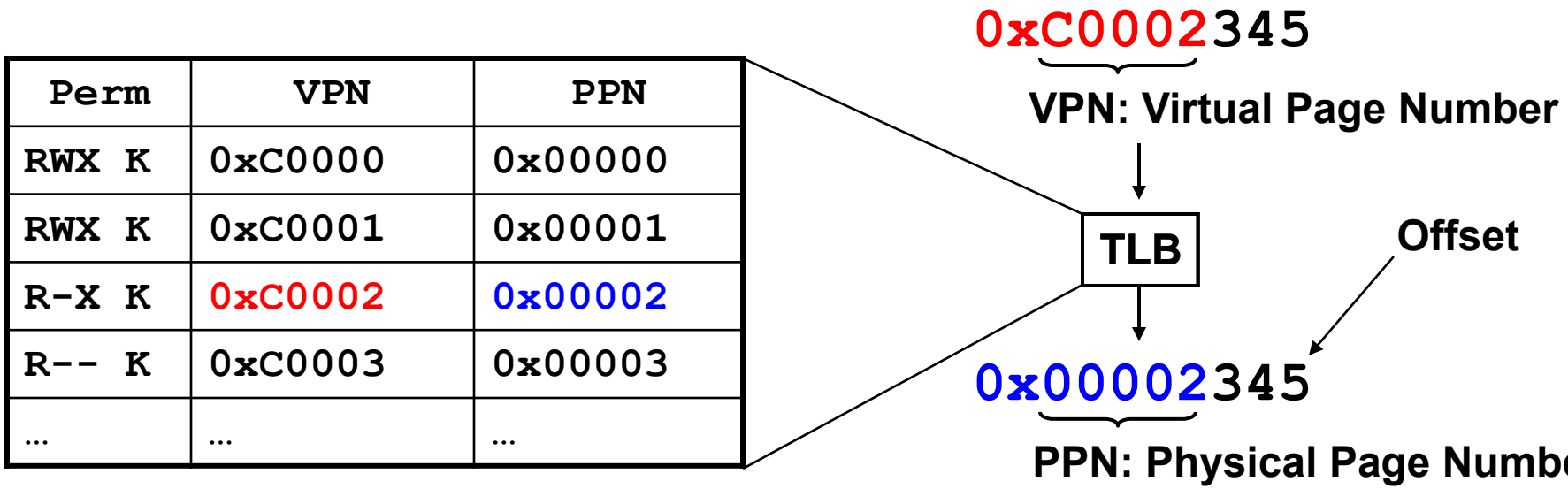
CS3204: Operating Systems

Lecture 17: Virtual Memory: *TLB, Demand Paging*

Presenter: Ali R. Butt

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*)

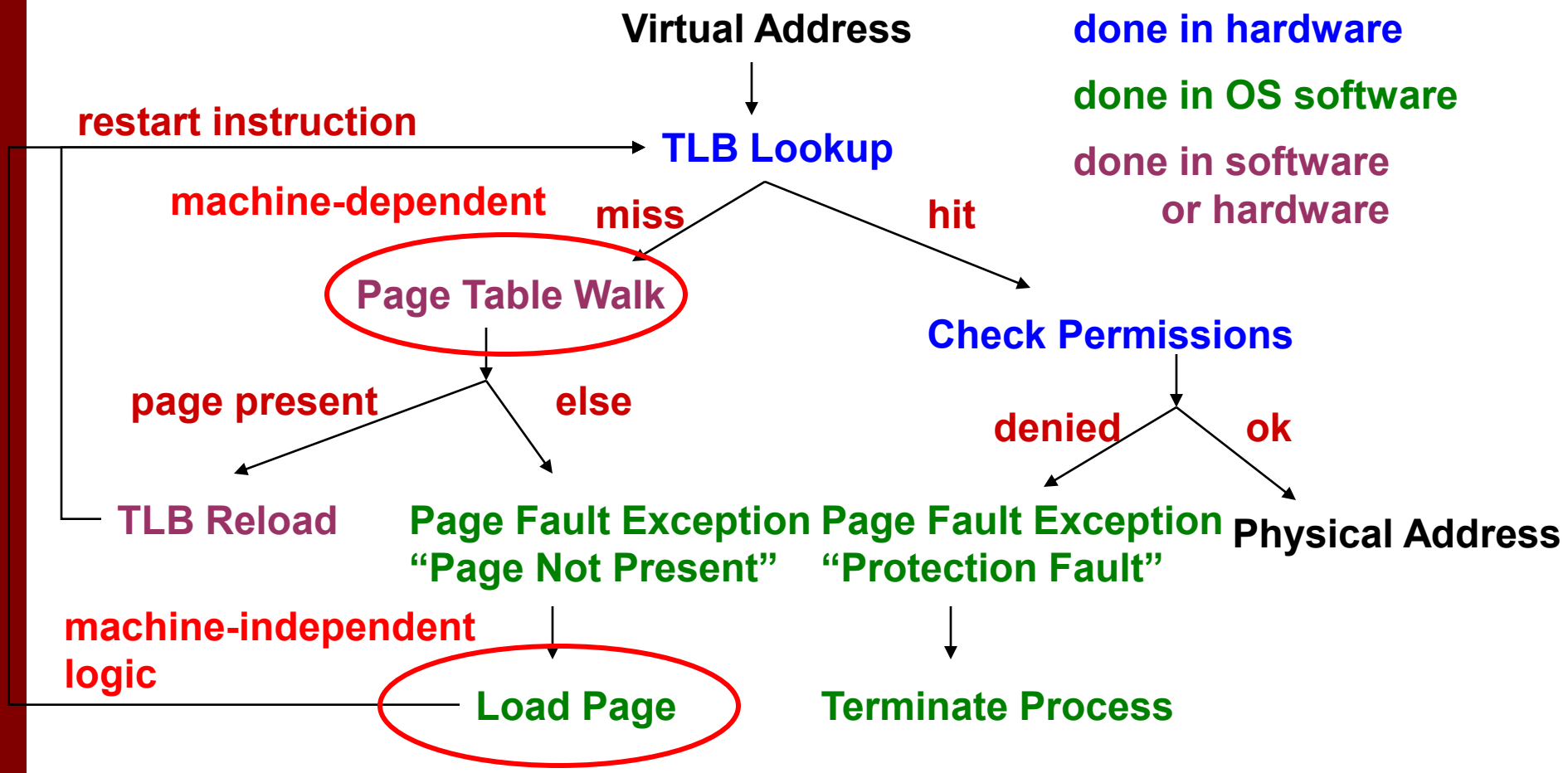


Perm	VPN	PPN
RWX K	0xC0000	0x00000
RWX K	0xC0001	0x00001
R-X K	0xC0002	0x00002
R-- K	0xC0003	0x00003
…	…	…

