Address Translation

Tore Brox-Larsen

Based on and including material developed by:
Kai Li, Andy Bavier, Princeton University

Topics

- Virtual memory
  - Virtualization
  - Protection
- Address translation
  - Base and bound
  - Segmentation
  - Paging
  - Translation look-ahead buffer (TLB)

The Big Picture

- DRAM is fast, but relatively expensive
- Disks are inexpensive, but slow
  - 100x less expensive
  - 100.000x longer latency
  - 1000x less bandwidth
- Goals
  - Run programs as efficiently as possible
  - Make the system as safe as possible

Issues

- Many processes running concurrently
  - The more processes the system can handle, the better
- Address space may exceed memory size
  - Many small processes whose total size may exceed memory
  - Even one large may exceed physical memory size
- Address space may be sparsely used
- Protection
  - User processes should not crash the system
  - User processes should be protected from each other
- Location transparency

Strategies

- **Size**: Can we use slow disks to “extend” the size of available memory?
  - Disk accesses must be rare in comparison to memory accesses so that each disk access is amortized over many memory accesses
- **Location**: Can we device a mechanism that delays the bindings of program address to memory location? Transparency and flexibility.
- Process **protection**: Must check access rights for every memory access
- **Sparsity**: Can we avoid reserving memory for non-used regions of address space?

Expansion - Location Transparency Issue

- Each process should be able to run regardless of location in memory
- Regardless of memory size?
- Dynamically relocateable?
- Memory fragmentation
  - External fragmentation – Unused area between processes
  - Internal fragmentation – Unused area within processes
- Approach
  - *Give each process large "virtual (fake)" address space*
  - *Relocate each memory access to actual memory address*

Protection Issue

- Errors in one process should not affect other processes
- For every process, we need to enforce that every load or store is to “legal” regions of memory only

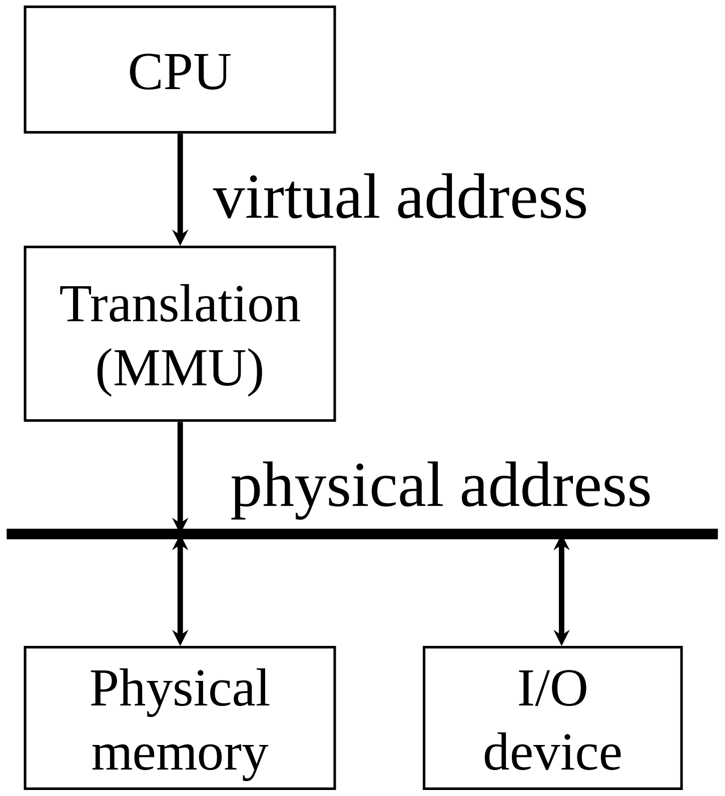
Virtual Memory

- Use secondary storage
  - Extend expensive DRAM with reasonable performance
- Flexibility
  - Processes may be located anywhere in memory, may be moved while executing, may reside partially in memory and partially on disk
- Efficient
  - 20/80 rule: 20 % of memory gets 80 % of references
  - Keep the 20 % in physical memory
- Convenience
  - Make sure memory address scheme fits programmer’s needs

Virtual Memory Design issues

- How is protection enforced
- How are processes relocated in physical memory
- How is memory partitioned

