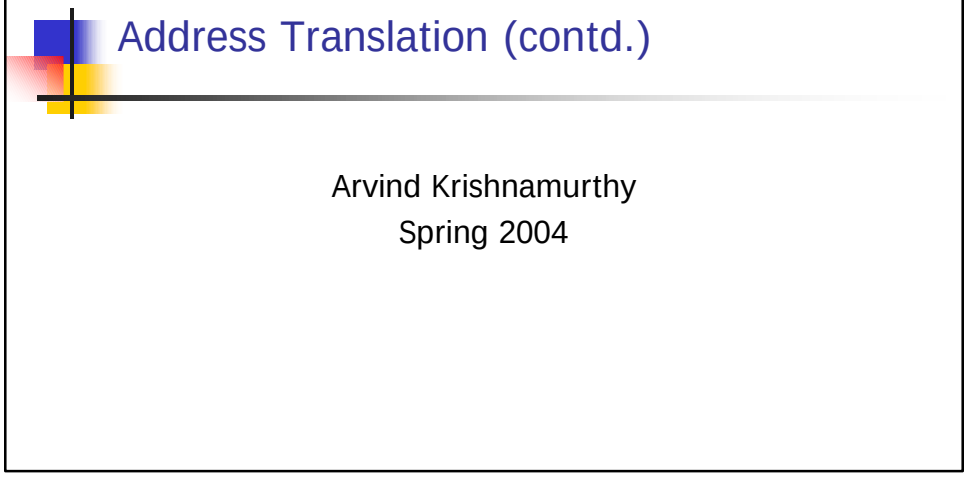
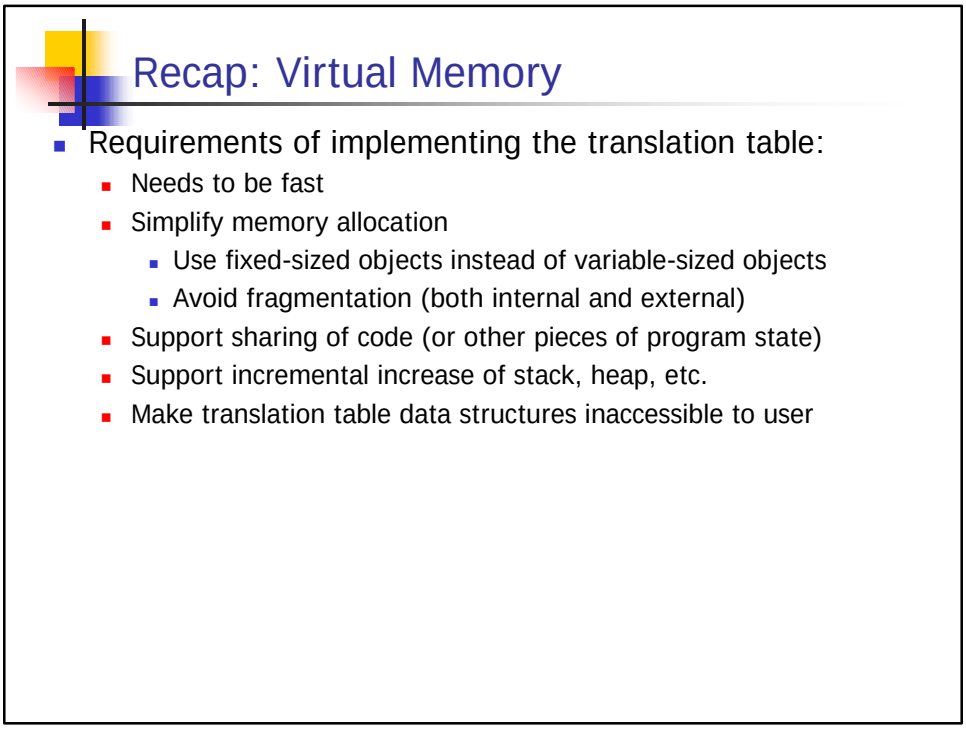


Address Translation (contd.)

