ECE 531 – Advanced Operating Systems
Lecture 13

Vince Weaver

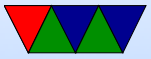
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

1 October 2025

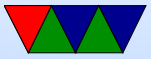
Announcements

- Homework #4 was posted. Due Monday (since I got it posted late)
  - I updated the assignment slightly on Tuesday to clarify and add some helper code for the command parsing part of things.



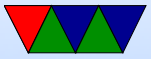
HW#3 Review – Code

- Be sure you comment your code!
- Serial port: most got value right
  Remember you can't use floating point in kernel
- printk for hex
  - Decimal algorithm is /10, digit remainder plus '0'
  - Ex can just /16 (which is a shift by 4) and two case: 0-9 same as before, but A-F you have to handle separately
  - Should digits be uppercase?



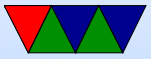
technically upper vs lowercase %X vs %x

  - Be careful shifting, what if print 0xffffffff? What if you shift right but it's signed? Also be careful shifting right by 32, might be undefined behavior in C compiler
- Be sure print hardware info (r1)



HW#3 Review – Questions

- Why serial port?
  Simplicity in code involved
  Note while in theory the hardware is simple too (just 3 wires) in practice just plugging in USB or HDMI is a lot easier than finding the right serial cable or hooking up USB-serial
- What is parity? Why is it disabled?
  Faster (one fewer bit per byte), much bigger infrastructure to handle, not even that great (only detect



one bit flip). Not that critical for text. What would a file transfer do? (checksum?)

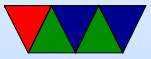
Some systems might not support? True, but why don't they support it?

- inline asm lets you write code that's not possible in C.

  Also lets you bypass compiler (if you think you can do better)

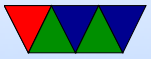
  Don't confuse it with the volatile keyword.
- Why no strtok()?

  strtok() given a string and set of delimiters (like space, tab) split up a line. So "led on" you'd get led, then run



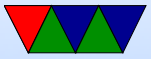
it again and get “on”

- People seem to mostly have usb-serial going OK



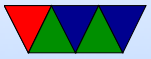
Blocking vs Non-blocking Syscall

- Blocking system calls – program stops, waits for reply before it can continue
- Nonblocking – system call returns right away, although the result might just be "no data available" and you have to check (callback or data structure) to find return value later

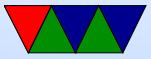


Implementing Blocking System Calls

- What happens if I/O can take a large amount of time?
- Should we just block whole kernel until hardware ready (maybe for seconds?)
- Ideally you can launch I/O and have HW send interrupt when done
- What happens to process in this case?
- Usually set up data structure (queue) and put process to sleep, and then OS moves to other things
- When I/O interrupt comes in, OS can wake up any

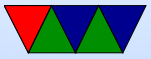


processes sleeping waiting for the data



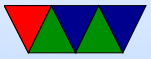
Syscalls are Slow!

- Doing a user to kernel transition is slow
- Exceptions are slow on modern CPUs
- Linux is highly optimized but still slow
- Security (Meltdown) mitigations might slow things further (need to flush TLB?)
- Are there alternatives?

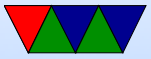


Linux vsyscalls

- Some common Linux syscalls don't really need any action from the kernel, but just return a static or easily calculated value (`getpid()`, `get_cpu()`, `gettimeofday()`
- Could we map some kernel memory into userspace to let the user access it without a syscall?
- vsyscalls do this. At fixed address, you can jump there and some code is in place to get the data without entering kernel

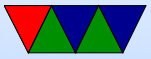


- stub call to get values in 0xffffffffff600400
- Security issue: as with ASLR, code in fixed place could be used by attacker
- Security issue: known location of syscall instruction
- Deprecated in Linux 3.1, use VDSO instead