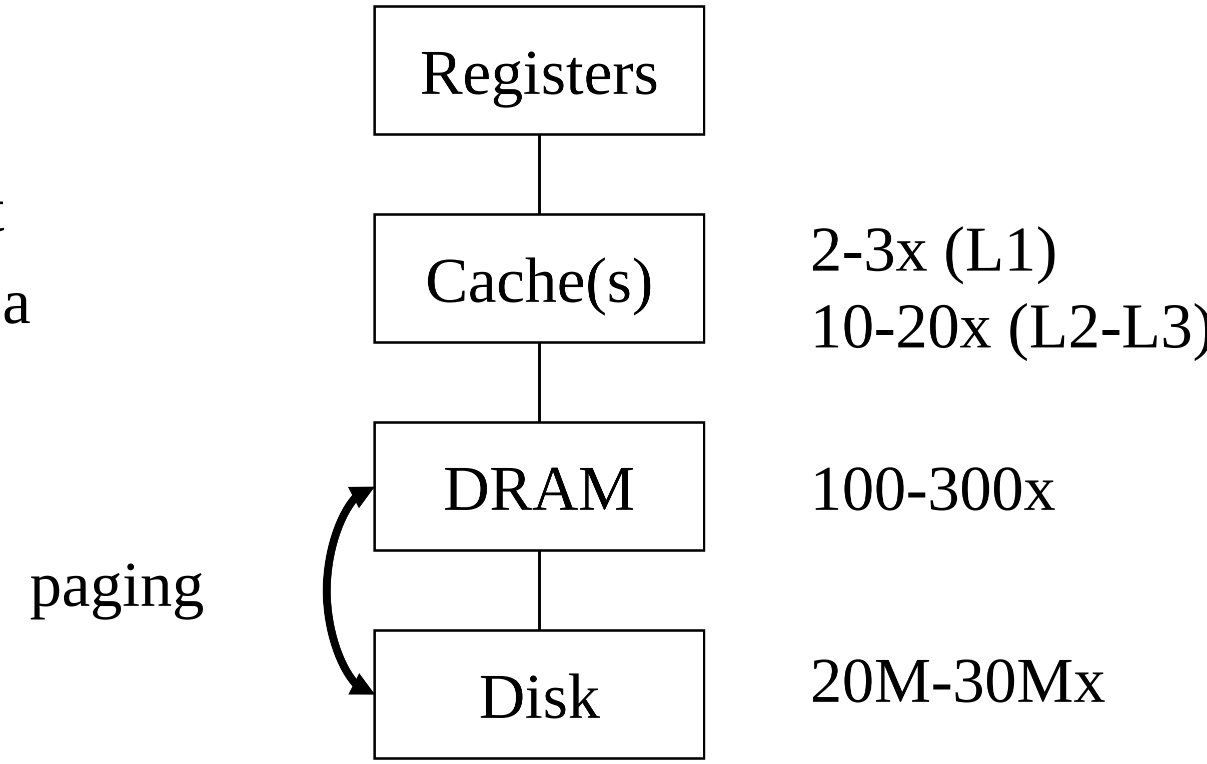
Generic Translation Overview



- Actual translation is in hardware (MMU)
- Controlled in privileged software
- CPU view
  - what program sees, virtual memory
- Memory & I/O view
  - physical memory

Goals of Translation

- Implicit translation for each memory reference
- A hit should be very fast
- Trigger an exception on a miss
- Protected from user's faults



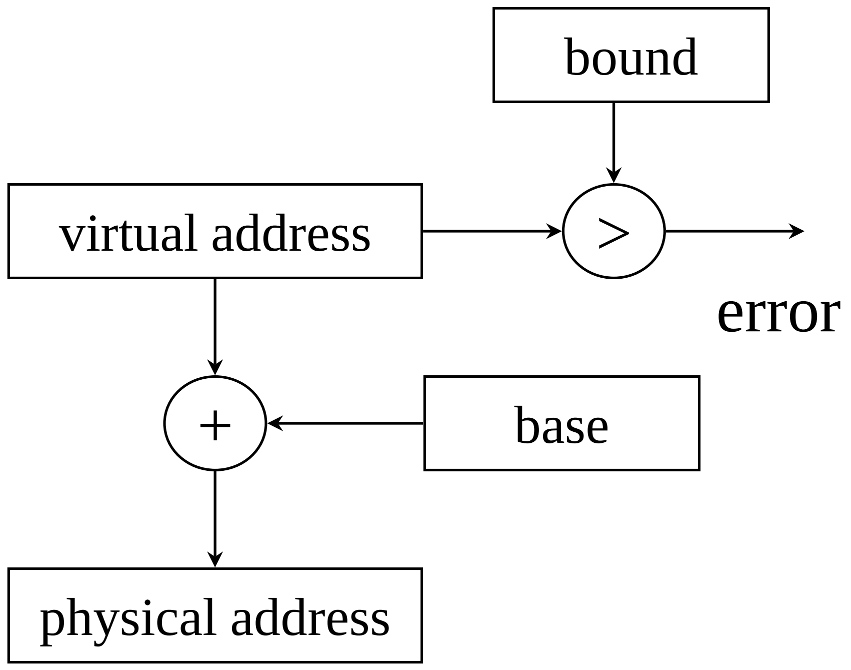
Address Mapping Granularity

- Mapping mechanism
  - Virtual addresses are mapped to DRAM addresses or onto disk
- Mapping granularity?
  - Increased granularity
    - Increases flexibility
    - Decreases internal fragmentation
    - Requires more mapping information & Handling
- Extremes
  - Any byte to any byte: Huge map size
  - Large segments: Smaller maps, internal fragmentation

