CSE380 - Operating Systems

Notes for Lecture 12 - 10/25/2005

(some examples by Insup Lee)

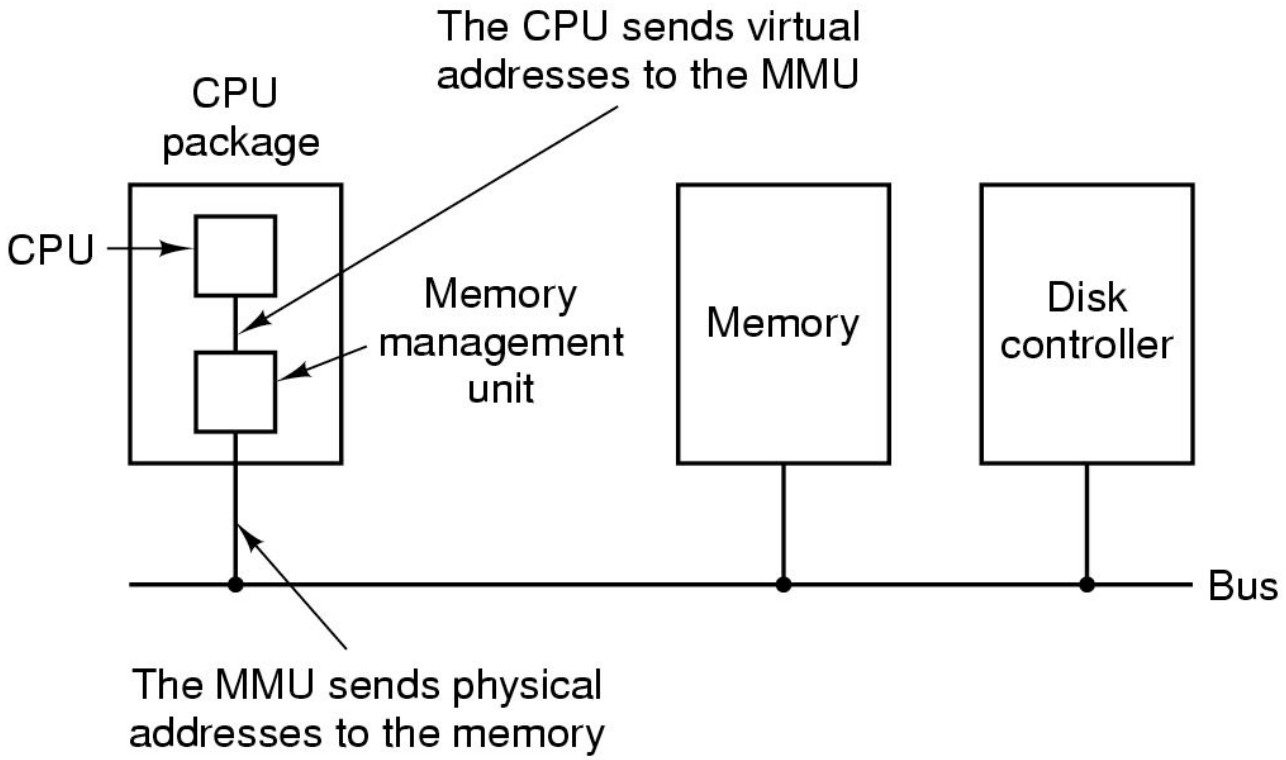
Issues in Virtual Memory

- How are virtual addresses structured?
  - paging, segmentation
- What is the interface to the MMU?
  - loading page table
  - dealing with references to unmapped addresses
  - TLBs
- What data structures maintain the mapping to physical addresses?
- How to decide what virtual addresses get to reside in physical memory
  - *page replacement rules*

Virtual Addresses

- Virtual memory gives a complete separation of logical and physical address-spaces
- Today, typically a virtual address is 32 bits
  - This allows a process to have 4GB of virtual memory
  - Physical memory is usually smaller than this, and varies from machine to machine
  - Virtual address spaces of different processes are distinct
- Structuring of virtual memory
  - Paging: Divide the address space into fixed-size pages
  - Segmentation: Divide the address space into variable-size segments (corresponding to logical units)