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)

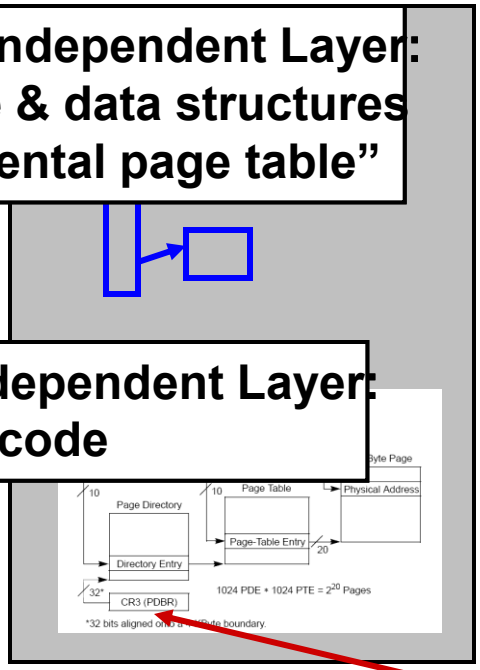
Page Tables vs TLB Consistency

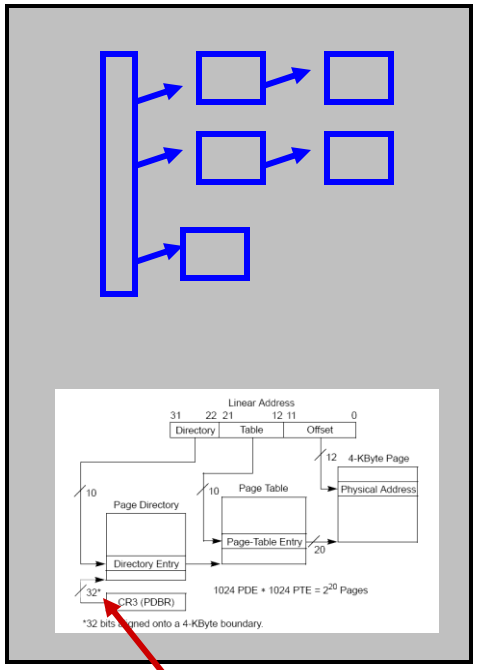
- No matter which method is used, OS must ensure that TLB & page tables are consistent
  - On multiprocessor, this may require "TLB shootdown"
- For software-reloaded TLB: relatively easy
  - TLB will only contain what OS handlers place into it
- For hardware-reloaded TLB: two choices
  - Use same data 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

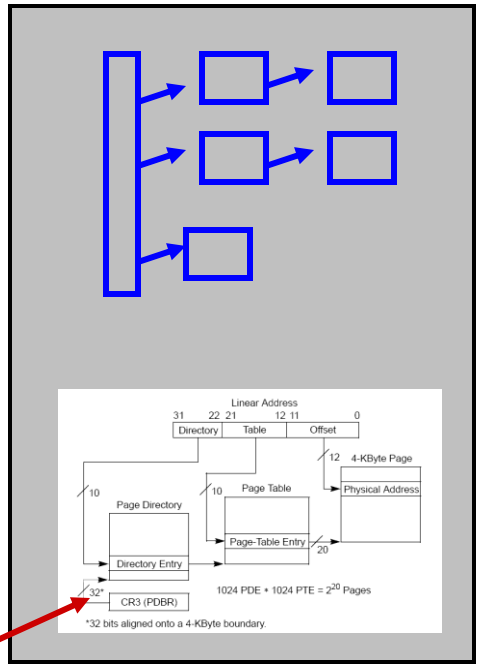
Hardware/Software Split in Pintos

Machine-independent Layer: your code & data structures "supplemental page table"

Machine-dependent Layer: pagedir.c code







CPU cr3

Representing Page Tables

- Choice impacts speed of access vs size needed to store mapping information:
  - Simple arrays (PDP-11, VAX)
    - Fast, but required space makes it infeasible for large, non-continuous address spaces
  - Search trees (aka "hierarchical" or "multi-level" page tables)
  - Hash table