Arvind Krishnamurthy
Spring 2004



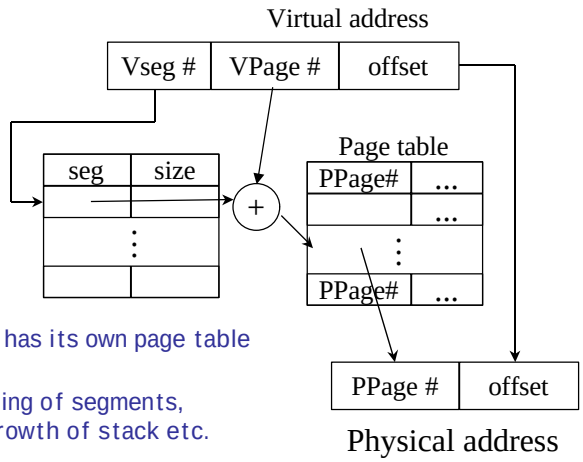
Recap: Virtual Memory

- Requirements of implementing the translation table:
  - Needs to be fast
  - Simplify memory allocation
    - Use fixed-sized objects instead of variable-sized objects
    - Avoid fragmentation (both internal and external)
  - Support sharing of code (or other pieces of program state)
  - Support incremental increase of stack, heap, etc.
  - Make translation table data structures inaccessible to user

Approaches

- Base & bounds approach:
  - Simple, fast
  - But does not support sharing, incremental increase
  - Complex memory allocation
- Segment table:
  - Top few bits encode segment number. Each segment has a base and bounds.
  - Supports sharing and allows holes in virtual address space
  - Complex memory allocation
- Page tables:
  - Memory allocation done in small page sizes (4K – 16K)
  - Supports sharing
  - But need to allocate page table for entire virtual address space

Segmentation with paging



Each segment has its own page table

Supports sharing of segments, incremental growth of stack etc.

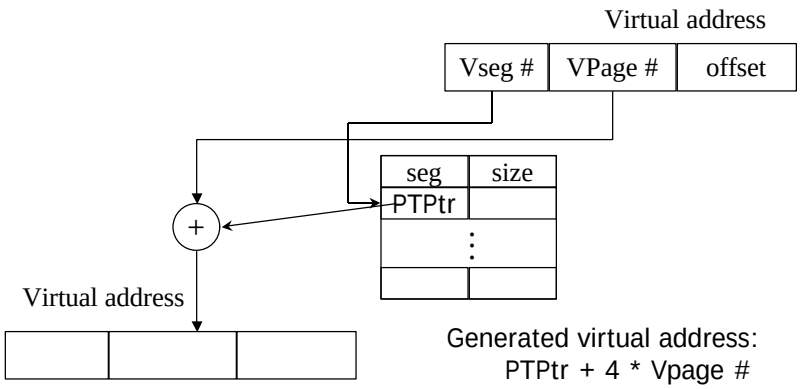
Almost simple memory allocation

Issues

- Is the page table pointer associated with each segment physical or virtual?
  - If physical, then the page table needs to be allocated contiguously
    - Assume 2 bits for segment number, 18 bits for page number, 12 bits for page offset
    - Each page table entry (PTE) contains physical page number and some permission bits
    - Assume the size of PTE is about 4 bytes
      Page table size = 2^18 * 4 = 1 MB in the worst case
    - Complicates memory allocation
  - If virtual, then need to keep the page table inaccessible to the user program!
  - Otherwise, user program could modify the entries to point to other programs' memory