Paging Architecture



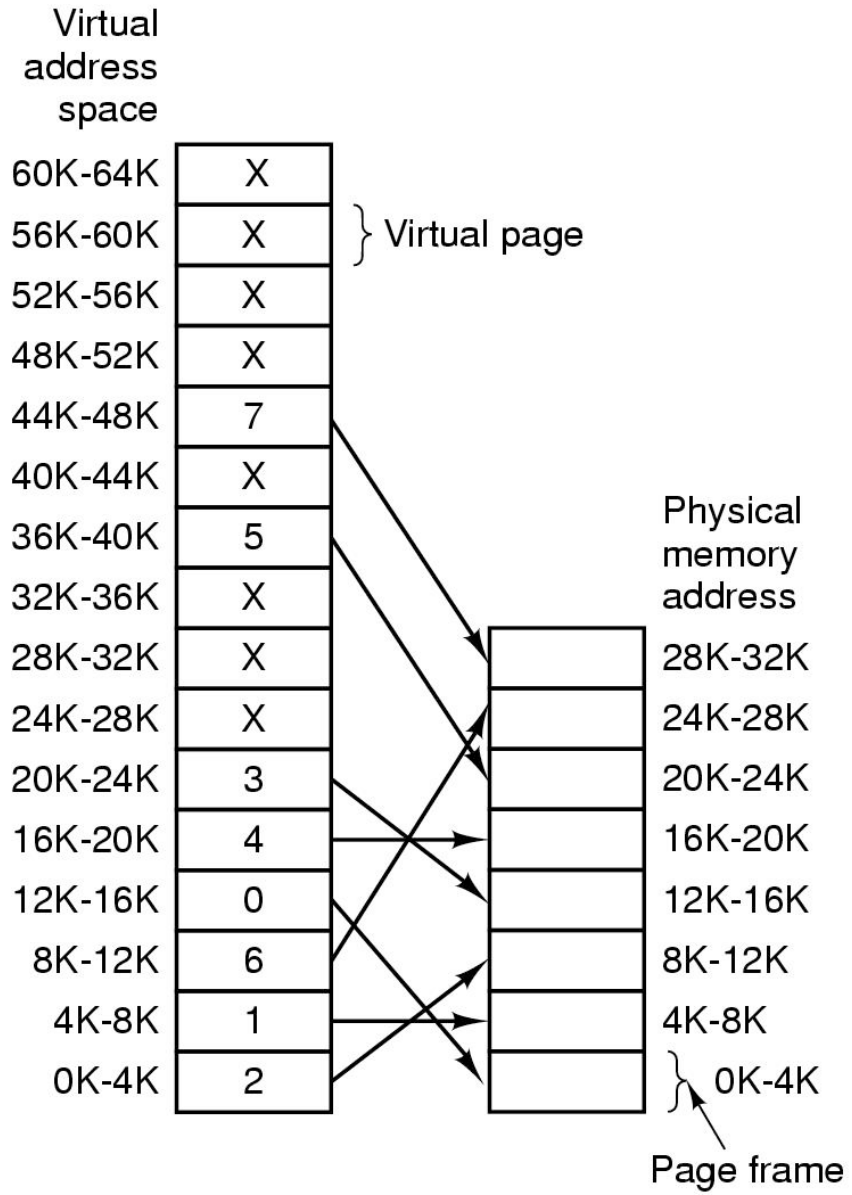
The position and function of the MMU

Paging

- Physical memory is divided into chunks (page-frames)
  - for example, on the Pentium, each page-frame is 4KB
- Virtual memory is divided into chunks called pages
  - size of a page is equal to the size of a page frame
  - therefore, on Pentium, 2^{20} different pages (a little over a million) in virtual memory address space
- OS keeps track of mapping of pages to page-frames
- MMU does the translation in hardware as addresses are referenced
- Remaining bits of address are offset into page-frame
  - offset bits are not translated by MMU

