Virtual Memory: Paging

COMP 321

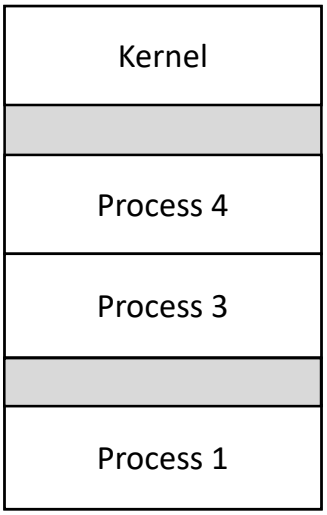
Dave Johnson

1

Kernel Memory Management and Allocation

Problem: internal and external fragmentation again, at a different level

- Each process's address space should be ***contiguous***
- Makes memory allocation difficult and expensive
- Growing a process's address space is a mess
  - Could allocate it larger than initially needed, but that wastes memory
  - And how do you plan for how much larger?
  - Could move processes around in memory to allow growing, but that's expensive
  - Example, move (copy) process 4's memory up to allow process 3 to grow

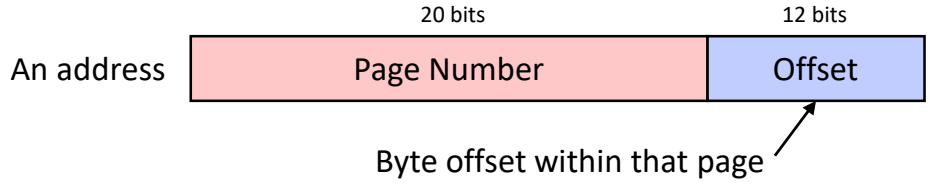


2

Virtual Memory Using Paging

An addition to the hardware, managed by the OS kernel

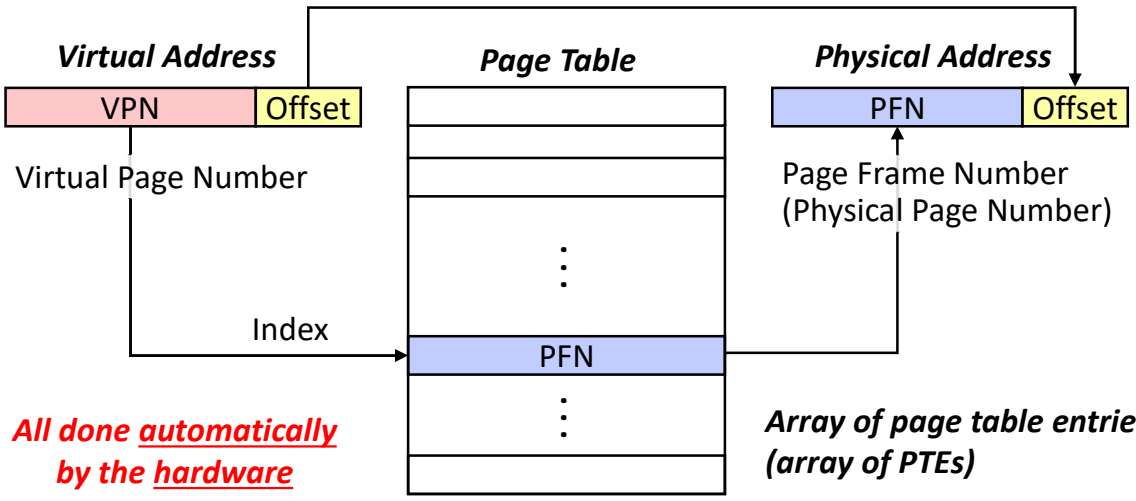
- Conceptually divide memory into fixed-size (power of 2 in size) ***pages***
- Each page is allocated and managed as an indivisible unit
- All pages here are the same size (common is 4096 = 0x1000)



Example: 4096 page size (0x1000)

- Offset is 0…4095 (12 bits for offset)