Locality of Reference

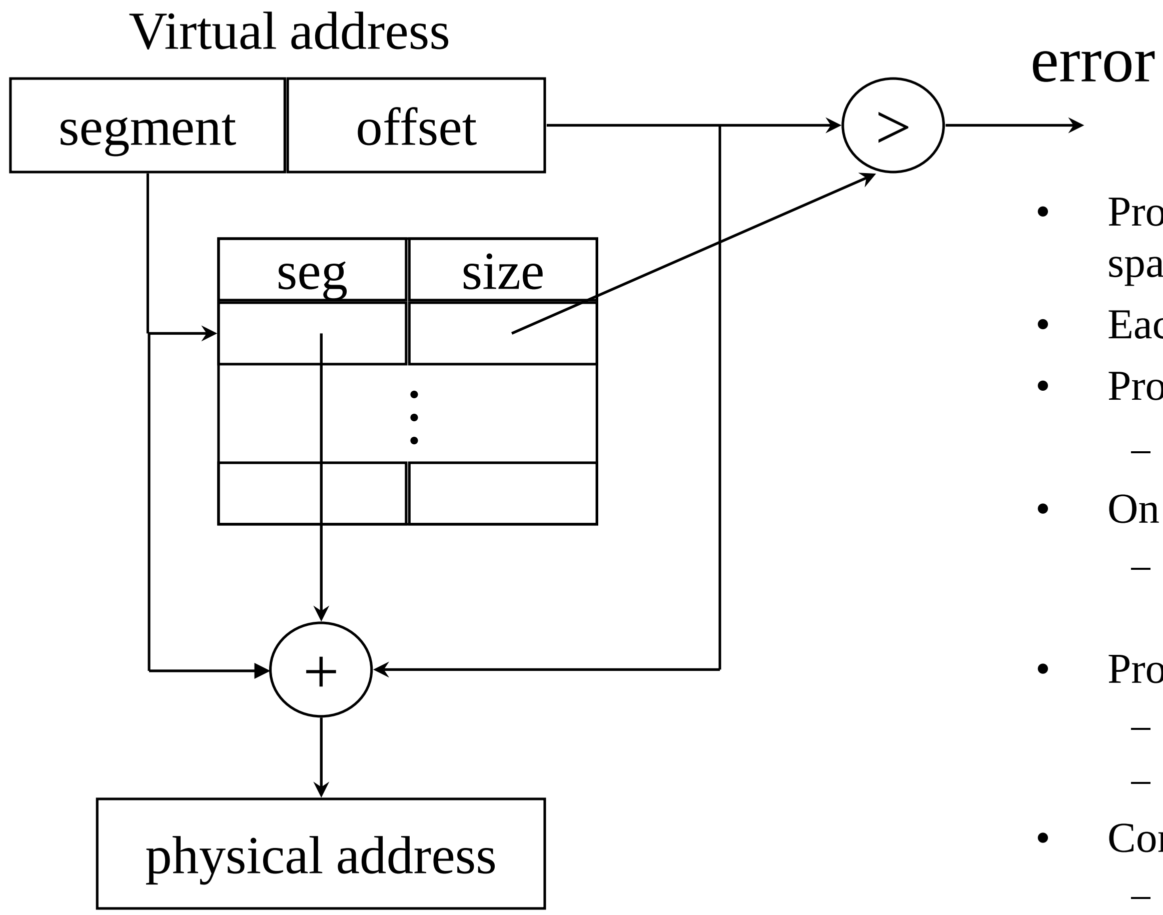
- Behaviors exhibited by most programs
- Locality in time
  - *When an item is addressed, it is likely to be addressed again shortly*
- Locality in space
  - *When an item is addressed, its neighboring items are likely to be addressed shortly*
- Basis of caching
- Argues that recently accessed items should be cached together with an encompassing region; A block (or line)
- 20/80 rule: 20 % of memory gets 80 % of references
- Keep the 20 % in memory

Base and Bound



- Built in Cray-1 (1976)
- Protection
  - A program can only access physical memory in [base, base+bound]
- On a context switch:
  - Save/restore base, bound registers
- Pros
  - Simple
  - Flat address
- Cons:
  - Fragmentation
  - Difficult to share
  - Difficult to use disks