Two-level Page Table

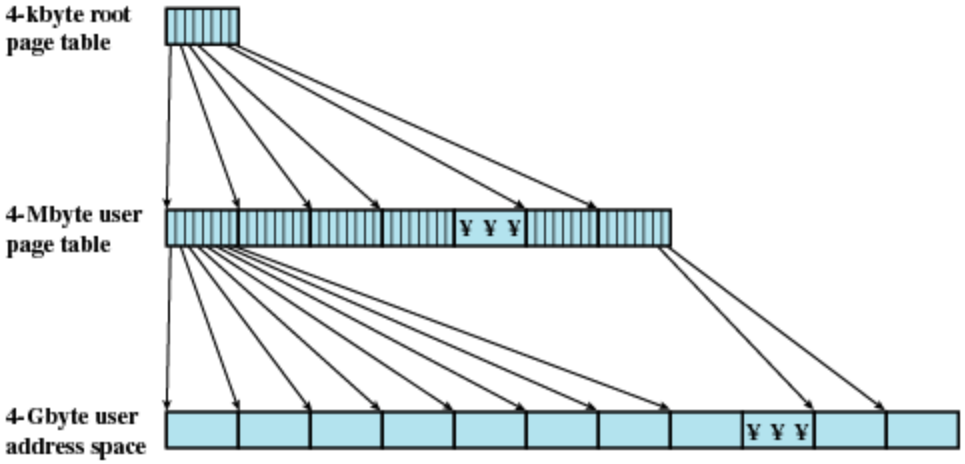
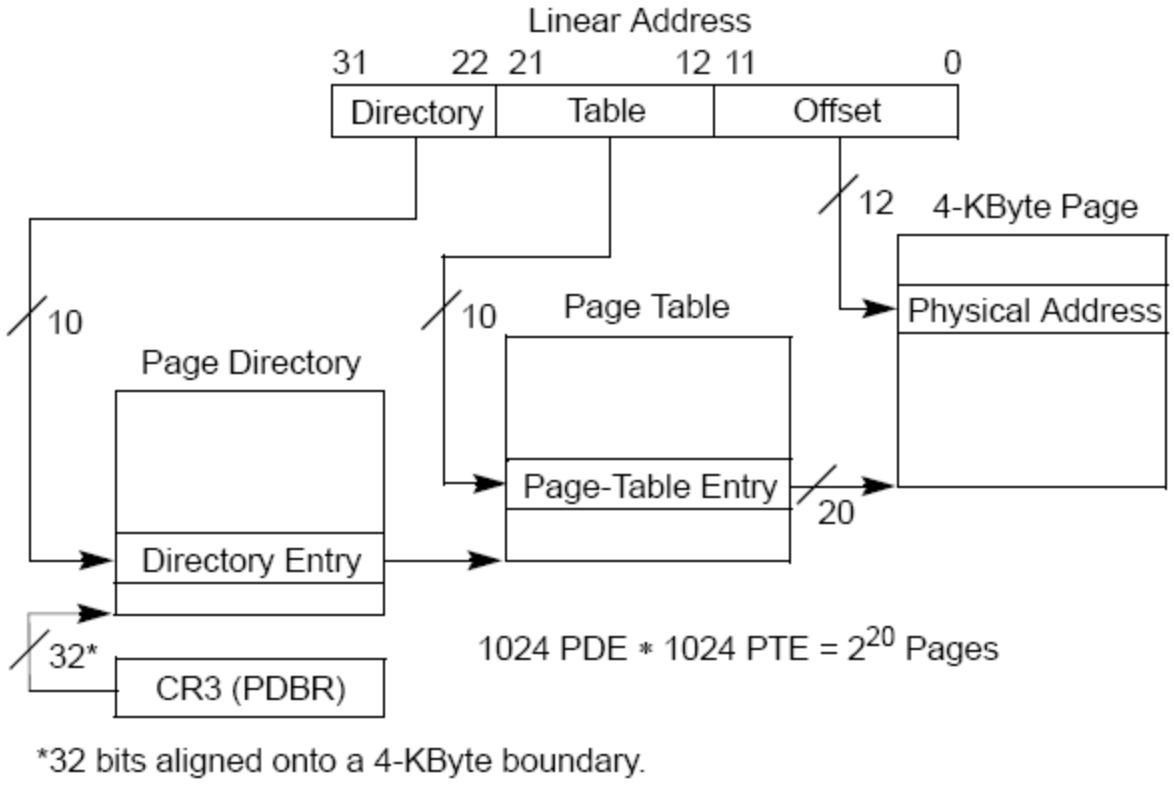


Figure 8.4 A Two-Level Hierarchical Page Table

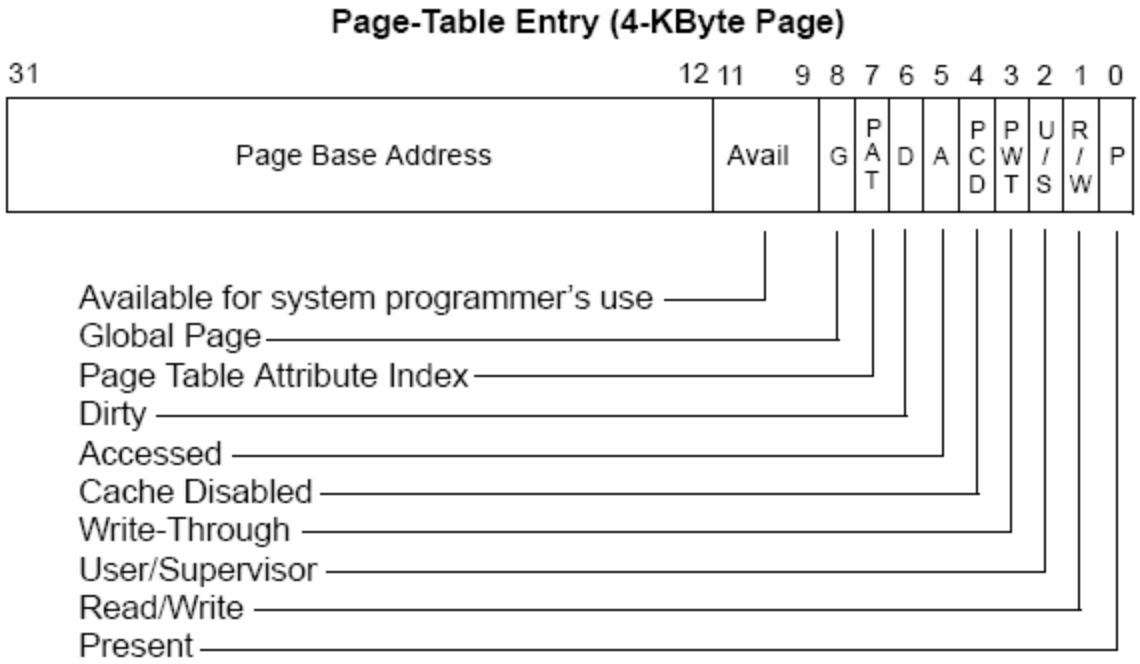
- Q.: how many pages are needed in
  - Minimum case
  - Worst case? (what is the worst case?)

