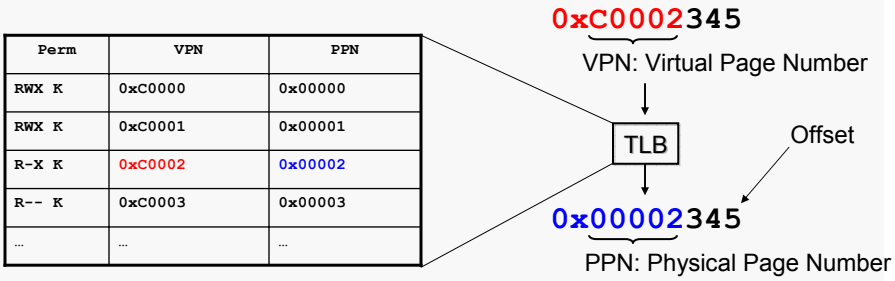
TLB: Translation Look-Aside Buffer

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

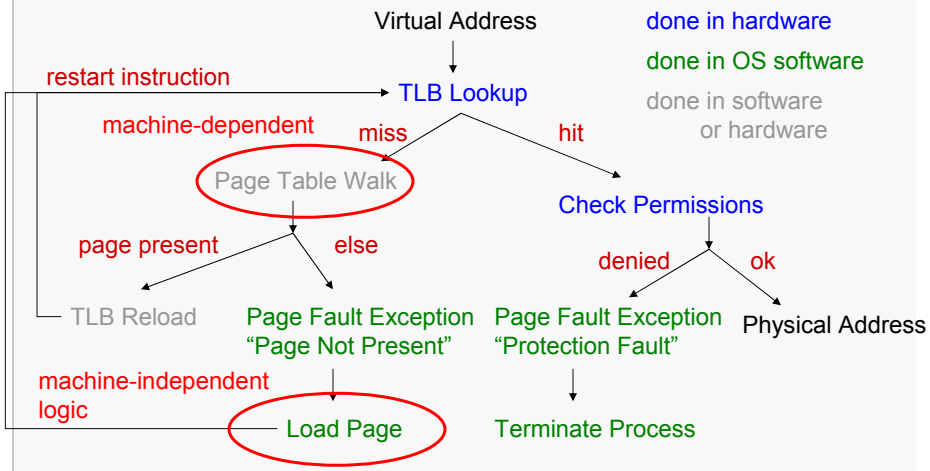
Then: if process changes, some or all TLB entries may become invalid

- 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



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"

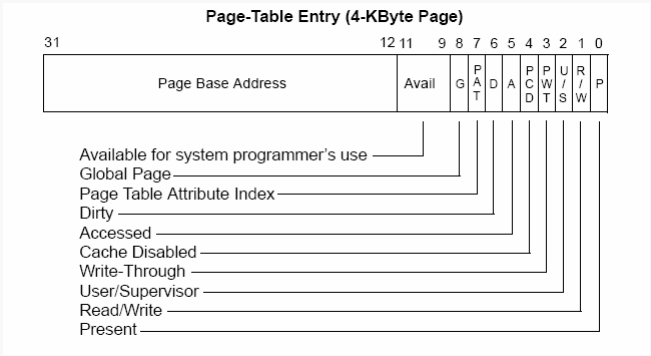
For software-reloaded TLB: relatively easy

- TLB will only contain what OS handlers place into it

For hardware-reloaded TLB: two choices

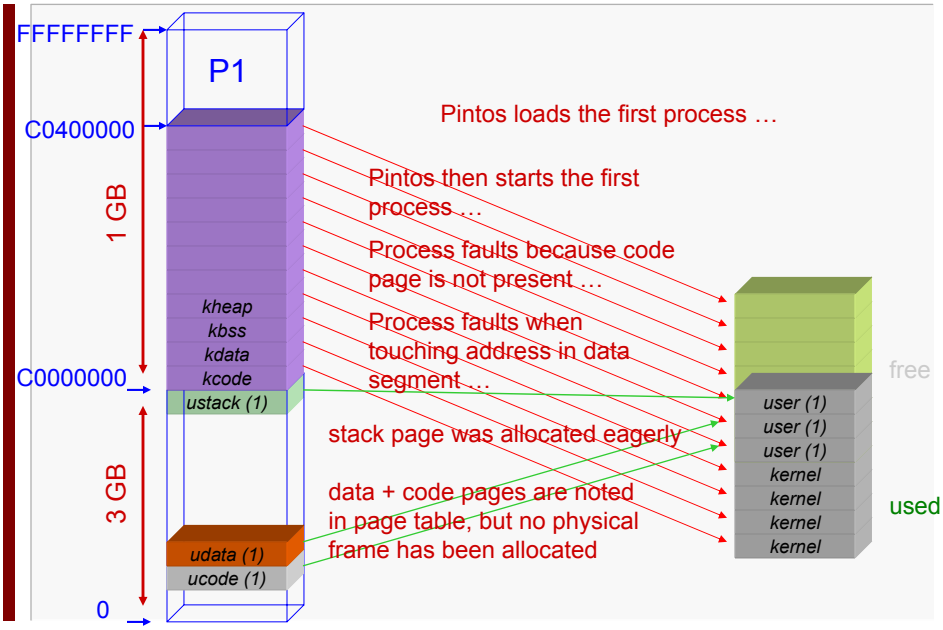
- Use same data structures for page table walk & page loading (hardware designers reserved bits for OS's use in page table)
- Use a layer on top (facilitates machine-independent implementation) – this is the recommended approach for Pintos Project 3
  - In this case, must update actual page table (on x86: "page directory") that is consulted by MMU during page table walk
  - Code is already written for you in pagedir.c