Paging

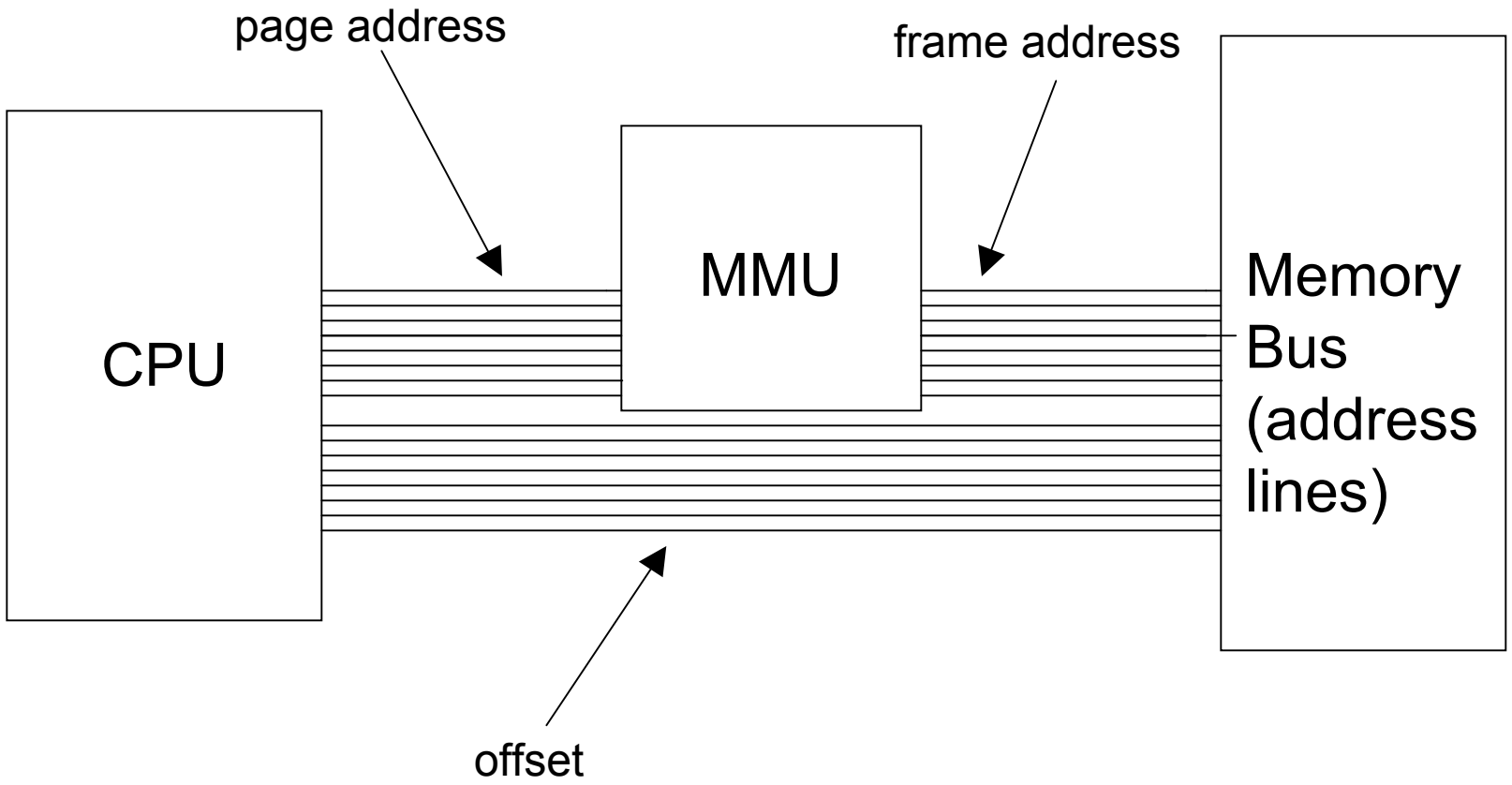
The mapping between virtual addresses and physical addresses is given by the page table



Paging

- A virtual address is really a pair (p,o) of addresses
  - Low-order bits give an offset o within the page
    - this part stays the same after translation to physical address
  - High-order bits specify the page p
    - this is the part that gets translated
  - E.g. If each page is 1KB and virtual address is 16 bits, then low-order 10 bits give the offset and high-order 6 bits give the page number
- The job of the Memory Management Unit (MMU) is to translate the page number p to a frame number f
  - The physical address is then (f,o)
    - this is what goes on the memory bus
- For every process, there is a *page table* (an array)
  - p is just an index into this array for the translation