Paged Page Tables

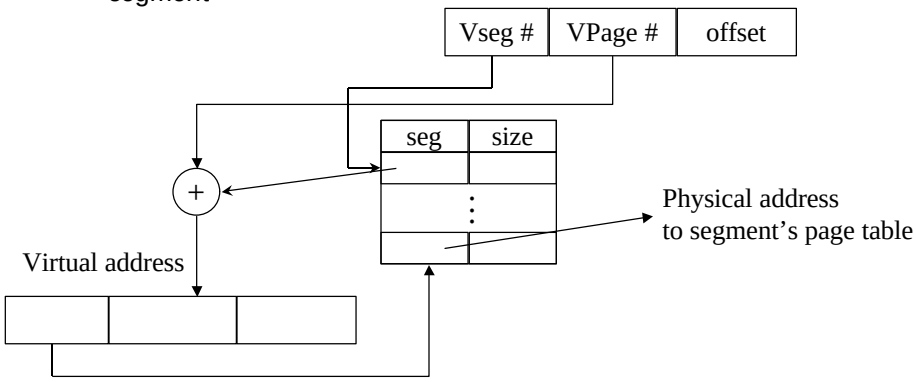
- Can we page them?
  - That is, can we replace page table pointers with virtual addresses
  - Implication: they can be allocated in page granularity



Generated virtual address:
PTPtr + 4 * Vpage #

Paged Page Tables

- Since it is a virtual address, it must be translated again
  - Need to lookup the segment table to do translation
  - To break "recursion," make one of the segment table entries contain a physical pointer to a page table
  - In other words, all the page table of all segments live in a special segment



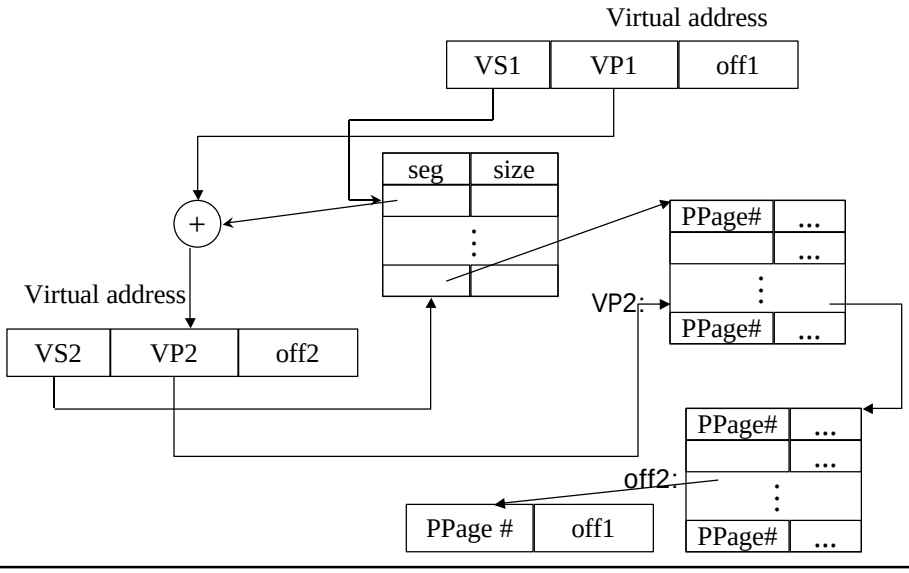
Address Translation Example

- Translate virtual address:
  - Segment: 0x1
  - Page number: 0x02002
  - Offset: 0xDEF

Segment table

0x3	0x00000	0x000
0x3	0x00100	0x000
0x3	0x00200	0x000
0x0	0x04010	0x000

Paged Page Tables



Paged Page Tables

```
TranslateAddr(vaddr) {
        <seg1, page1, offset1> = vaddr;
        PTPtr = LookupSegmentTable(seg1);
        PTEPtr = PTPtr + page1 * sizeof(PTE);
        if (PTEPtr is virtual)
              PTEPtr = TranslateAddr(PTEPtr);
        PPageNumber = *PTEPtr;  // ignore permission bits
        paddr = PPageNumber*page_size + offset1;
        return paddr;
}
```

Back of the envelope calculations

- Assume: 4 segments, 4KB pages, 18 bits to encode page number
- Maximum extent of each segment = 2^18 pages
- Maximum size of page table of each segment = 4*2^18 = 1MB = 256 pages
- 3*256 pages are stored in the system segment and there is a page table with 3*256 entries to address these pages
- Note that 3*256 entries fits into a single page

- Sample layout: 3*256 pages (representing page tables) are stored at the beginning of the system segment
  - System page table would have pointers to these pages

Cache concept

- **Cache**: copy that can be accessed more quickly than the original.