Example: x86 Page Table Entry

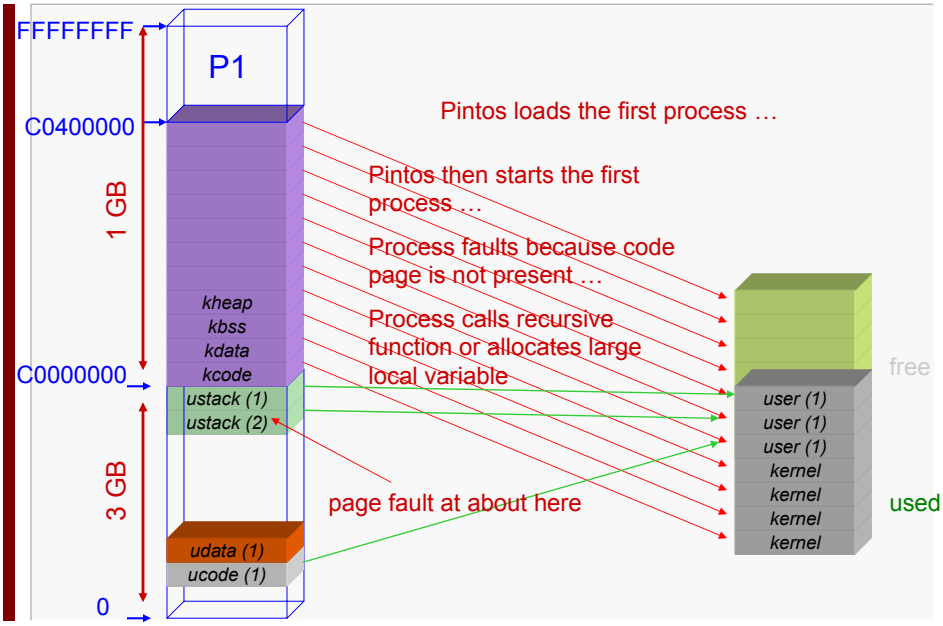


Note: if bit 0 is 0 ("page not present") MMU will ignore bits 1-31 – OS can use those at will