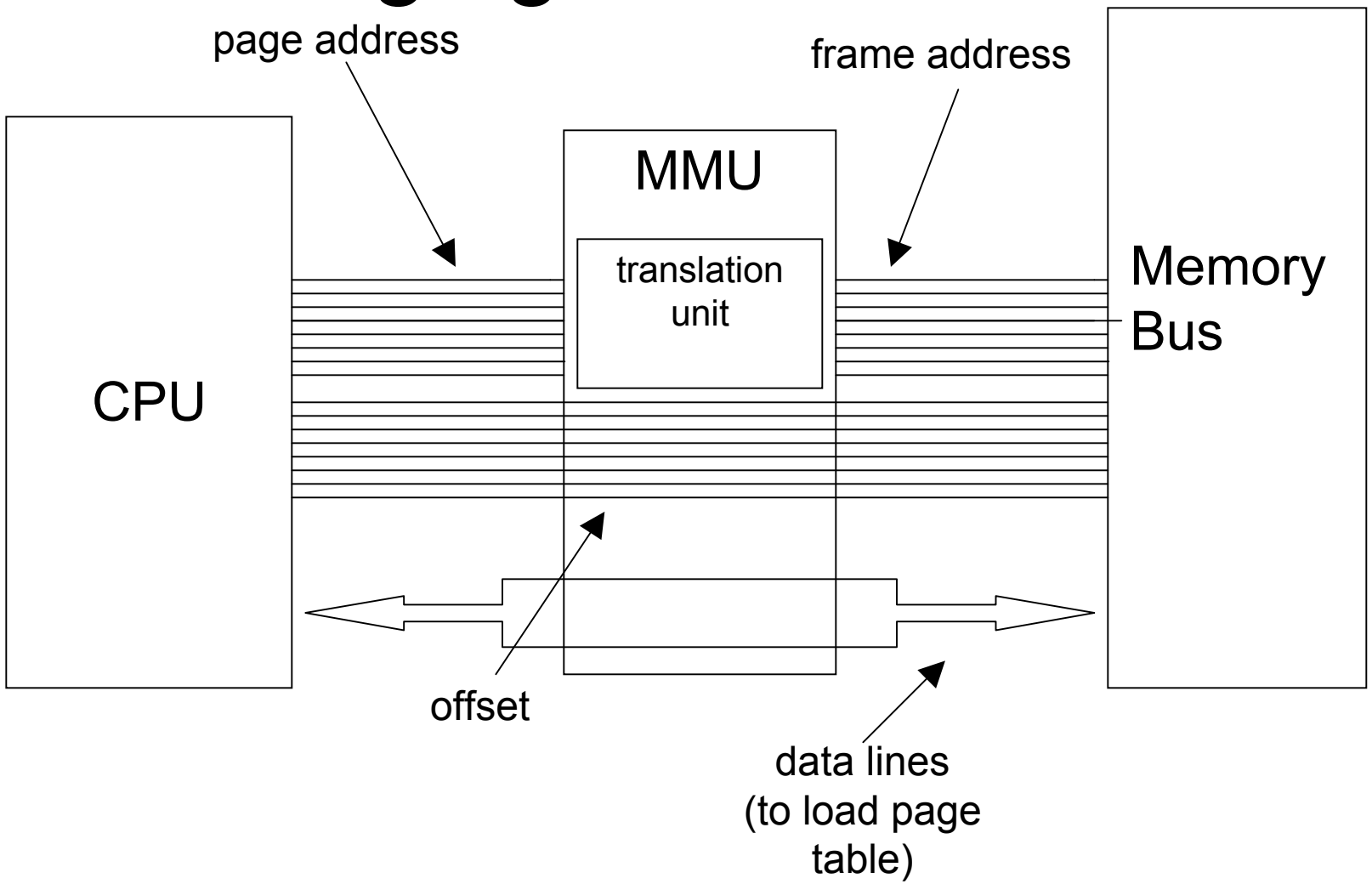
Oversimplified Paging Architecture



Managing the page table

- Page table is pretty big
  - and it's managed by the OS
- Table is stored in main memory itself
- Huh? What? How can this work?
  - OS loads address of page table into MMU
  - MMU reads page table from memory as needed
    - MMU connected to all memory address lines and data lines, not just the page and frame address bits
  - MMU is tightly coupled to CPU architecture
  - Sometimes CPU must wait for MMU to finish

Less Oversimplified Paging Architecture



Typical Page Table Entries

- *Validity* bit
  - set to 0 if the corresponding page is not in memory
- *Frame number*
  - Number of bits required depends on size of physical memory
- *Protection* bits:
  - Read, write, execute accesses
- *Referenced* bit
  - set to 1 by hardware when the page is accessed: used by page replacement policy
- *Modified* bit (*dirty* bit)
  - set to 1 by hardware on write-access: used to avoid writing when swapped out

