ECE 598 – Advanced Operating Systems
Lecture 14

Vince Weaver

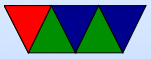
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

22 March 2018

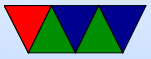
Announcements

- HW#6 was due.
- HW#7 will be posted eventually.
- Project posted. See course website for details. Topics due 29 March.
- Midterms not graded yet.



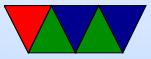
Pi-3B+ Notes

- Pi3B+ released over break (on Pi day)
- Ordered one
- Not sure if it will make a difference in this class
- Bigger difference for Pi cluster. Gigabit ethernet (still not staturate), better thermal support.
- Still only 1GB of RAM. Partly SoC limitation, but also they said it's because of high DRAM prices

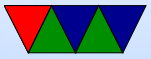


HW#6 Notes

- Sorry for the delay getting it out. Part of it was finally getting device tree support (it detects memory properly now, as well as should tell you pi2/pi3)
- Processes
  - Meant to have you do more work on the scheduling side. Getting A/B working is always a huge milestone in making your own OS
  - Took me weeks the first time, thought I could simplify it down. But no, still a huge mass of assembly and

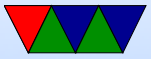


had trouble sorting it out (lack of comments and code 2 years old!)
  - Each process has a structure that holds info on it, plus the save state.
- Userspace/Executables
  - Entering into userspace for first time is a pain.
  - Previous homework just called a function and treated as an exe, but that a bit of a hack.
  - So had to implement executables (right now, bare code/data blob. A problem as working on HW#7 issue with BSS not actually being allocated so program



crashes) Working on bFLT support.

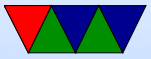
- Filesystem
  - But, needed a place for them to live. So a simple ramdisk and romfs (we'll talk about filesystems soon)
- Fork/Exec
  - How to executables start? Unix we mentioned fork/exec. However a true fork requires virtual memory and we don't have that yet.
  - So there's a stripped-down version of fork called vfork() you use in this case. The way it works is that as soon as you fork, the parent goes to sleep and the child



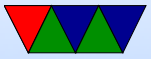
is running inside the parent and the *only* thing the child it is allowed to do is either call exec or _exit() (not even plain exit() as that would exit the parent)

  - execve you pass in the program you want to run, as well as the command line arguments. It loads from disk the executable, allocates memory, sets up the process, marks as ready to run.

- Scheduler/Idle Thread
  - How does the scheduler work? Simple, nothing fancy. There's a doubly linked-list of all processes and when a timer interrupt happens the list is walked to find the

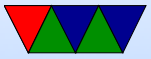


next one that's runnable.
  - What if none available? Then run the idle thread.
- Waitqueues
  - Also implements wait queues. If you are sleeping (because of a vfork) or waiting on I/O (waiting for keypress) you get put to sleep and put on a linked-list waitqueue. Then when I/O comes in, you are woken up, removed from the queue, and marked as ready.
  - This is tricky as in theory you are sort of sleeping in the kernel and that's how we implement it, so we need to save our kernel register state as well as the user

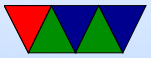


space. There's probably better ways to do this.

- Waitpid
  - In UNIX like operating systems once you have children via fork, if they die they don't go away. zombies. You can wait using waitpid() to see when they die, and once you use waitpid they are finally freed.
  - So how do you wait in the background like in the HW? Had to implement waitpid(NOHANG) which means check to see if any children have died. If not, continue. Otherwise handle them so they can die.
  - So in the shell after every command is typed it does

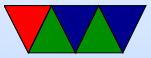


a waitpid(NOHANG) to see if any of the background tasks finished.

