CS 3204
Operating Systems

Lecture 15

Godmar Back

Announcements

- Project 2 due Oct 20, 11:59pm
  - See forum for additional office hours
- Reminder: need to pass 90% of tests of project 2 by the end of the semester to pass the class
  - That's all tests (except for possibly multi-doom)
  - P2 score will depend on what you pass by the deadline.
  - P2 is a prerequisite for projects 3 and 4 – if you can't get all tests to pass by submission deadline, fix them quickly

Virtual Memory

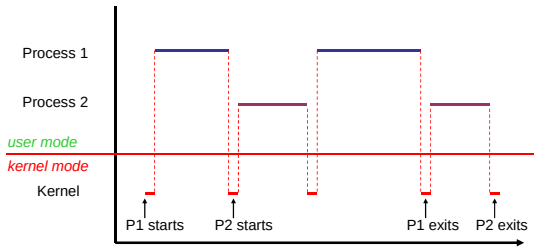
Virtual Memory

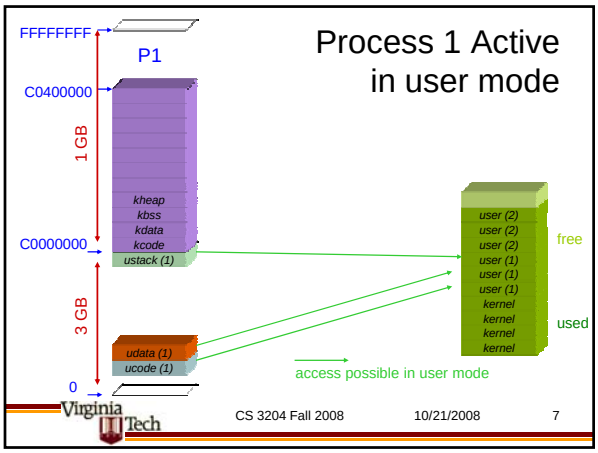
- Is not a "kind" of memory
- Is a technique that combines one or more of the following concepts:
  - Address translation (always)
  - Paging (not always)
  - Protection (not always, but usually)
- Can make storage that isn't physical random access memory appear as though it were

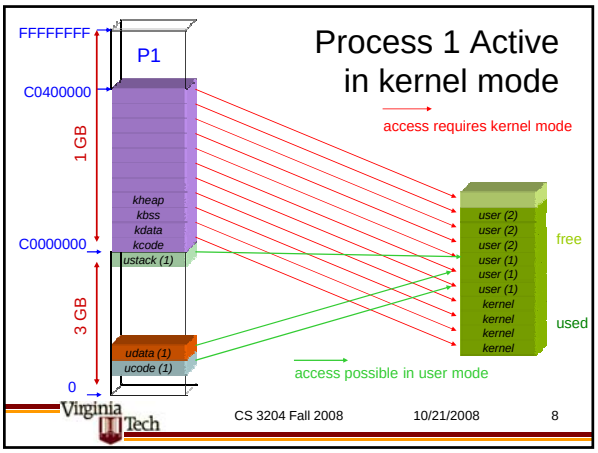
Goals for Virtual Memory

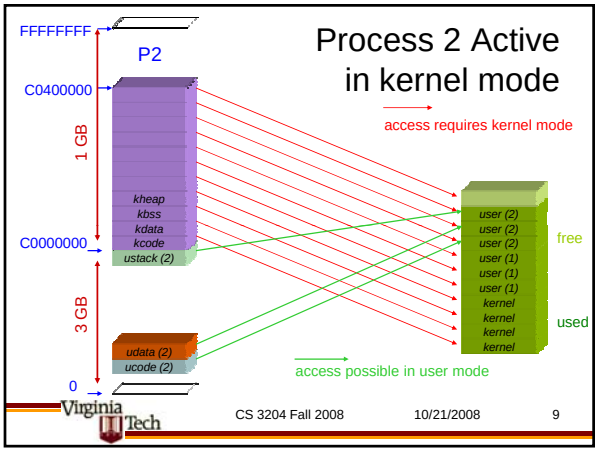
- Virtualization
  - Maintain illusion that each process has entire memory to itself
  - Allow processes access to more memory than is really in the machine (or: sum of all memory used by all processes > physical memory)
- Protection
  - make sure there's no way for one process to access another process's data