Example: x86 Address Translation



- Two-level page table
- Source: [IA32-v3] 3.7.1

Example: x86 Page Table Entry

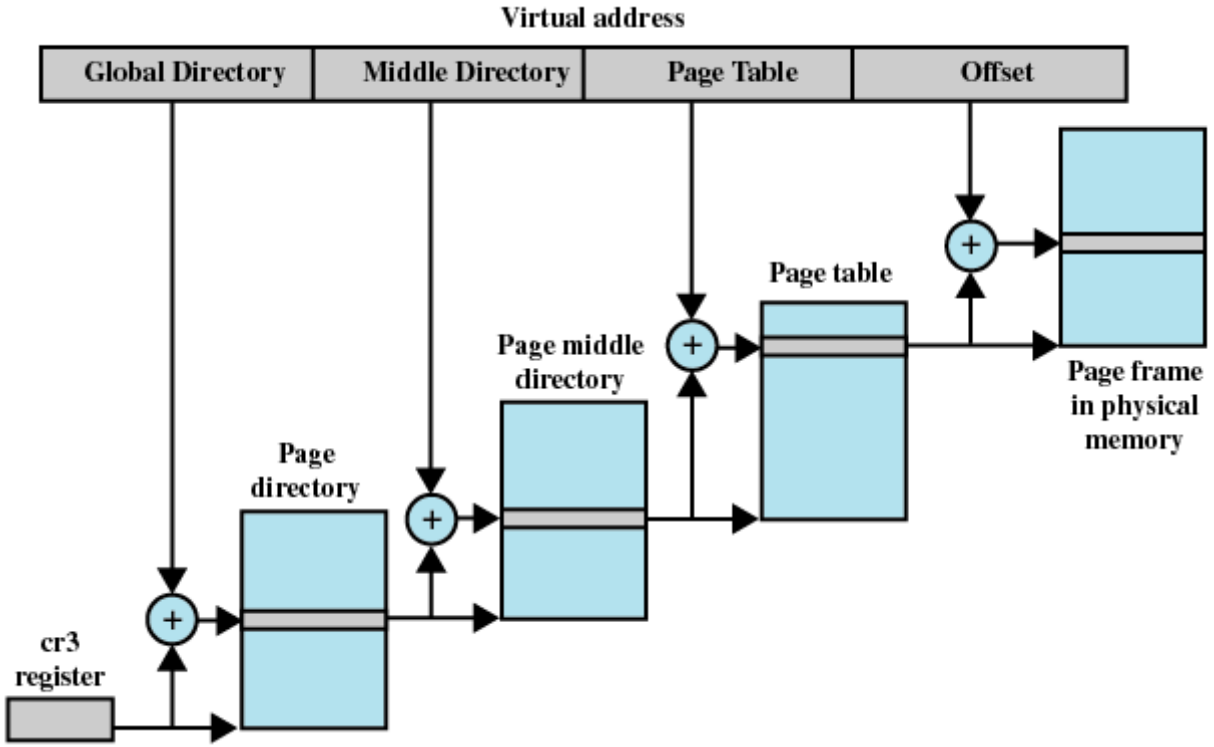


- Note: if bit 0 is 0 ("page not present") MMU will ignore bits 1-31 – OS can use those at will

Page Table Management on Linux

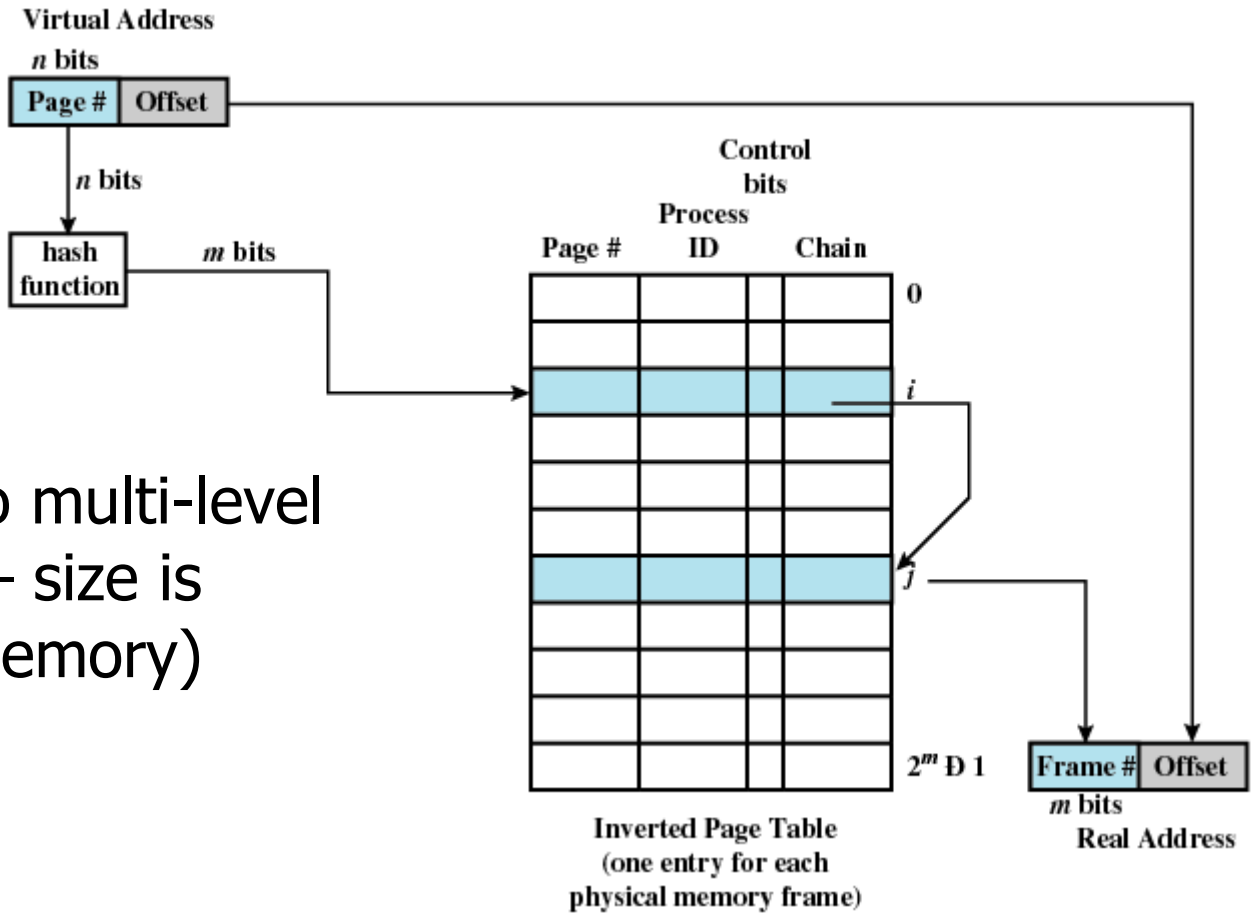
- Interesting history:
  - Linux was originally x86 only with 32bit physical addresses. Its page table matched the one used by x86 hardware
  - Since:
    - Linux has been ported to other architectures
    - x86 has grown to support 36bit physical addresses (PAE) – required 3-level page table
- Linux's now uses 4-level page table to support 64-bit architectures