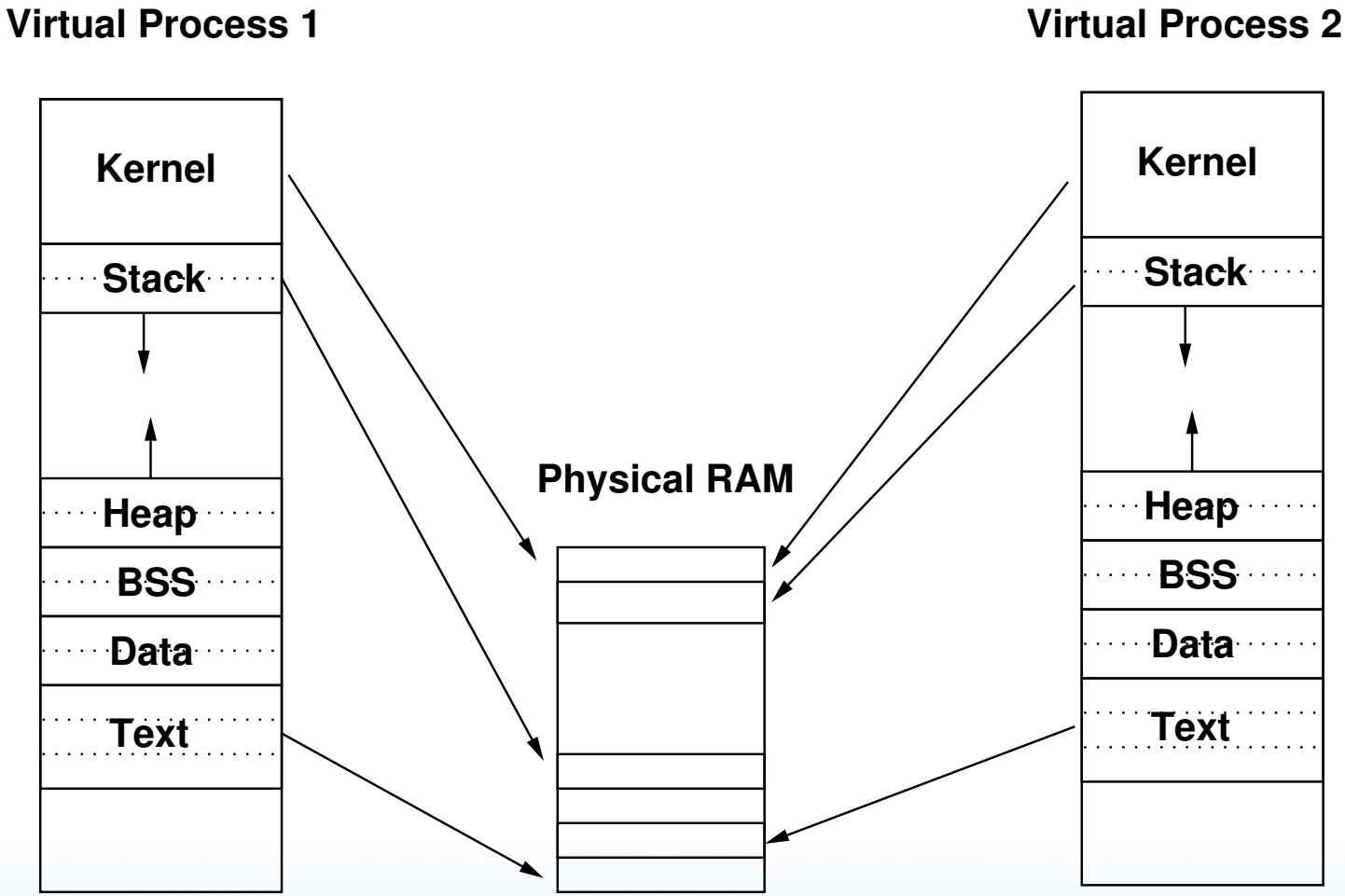


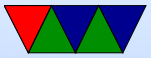
Virtual Memory Redux

- Original purpose was to give the illusion of more main memory than available, with disk as backing store.
- Give each process own linear view of memory.
- Demand paging (no swapping out whole processes).
- Execution of processes only partly in memory, effectively a cache.
- Memory protection



Diagram

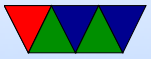

Virtual Process 1
Virtual Process 2
Kernel
Stack
Heap
BSS
Data
Text
Physical RAM
Kernel
Stack
Heap
BSS
Data
Text



Memory Management Unit

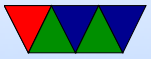
Can run without MMU. There's even MMU-less Linux.

How do you keep processes separate? Very carefully...