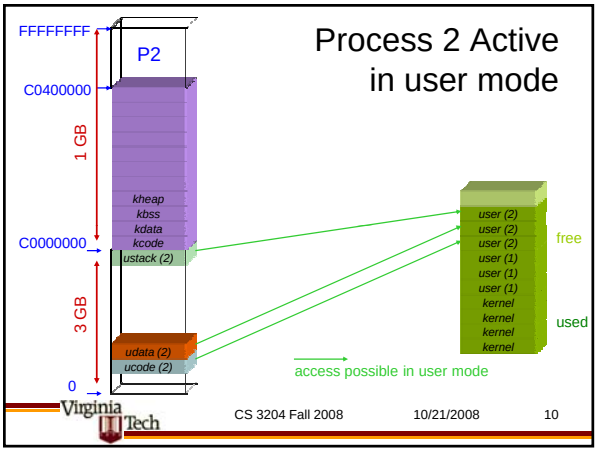
Context Switching











Page Tables

- How are the arrows in previous pictures represented?
  - Page Table: mathematical function "Trans"

Trans:
{ Process Ids } × { Virtual Addresses } × { user, kernel } × ℘({ read, write, execute })

→ { Physical Addresses } ∪ { INVALID }

- Typically (though not on all architectures) have
  - Trans(p_i, v_a, user, *) = Trans(p_i, v_a, kernel, *)
    OR
    Trans(p_i, v_a, user, *) = INVALID
  - E.g., user virtual addresses can be accessed in kernel mode

Sharing Variations

- We get user-level sharing between processes p_1 and p_2 if
  - Trans(p_1, v_a, user, *) = Trans(p_2, v_a, user, *)
- Shared physical address doesn't need to be mapped at same virtual address, could be mapped at v_a in p_1 and v_b in p_2:
  - Trans(p_1, v_a, user, *) = Trans(p_2, v_b, user, *)
- Can also map with different permissions: say p_1 can read & write, p_2 can only read
  - Trans(p_1, v_a, user, {read, write}) = Trans(p_2, v_b, user, {read})
- In Pintos (and many OS) the kernel virtual address space is shared among all processes & mapped at the same address:
  - Trans(p_i, v_a, kernel, *) = Trans(p_k, v_a, kernel, *) for all processes p_i and p_k and v_a in [0xC0000000, 0xFFFFFFFF]

Per-Process Page Tables

- Can either keep track of all mappings in a single table, or can split information between tables
  - one for each process
  - mathematically: a projection onto a single process
- For each process pi define a function PTransi as
  - PTransi (va, *, *) = Trans(pi, va, user, *)
- Implementation: associate representation of this function with PCB, e.g., per-process hash table
  - Entries are called "page table entries" or PTEs
- Nice side-effect:
  - reduced need for synchronization if every process only adds/removes entries to its own page table

Per-Process Page Tables (2)

- Common misconception
  - "User processes use 'user page table' and kernel uses 'kernel page table'" – as if those were two tables
- Not so (on x86): mode switch (interrupt, system call) does not change the page table that is used
  - It only "activates" those entries that require kernel mode within the current process's page table
- Consequence: kernel code also cannot access user addresses that aren't mapped

Non-Resident Pages

- When implementing virtual memory, some of a process's pages may be swapped out
  - Or may not yet have been faulted in
- Need to record that in page table:

Trans (with paging):
{ Process Ids } × { Virtual Addresses } × { user, kernel } × $\wp$({ read, write, execute })