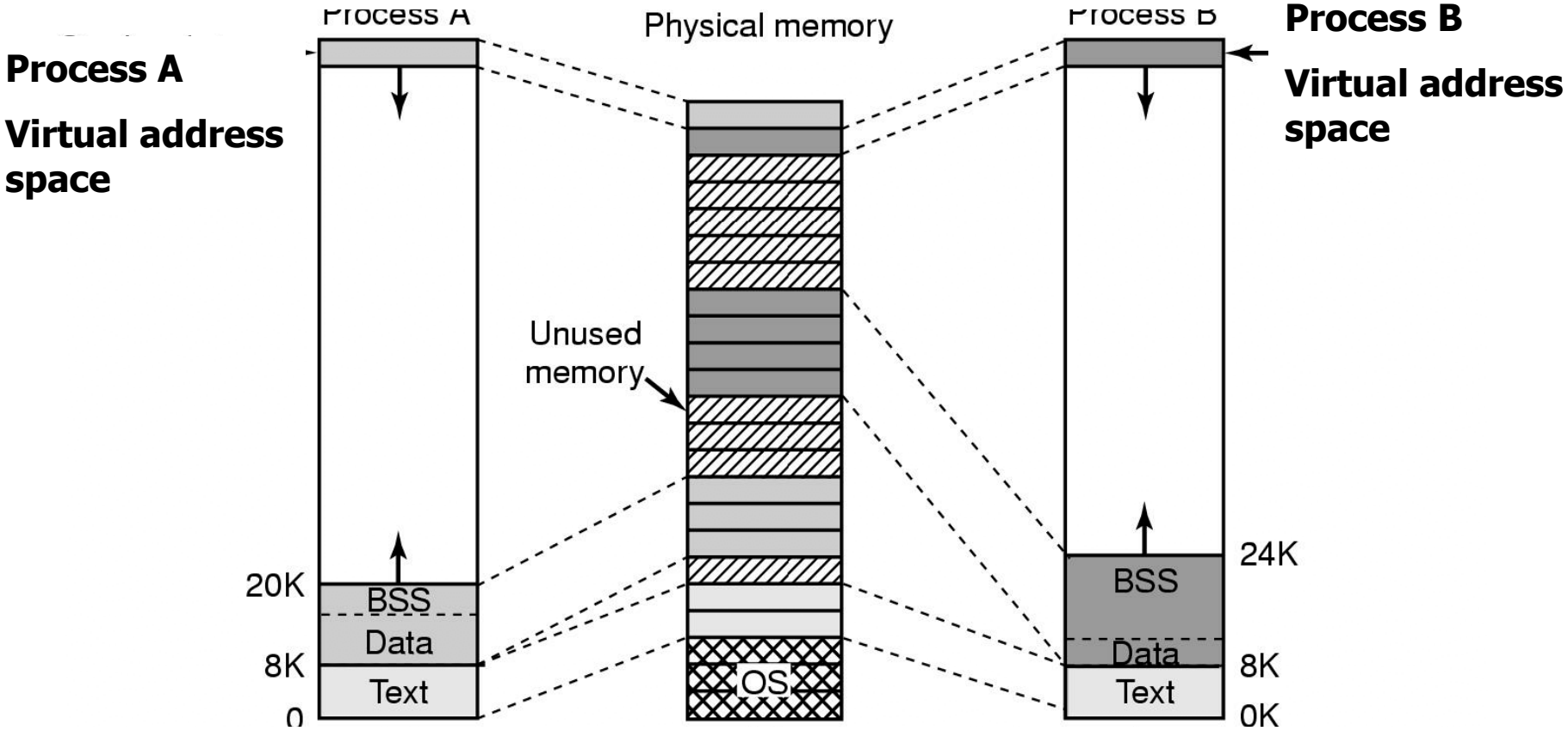
Design Issues

- What is the “optimal” size of a page frame ?
  - Typically 1KB – 4KB, but more on this later
- How to save space required to store the page table
  - With 20-bit page address, there are over a million pages, so the page-table is an array with over million entries
  - Solution: Two-level page tables, TLBs (Translation Lookaside Buffers), Inverted page tables
- What if the referenced page is not currently in memory?
  - This is called a *page fault*, and it traps to kernel
  - On many systems, the OS runs a *page daemon* periodically to ensure that there is enough free memory that new pages can be loaded from disk upon a page fault
- Page replacement policy: how to free physical memory?

