ECE 531 – Advanced Operating Systems
Lecture 21

Vince Weaver

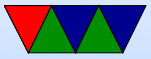
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

27 October 2025

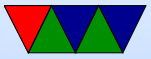
Announcements

- HW#6 was posted

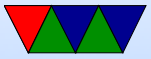


Virtual Memory Continued

This is a hard concept, so go over things again

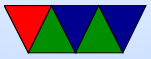


Quick run-through, the path of a load

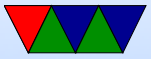


Load Path – First Hardware Stuff

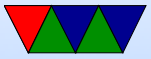
- For this class we can mostly ignore this
- On an Out-of-Order processor there are potentially a lot of corner cases, out-of-order loads, load buffers, etc
- Caches cause lots of complications
- Caches make fast, local copies of memory on the CPU for fast access. They keep track of what memory they have by address, so they might have to be virtual-memory aware
  - Physically Indexed/Physically Tagged caches (PIPT)



only use physical addresses, so the cache happens after the TLB
  - Virtually Indexed/Virtually Tagged (VIVT) happen before the TLB but there are complications (what happens on context switch? flush?)
  - Virtually Indexed/Physically Tagged (VIPT). Our Pi boards are this. The TLB lookup happens at the same time as the cache lookup which can be faster
- Also remember the CPU is keeping things cache coherent too which adds complications.
- Each core has its own TLB which has performance

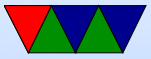


implications



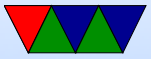
Load Path – the TLB

- If tag from TLB matches a tag from cache, hit! Good! Cache hit!
- If tag in TLB but not in cache, cache miss.
- If tag not in TLB, TLB miss. Won't know if cache hit until later.
- If TLB miss need to walk the page tables



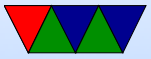
Load Path – Walking the Page Tables

- Now let the hardware walk the page tables.
- If hardware finds the page, great! Return it back up to the TLB
- If hardware can't find the page, time to get the Operating System involved. Page fault.

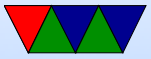


Load Path – OS Lookup

- OS has a list of what should be in memory where (from the executable). Typically these are demand-loaded
  - Text/code — read from disk
  - Data — read from disk
  - BSS — allocate zeros
  - Stack — if near top growing down, auto-grow
  - Heap — similar to stack
  - Shared page — could already be in memory (shared lib?) Just need to point to it.

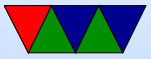


- Zeros — just have one page of zeros you can point to
- Paged out to disk — have offset in page file, need to load it
- What if page is invalid/not part of process? segfault!



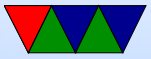
Load Path – Making Room for a Page

- Time to bring in the page!
- Need to find room in Physical RAM. Search RAM map for a free page
- If no room, need to make room. Possibly paging out to disk (this is what LRU/dirty bits are used for).
- We will discuss what happens if no memory available later



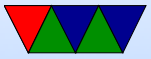
Load Path – We now have a physical page allocated!

- Page now in physical RAM
- Fill the page with contents (zero, data from disk)
- Add a new PTE that points to the page
- Insert the PTE into the page tables. Depending on how full the existing page tables are you might need to create new ones and this involves allocating space in memory too



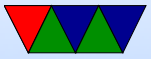
Load Path – Finishing Up

- Fill in the TLB with the new entry
  This might involve kicking out the oldest TLB entry to make room
- Need to return to userspace
  - Unlike other interrupts, we want to re-run the instruction this time hopefully it will complete successfully



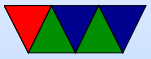
Load Path – Possible Issues

- Could you have race where you re-execute it and the page had gotten swapped out again?
- Can we page out the page tables? What can go wrong there? Double faults? How many nested page faults



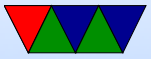
Page Fault Exceptions

- On ARM32 at least a page fault triggers like an interrupt
- Memory Abort if a load/store fails
- Prefetch Abort if an instruction fetch fails
- This calls into the operating system and starts the page fault handling
- Generally when done you return to the faulting instruction to be re-executed, and hopefully this time the memory access works



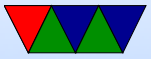
What happens in Page Fault Handler

- The OS process structure has info on what memory regions are valid and what should be there
  - text/data comes from executable on disk
  - bss pages pre-zeroed by OS
  - heap/stack might be auto-allocated zero pages
- The OS determines if the access is a valid region, if not, SEGFAULT
- If it is valid, it will bring things up and add pagetable entry



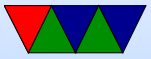
page fault types

- "minor" fault – page is already in memory, just need to point a PTE at it. For example, shared memory, shared libraries, etc.
- "major" fault – page needs to be created or brought in from disk
  - Demand paging
  - Needs to find room in physical memory. If no free space available, needs to kick something out. Disk-backed (and not dirty) just discarded. Disk-backed



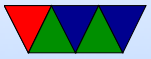
and dirty, written back.

- Memory can be paged to disk. Eventually can OOM.
- Memory is then loaded, or zeroed, and PTE updated.
- Can it be shared? (zero page)



Virtual Address on Linux

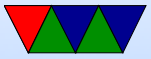
- TODO: research these a bit more
- You can view these under /proc on Linux /proc/pid/maps had virtual maps (in text format)
- /proc/pid/mem
- /proc/pid/pagemaps



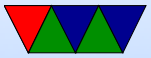
Uses of VM in an operating system

Some of these were initially hacks that were convenient on systems with VM available

- Process separation, security
- Each process own view of memory
- Kernel mapped into each process address space
- Auto-growing stack
- zero page?
- Memory overcommit
- Demand paging

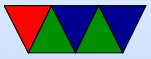


- Copy-on-write with fork



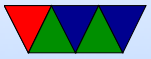
Each process has a page table

- When you context switch, simply update the hardware pointer to this
- On x86, CR3
- ARM: TTBR0

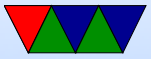


Memory Protection

- Can mark pages as read-only, execute-only, etc
  - Code might be read-only
  - Stack might want read/write but no execute
  - Some of data segment (const) might be read-only
- Why is this good?
- x86 late with NX (no-execute support) only came in with PAE support when larger pagetable option introduced and there was room to add bits. Good for security but meant separate kernel that handled it on both Linux and

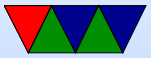


Windows.



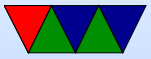
What happens on a fork?

- Do you actually copy all of memory?
  Why would that be bad? (slow, also often exec() right away)
- Page table marked read-only, then shared
- Only if writes happen, take page fault, then copy made
  Copy-on-write (COW)



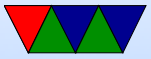
What happens when Low on Memory

- What kind of issues come up when low on RAM and constantly paging same pages in and out (thrashing?)



What happens on out-of-memory

- Crash the operating system?
- Linux will run an OoM (Out of Memory) killer that tries to kill off the worst offender. Often gets it wrong and kills something useful
- Related issue, "overcommit"
  - Modern programs rarely use whole working set
  - Let programs allocate large chunk of RAM without error on assumption they won't use it all
  - Problem if they do try to use it all and RAM isn't



actually there

- You can turn off overcommit, but then have problem where fewer programs can fit on a system