Computer Systems Lecture 18

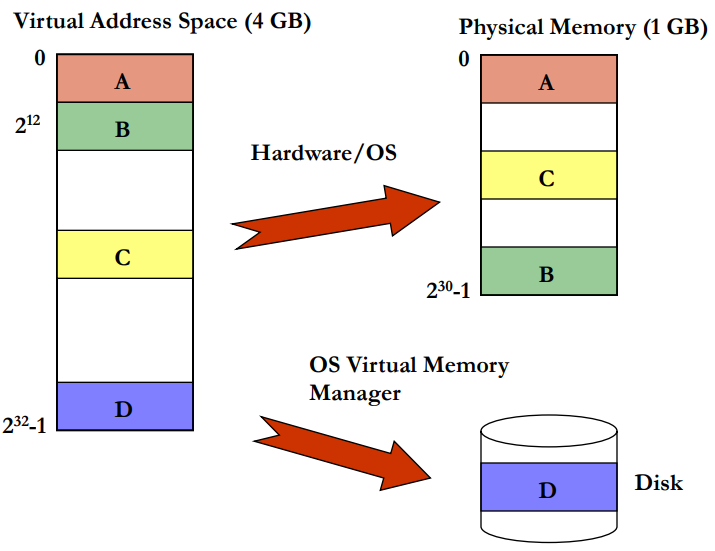
Virtual Memory: Motivation

Virtual memory addresses two main problems:

1. Capacity: how do we relieve the programmers/users from dealing with limited main memory?
   1. We want to allow for the physical memory to be smaller than the program’s address space (e.g. 32 bits -> 4GB; 64 bit -> A LOT MORE)
   2. We want to allow multiple programs to share the limited physical memory with no human intervention
2. Safety: how do we allow for safe and efficient sharing of memory among multiple programs?
   1. We want to prevent user programs from accessing the memory used by the OS
   2. We want strict control of access by each user program to the memory of other user programs.

Virtual Memory

* The basic idea: each program thinks it owns the entire memory -> the virtual address space
  + PC and load/store addresses are virtual addresses
* Actual main memory: physical address space
  + Virtual addresses are translated on the fly to physical addresses
  + Parts of virtual address space not recently used are stored on disk
* Address translation is done jointly by the OS and hardware

Address Translation for a Program

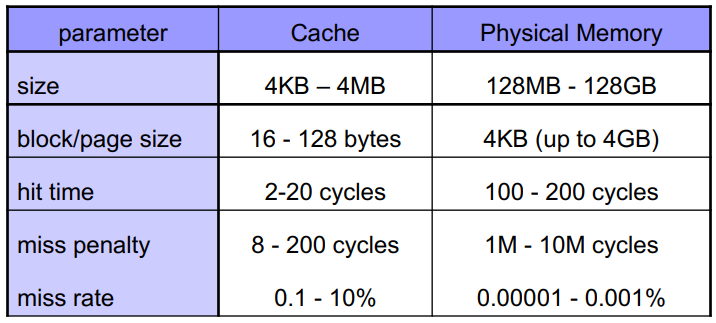
Physical Memory as a Cache for VM

* Virtual memory space can be larger than physical memory
  + Programmer always sees the full address space
* Physical memory is a cache for the virtual memory
  + Physical memory holds the currently used portions of a program’s code and data (exploits locality)
* Secondary storage (disk or SSD) ‘backs’ the physical memory
  + OS reserves a portion of the disk for swap space
  + OS swaps portions of each process’ code and data areas in and out of physical memory on demand (called paging)
  + Paging is transparent to the application ad the programmer

Paging

* A cache line or block of VM is called a page
  + Simply page or virtual page for virtual memory
  + Page frame or physical page for physical memory
* Typical page sizes are 4-16 KB (MB or GB in servers)
  + Large enough for efficient disk use and to keep translation tables (called page tables) small
* Mapping is done through a per-program page table
  + This allows control of which pages each program can access
  + Different programs can use the same virtual addresses

Typical Virtual Memory Parameters

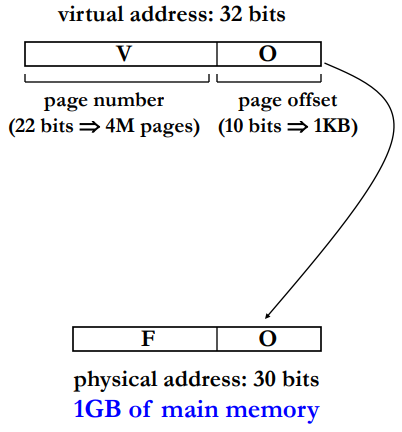
When a valid virtual address is not in memory, this is called a page fault.

