ECE 598 – Advanced Operating Systems
Lecture 10

Vince Weaver

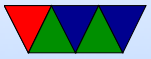
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

22 February 2018

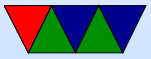
Announcements

- Homework #5 will be posted

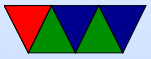


Blocking vs Nonblocking Syscall

- 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.



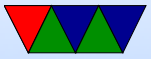
Userspace Executables



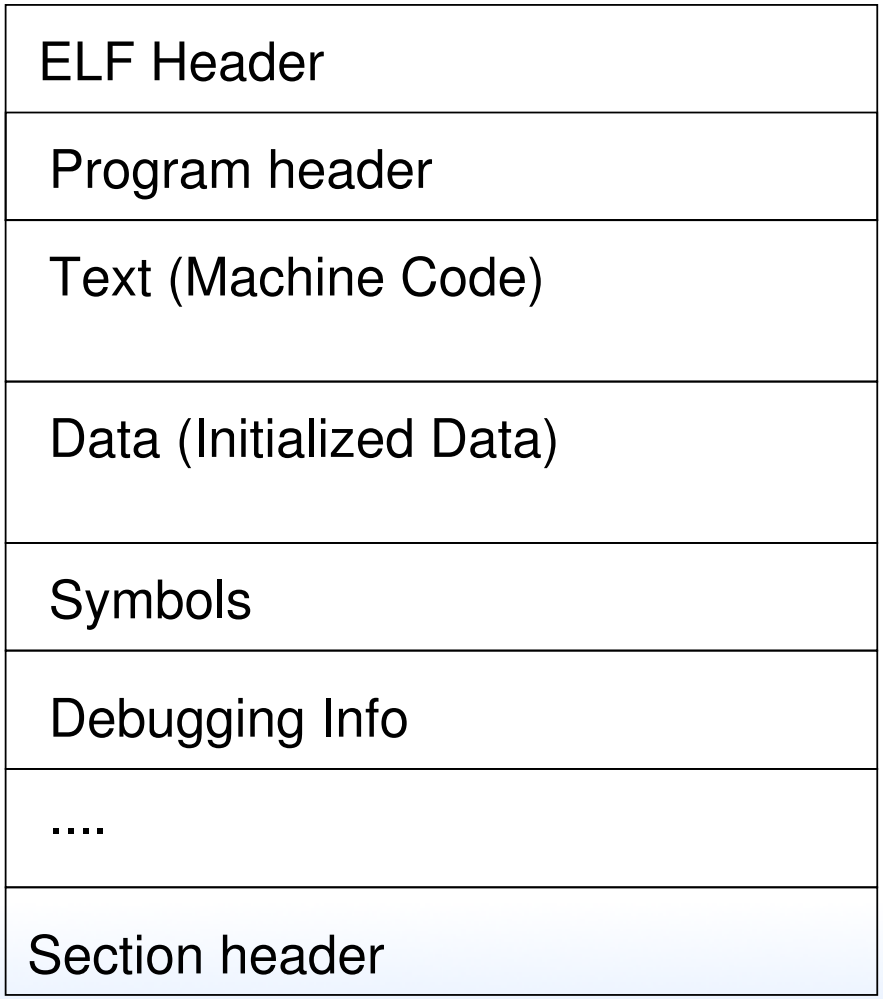
Executable Format

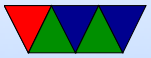
- ELF (Executable and Linkable Format, Extensible Linking Format)
  Default for Linux and some other similar OSes
  header, then header table describing chunks and where they go

- Other executable formats: a.out, COFF, binary blob



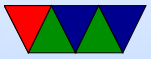
ELF Layout


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



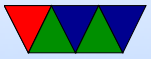
ELF Description

- 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



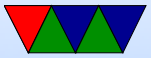
- Other data: things like symbol names, debugging info (DWARF), etc.
  DWARF backronym = "Debugging with Attributed Record Formats"

