CMSC 411
Computer Systems Architecture
Lecture 12
Virtual Memory

Alan Sussman
als@cs.umd.edu

Administrivia

- Homework #3 posted tomorrow, due after spring break
  - answers to Homework #2 posted too, and papers returned today
- Exam #1 Thursday
  - practice exam posted, with answers
- Project #1 due Wed. after break
- Reminder: 400-level lecture series for fall semester, Tuesday-Thursday starting at 5PM

Virtual addressing

- Computers are designed so that multiple programs can be active at the same time
- At the time a program is compiled, the compiler has to assign addresses to each data item. But how can it know what memory addresses are being used by other programs?
- Instead, the compiler assigns virtual addresses, and expects the loader/OS to provide the means to map these into physical addresses

Memory protection

- Each program "lives" in its own virtual space, called its *process*
- When the CPU is working on one process, others may be partially completed or waiting for attention
- The CPU is *time shared* among the processes, working on each in turn
- And main memory is also shared among processes

In the olden days …

- The loader would locate an unused set of main memory addresses and load the program and data there
- There would be a special register called the *relocation register*, and all addresses that the program used would be interpreted as addresses relative to the base address in that register
- So if the program jumped to location 54, the jump would really be to 54 + contents of relocation register. A similar thing, perhaps with a second register, would happen for data references

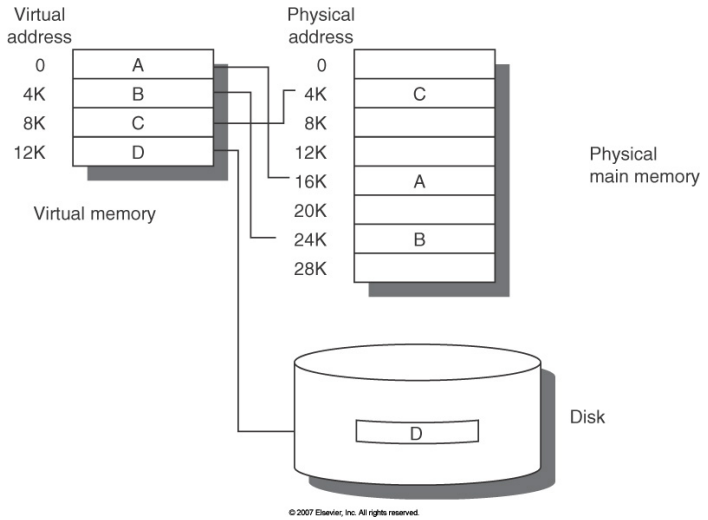
In the less-olden days ...

- It became difficult to find a contiguous segment of memory big enough to hold program and data, so the program was divided into *pages*, with each page stored contiguously, but different pages in any available spot, either in main memory or on *disk*
- This is the *virtual addressing* scheme
  - to the program, memory looks like a contiguous segment, but actually, data is scattered in main memory and perhaps on disk

But we know all about this!

- Already know that a program and data can be scattered between cache memory and main memory
- Now add the reality that its location in main memory is also determined in a scattered way, and some pages may also be located on disk
- So each page has its own relocation value

Virtual Memory – Fig. B.19



Parameters – Fig. B.20

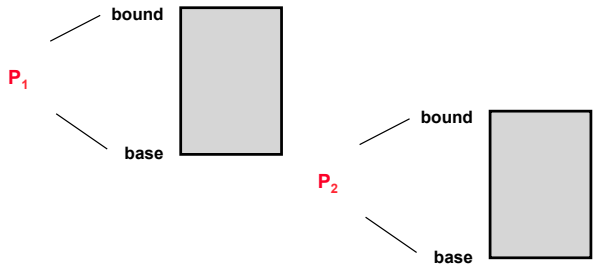
Parameter	First-level cache	Virtual memory
Block (page) size	16-128 bytes	4096-65,536 bytes
Hit time	1-3 clock cycles	100-200 cc
Miss penalty	8-200 cc	10^6-10^7 cc
(access time)	(6-160 cc)	(.8-8)*10^6 cc
(transfer time)	(2-40 cc)	(.2-2)*10^6 cc
Miss rate	0.1-10%	0.00001-0.001%
Address mapping	25-45 bit physical address to 14-20 bit cache address	32-64 bit virtual address to 25-45 bit physical address