Segmentation

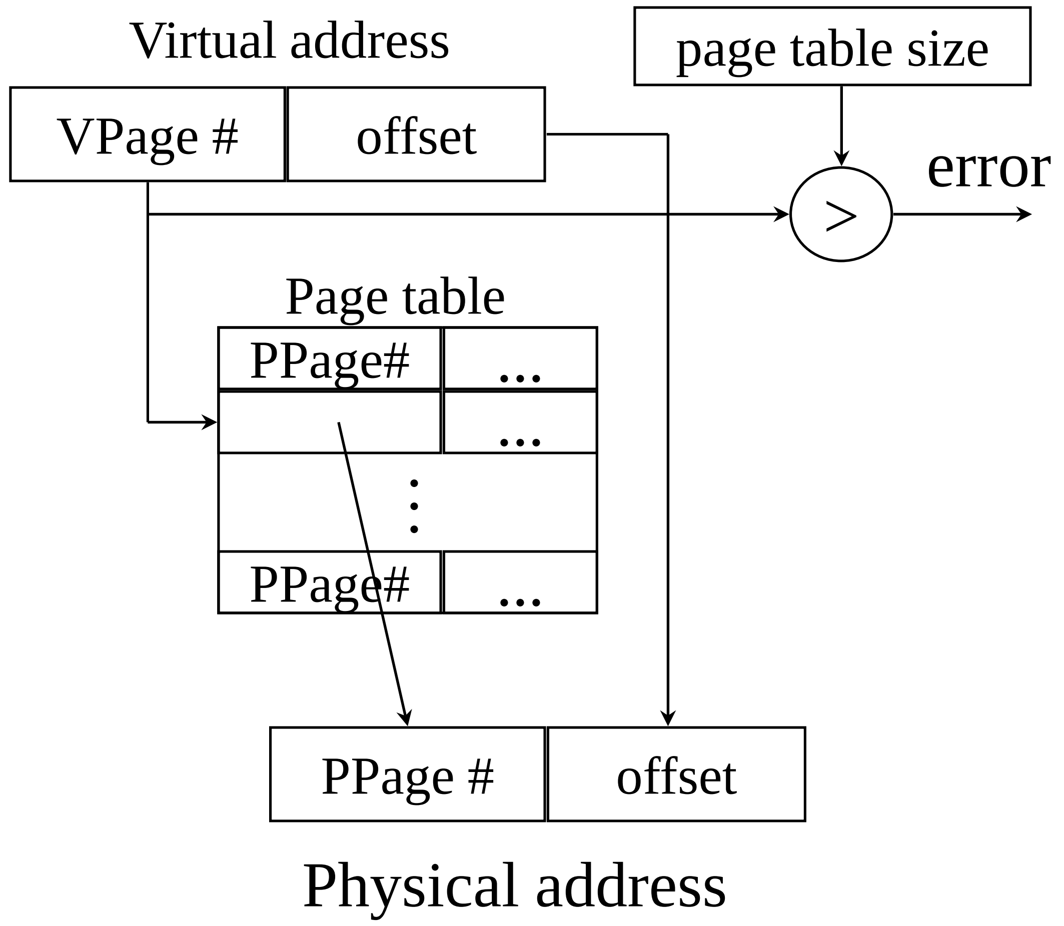


- Provides multiple separate virtual address spaces (segments)
- Each process has a table of (seg, size)
- Protection
  - Each entry has (nil,read,write)
- On a context switch
  - Save/restore the table or a pointer to the table in kernel memory
- Pros
  - Efficient
  - Easy to share
- Cons:
  - Complex management
  - Fragmentation within a segment

Historical Computers Applying Segmentation Schemes

- Burroughs B5000,
- GE645 (Multics)
- Intel iAPX 432
- IBM System/38
- IBM AS/400
- All above designs attempted to provide memory model (and other features) more directly supporting programming structures
- See Glenford J. Myers, Advances in Computer Architecture, for a proponent description of this historic design trend