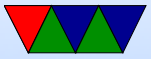
Vsyscall Example

```
int vsyscall_getcpu(int *cpu, int *node) {

        register int64_t rax __asm__("rax") = 0xffffffffff600800;
        register void *rdi __asm__("rdi") = cpu;
        register void *rsi __asm__("rsi") = node;

        __asm__ volatile (
                "callq␣*%%rax␣\n\t"
                : "+r" (rax)
                : "r" (rdi), "r" (rsi)
        );

        return rax;
}
```

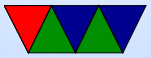


Linux Virtual Dynamic Shared Object (VDSO)

- New way of doing things is VDSO
- Similar to vsyscall, but appears as virtual shared library that can be moved around in memory
- Can run "ldd /bin/ls" and you'll see the vdso mapped on modern Linux executables
- Same functions as well as gettime()
- Also some other things: has stub for calling syscalls so on 32-bit x86 could use it to automatically call int 0x80

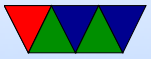


vs SYSCALL when avail



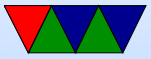
Sample VDSO Code

```
void *vdso;

vdso = dlopen("linux-vdso.so.1", RTLD_LAZY|RTLD_LOCAL|RTLD_NOLOAD);
if (!vdso) {
   vdso = dlopen("linux-gate.so.1", RTLD_LAZY|RTLD_LOCAL|RTLD_NOLOAD);
   vdso_gtod = (gtod_t)dlsym(vdso, "__vdso_gettimeofday");
}

vdso_gtod(&ours,NULL);
printf("gettimeofday: %ld %ld\n",ours.tv_sec,ours.tv_usec);
```



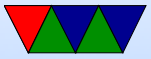
Asynchronous I/O

- What if want to launch a bunch of system calls and you're not particular in what order they run
- It's possible to do this with the standard interface but complicated
- Wouldn't it be great if you could also launch a bunch all at once without a slow syscall for each one?

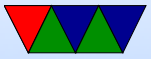


Linux io_uring

- input/output user ring-buffer
- This is recent, Linux 5.1 (2019)
- Most useful for asynchronous I/O
- Can set up two circular queues
  - Submission Queue (SQ)
  - Completion Queue (CQ)
- Use syscalls to set this up, with head and tail pointers
- Add info for a syscall-like request to submission queue, update tail pointer

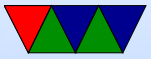


- Kernel checks and sees there's a request and handles it
- When kernel is done it updates head/tail pointers and puts results in completion queue
- This allows kernel communication without constant syscalls



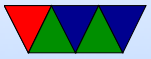
Linux io_uring Setup

- Use `mmap()` to allocate SQ/CQ buffers
- Use `io_uring_setup()` also
- There's a list of supported syscalls (see manpage). Put something like `IORING_OP_WRITE` in SQ
- Syscall to get things running



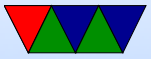
Linux io_uring Completion

- Monitor CQ for results
- Can finish out of order
- If you require things to be ordered you can set that up
- Can also configure free-running mode where kernel thread constantly watches SQ for more syscalls to end w/o syscall overhead



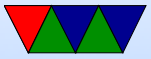
Linux io_uring Sample Code

TODO



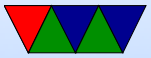
Linux io_uring security issues

- Under current development, some security issues
- Google said 60% of their reported security bugs from io_uring (2023)
- For a while they turned off io_uring on all products
- Maybe it has gotten better?



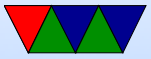
Userspace Executables

- Now that we can run code in userspace, what does that look like?