Cache vs. virtual memory

Cache	Virtual memory
Cache miss handled by hardware	Page faults handled by operating system
Cache size fixed for a particular machine	Virtual memory size fixed for a particular program
Fundamental unit is a block	Fundamental unit is a fixed-length page or a variable-length segment
cache fault	page fault

Old Protection mode: Base & Bound

- User processes need to be protected from each other
- Two registers, *base* and *bound* test whether this virtual address belongs to this process
- If not, a *memory protection violation* exception is raised
- Users cannot change the base and bound registers

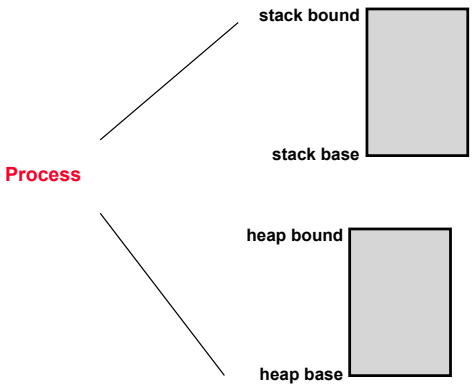


Who can change them?

- The operating system needs access to the base and bound registers
- So a process that is labeled *kernel* (also called *supervisor* or *executive*) can access any memory location and change the registers
- Kernel processes are accessed through *system calls*, and a return to user mode is like a subroutine return, restoring the state of the user process

Segmentation

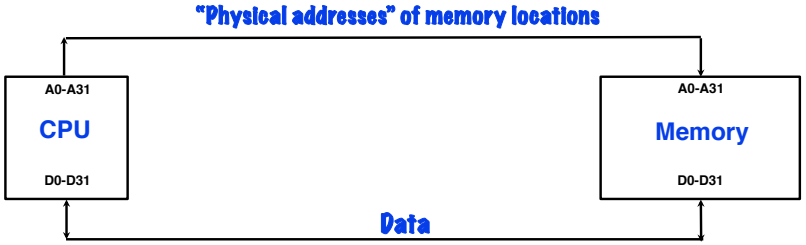
- Basically multiple base&bounds
  - w/ virtual memory



Each segment can be located anywhere in physical memory

Paging

The Limits of Physical Addressing

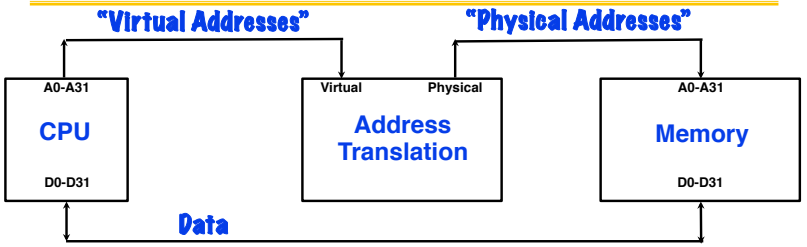


All programs share one address space:
The physical address space

Machine language programs must be aware of the machine organization

No way to prevent a program from accessing any machine resource

Solution: Add a Layer of Indirection



User programs run in a standardized virtual address space

Address Translation hardware managed by the operating system (OS) maps virtual address to physical memory

Hardware supports "modern" OS features: Protection, Translation, Sharing

Paging overview

- Fully-associative mapping, because page faults are *really, really* expensive
- Page is located using a *page table*, one entry per page in the virtual address space
  - Size is sometimes reduced by hashing, to make one entry per physical page in main memory – an *inverted* page table
- Since locality says that a page will be used multiple times, address translation usually tests the addresses of the recently referenced pages before looking in other places
- So address translation information is held in the *translation look-aside buffer* (TLB)

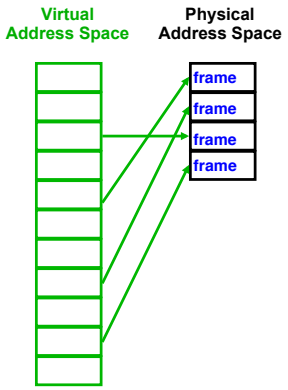
Paging overview (cont.)

