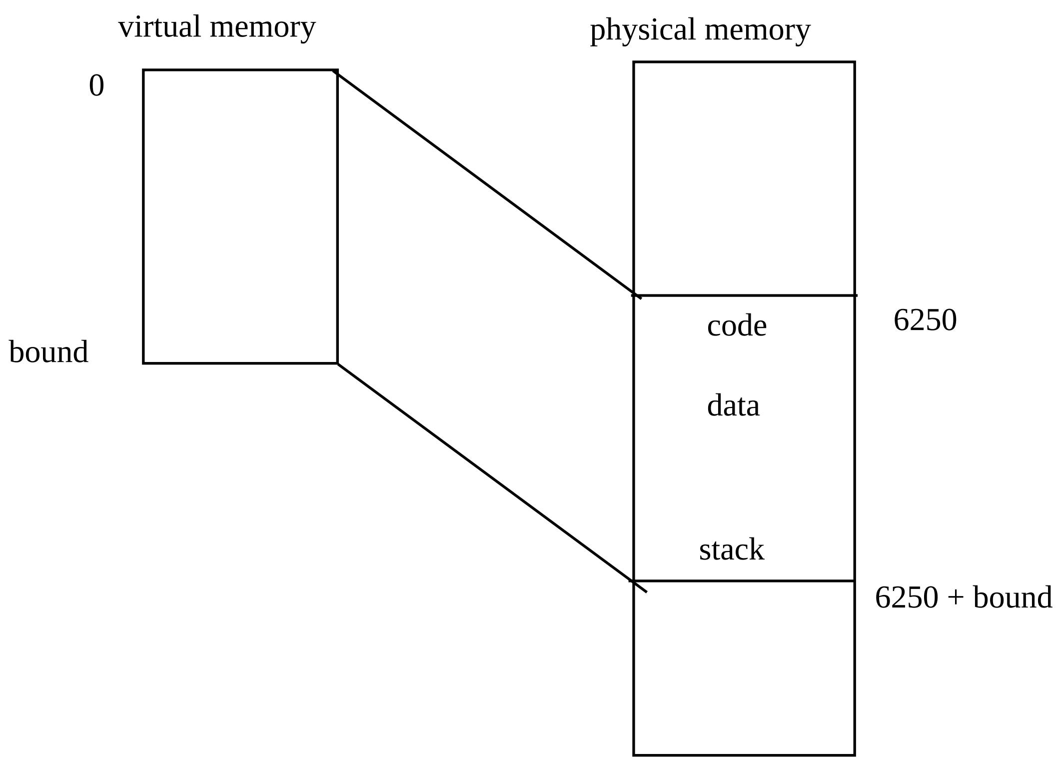
Translation implemented in hardware; controlled in software.

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



Virtual and Physical Memory Views in Base and Bounds System

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

With base&bounds system, what gets saved/restored on a context switch?
Hardware cost:

2 registers
adder, comparator

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

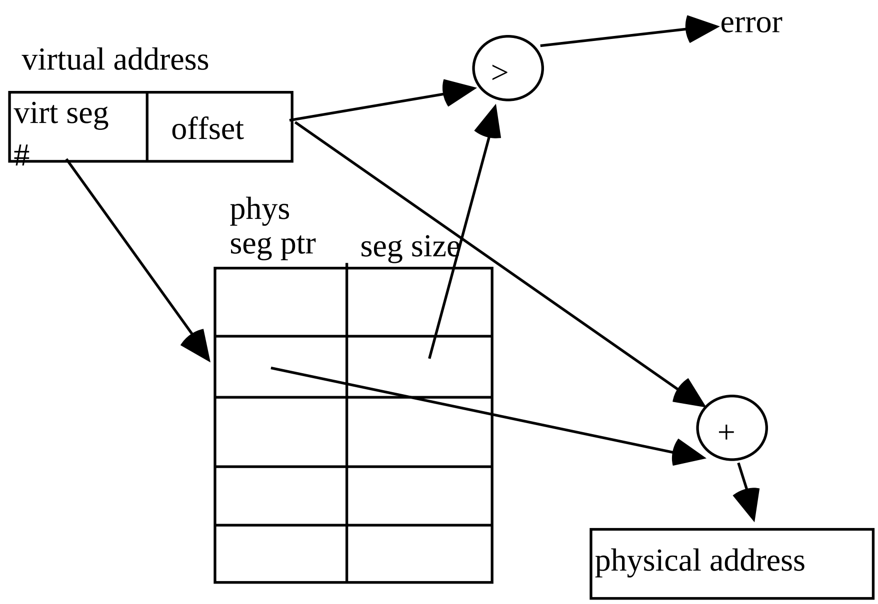
Base and bounds:
+ simple, fast
- hard to share between programs