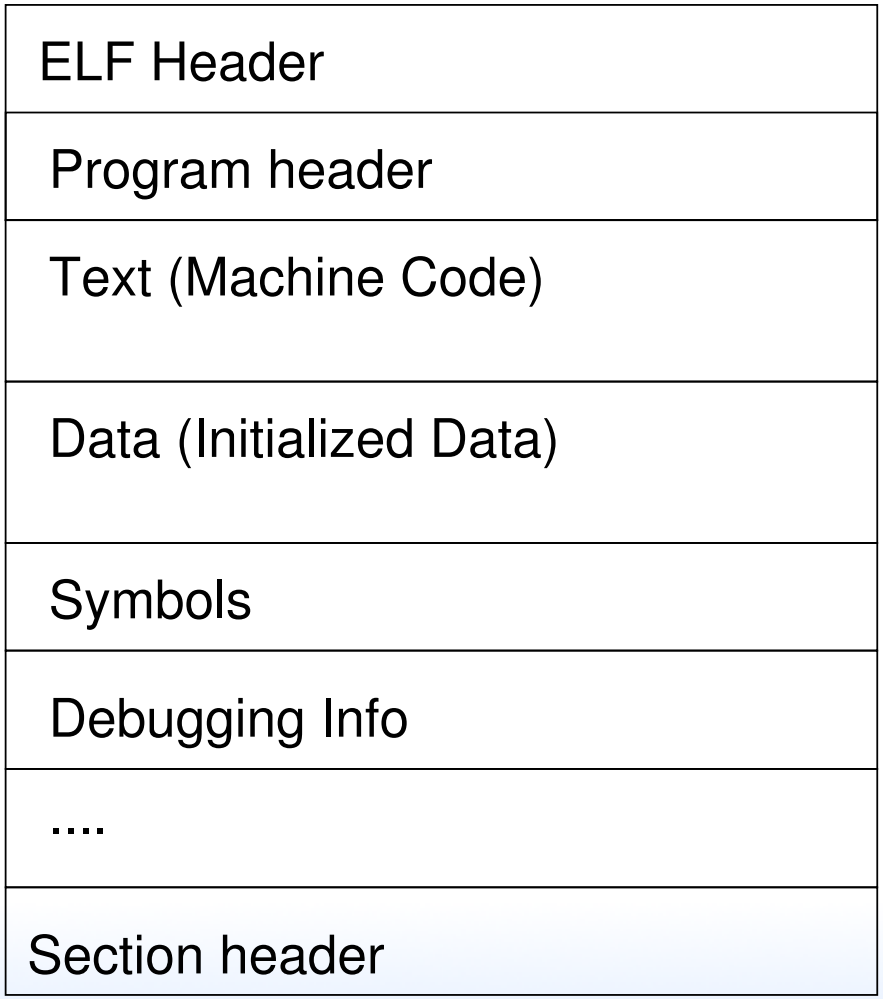


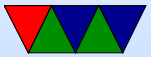
Executable Format

- ELF (Executable and Linkable Format, Extensible Linking Format)
  Default for Linux and some other similar OSes
  header, then header table describing chunks and where they go
- Other executable formats: a.out, COFF, binary blob
- Can install "elfutils" and use something like "readelf -a /bin/ls" to get info on what's inside



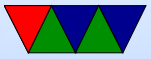
ELF Layout


ELF Header
Program header
Text (Machine Code)
Data (Initialized Data)
Symbols
Debugging Info
....
Section header



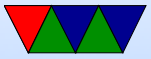
ELF Description

- ELF Header includes a "magic number" saying it's 0x7f,ELF, architecture type, OS type, etc. Also location of program header and section header and entry point.
- Program Header, used for execution:
  has info telling the OS what parts to load, how, and where (address, permission, size, alignment)
- Program Data follows, describes data actually loaded into memory: machine code, initialized data