Linux Page Tables (2)



- On x86 – hardware == software
  - On 32-bit (no PAE) middle directory disappears
- With four-level, "PUD" page upper directory is added (not shown)

Inverted Page Tables



- Alternative to multi-level page tables – size is O(physical memory)

Summary

- Page tables store mapping information from virtual to physical addresses, or to find non-resident pages
  - Input is: process id, current mode (user/kernel) and kind of access (read/write/execute)
- TLBs cache such mappings
- Page tables are consulted when TLB miss occurs
  - Either all software, or in hardware
- OS must maintain its page table(s) and, if hardware TLB reload is used, the page table (on x86 aka "page directory + table") that is consulted by MMU
  - These two tables may or may not be one and the same
- The OS page table must have sufficient information to load a page's content from disk

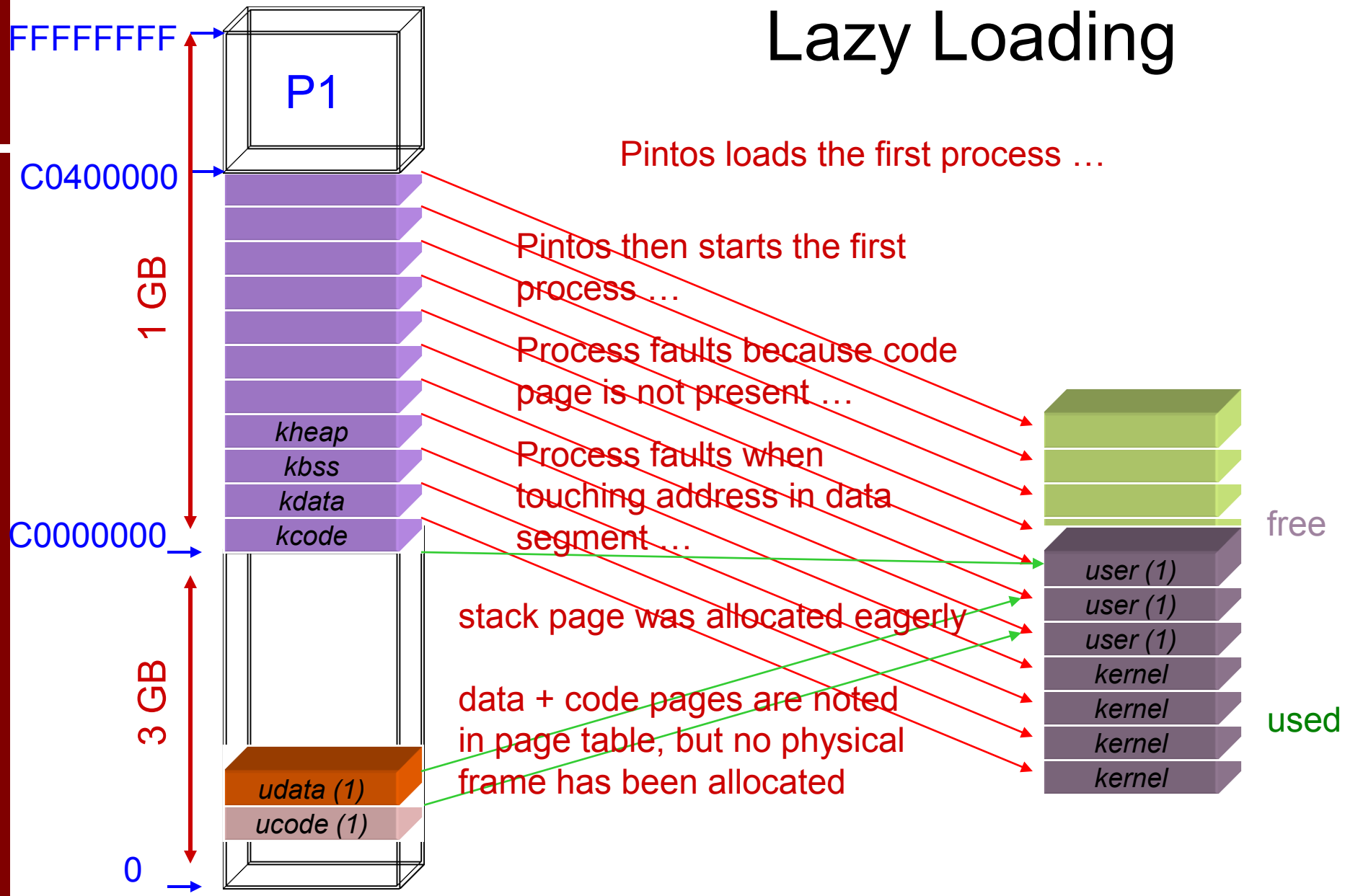
Virtual Memory

Paging Techniques

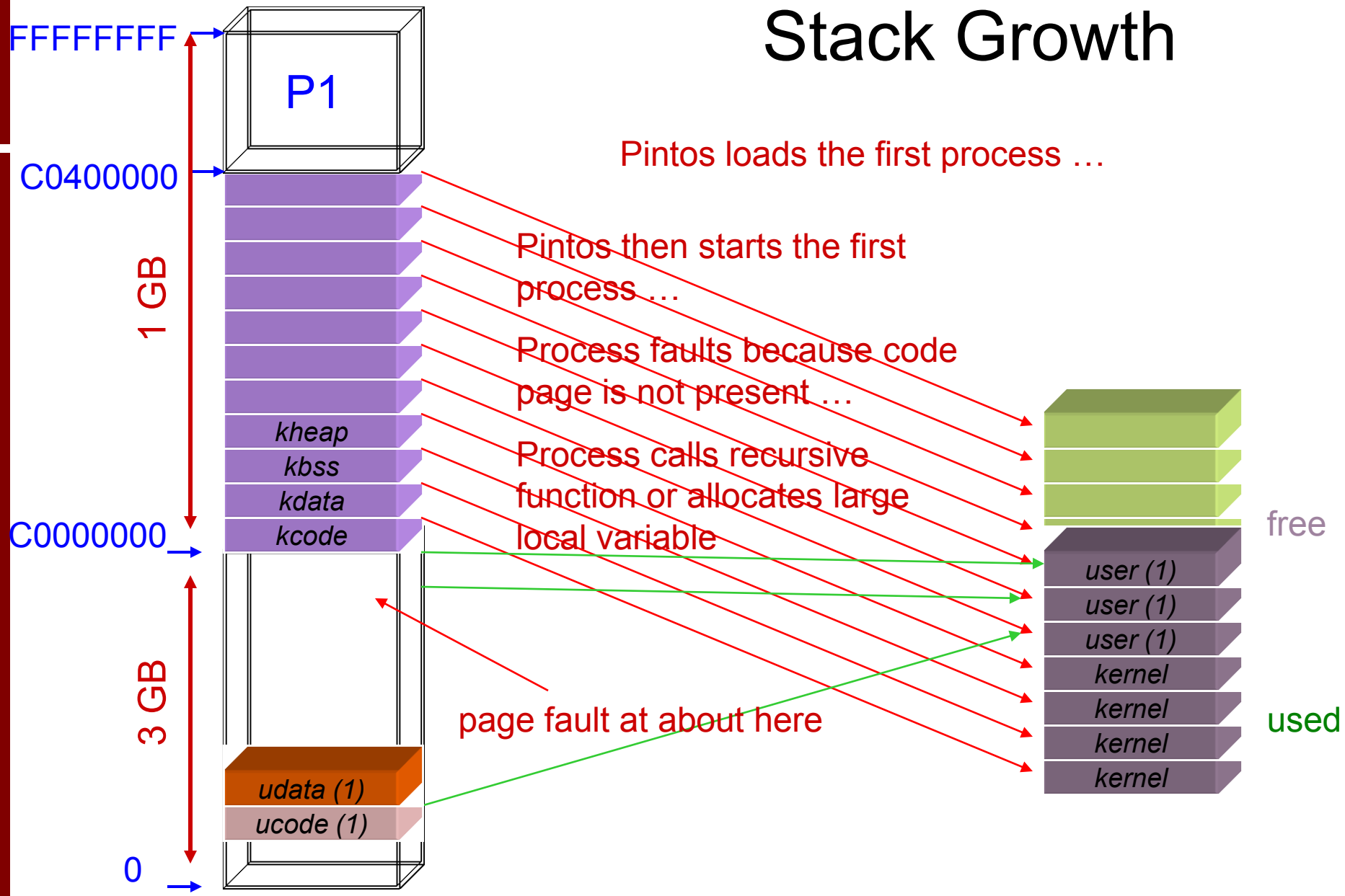
Demand paging

- Idea: only keep data in memory that's being used
  - Needed for virtualization – don't use up physical memory for data processes don't access
- Requires that actual allocation of physical page frames be delayed until first access
- Many variations
  - Lazy loading of text & data, mmapped pages & newly allocated heap pages
  - Copy-on-write