- complex memory allocation (particularly when address space needs to grow dynamically)
- Can't put heap, stack at opposite ends of address space

13.4 Segmentation

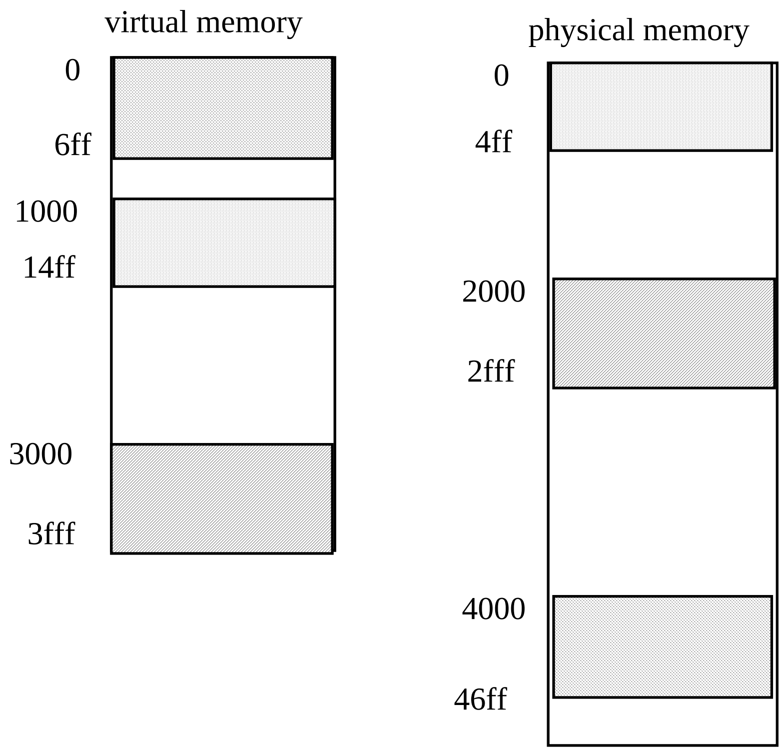
A **segment** is a region of logically contiguous memory.

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.

A correct program will never address gaps; if it does, trap to kernel and then core dump. 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.

Detail: Protection mode in segmentation table entries. For example, code segment would be read-only (only execution and loads are allowed). Data and stack segment would be read-write (stores allowed).

What must be saved/restored on context switch? Typically, segment table stored in CPU, not in memory, because it's small.

Example: What happens with the above segment table, with the following as virtual memory contents? Code does:

```
strlen(x);
```

Virtual memory

Main: 240	store 1108, r2
244	store pc +8, r31
248	jump 360
24c	
...	
Strlen: 360	loadbyte (r2), r3
...	
420	jump (r31)
...	
x: 1108	a b c \0
...	

Initially, pc = 240.