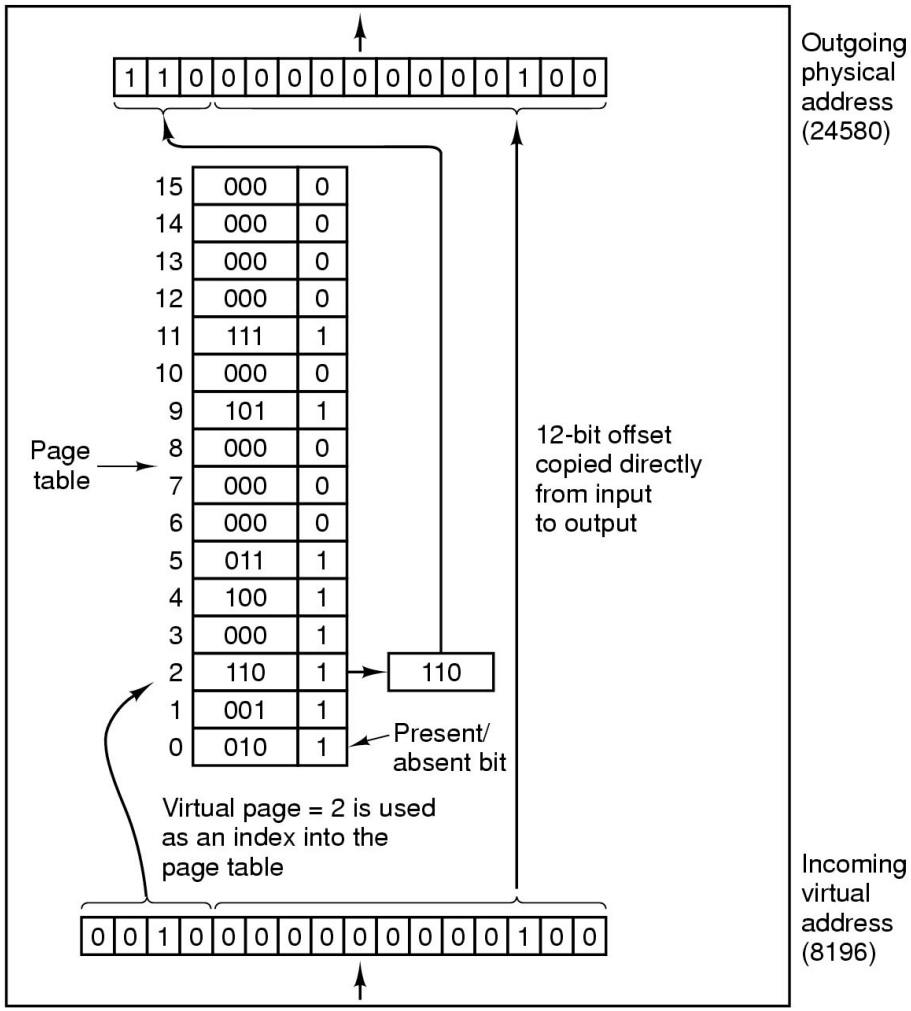
Some properties of VM systems

- Within a page/frame, virtual addresses map to a contiguous section of physical memory
- But the page-frames themselves can map anywhere in physical memory
  - out of order within a process
  - interleaved with pages from other processes

Virtual Memory in Unix



Page Tables

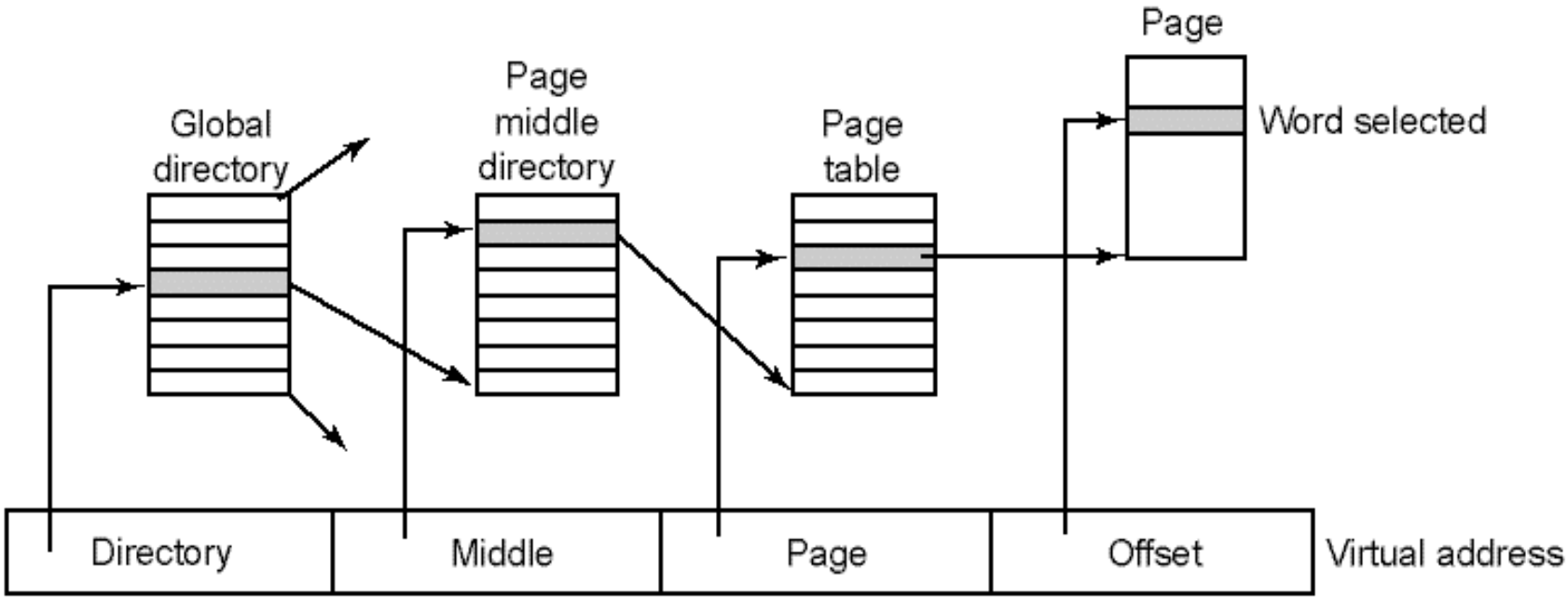


Internal operation of MMU with 16 4KB pages

Multi-Level Paging

- A page-table with 2^{20} entries in memory is pretty big
  - and you need one per process!
- Solution: Make the page table hierarchical
  - Pentium supports two-level paging
- Example: first 10-bits index into a top-level page-entry table T1 (1024 or 1K entries)
  - Each entry in T1 points to another (second-level) page table with 1K entries (4 MB of memory since each page is 4KB)
  - Next 10-bits of physical address index into the second-level page-table selected by the first 10-bits
  - Total of 1K potential second-level tables, but many are likely to be unused
  - If a process uses 16 MB virtual memory then it will have only 4 entries in top-level table (rest will be marked unused) and only 4 second-level tables