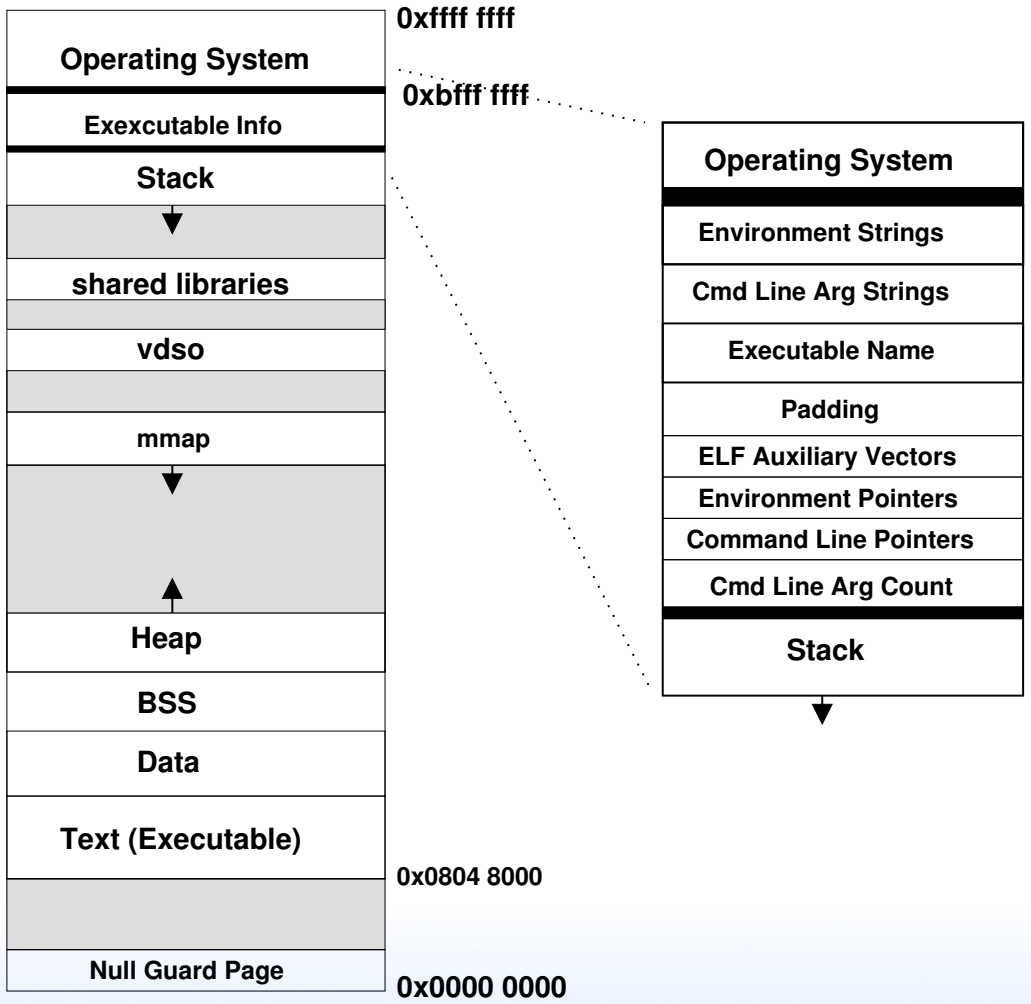


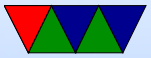
- Other data: things like symbol names, debugging info (DWARF), etc.
  DWARF backronym = "Debugging with Attributed Record Formats"

- Section Header, used when linking:
  has info on the additional segments in code that aren't loaded into memory, such as debugging, symbols, etc.



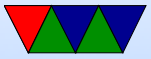
Linux Virtual Memory Map





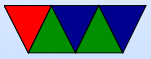
Program Memory Layout on Linux

- Text: the program's raw machine code
- Data: Initialized data
- BSS: uninitialized data; on Linux this is all set to 0.
- Heap: dynamic memory. `malloc()` and `brk()`. Grows up
- Stack: LIFO memory structure. Grows down.



Program Layout

- Kernel: is mapped into top of address space, for performance reasons
  DANGER: MELTDOWN
- Command Line arguments, Environment, AUX vectors, etc., available above stack



Address Space Layout Randomization (ASLR)

- For security ASLR is enabled by default (you can disable)
- Each run of a program the location of text / data / bss / heap / stack might be moved around
- This in theory makes it harder for attackers to find functions/data they want to use
- Makes performance analysis hard as execution ends up being less deterministic (yes, some code behaves differently depending on memory addresses)