Address Translation

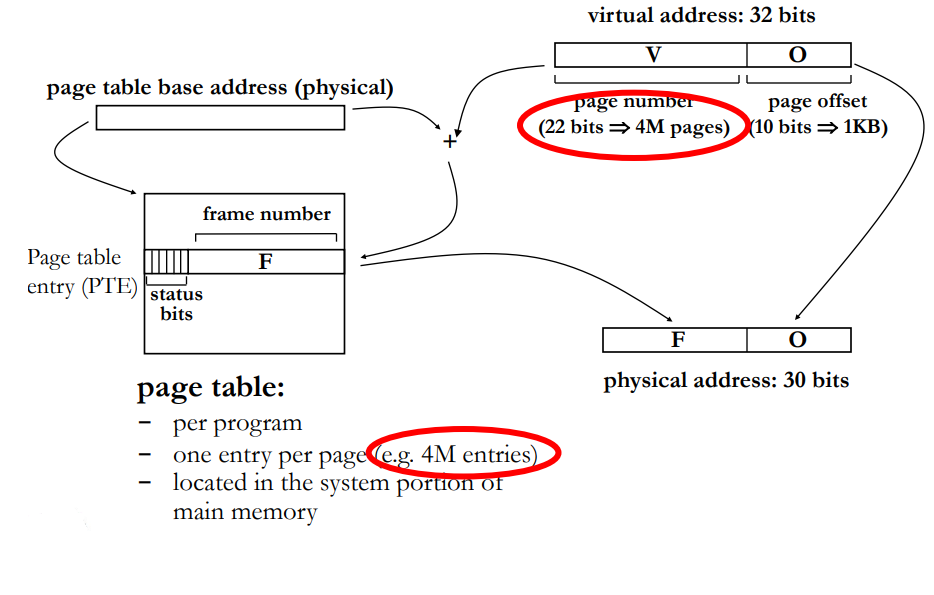
We need:

* A mapping from virtual (V) to physical (F) page numbers
* Page offset not translated
* Must be efficient (in time and space)

The solution to this is a page table.



Here with 1KB of pages we will need a 10 bit offset, leaving 22 bits for the virtual page number. Here we have 1GB of memory for the machine, this would need 30 bits to make this memory byte addressable.



The conversion from V to F is not really a conversion, V is used as an offset from the base address of the page table, which stores the value for F.

Example Problem

What is the size of the page table given a 32-bit virtual address space, 4KB physical pages and 1GB of main memory?

With 4KB of pages, will mean 12 page offset bits will be needed, leaving 20 bits for the page number with means 1M entries. With 1GB of main memory, we’ll need 30 bits to address the memory, but the last 12 of these will be covered by the page offset, so we only need to store the first 18 bits in each entry. This means the page table will be 18Mbits ~= 2.25MBs large.

Moving Pages To/From Memory

Pages are allocated on demand, for example when a program is launched, resulting pages are allocated for its code, data and stack.

Pages are replaced and swapped to disk when the system runs out of free page frames. The aim being to replace pages not recently used (principle of locality), The A(ccess) status bit for a page is set whenever a page is accessed and is reset periodically to track locality. If any data in a page has been modified, the page must be written back to disk: the M(odified) status bit is set.

Access to a swapped-out page causes a page-fault which invokes the OS through the interrupt mechanism, the R(esidence) status bit is zero

Providing Protection

Each page table entry can have permission bits that control whether

* The process is allowed to access a page
* Read & write (W), read-only (R) or execute-only (E) access is allowed

This enables per process memory protection, for example it allows private and shared areas

It’s important that only the OS can change page tables

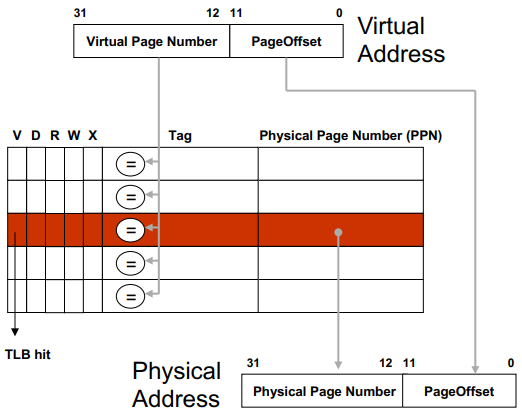
Fast Address Translation

Problem: page table accesses add latency to each memory access, there are two memory accesses per load or store (1 to get the page table entry + 1 for the actual load/store).

Fast address translation: Translation Lookaside Buffer (TLB) is a solution for this (it’s essentially just a cache):

* A cache of page table entries
  + Each TLB entry holds translation information, not program data
  + Tag: virtual page number. Entry: physical page number
* Small and fast table in hardware, located close to processor
* Can capture most translations due to principle of locality
* When page not in TLB: access the page table and save the translation entry in TLB

Translation Look-Aside Buffer (TLB)

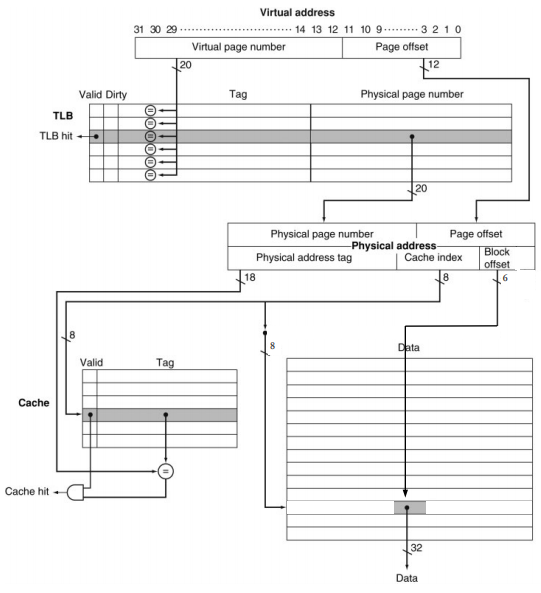
  
TLB: a small, fully-associative cache of page table entries

* Accessed with Virtual Page Number (VPN)
* Each entry stores the translation (PPN) for a given VPN
* Physical address formed from PPN and Page Offset
* Page table accessed on a TLB miss

TLB status bits:

* V (valid) bit indicates a valid entry
* D (dirty) bit indicates whether the page has been modified
* R,W,X permission bits. Checked on every memory access

Note: physical address is always 32 bits, regardless of actual physical memory size

Integrating a TLB with the Cache

Virtual Memory: Full Picture

