ECE 531 – Advanced Operating Systems
Lecture 14

Vince Weaver

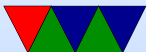
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

19 October 2023

Announcements

- HW#6 will be posted soon hopefully
- Project was posted. Topics due 30 October
  the PDF handout has past project examples
- Midterms not graded yet.

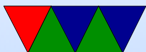


HW#5 Note – Pointers

- All data structures live in memory somewhere
  You can take the address of them with the & operator
- You can have a pointer to an object, those are declared with *
  These are 32-bit or 64-bit depending on architecture
- For the get_time() example, you want to pass in a pointer to an integer (really, we should be explicit with int32_t)

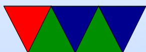
```
int ticks;
get_time(&ticks);
```

- Don't declare int *ticks; then pass in &ticks, that's a



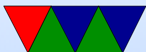
pointer to a pointer and C only isn't complaining because we're having to cast things into a long because we're going into inline assembly for the syscall and assembly language has no concept of types/pointers

- Since we have no memory protection enabled, a lot of bad memory/pointer accesses might work by accident



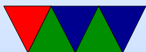
HW#5 Note – Ignoring Syscall Argument

- Traditionally if a syscall takes a pointer as input but it's optional then you pass in NULL instead
- Your syscall handler code should check for NULL before writing out the value
- Have your code actually use NULL, don't assume NULL is 0 because it doesn't necessarily have to be. Since we are casting we might not get errors



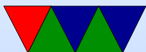
HW#6 Notes

- **NOTE FOR NEXT TIME** a lot of this is broken out better in the following two lectures
- Sorry for the delay getting it out. Various issues including trying to get Pi4 support
- There's a lot going on just trying to get processes running
- 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



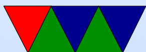
- Took me weeks the first time, thought I could simplify it down. But no, still a huge mass of assembly and had trouble sorting it out (lack of comments and code 2 years old!)
- Each process has a structure that holds info on it, plus the save state.

- 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.

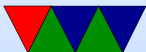
- Filesystem
  - But, needed a place for them to live. So a simple ramdisk and romfs (we'll talk about filesystems soon)
- Fork/Exec
  - How to executables start? Unix we mentioned fork/exec. However a true fork requires virtual memory and we don't have that yet.
  - So there's a stripped-down version 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.

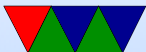
- Scheduler/Idle Thread
  - How does the scheduler work? Simple, nothing fancy.



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.

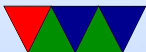
  - What if none available? Then run the idle thread.

- 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



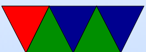
the kernel and that's how we implement it, so we need to save our kernel register state as well as the user space. There's probably better ways to do this.

- 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.



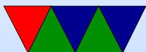
Otherwise handle them so they can die.

◦ So in the shell after every command is typed it does a waitpid(NOHANG) to see if any of the background tasks finished.