- **Idea**: make frequent case efficient, infrequent case does not matter as much

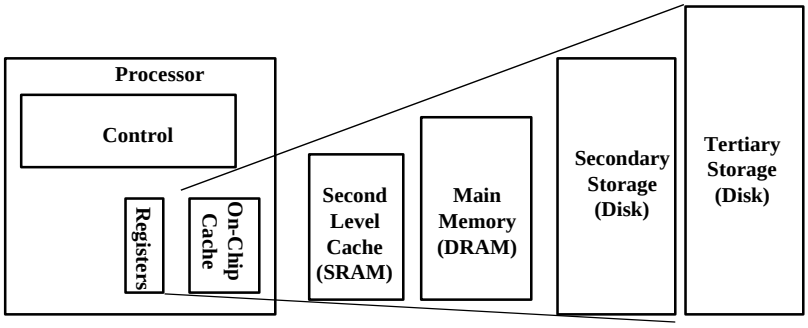
- Caching is a fundamental concept; used widely in:
  - page translations
  - memory locations
  - file blocks
  - network routes
  - authorizations for security systems

Generic Issues in Caching

- Cache hit: item is in cache
- Cache miss: item is not in cache, have to do full operation
- Effective access time = P(hit) * cost of hit + P(miss) * cost of miss
- Issues:
  - How do you find whether the item is in the cache or not?
  - If not in the cache, how do you choose what to replace from cache to make room?
  - Consistency – how do you keep cache copy consistent with real version?

Memory Hierarchy

- Two principles:
  - Smaller the amount of memory, faster it can be accessed
  - Larger the amount of memory, cheaper per byte



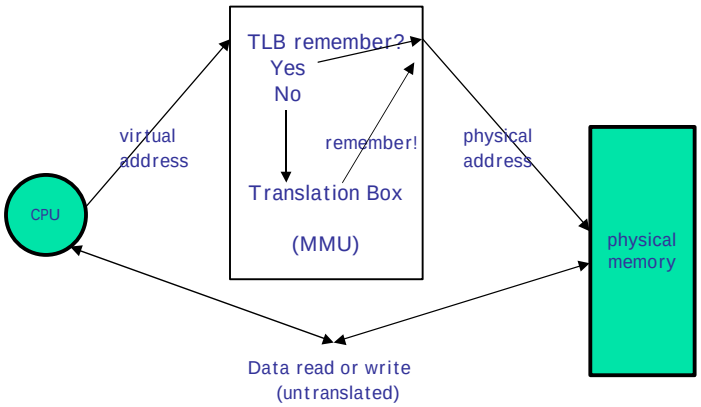
Speed (ns):	1s	10s		100s	10,000,000s (10s ms)	10,000,000,000s (10s sec)
Size (bytes):	100s	Ks		Ms	Gs	Ts

Why caching works?

- Present the user with as much memory as is available in the cheapest technology
- Provide access at the speed offered by the fastest technology
- By taking advantage of the principle of locality
  - Temporal locality: will reference same locations as accessed in the recent past
  - Spatial locality: will reference locations near those accessed in the recent past

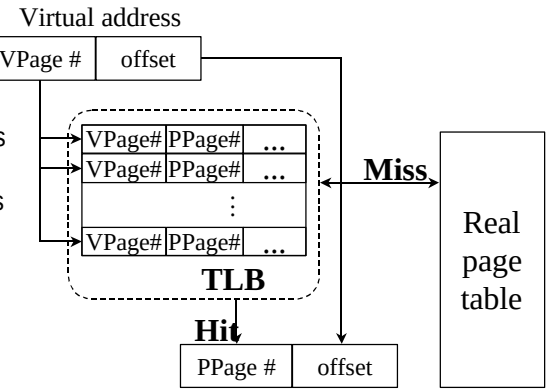
Caching applied to address translation

- Often reference same page repeatedly, why go through entire translation each time ?
- Use Translation Look-aside Buffer (TLB)