Paging

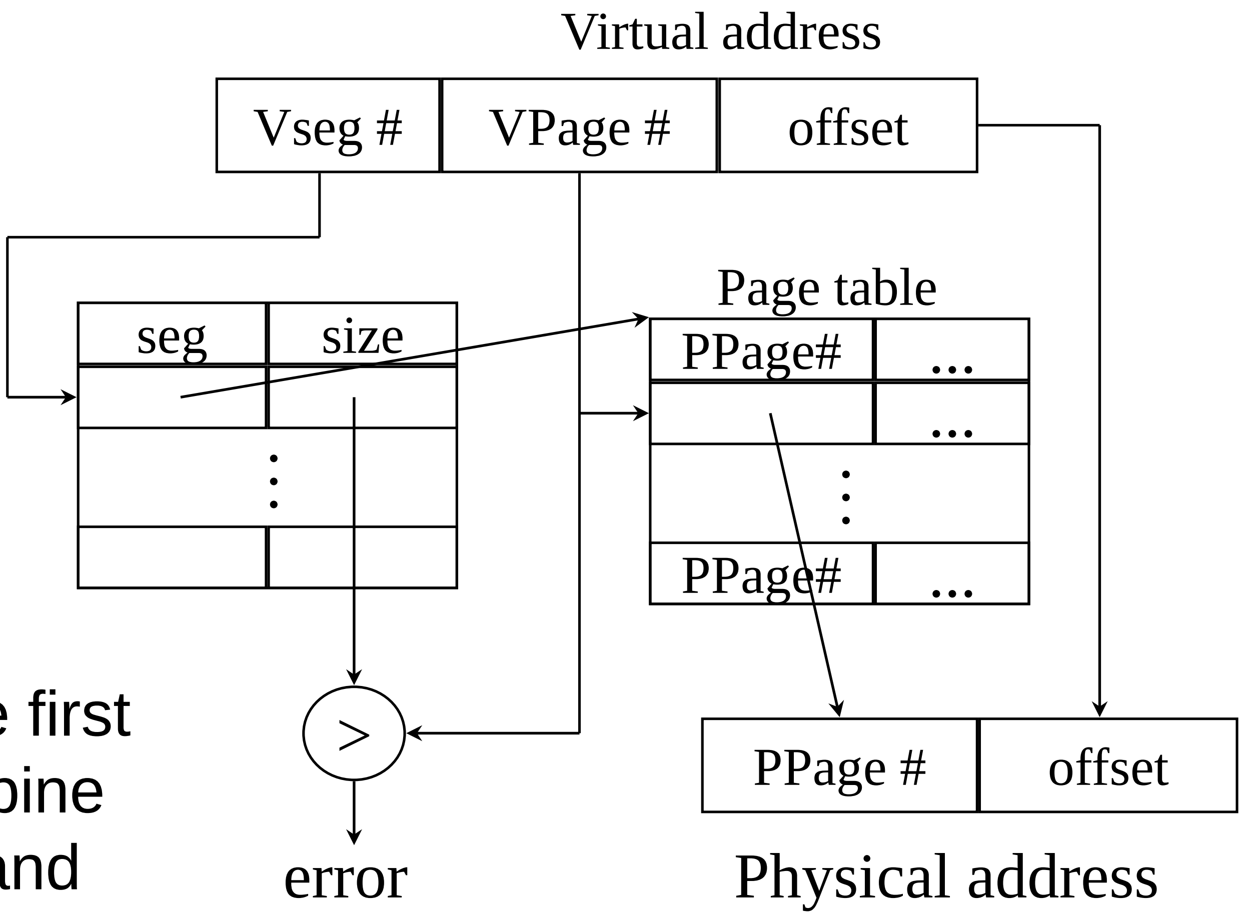


- Use a fixed size unit called page
- Pages not visible from program
- Use a page table to translate
- Various bits in each entry
- Context switch
  - Similar to the segmentation scheme
- What should be the page size?
- Pros
  - Simple allocation
  - Easy to share
- Cons
  - Big page tables
  - How to deal with holes?

How Many PTEs Do We Need?

- Assume 4KB page size
  - 12 bit (low order) displacement within page
  - 20 bit (high order) page#
- Worst case for 32-bit address machine
  - # of processes × 2^{20}
  - 2^{20} PTEs per page table (~4MBytes). 10K processes?
- What about 64-bit address machine?
  - # of processes × 2^{52}
  - Page table won't fit on disk (2^{52} PTEs = 16PBytes)