Physical Memory

x: 108	666
...	
Main: 4240	store 1108, r2
4244	store pc +8, r31
4248	jump 360
424c	
...	
Strlen: 4360	loadbyte (r2), r3
...	
420	jump (r31)

Segmentation pros & cons:

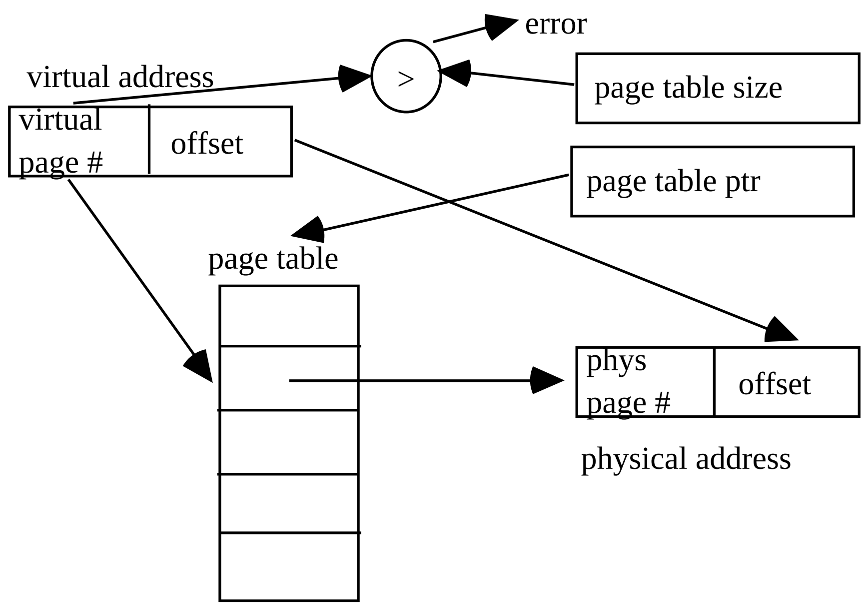
+ efficient for sparse address spaces

+ easy to share whole segments (for example, code segment)

- complex memory allocation

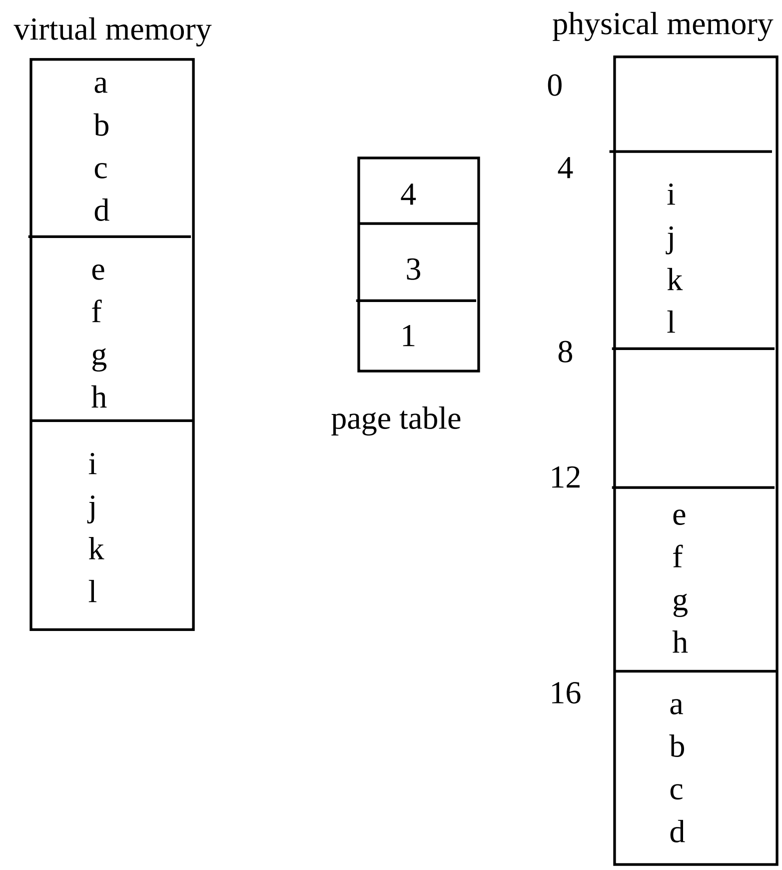
13.5 Paging

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



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:

0. What must be saved and restored on a context switch?

1. What if page size is very small? For example, VAX had a page size of 512 bytes.

2. What if page size is really big? Why not have an infinite page size?

Fragmentation: wasted space

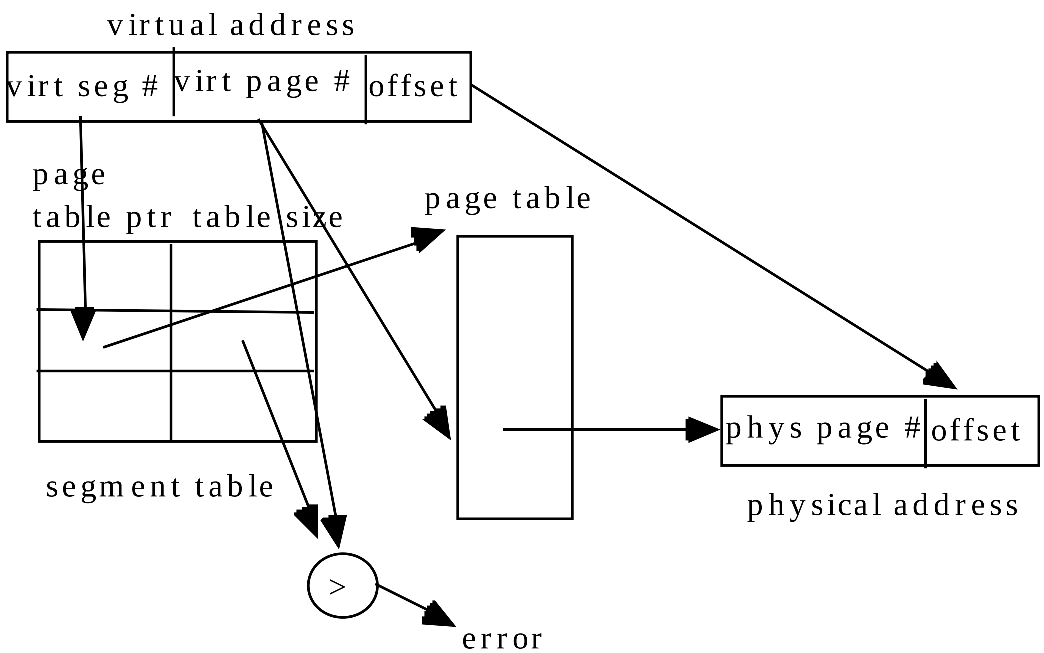
external -- free gaps between allocated chunks
internal -- free gaps because don't need all of allocated chunk

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!

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

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:



Questions:

What must be saved/restored on context switch?