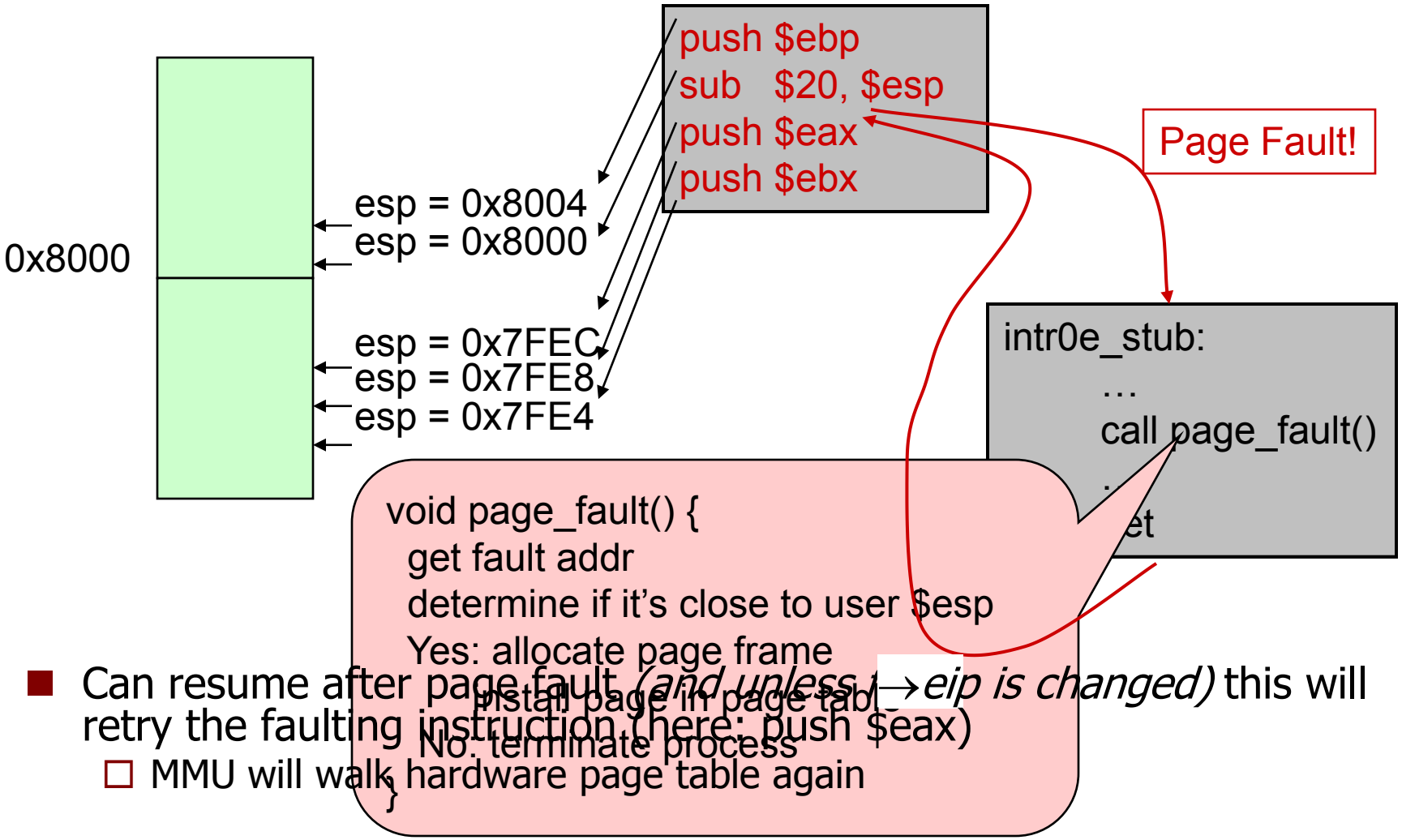
Lazy Loading



Stack Growth



Microscopic View of Stack Growth



- Can resume after page fault *(and unless $i \rightarrow$ eip is changed)* this will retry the faulting instruction (here: push $eax)
  - MMU will walk hardware page table again

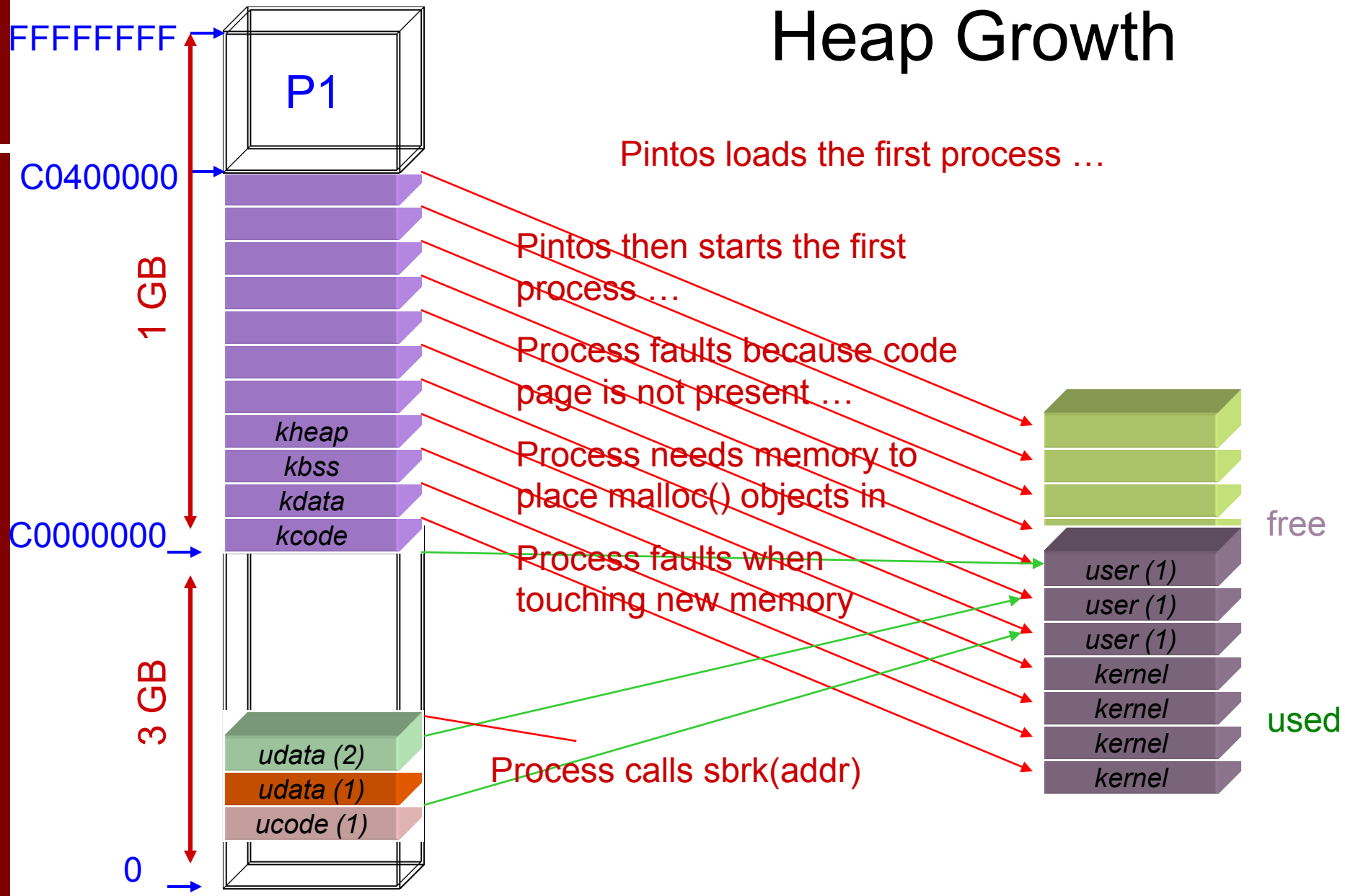
Fault Resumption

- Requires that faulting CPU instruction be restartable
  - Most CPUs are designed this way
- Very powerful technique
  - Entirely transparent to user program: user program is frozen in time until OS decides what to do
- Can be used to emulate lots of things
  - Programs that just ignore segmentation violations (!?) (here: resume with next instruction – retrying would fault again)
  - Subpage protection (protect entire page, take fault on access, check if address was to an valid subpage region)
  - Virtual machines (vmware, qemu – run entire OS on top of another OS)
  - Garbage collection
  - Distributed Shared Memory