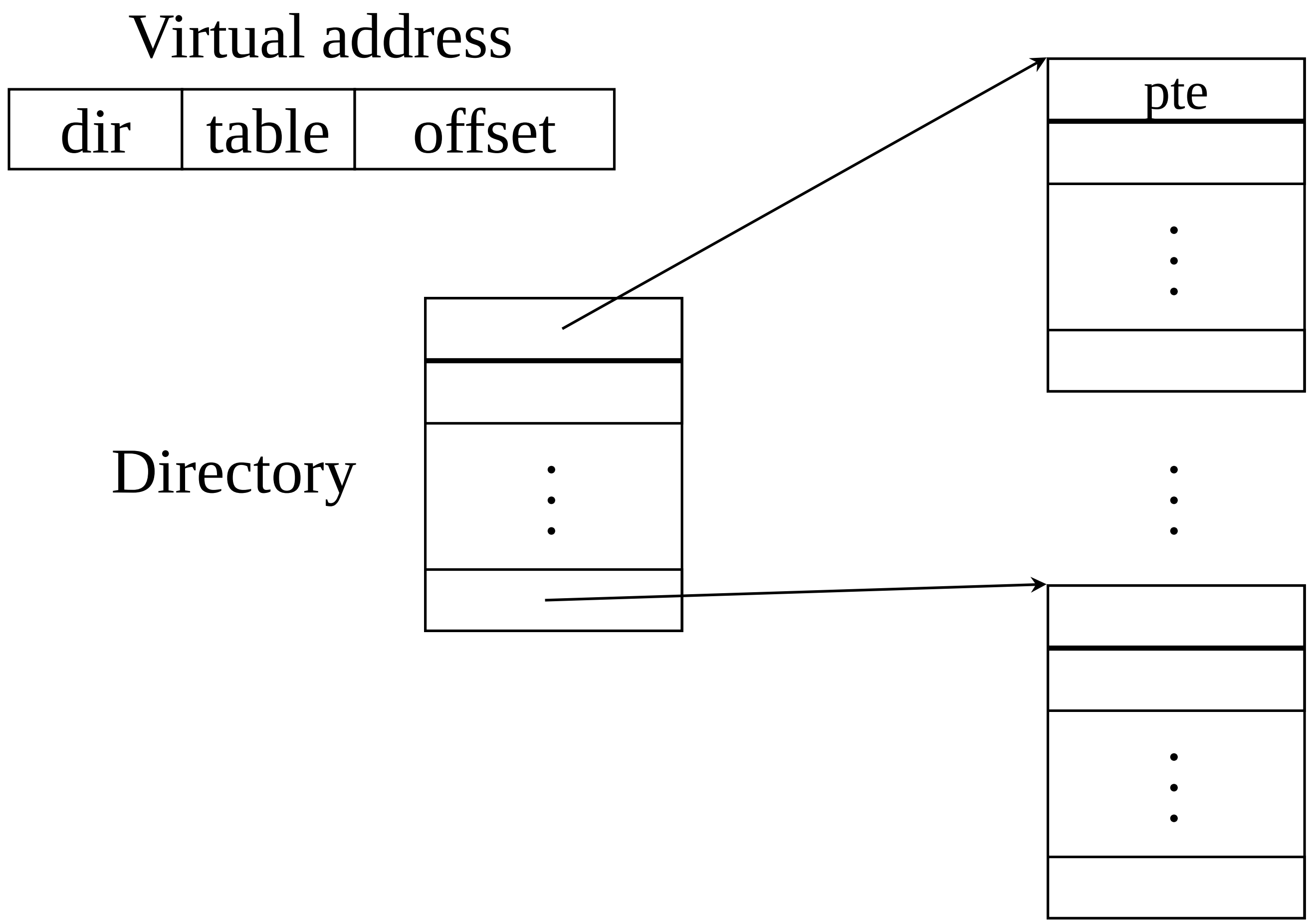
Segmentation with Paging



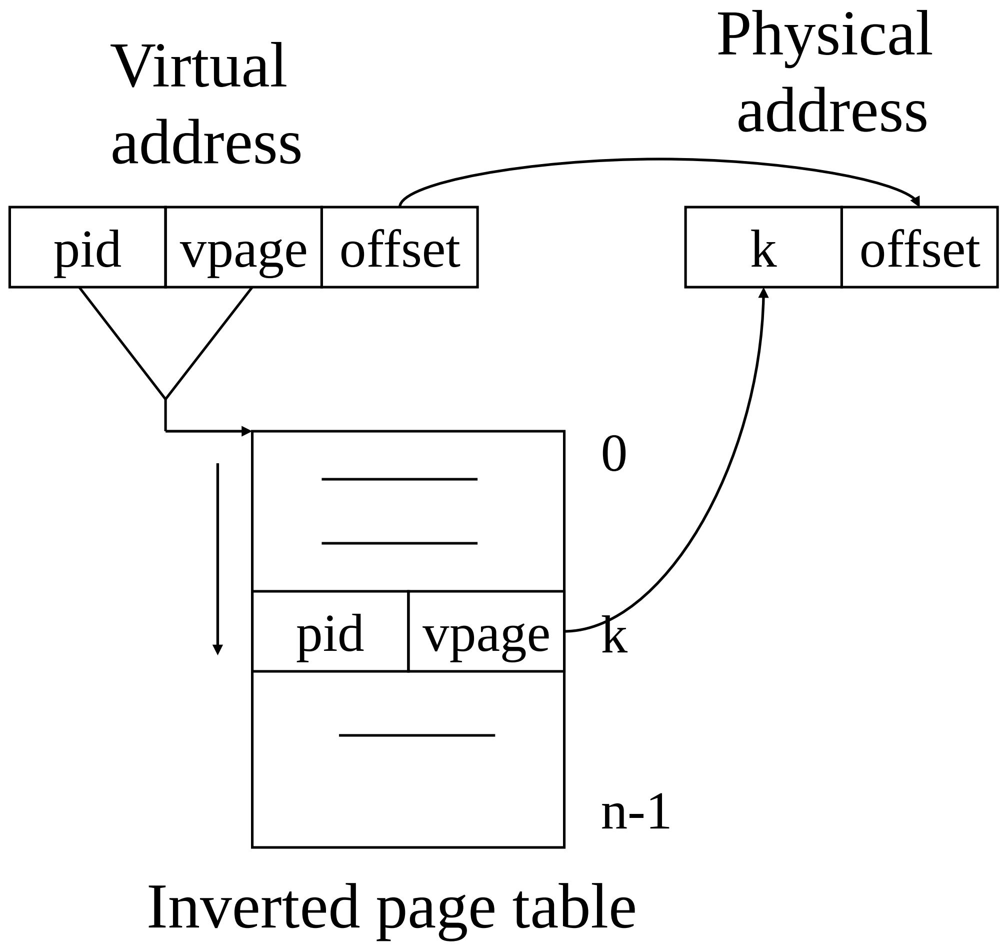
Multics was the first system to combine segmentation and paging.