Page Table

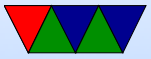
- Collection of Page Table Entries (PTE)
- Some common components: ID of owner, Virtual Page Number, valid bit, location of page (memory, disk, etc), protection info (read only, etc), page is dirty, age (how recent updated, for LRU)



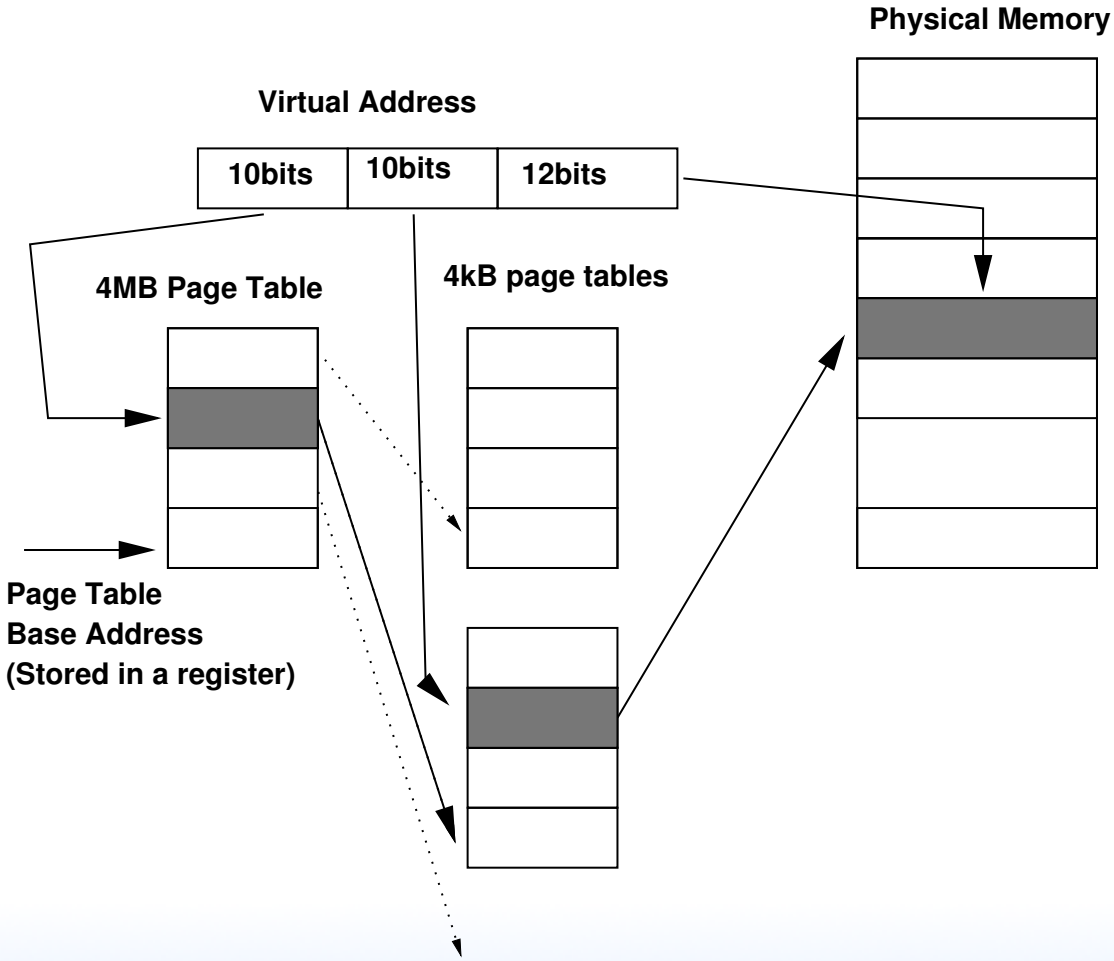
Hierarchical Page Tables

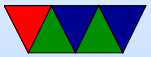
- With 4GB memory and 4kb pages, you have 1 Million pages per process. If each has 4-byte PTE then 4MB of page tables per-process. Too big.

- It is likely each process does not use all 4GB at once. (sparse) So put page tables in swappable virtual memory themselves!
  4MB page table is 1024 pages which can be mapped in 1 4KB page.



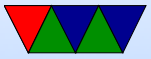
Hierarchical Page Table Diagram





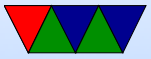
Hierarchical Page Table Diagram

- 32-bit x86 chips have hardware 2-level page tables
- ARM 2-level page tables



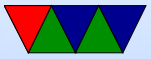
Walking the Page Table

- Can be walked in Hardware or Software
- Hardware is more common
- Early RISC machines would do it in Software. Can be slow. Has complications: what if the page-walking code was swapped out?