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

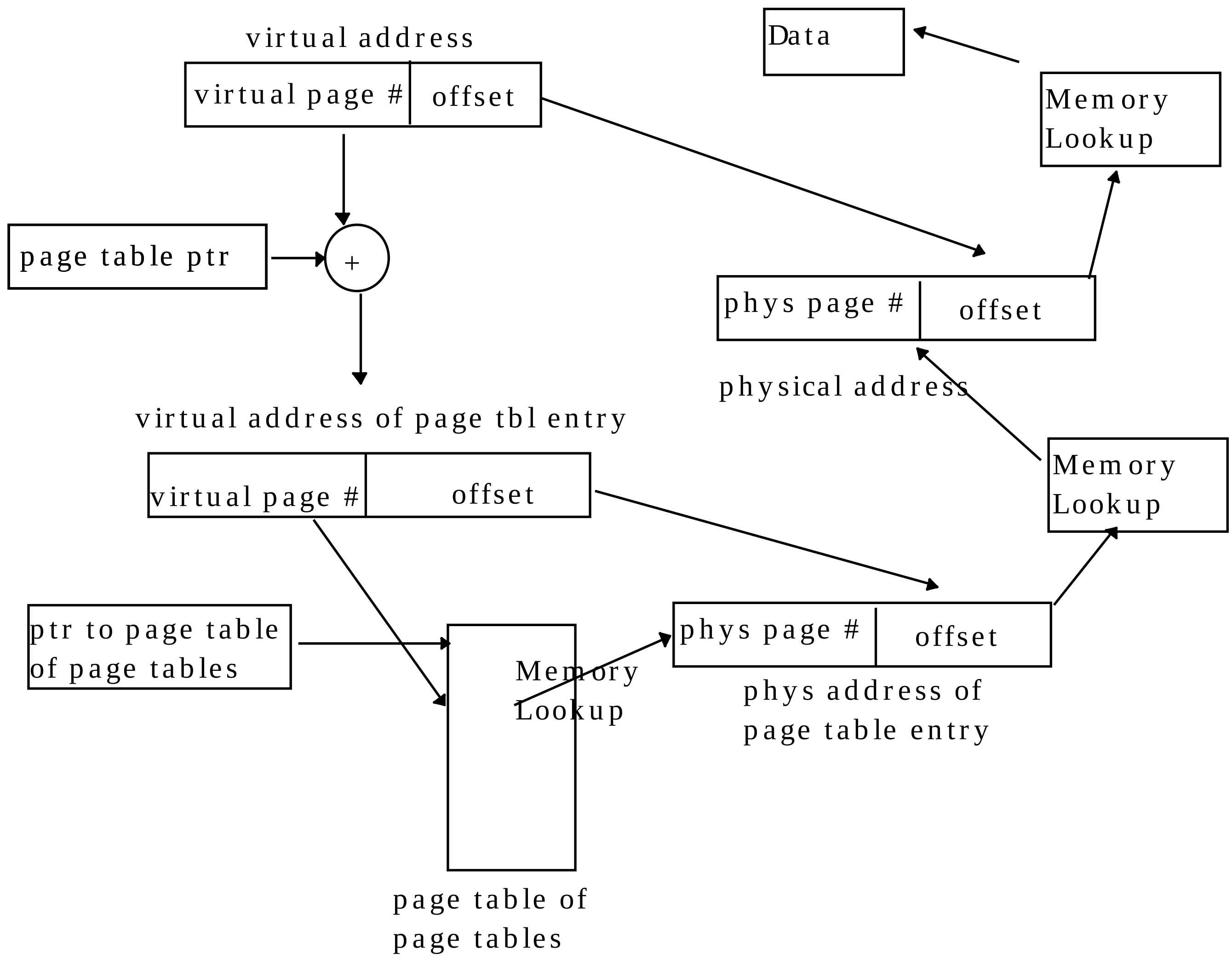
...	
1000	0x6
	0xb
	0x4
...	
2000	0x13
	0x2a
	0x3
....	

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.
+ easy memory allocation
+ share at segment or page level
- pointer per page (typically 4KB - 16KB pages today)
- page tables need to be contiguous
- two (or more) lookups per memory reference

13.7 Paged page tables

A different solution to sparse address spaces is to allow the page tables to be paged -- only need to allocate physical memory for page table entries you really use. Top level page table is in physical memory, all lower levels of hierarchy are in virtual memory (and therefore can be allocated in fixed size page frames in physical memory).



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).

13.8 Inverted page tables

What is an efficient data structure for doing lookups? Hash table. Why not use a hash table to translate from virtual address to a physical address.

This is called an inverted page table for historical reasons.

Take virtual page #, run hash function on it, index into hash table to find page table entry with physical page frame #.

Independent of size of address space,
Advantages:
+ O(1) lookup to do translation
+ Requires page table space proportional to how many pages are actually being used, not proportional to size of address space -- with 64 bit address spaces, big win.
- overhead of managing hash chains, etc.