www.multicians.org

Multiple-Level Page Tables



Inverted Page Tables

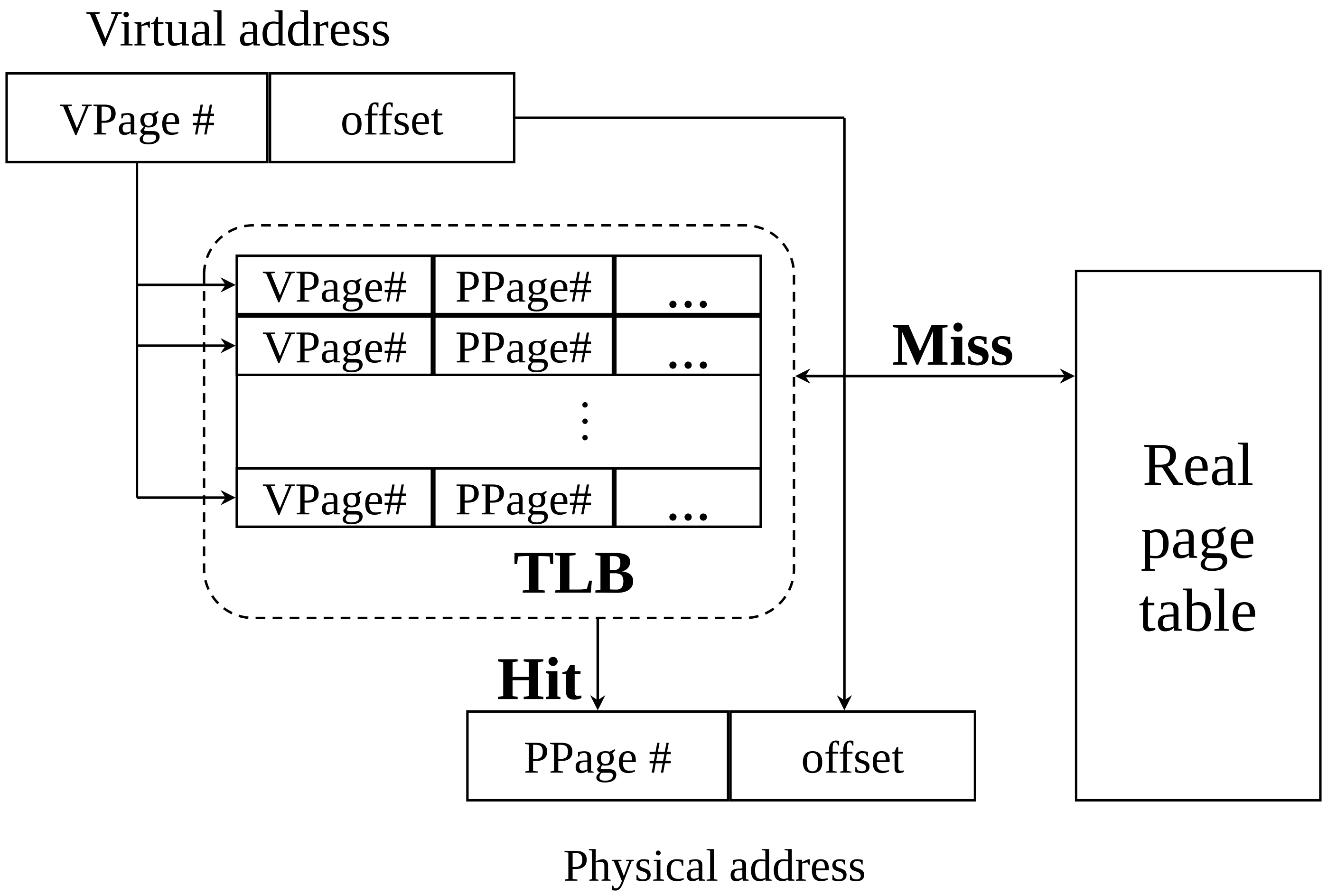


- Main idea
  - One PTE for each physical page frame
  - Hash (Vpage, pid) to Ppage#
- Pros
  - Small page table for large address space
- Cons
  - Lookup is difficult
  - Overhead of managing hash chains, etc

Virtual-To-Physical Lookup

- Program only knows virtual addresses
  - Each process goes from 0 to highest address
- Each memory access must be translated
  - Involves walk-through of (hierarchical) page tables
  - Page table is in memory
    - An extra memory access for each memory access???
- Solution
  - Cache part of page table (hierarchy) in fast associative memory – Translation-Lookahead-Buffer (TLB)
  - Introduces TLB hits, misses etc.

Translation Look-aside Buffer (TLB)



Bits in A TLB Entry

- Common (necessary) bits
  - Virtual page number: match with the virtual address
  - Physical page number: translated address
  - Valid
  - Access bits: kernel and user (nil, read, write)
- Optional (useful) bits
  - Process tag
  - Reference
  - Modify
  - Cacheable

Hardware-Controlled TLB

- On a TLB miss
  - Hardware loads the PTE into the TLB
    - Need to write back if there is no free entry
  - Generate a fault if the page containing the PTE is invalid
  - VM software performs fault handling
  - Restart the CPU
- On a TLB hit, hardware checks the valid bit
  - If valid, pointer to page frame in memory
  - If invalid, the hardware generates a page fault
    - Perform page fault handling
    - Restart the faulting instruction

Software-Controlled TLB

- On a miss in TLB
  - Write back if there is no free entry
  - Check if the page containing the PTE is in memory
  - If not, perform page fault handling
  - Load the PTE into the TLB
  - Restart the faulting instruction
- On a hit in TLB, the hardware checks valid bit
  - If valid, pointer to page frame in memory
  - If invalid, the hardware generates a page fault