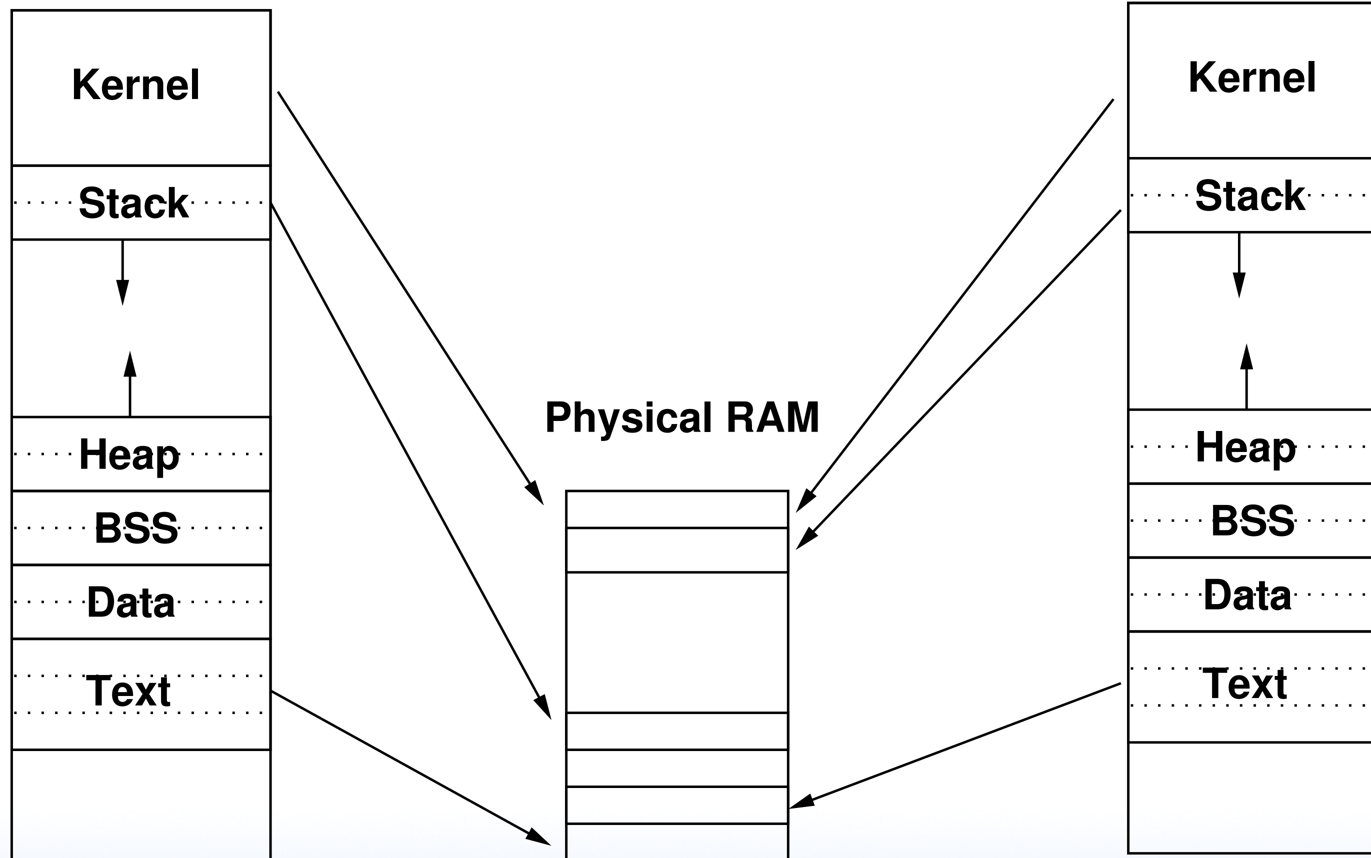


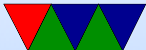
Virtual Memory Redux

- 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



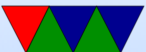
Diagram





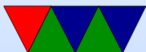
Memory Management Unit

- In very old days was a separate (optional!) chip
- Can run run OS without an MMU?
  - There's MMU-less Linux (uclinux)
  - How do you keep processes separate? Very carefully...



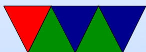
Page Lookup

- Simplest would just be a table, with virtual page as index and physical page as value
- Ends up being more complex than this



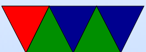
Page Tables – Hold Virt/Phys Mappings

- 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)



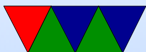
Page Table Encoding

- If 4k pages, bottom 12 bits of mappings unused
- Could you squeeze all the PTE info in those bits?
- Can be complex, ARM32 and ARM64 have really complicated page table setups