Translation look-aside buffer

- Typically on chip, access time of a few ns instead of several hundred of ns for main memory



- Example:

virtual page #	physical page #	control bits
2	1	valid, rw
-	-	invalid
0	4	valid, rw

Issues

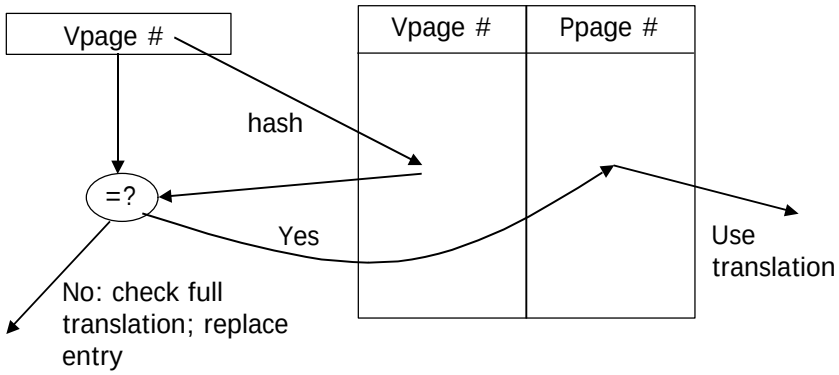
- Main questions
  - How do we tell if needed translation is in TLB?
  - How do we choose which item to replace?
  - What happens at context switch?
  - What if the page table changes? (consistency)

How to tell if needed translation is in TLB?

Option 1: Search table in sequential order

Option 2: “direct mapped”

- Restrict each virtual page to use specific slot in TLB

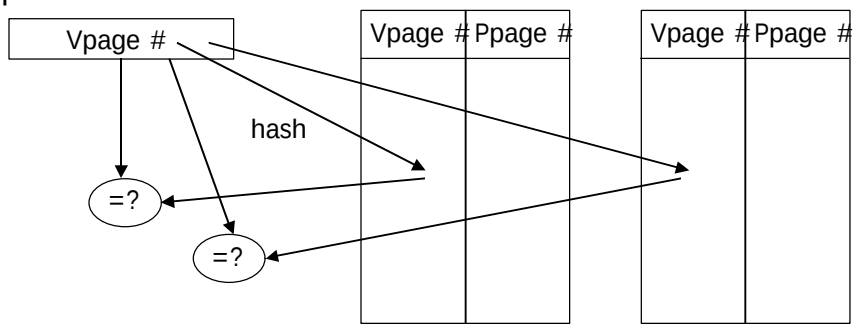


Hash functions

- What is a good hash function?
  - Table entry = (Vpage# / NUM_TLB_ENTRIES)
  - Table entry = (Vpage# % NUM_TLB_ENTRIES)

Other strategies

Option 3: Set associative



Option 4: Check all entries of TLB in parallel

- Requires more hardware
- Translation can be store anywhere in TLB
- Referred to as “fully associative”

How do we choose which item to replace?

- For direct mapped, never any choice as to which item to replace.
  - But for set associative or fully associative cache have a choice
- What policy?
  - Least recently used?
  - Random?
  - Most recently used?
- In hardware: often choose item to replace randomly
  - Simple and fast
- In software: do something more sophisticated
  - Tradeoff: spend CPU cycles to try to improve cache hit rate

Consistency between TLB and page tables

- What happens on context switch?
- Have to invalidate entire TLB contents
  - When new program stats running, will bring in new translations
  - Alternatively, include process id in TLB comparator
- What if translation tables change?
  - For example, to move page from memory to disk
  - Have to invalidate TLB entry