Paging in Linux

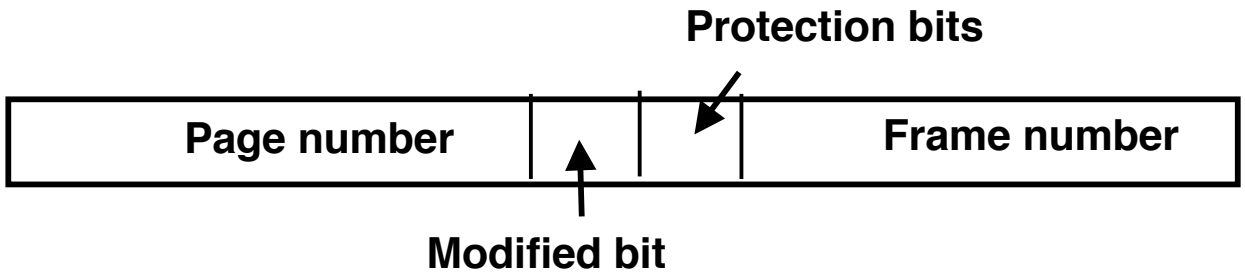


Linux uses three-level page tables

Translation Lookaside Buffers (TLB)

- Problem: page tables are stored in main memory
- Access to main memory is slow compared to clock cycle on CPU (factor of about 10 -- 10ns vs 1 ns)
  - An instruction such as **MOVE REG, ADDR** has to decode **ADDR** and thus go through page tables
  - This would make things very, very slow
- Standard solution: TLB in MMU
  - TLB stores small cache (say, 64) of page table entries to avoid the usual page-table lookup
  - TLB is *associative memory (content addressable memory)* that contains, basically, pairs of the form (page-no, page-frame)
  - Special hardware compares incoming page number in parallel with all entries in TLB to retrieve page-frame
  - If no match found in TLB, we do standard lookup via main memory
- TLB is like a cache for MMU