- Most machines replace the LRU page
  - moving pages between memory and disk is so slow that it's worth doing something close to real LRU
- Disks are *so slow* that machines use write-back, not write-through, and keep a *dirty bit* for each page

Advantages of Paged Virtual Memory

- Translation:
  - Program can be given consistent view of memory, even though physical memory is scrambled
  - Only the most important part of program ("Working Set") must be in physical memory.
  - Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later.
- Protection:
  - Different threads (or processes) protected from each other.
  - Different pages can be given special behavior
    - » (Read Only, Invisible to user programs, etc).
  - Kernel data protected from User programs
    - » Very important for protection from malicious programs
- Sharing:
  - Can map same physical page to multiple users ("Shared memory")

Page tables encode virtual address spaces



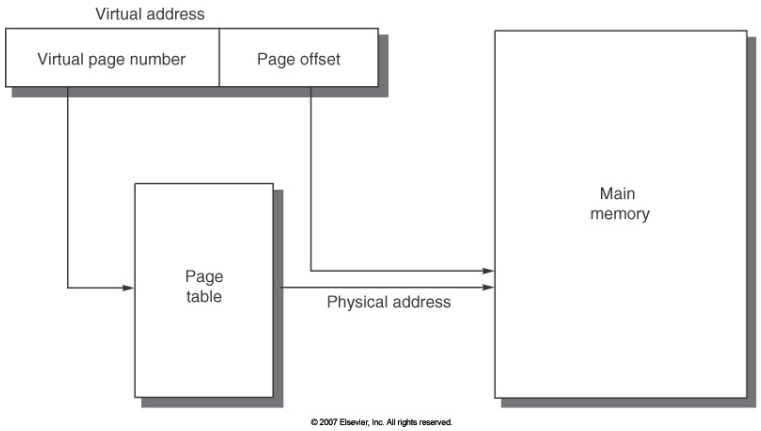
A virtual address space is divided into blocks of memory called pages

A machine usually supports pages of a few sizes (MIPS R4000):

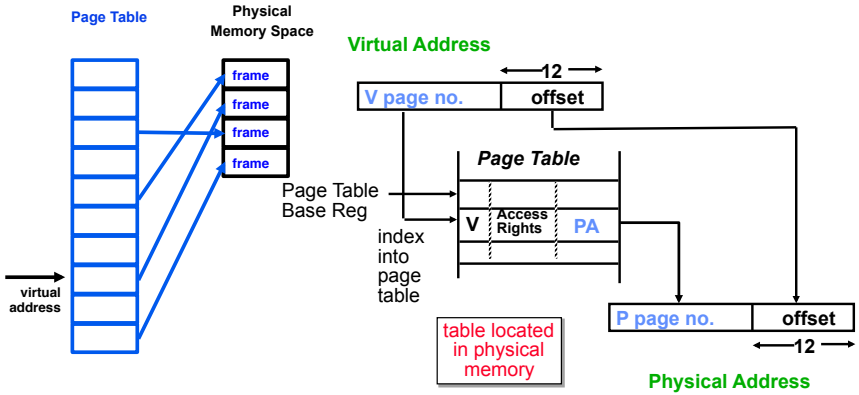
Page Size
4 Kbytes
16 Kbytes
64 Kbytes
256 Kbytes
1 Mbyte
4 Mbytes
16 Mbytes

A valid page table entry encodes physical memory "frame" address for the page

Page tables – Fig. B.23



Details of Page Table



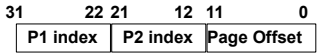
- Page table maps virtual page numbers to physical frames ("PTE" = Page Table Entry)
- Virtual memory => treat memory ≈ cache for disk

Two-Level Page Tables

Each process needs its own address space!