→ { Physical Addresses } ∪ { INVALID } ∪ { Some Location On Disk }

Implementing Page Tables

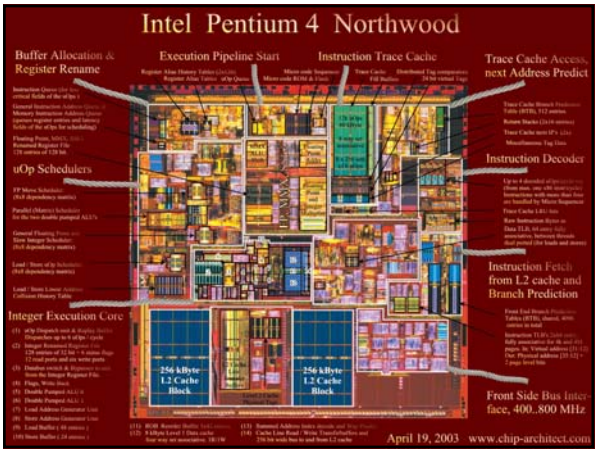
- Many, many variations possible
- Done in combination of hardware & software
  - Hardware part: dictated by architecture
  - Software part: up to OS designer
    - Machine-dependent layer that implements architectural constraints (what hardware expects)
    - Machine-independent layer that manages page tables
- Must understand how TLB works first

Page Tables Function & TLB

Trans (with paging):
{ Process Ids } × { Virtual Addresses } × { user, kernel } × { read, write, execute }

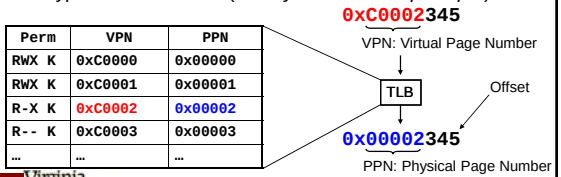
→ { Physical Addresses } ∪ { INVALID } ∪ { Some Location On Disk }

- For each combination (process id, virtual_addr, mode, type of access) must decide
  - If access is permitted
  - If permitted:
    - if page is resident, use physical address
    - if page is non-resident, page table has information on how to get the page in memory
- CPU uses TLB for actual translation – page table feeds the TLB on a TLB miss



TLB: Translation Look-Aside Buffer

- Virtual-to-physical translation is part of every instruction (why not only load/store instructions?)
  - Thus must execute at CPU pipeline speed
- TLB caches a number of translations in fast, fully-associative memory
  - typical: 95% hit rate (*locality of reference principle*)

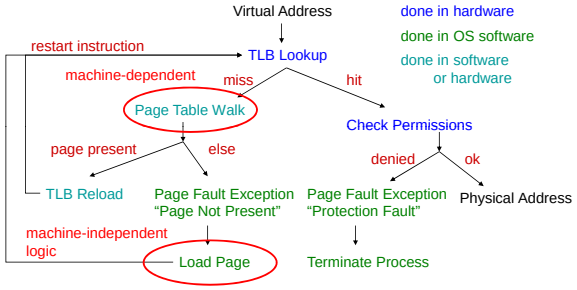


Perm	VPN	PPN
RWX K	0xC0000	0x00000
RWX K	0xC0001	0x00001
R-X K	0xC0002	0x00002
R-- K	0xC0003	0x00003
…	…	…

TLB Management

- Note: on previous slide example, TLB entries did not have a process id
  - As is true for x86
- Then: if process changes, some or all TLB entries may become invalid
  - X86: flush entire TLB on process switch (refilling adds to cost!)
- Some architectures store process id in TLB entry (MIPS)
  - Flushing (some) entries only necessary when process id reused

Address Translation & TLB



TLB Reloaded

- TLB small: typically only caches 64-2,048 entries
  - What happens on a miss? – must consult ("walk") page table – TLB Reload or Refill
- TLB Reload in software (MIPS)
  - Via TLB miss handlers – OS designer can pick any page table layout – page table is only read & written by OS
- TLB Reload in hardware (x86)
  - Hardware & software must agree on page table layout *inasmuch* as TLB miss handling is concerned – page table is read by CPU, written by OS
- Some architectures allow either (PPC)

Page Tables vs TLB Consistency

- No matter which method is used, OS must ensure that TLB & page tables are consistent
  - On multiprocessor, this may require "TLB shootdown"
- For software-reloaded TLB: relatively easy
  - TLB will only contain what OS handlers place into it
- For hardware-reloaded TLB: two choices
  - Use same data structures for page table walk & page loading (hardware designers reserved bits for OS's use in page table)
  - Use a layer on top (facilitates machine-independent implementation) – this is the recommended approach for Pintos Project 3
    - In this case, must update actual page table (on x86: "page directory") that is consulted by MMU during page table walk
    - Code is already written for you in pagedir.c

Hardware/Software Split in Pintos