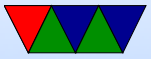


TLB

- Translation Lookaside Buffer
  (Lookaside Buffer is an obsolete term meaning cache)
- Caches page tables
- Much faster than doing a page-table walk.
- Historically fully associative, recently multi-level multi-way
- TLB shootdown – when change a setting on a mapping

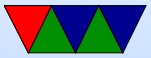


and TLB invalidated on all other processors



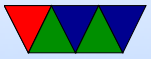
Flushing the TLB

- May need to do this on context switch if doesn't store ASID or ASIDs run out (ASID=Address Space ID)
- Sometimes called a "TLB Shootdown"
- Hurts performance as the TLB gradually refills
- Avoiding this is why the top part is mapped to kernel under Linux



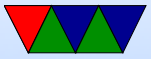
What happens on a memory access

- If in TLB, not a problem, right page fetched from physical memory, TLB updated
- If not in TLB, then the page tables are walked
- It no physical mapping in page table, then page fault happens



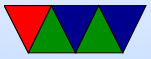
What happens on a page fault

- Walk the page table and see if the page is valid and there
- "minor" – page is already in memory, just need to point a PTE at it. For example, shared memory, shared libraries, etc.
- "major" – page needs to be created or brought in from disk. Demand paging.
  Needs to find room in physical memory. If no free space



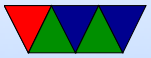
available, needs to kick something out. Disk-backed (and not dirty) just discarded. Disk-backed and dirty, written back. Memory can be paged to disk. Eventually can OOM. Memory is then loaded, or zeroed, and PTE updated. Can it be shared? (zero page)

- "invalid" – segfault



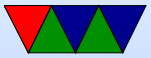
Uses of VM in an operating system

- 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

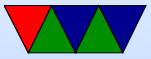


What happens on a fork?

- 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)

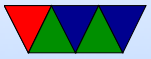


Virtual Memory Wrapup

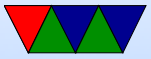


Large Pages

- Another way to avoid problems with 64-bit address space
- Larger page size (64kB? 1MB? 2MB? 2GB?)
- Less granularity. Potentially waste space
- Fewer TLB entries needed to map large data structures
- Compromise: multiple page sizes.
  Complicate O/S and hardware. OS have to find free blocks of contiguous memory when allocating large page.

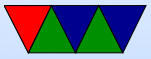


- Transparent usage? Transparent Huge Pages? Alternative to making people using special interfaces to allocate.

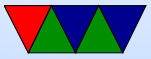


Having Larger Physical than Virtual Address Space

- 32-bit processors cannot address more than 4GB
  x86 hit this problem a while ago, ARM just now
- Real solution is to move to 64-bit
- As a hack, can include extra bits in page tables, address more memory (though still limited to 4GB per-process)
- Linus Torvalds hates this.

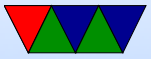


- Hit an upper limit around 16-32GB because entire low 4GB of kernel addressable memory fills with page tables

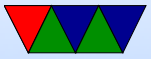


ARMv7 Virtual Memory

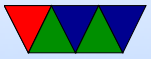
- Our OS we set up a 1:1 Virtual to Physical "Section" Mapping with 1MB pages
- Set up a pagetable, 4k table that is 14-bit aligned
- ARMv7 supports two pagetables, one for kernel-type thing that's fixed and always there, one for process that you can swap in/out
- Pagetable has lots of fields
  - Address (31-20)
  - NS - not secure



  - nG - not global
  - S - shared
  - AP[2:0] access permissions (kernel r/w, kernel r/o, anyone r/w, anyrone r/o, no access)
  - TEX - caching (cachable, no cache)
  - Domain - up to 16 domains with different checking permissions
- Setup pagetable, point to it
- Setup domains (want 0x55555555 not 0xffffffff or won't check)
- Flush TLB/Caches?

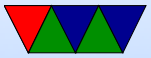


- Enable MMU



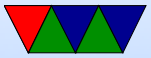
ARMv7 Caches

- Caches are small, fast memories that mirror parts of DRAM for speed
- Important for performance, really hard to set up on ARMv7



Devices

- Character devices – read a sequence of characters incoming, like a serial connection
  open/read/write
- Block devices – read chunks of data with random access, can seek back and forth
  open/read/write/lseek
- /dev, mknod



Block Devices

- Disks?
- Ramdisk?