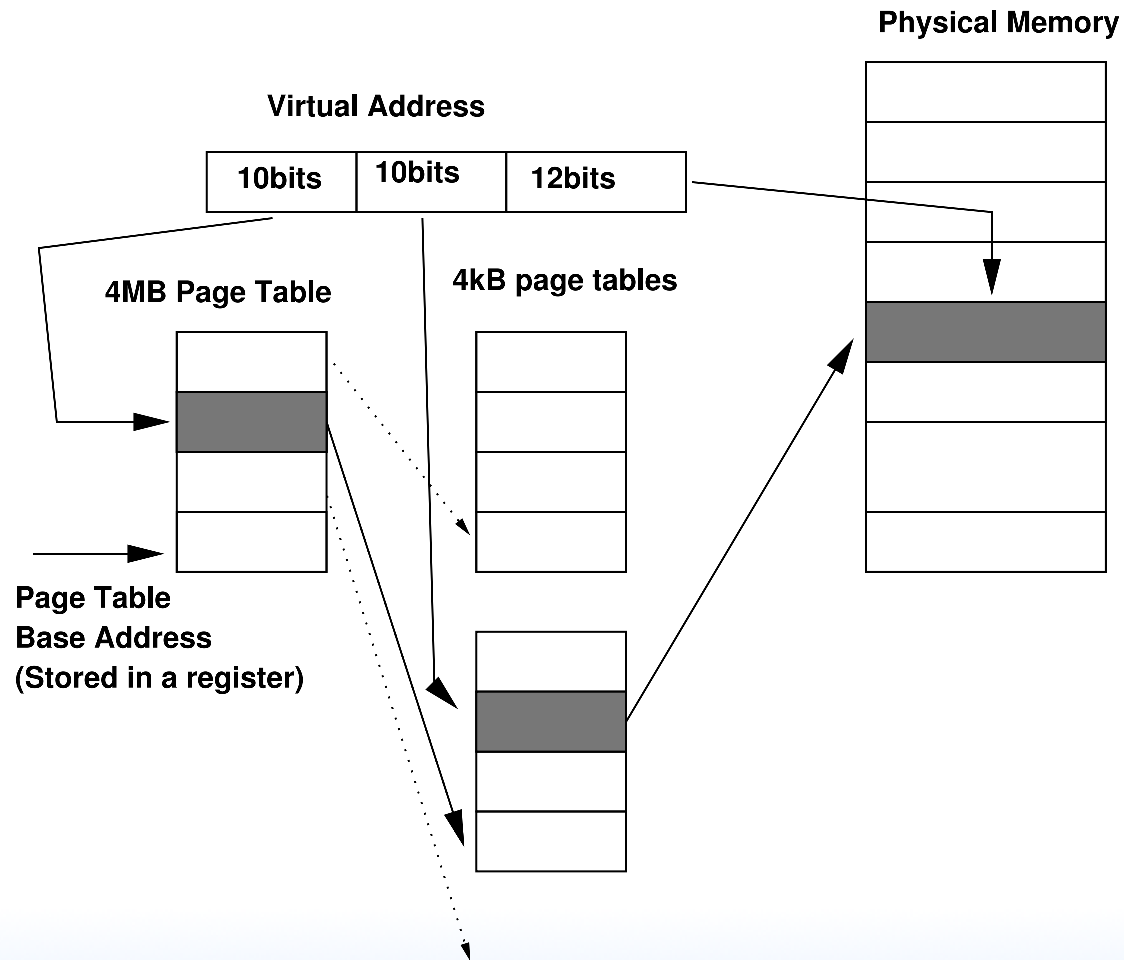


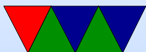
Hierarchical Page Tables

- It is likely each process does not use all 4GB at once (sparse)
- Put page tables in swappable virtual memory
- 4MB page table is 1024 entries which can be mapped by one 4kB page



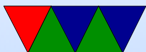
Hierarchical Page Table Diagram





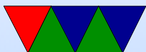
Hierarchical Page Table Diagram

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



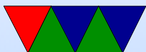
64-bit Systems

- Virtual address space much bigger, how to handle?
- Physical memory usually not 64-bit yet, often from 40-48 bits (recent push for 56?)
- Can we just add more levels of page tables?
  - 64 bit x86 has 4-level page tables (256TBv/64TBp) 44/40 bits
  - Push by Intel for 5-level tables (128PBv/4PBp) 57 bits



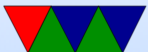
Another approach (Historical) Inverted Page Table

- IBM Power, Ultrasparc, ia64
- 4/5 level tables can be slow
- Have one single mapping, page mapping for each physical to virtual page
- Almost like having a large software TLB
- Note: Linus Torvalds wasn't a fan
- A linear search to find a mapping is slow, so can use hash to find page. Better best case performance, can



perform poorly if hash algorithm has lots of aliasing.

- Also has poor cache performance due



Inverted Page Table Diagram

Physical Memory

Page Tables

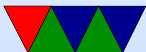
Virtual
Address

HASH

hit

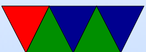
alias

re–hash



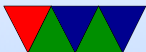
Walking the Page Table

- Can be walked in Hardware or Software
- Hardware is more common
  - Generally have a register pointing to the current process page table (CR3 ox x86)? CR4?
- Early RISC machines would do it in Software.
  - Can be slow
  - Has complications: what if the page-walking code was swapped out?



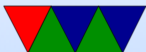
TLB

- 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

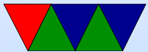


Flushing the TLB

- May need to do this on context switch if doesn't store ASID or ASIDs run out (ASID=Address Space ID, intel only added support recently)
- 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 (security issue with Meltdown bug!)

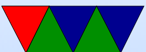


When do we flush TLB?



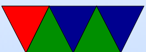
What happens on a memory access

- This gets complicated for caches, we'll ignore that for now
- If in TLB, not a problem, right page fetched from physical memory, TLB updated
- If not in TLB, then the page tables are walked
- If no physical mapping in page table, then page fault happens

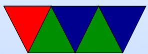


What happens on a page fault

- 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 should be zeroed pages
  - heap/stack might be auto-allocated zero pages
  - You can view these under /proc on Linux
- 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

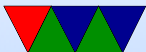


entry



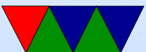
page fault types

- "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



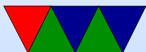
and dirty, written back.

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



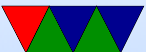
Uses of VM in an operating system

- 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



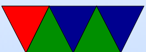
Each process has a page table

- 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?

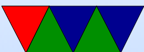


What happens on a fork?

- Do you actually copy all of memory?
  Why would that be bad? (slow, also often exec() right away)

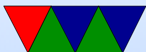
- Page table marked read-only, then shared

- Only if writes happen, take page fault, then copy made
  Copy-on-write (COW)



What happens 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