## Lecture 21 (Actually covered on Mon 11/17/2008)

- Lab #4 Software Simulation – Due Fri Nov 21 at 5pm
- HW #3 Cache Simulator & code optimization – Due Mon Nov 24 at 5pm

- Virtual Memory



## Virtual Memory (and Indirection)

- Virtual Memory
  - We'll talk about the motivations for virtual memory
  - We'll talk about how it is implemented
  - Lastly, we'll talk about how to make virtual memory fast: Translation Lookaside Buffers (TLBs).



## A Real Problem

- What if you wanted to run a program that needs more memory than you have?
  - You could store the whole program on disk, and use memory as a cache for the data on disk. This is one feature of virtual memory.
  - Before virtual memory, programmers had to manually manage loading "overlays" (chunks of instructions & data) off disk before they were used. This is an incredibly tedious, not to mention error-prone, process.



## More Real Problems

- Running multiple programs at the same time brings up more problems.

1. Even if each program fits in memory, running 10 programs might not.
2. Multiple programs may want to store something at the same address.
3. How do we protect one program's data from being read or written by another program?

## More Real Problems

- Running multiple programs at the same time brings up more problems.

1. Even if each program fits in memory, running 10 programs might not.
   - This is really the same problem as on the previous slide.

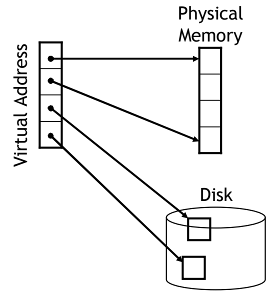
2. Multiple programs may want to store something at the same address.
   - I.e., what if both Program A and B want to use address 0x10000000 as the base of their stack?
   - It is impractical (if not impossible) to compile every pair of programs that could get executed together to use distinct sets of addresses.

3. How do we protect one program's data from being read or written by another program?



## Virtual Memory

- We translate "virtual addresses" used by the program to "physical addresses" that represent places in the machine's "physical" memory.
  - The word "translate" denotes a level of indirection

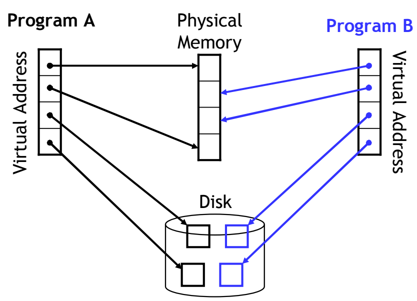



A virtual address can be mapped to either physical memory or disk.



## Virtual Memory

- Because different processes will have different mappings from virtual to physical addresses, two programs can freely use the same virtual address.
- By allocating distinct regions of physical memory to A and B, they are prevented from reading/writing each others data.






## Caching revisited

- Once the translation infrastructure is in place, the problem boils down to caching.
  - We want the size of disk, but the performance of memory.

- The design of virtual memory systems is really motivated by the high cost of accessing disk.
  - While memory latency is ~100 times that of cache, disk latency is ~100,000 times that of memory.

- Hence, we try to minimize the miss rate:
  - VM "pages" are much larger than cache blocks. Why?
  - A fully associative policy is used.
    - With approximate LRU

- Should a write-through or write-back policy be used?

## Finding the right page

- If it is fully associative, how to we find the right page without scanning all of memory?

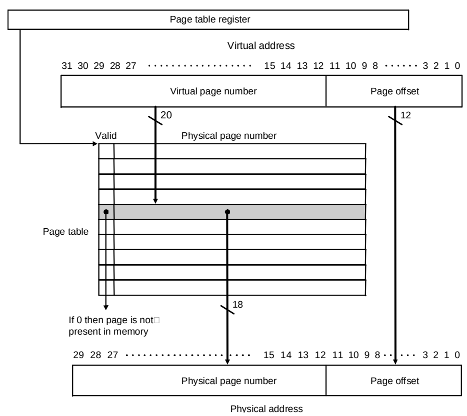


## Finding the right page

- If it is fully associative, how do we find the right page without scanning all of memory?
  - Use an index, just like you would for a book.
- Our index happens to be called the page table:
  - Each process has a separate page table
    - A “page table register” points to the current process’s page table
  - The page table is indexed with the virtual page number (VPN)
    - The VPN is all of the bits that aren’t part of the page offset.
  - Each entry contains a valid bit, and a physical page number (PPN)
    - The PPN is concatenated with the page offset to get the physical address
  - No tag is needed because the index is the full VPN.