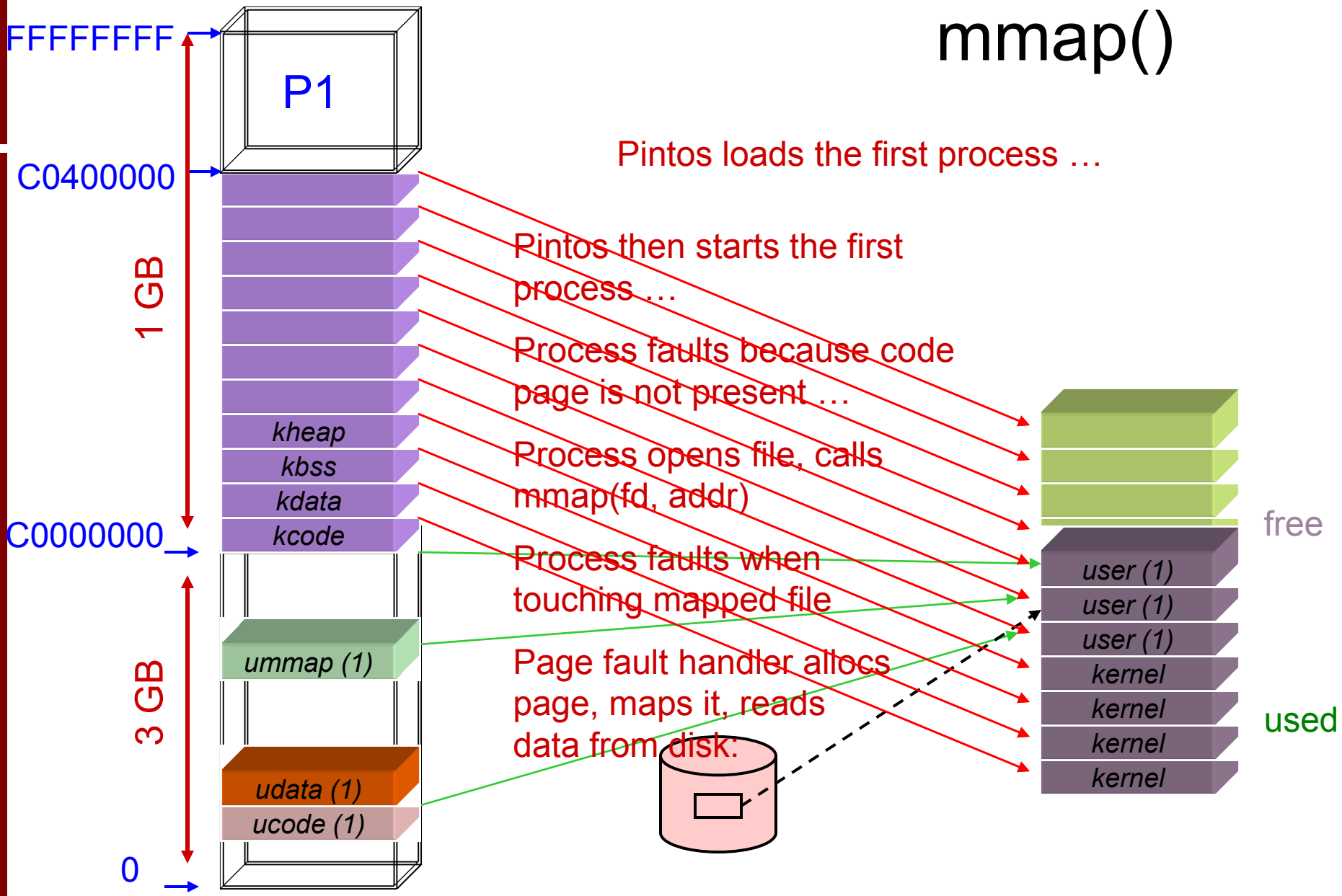
Distributed Shared Memory

- Idea: allows accessing other machine's memory as if it were local
- Augment page table to be able to keep track of network locations:
  - local virtual address $\rightarrow$ (remote machine, remote address)
- On page fault, send request for data to owning machine, receive data, allocate & write to local page, map local page, and resume
  - Process will be able to just use pointers to access all memory distributed across machines – fully transparent
- Q.: how do you guarantee consistency?
  - Lots of options

Heap Growth


FFFFFFFF
P1
C0400000
1 GB
kheap
kbss
kdata
kcode
C0000000
3 GB
udata (2)
udata (1)
ucode (1)
0
Pintos loads the first process …
Pintos then starts the first process …
Process faults because code page is not present …
Process needs memory to place malloc() objects in
Process faults when touching new memory
Process calls sbrk(addr)
user (1)
user (1)
user (1)
kernel
kernel
kernel
kernel
free
used

mmap()


FFFFFFFF
P1
C0400000
1 GB
kheap
kbss
kdata
kcode
C0000000
3 GB
ummap (1)
udata (1)
ucode (1)
0
Pintos loads the first process …
Pintos then starts the first process …
Process faults because code page is not present …
Process opens file, calls mmap(fd, addr)
Process faults when touching mapped file
Page fault handler allocs page, maps it, reads data from disk:
user (1)
user (1)
user (1)
kernel
kernel
kernel
kernel
free
used

Lazy Loading & Prefetching

- Typically want to do some prefetching when faulting in page
  - Reduces latency on subsequent faults
- Q.: how many pages? which pages?
  - Too much: waste time & space fetching unused pages
  - Too little: pay (relatively large) page fault latency too often
- Predict which pages the program will access next (how?)
- Let applications give hints to OS
  - If applications knows
  - Example: madvise(2)
  - Usual conflict: what's best for application vs what's best for system as a whole