More on TLBs

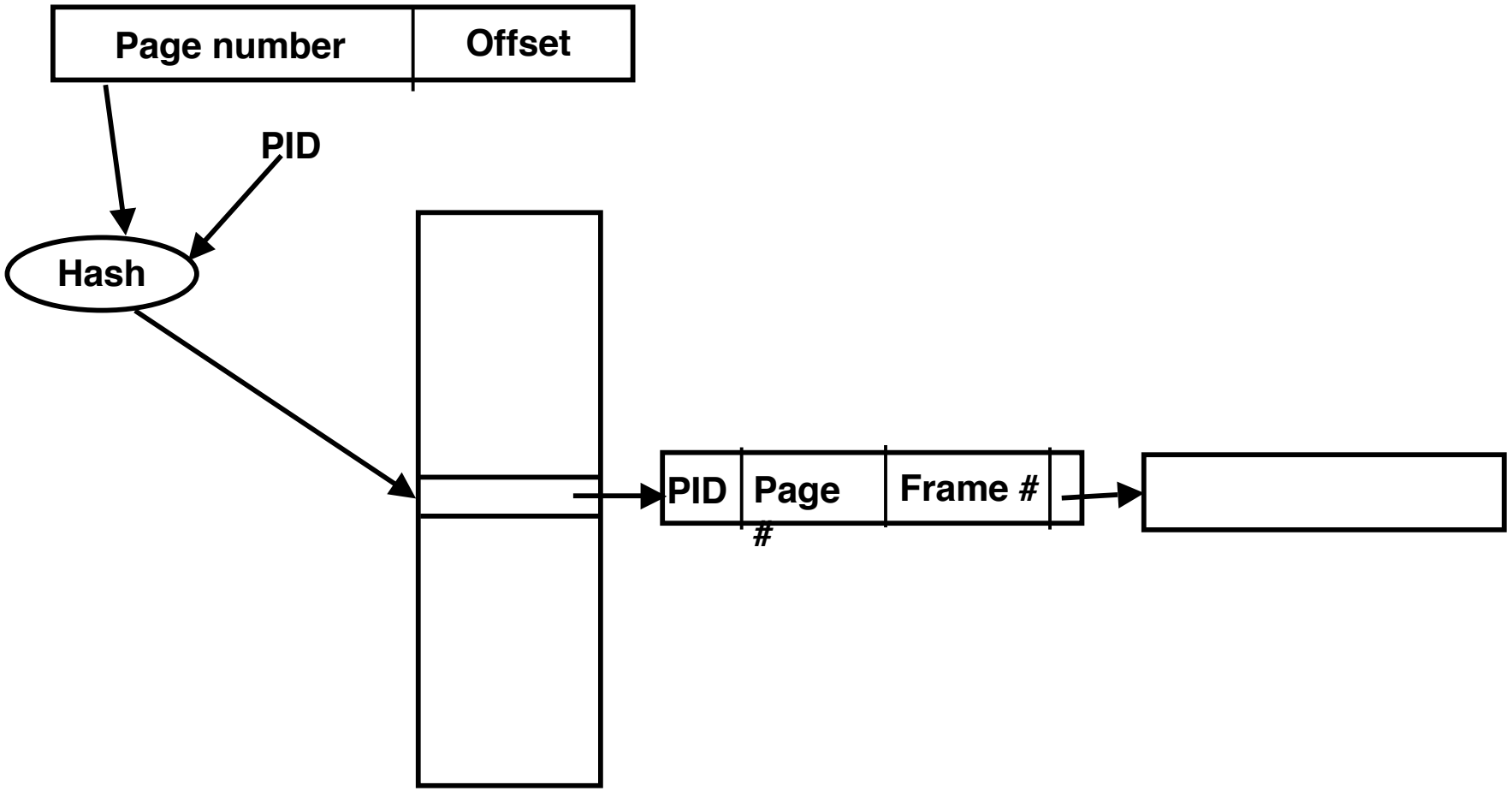


- Main design issue: maximizing TLB hit rate
  - Which pages should be in TLB?
    - ideally, the next ones to be accessed
    - approximated by the most recently accessed
- Also, how to update the TLB?
  - Modern architectures provide hardware support to do this
  - Alternative: TLB miss generates a fault and invokes OS, which then decides how to use the TLB entries effectively.

Inverted Page Tables

- When virtual memory is much larger than physical memory, overhead of storing page-table is high
  - For example, in 64-bit machine with 4KB per page and 256 MB memory, there are just 64K page-frames but 2^{52} pages !
- Solution: Inverted page tables that store entries of the form (page-frame, process-id, page-no)
  - Advantage: now at most 64K entries required
- Problem: Given a page p, how to find the corresponding page frame?
  - Linear search is too slow, so we use hashing
    - now issues like hash-collisions must be handled
- Used in some IBM and HP workstations; will be used more with 64-bit machines

Hashed Inverted Page Tables



Hash table
Number of entries: Number of page frames

Steps in Paging

- Modern systems use TLBs and multi-level paging
- Paging assumes special hardware support
- Overview of steps
  1. Input to MMU: virtual address = (page p, offset o)
  2. Check if there is a frame f with (p,f) in TLB
  3. If so, physical address is (f,o)
  4. If not, lookup page-table in main memory
     - requires several slow accesses due to multi-level paging
  5. If page is present, compute physical address
  6. If not, issue page fault interrup to OS kernel
     - even slower
  7. Update TLB/page-table entries (e.g. Modified bit)

Paging Summary

- How long will access to a location in page p take?
  - If the address of the corresponding frame is found in TLB?
  - If the page-entry corresponding to the page is valid?
    - Using two-level page table
    - Using Inverted hashed page-table
  - If a page fault occurs?
- How to save space required to store a page table?
  - Two-level page-tables exploit the fact only a small and contiguous fraction of virtual space is used in practice
  - Inverted page-tables exploit the fact that the number of valid page-table entries is bounded by the available memory
- Page-table for a process is usually stored in user space

Role of the OS

- Basic paging architecture is determined by hardware
  - OS loads address of page table into MMU, MMU takes it from there
- OS's main issue is responding to page faults
  - may also run page daemon
- Page replacement handled by OS

Evaluating VM performance

- Parameters: address size, page size, multi-level page size, TLB size, replacement rule
- Performance: size of page table (memory overhead), time to operate on page table (CPU overhead), frequency of page faults, average memory access time

Page Replacement

- We might page fault if:
  - a process is allocating new memory
  - a process accesses memory swapped out to disk
- If there's available space in physical memory, great, just load the page from disk
- Otherwise, we have to select a page and swap it out (write page contents to disk)
- Two problems
  - which page? page replacement rule
  - managing each process' list of mapped pages

Swapping out an old page and swapping in a new page

- Consult the page replacement rule to select a victim (more on that in a moment)
  - might belong to current process or might belong to another
- Has page been written since it was last swapped out (dirty bit)?
  - if not, just deallocate the page, no need to re-write
  - if so, schedule transfer (write to disk) and note block number in that process' page table
    - block current process until complete
- Look up address of page to load and schedule transfer (read from disk)
  - block current process until complete
- Update current process' page table

So we need a…
Page Replacement Rule

- Performance influenced by which page we pick
  - wouldn't want to pick one we're just going to have to swap in again soon
  - so we aim for "inactive" pages not likely to be used in the near future
- Optimal page replacement rule (OPT):
  - pick the page to be accessed farthest in the future
  - can't actually do this, of course
- Real replacement rules aim to approximate OPT
  - we determine out how well they do experimentally