ECE 531/598 – Advanced Operating Systems
Lecture 10

Vince Weaver

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

vincent.weaver@maine.edu

28 September 2023

Announcements

- Homework #4 Deadline extended
- Homework #5 will be posted
- Raspberry Pi 5 was released

More notes for HW#4

Accessing MMIO registers

- For the BCM2835 peripherals, use `bcm2835_read()` and `bcm2835_write()` to do MMIO access
  These adjust for the differing location of the MMIO regions on different Pis
- Don't use the `mmio_read()/write()` routines unless you have to (I renamed them to make this hard to do accidentally). The only code that uses them is the new gic-400 irq controller code on Pi4 because it lives in a different mmio region separate from the peripheral IO

Blocking vs Nonblocking 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"
- What if a blocking system call tried to block inside the kernel with interrupts disabled? Real OS uses queues and wakeups to put processes to sleep when blocking, not just busy spinning.

Syscalls are Slow!

- Doing a user-¿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/VDSO

- 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 could jump there to get the data without entering kernel
- Security issue: as with ASLR, code in fixed place could

be used by attacker

- Solution was VDSO which does something similar but the location can be mapped to different locations
- Can run “ldd /bin/ls” and you’ll see the vdso mapped on modern Linux executables

Linux io_uring

- This one is more recent, Linux 5.1 (2019)
- Most useful for asynchronous I/O
- Can set up two circular queues, submission an completion
- 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
- Under current development, some security issues recently (2023)

Userspace Executables

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)

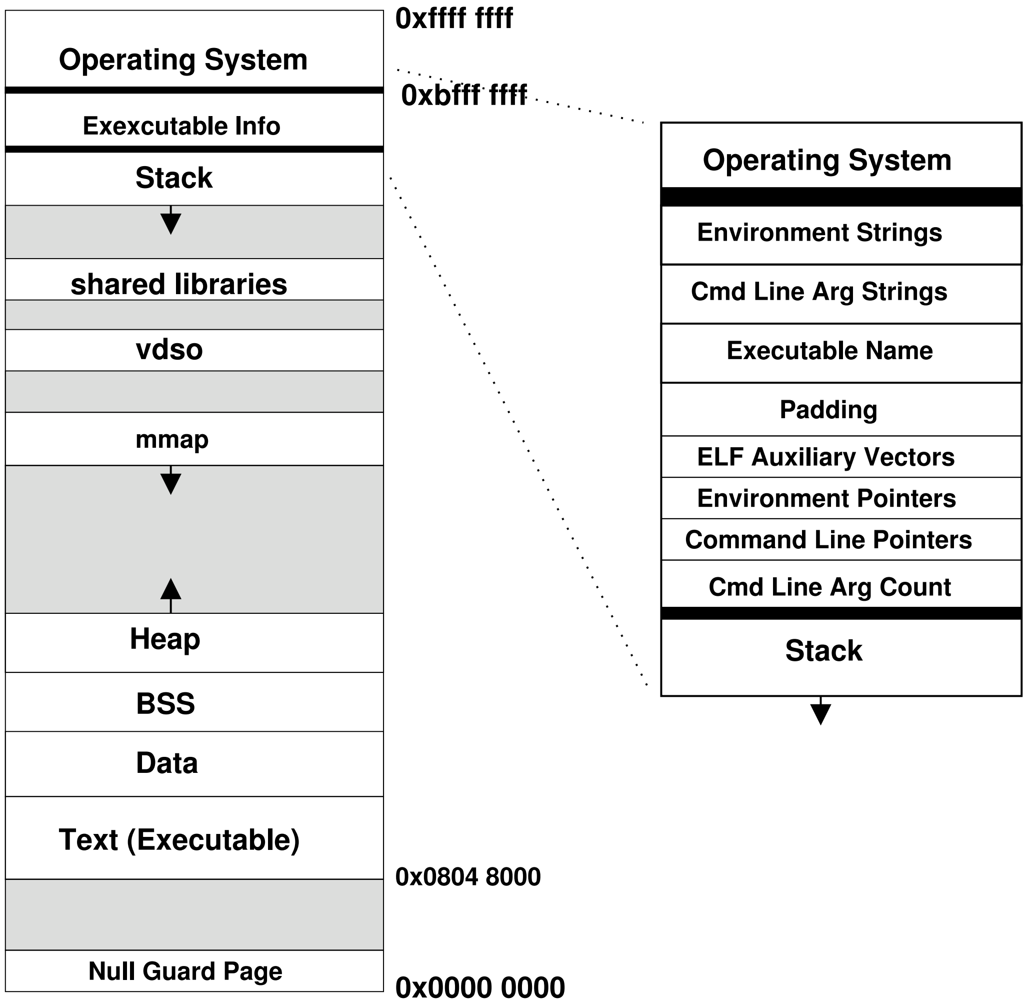
- Program Data follows, describes data actually loaded into memory: machine code, initialized data

- 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

- The view a Linux program has of memory, note it doesn't match Physical memory via CPU/OS magic
- We will go over virtual memory in much greater detail in a future lecture



Operating System
Exexcutable Info
Stack
shared libraries
vdso
mmap
Heap
BSS
Data
Text (Executable)
Null Guard Page
0xffff ffff
0xbfff ffff
0x0804 8000
0x0000 0000
Operating System
Environment Strings
Cmd Line Arg Strings
Executable Name
Padding
ELF Auxiliary Vectors
Environment Pointers
Command Line Pointers
Cmd Line Arg Count
Stack

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 reasons 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)

Loader

- /lib/ld-linux.so.2
- loads the executable

Static vs Dynamic Libraries

- Static: includes all code in one binary.
  Large binaries, need to recompile to update library code, self-contained
- Dynamic: library routines linked at load time.
  Smaller binaries, share code across system, automatically links against newer/bugfixes
- Lots of debate about what is better: apt-get install vs the app-store (flatpack, etc)

How Dynamic Linking Works

- Can read about how things load on Linux here:
  `https://lwn.net/Articles/630727/`
  `https://lwn.net/Articles/631631/`
- ELF executable can have interp section, which says to load /lib/ld-linker.so first
- This loads things up, then initialized dynamic libraries.
- Links things in place, updates function pointers and shared variables, offset tables, etc.
- Lazy-Linking is possible. Function calls just call to a

stub that calls into linker. Only resolves the link if you actually use it. Why is this a benefit (faster startup, not load things not need). Does add indirection every time you call.

- Can use "ldd /bin/ls" to see what dynamic libraries a program is using

How a Program is Loaded on Linux

- Kernel Boots
- `init` started
- `init` calls `fork()`
- child calls `exec()`
- Kernel checks if valid ELF. Passes to loader
  Possibly not ELF. Shell scripts, etc.

- Loader loads it. Clears out BSS. Sets up stack. Jumps to entry address (specified by executable)
- Program runs until complete.
- Parent process returned to if waiting. Otherwise, init.