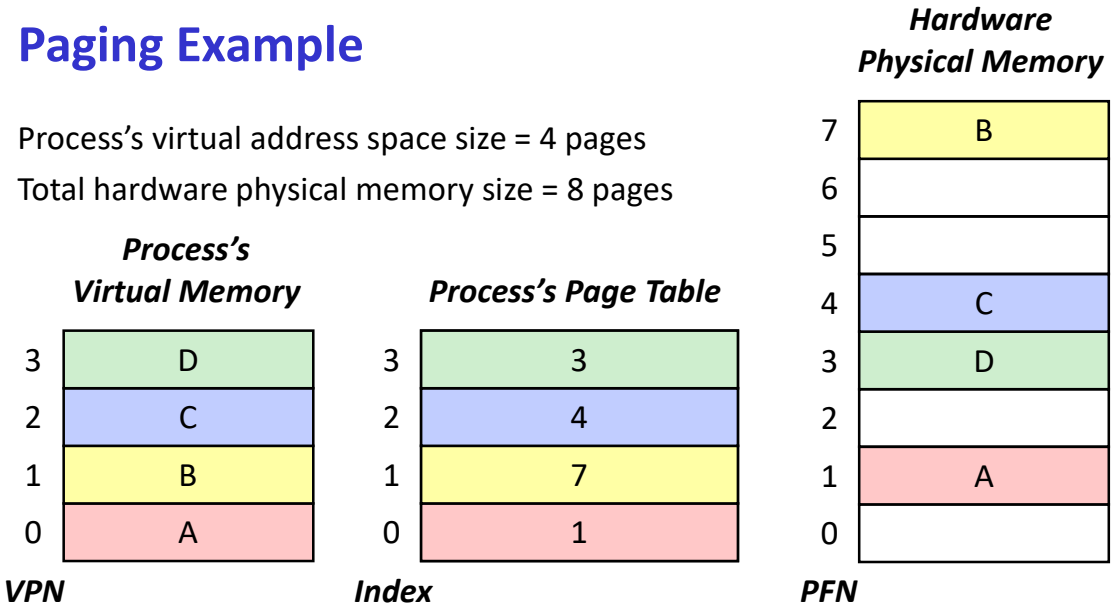
Converting Virtual Address to Physical Address



Paging Example

Process's virtual address space size = 4 pages

Total hardware physical memory size = 8 pages



Advantages and Disadvantages of Paging

Advantages

- Can map ***any*** physical page to appear ***anywhere*** in process's address space
- Pages need not be contiguous in physical memory
- Can make any physical pages ***appear*** to be contiguous in virtual memory
- Thus, no ***external*** fragmentation
- Memory allocation is ***easy***, just allocate ***any*** available physical page
- Can easily grow a process's virtual address space
- Provides support for ***demand paging*** (later)

Disadvantages

- A small amount of ***internal*** fragmentation (up to size PAGESIZE – 1 bytes)

Where is the Page Table?

Could be in an "array" of special hardware registers

- Can be very fast to translate virtual address to physical address
- But very limited; need a separate register for every virtual page of a process
- And what about the address space for other processes?

Instead, most CPUs define the page table to be stored in physical memory

- Can hold essentially arbitrarily large page tables for any number of processes
- Hardware just needs to know ***physical address*** and ***size*** of the page table
- Add two special CPU registers to the ***hardware***
  - Page Table Base Register (***PTBR***): ***physical address*** of the page table
  - Page Table Limit Register (***PTLR***): the number of PTEs in the page table (the array ***size***)

The Page Table Limit Register (PTLR)

Any VPN ≥ PTLR is not accessible (that page effectively doesn't exist)

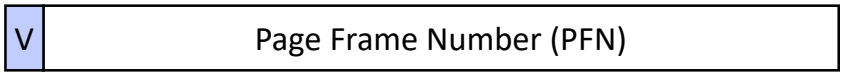
- Any process's address space can have any number of pages in it
- One process's page table may have a different number of entries from another process's page table
- PTLR is thus required so that the ***hardware*** knows the size (# of entries)
  - The hardware won't access beyond the end of the PTE array
  - Attempting to access any VPN ≥ PTLR causes an ***exception***

And changing PTBR or PTLR requires a privileged instruction, so a process can't go beyond the pages defined in its own page table!

Adding a PTE "Valid" Bit

The hardware finds the PTE in physical memory using PTBR and PTLR

- Add a "valid" bit to the format for each PTE (***redefine*** 1 bit of the PTE)



- If PTE valid bit in needed PTE is 0, ***hardware*** causes ***exception*** on any access
- The kernel can set the valid bit to 0 in any PTE as needed or desired
  - No physical page needs to be allocated if page's PTE.valid == 0
  - Basically, a virtual "hole" in the process's address space ***that consumes zero physical pages (the other PTE fields are <u>ignored</u> by the hardware)***

Example Use of Setting PTE.valid == 0

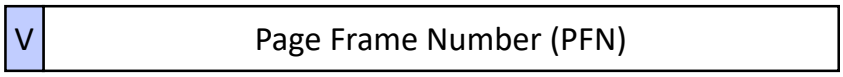
Separating the stack "far away" from the heap to allow room to grow

- The stack is up at high VPNs (high virtual addresses)
- The text, data, bss, and heap are down at low VPNs (low virtual addresses)
- All PTEs in between are set up with PTE.valid == 0
- ***This separation for space between heap and stack consumes zero physical pages!***
- (other than the size of the page table)