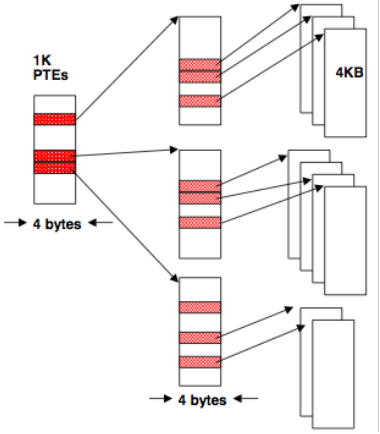
Two-level Page Tables

32 bit virtual address



Top-level table *wired* in main memory

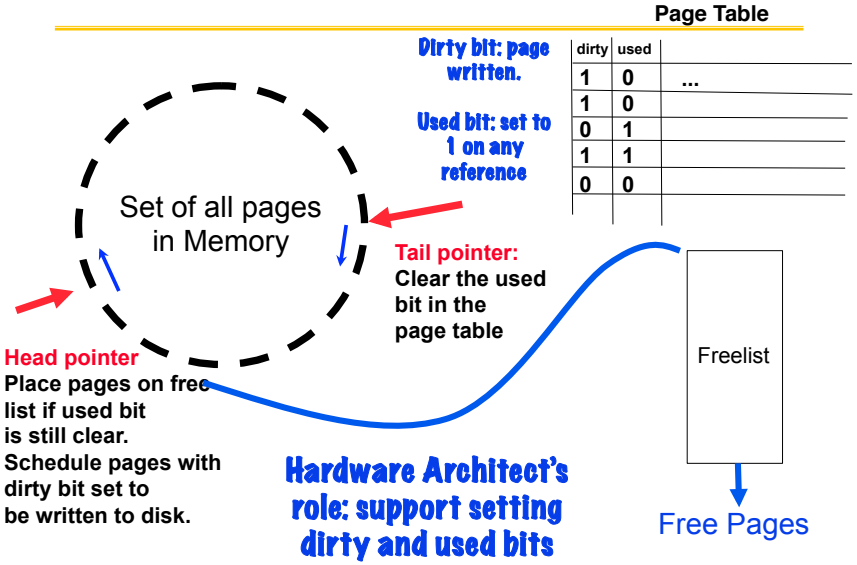
Subset of 1024 second-level tables in main memory; rest are on disk or unallocated



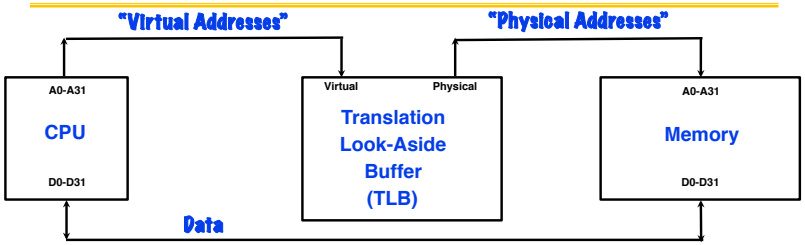
Choosing page size

- A large page size
  - keeps page table small.
  - reduces cache miss times, if accesses have locality
  - reduces start-up overhead in moving data from disk to memory
  - means fewer TLB misses
- but also
  - wastes memory (internal fragmentation)
  - increases the time to start up a program

VM and Disk: Page replacement policy



MIPS Address Translation: How does it work?



Translation Look-Aside Buffer (TLB)
A small fully-associative cache of mappings from virtual to physical addresses

TLB also contains protection bits for virtual address

Fast common case: Virtual address is in TLB, process has permission to read/write it.

The TLB caches page table entries