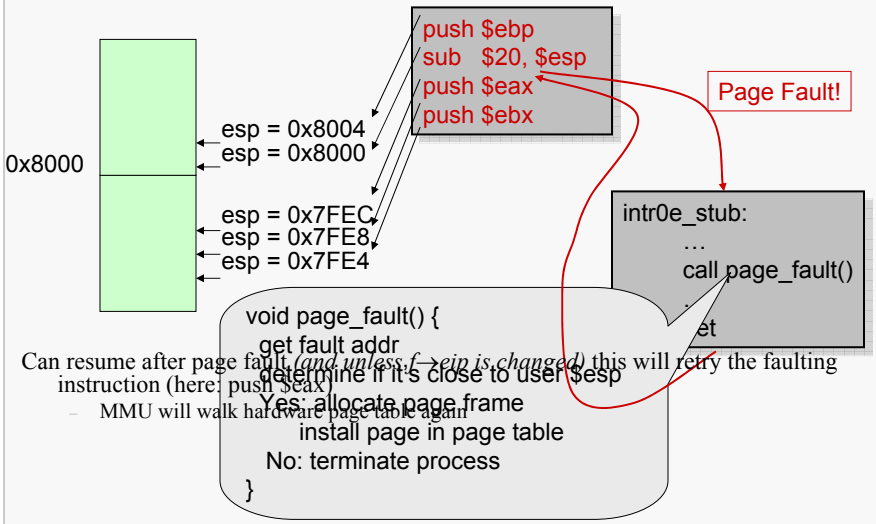
Lazy Loading



Stack Growth



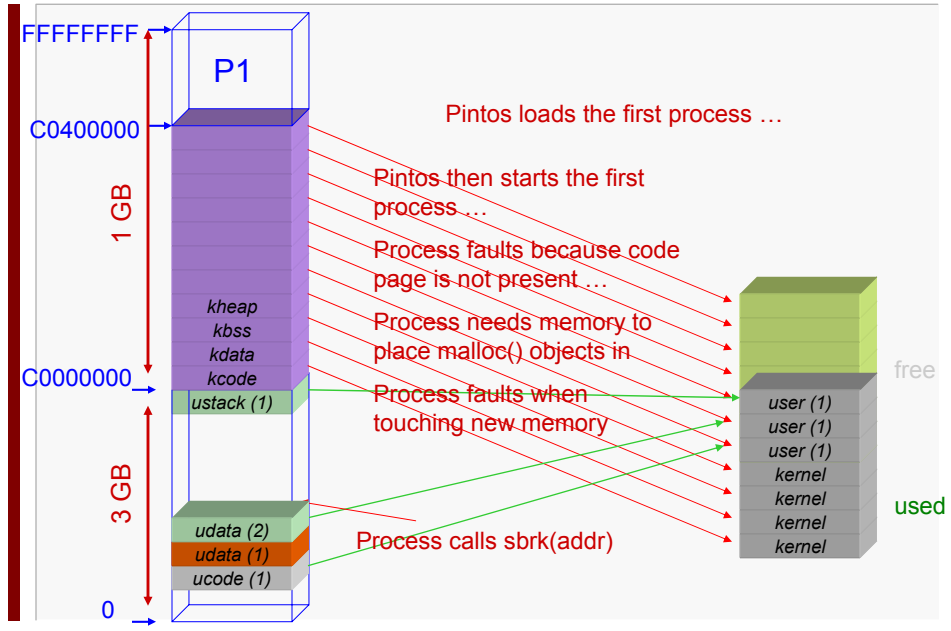
Microscopic View of Stack Growth



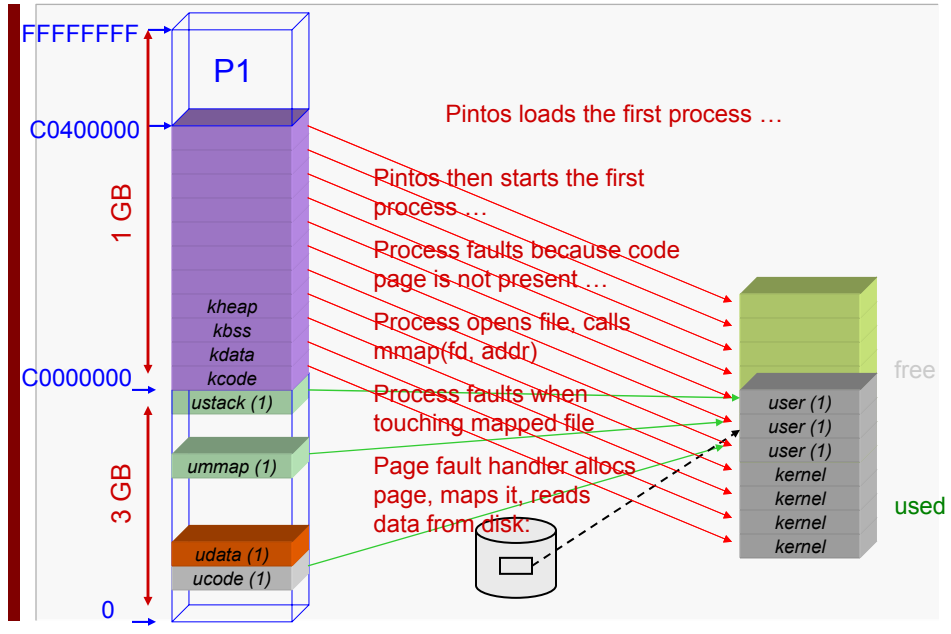
Can resume after page fault *(and unless $f->eip is changed)* this will retry the faulting instruction (here: push $eax)

- MMU will walk hardware page table again

Heap Growth



mmap()



Copy-On-Write

Sometimes, want to create a copy of a page:

- Example: Unix fork() creates copies of all parent's pages in the child

Optimization:

- Don't copy pages, copy PTEs – now have 2 PTEs pointing to frame
- Set all PTEs read-only
- Read accesses succeed
- On Write access, copy the page into new frame, update PTEs to point to new & old frame

Looks like each have their own copy, but postpone actual copying until one is writing the data

- Hope is at most one will ever touch the data – never have to make actual copy