1	11
1	50
1	2
0	. . .
1	61
1	15
1	7
1	29

Controlling Protection on Individual Virtual Pages

Our current PTE definition, including the "valid" bit



Add hardware to treat part of each PTE as defining page protection

- ***Example***: prot (e.g., a 3-bit field) define types of ***hardware-allowed*** accesses
- R (0x1 = 001): ***Reading*** the contents of the page (e.g., LOAD) is allowed
- W (0x2 = 010): ***Writing*** on the contents of the page (e.g., STORE) is allowed
- X (0x4 = 100): ***Executing*** the contents of the page (CPU instructions) is allowed

Example Page Table for a Process

Kernel can set up protection to allow only necessary types of access

- The pages holding the ***text*** of the program have prot set to RX (0x5 = 101)
- The pages holding the ***data***, ***bss***, and ***heap*** have prot set to RW (0x3 = 011)
  - The division between data, bss, and heap need not be on page boundaries
- The pages holding the ***stack*** have prot set to RW (0x3 = 011)

1	RW	11
1	RW	50
1	RW	2
0	. . .	. . .
1	RW	61
1	RW	15
1	RX	7
1	RX	29

Separate Protection for User vs. Kernel Mode

Our current PTE definition, including the "valid" and "prot" bits

Example: change hardware to define <u>two</u> <u>separate</u> protection fields

- Hardware automatically uses ***kprot*** or ***uprot***, depending on current mode
  - **k**prot if CPU is currently in ***kernel mode***, **u**prot if CPU is in ***user mode***
- ***The kernel can make some pages kernel accessible but not user accessible***

A Problem with Our Current Paging Support

Every memory access requires <u>two</u> memory accesses

- The relevant PTE is required by the hardware in order to translate the original memory address from a ***virtual*** address to a ***physical*** address
- With the page table stored in physical memory, accessing that PTE requires a physical memory read to get the PTE
- Thus, two ***physical*** memory accesses for each original ***one*** access
  - One access to get the relevant PTE, and then
  - One access to do the original program memory access
- Thus, the memory accesses run at basically half-speed (two memory accesses for each one memory access)

Solution: The "Translation Lookaside Buffer" (TLB)

A hardware cache of recently used PTEs

- Typically maintained automatically ***entirely by the hardware***
  - ***Hardware*** extracts the VPN from address being accessed
  - If the PTE for this VPN is not in the TLB, ***hardware*** goes to physical memory to get it (using PTBR and PTLR) and adds PTE to the TLB
  - In either case, ***hardware*** uses TLB copy of PTE to translate the address
- Basically as fast as a complete set of special hardware registers to hold the whole page table, but really just a cache of recently used PTEs
  - Even with a moderately small TLB size, hardware gets a high hit rate

A Simple Example TLB Implementation

A fixed number of entries, built into the hardware

V	VPN	Cached PTE for That VPN
V	VPN	Cached PTE for That VPN
⋮	⋯	⋯
V	VPN	Cached PTE for That VPN
V	VPN	Cached PTE for That VPN

- If the "V" (Valid) bit is set, that TLB entry is "valid" (i.e., it's not "empty")
- ***Hardware*** searches all entries for a valid entry with the needed VPN
- If not found, ***hardware*** loads the needed PTE from physical memory into the TLB, caching it, replacing some other TLB entry if needed

Flushing the TLB

The kernel has a very "narrow window" to interface with the TLB hardware

- The kernel ***cannot*** look into the TLB to see what's in it
- The kernel ***cannot*** explicitly modify any entries in the TLB
- The kernel can only "flush" the TLB
  - Can flush ***all*** entries (set V = 0 for all TLB entries)
  - Or flush the entry for a ***specific*** virtual address (if it's in the TLB)
  - Either one forces PTE to be reloaded from physical memory ***on next use***
- Two reasons the kernel ***must*** flush one or all TLB entries
  - If the kernel ***modifies a PTE*** in memory, to enable the TLB see it
  - Or if the kernel ***context switches*** to a new process, since the two process' VPNs otherwise get confused in TLB entries (the same numbers)

Adding Hardware Address Space ID in the TLB

Another small addition to the hardware

- A new special register ASID to hold a unique id for the current address space
  - Kernel changes this register value on a context switch
  - Sometimes called PID (even though different than software pid)
- Add an ASID field to each TLB entry
  - Hardware records current ASID register value in TLB entry when loading
  - Hardware compares VPN ***and ASID*** fields (and TLB valid bit) on searching for a matching entry
- ***Makes the incoming process after a context switch run much faster***

V	VPN	ASID	Cached PTE
V	VPN	ASID	Cached PTE
V	VPN	ASID	Cached PTE
V	VPN	ASID	Cached PTE