    - Perform page fault handling
    - Restart the faulting instruction

Hardware vs. Software Controlled

- Hardware approach
  - Efficient
  - Inflexible
  - Need more space for page table
- Software approach
  - Flexible
  - Software can do mappings by hashing
    - PP# $\rightarrow$ (Pid, VP#)
    - (Pid, VP#) $\rightarrow$ PP#
  - Can deal with large virtual address space

Cache vs. TLB

- Similarity
  - Both are fast and expensive with respect to capasity
  - Both cache a portion of memory
  - Both write back on a miss
- Differences
  - TLB is usually fully set-associative
  - Cache can be direct-mapped
  - TLB does not deal with consistency with memory
  - TLB can be controlled by software
- Logically TLB lookup appears ahead of cache lookup, careful design allows overlapped lookup
- Combine L1 cache with TLB
  - Virtually addressed cache
  - Why wouldn’t everyone use virtually addressed cache?

TLB Related Issues

- What TLB entry to be replaced?
  - Random
  - Pseudo LRU
- What happens on a context switch?
  - Process tag: change TLB registers and process register
  - No process tag: Invalidate the entire TLB contents
- What happens when changing a page table entry?
  - Change the entry in memory
  - Invalidate the TLB entry

Consistency Issue

- Snoopy cache protocols
  - Maintain cache consistency with DRAM, even when DMA happens
- Consistency between DRAM and TLBs:
  - You need to flush (SW) related TLBs whenever changing a page table entry in memory
- Multiprocessors need TLB "shootdown"
  - When you modify a page table entry, you need to do to flush ("shootdown") all related TLB entries on every processor

Summary

- Virtual memory
  - Easier SW development
  - Better memory utilization
  - Protection
- Address translation
  - Base & bound: Simple, but limited
  - Segmentation: Useful but complex
- Paging: Best tradeoff currently
  - TLB: Fast translation
  - VM needs to handle TLB consistency issues