## Page Table picture



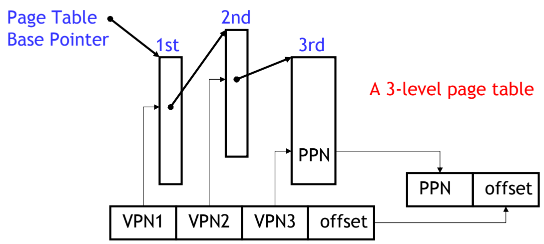



## How big is the page table?

- From the previous slide:
  - Virtual page number is 20 bits.
  - Physical page number is 18 bits + valid bit -> round up to 32 bits.
- How about for a 64b architecture?

## Dealing with large page tables

- Multi-level page tables
  - "Any problem in CS can be solved by adding a level of indirection"
    - or two…




- Since most processes don't use the whole address space, you don't allocate the tables that aren't needed
  - Also, the 2nd and 3rd level page tables can be "paged" to disk.

13

14

## Waitaminute!

- We've just replaced every memory access MEM[addr] with:

  MEM[MEM[MEM[MEM[PTBR + VPN1<<2] + VPN2<<2] + VPN3<<2] + offset]

  - *i.e.*, 4 memory accesses

- And we haven't talked about the bad case yet (*i.e.*, page faults)…

"Any problem in CS can be solved by adding a level of indirection"

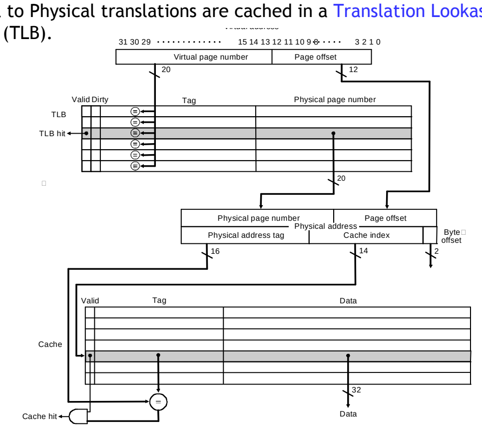
– except too many levels of indirection…

- How do we deal with too many levels of indirection?

15

## Caching Translations

- Virtual to Physical translations are cached in a Translation Lookaside Buffer (TLB).




16

## What about a TLB miss?

- If we miss in the TLB, we need to “walk the page table”
  - In MIPS, an exception is raised and software fills the TLB
  - In x86, a “hardware page table walker” fills the TLB
- What if the page is not in memory?
  - This situation is called a **page fault**.
  - The operating system will have to request the page from disk.
  - It will need to select a page to replace.
    - The O/S tries to approximate LRU (see CSE 451)
  - The replaced page will need to be written back if dirty.



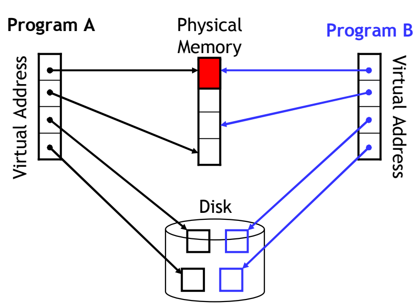
## Memory Protection

- In order to prevent one process from reading/writing another process’s memory, we must ensure that a process cannot change its virtual-to-physical translations.
- Typically, this is done by:
  - Having two processor modes: user & kernel.
    - Only the O/S runs in kernel mode
  - Only allowing kernel mode to write to the virtual memory state, *e.g.*,
    - The page table
    - The page table base pointer
    - The TLB



## Sharing Memory

- Paged virtual memory enables sharing at the granularity of a page, by allowing two page tables to point to the same physical addresses.
- For example, if you run two copies of a program, the O/S will share the code pages between the programs.






## Summary

- Virtual memory is great:
  - It means that we don’t have to manage our own memory.
  - It allows different programs to use the same memory.
  - It provides protect between different processes.
  - It allows controlled sharing between processes (albeit somewhat inflexibly).
- The key technique is **indirection**:
  - Yet another classic CS trick you’ve seen in this class.
  - Many problems can be solved with indirection.
- Caching made a few appearances, too:
  - Virtual memory enables using physical memory as a cache for disk.
  - We used caching (in the form of the Translation Lookaside Buffer) to make Virtual Memory’s indirection fast.