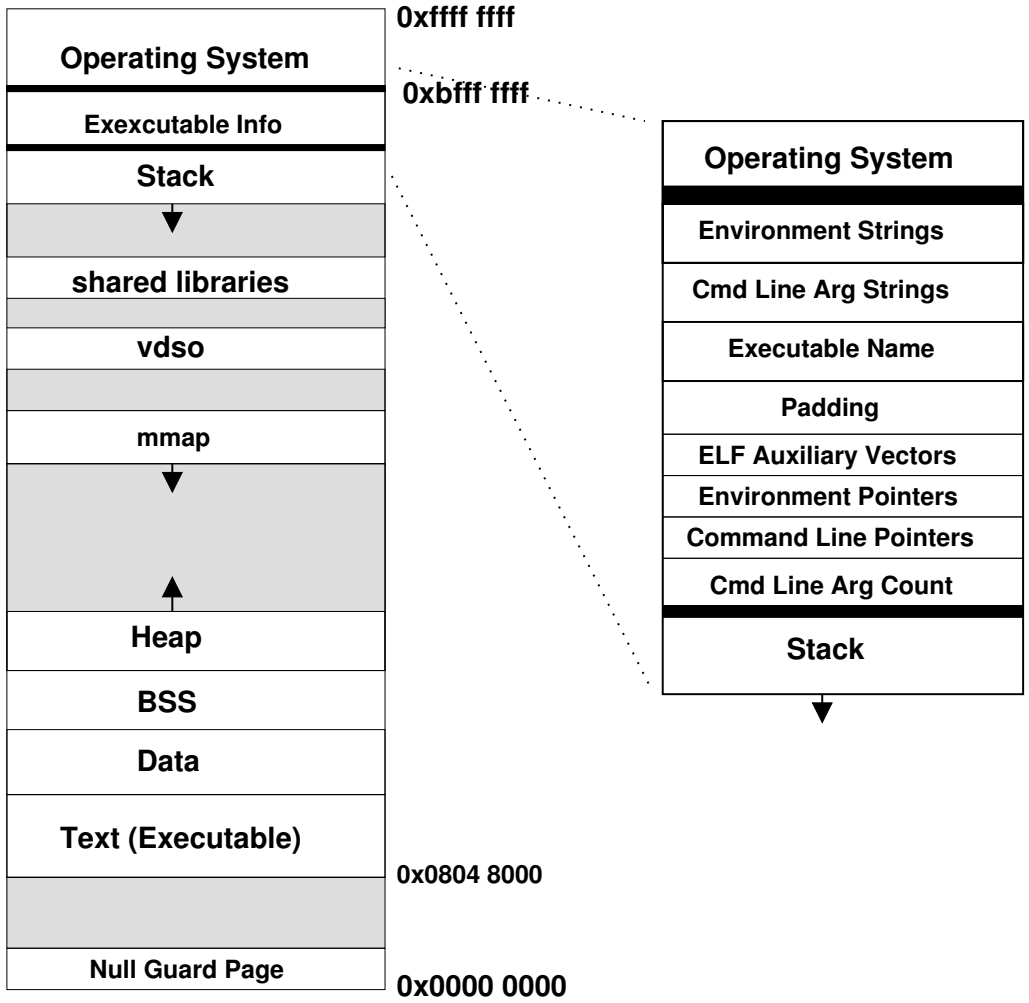
- Section Header, used when linking:
  has info on the additional segments in code that aren't loaded into memory, such as debugging, symbols, etc.

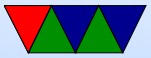


Linux Virtual Memory Map

We will go over virtual memory in much greater detail later.

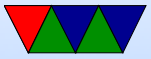



0xffff ffff
Operating System
0xbfff ffff
Exexcutable Info
Stack
shared libraries
vdso
mmap
Heap
BSS
Data
Text (Executable)
0x0804 8000
Null Guard Page
0x0000 0000
Operating System
Environment Strings
Cmd Line Arg Strings
Executable Name
Padding
ELF Auxiliary Vectors
Environment Pointers
Command Line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



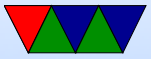
Program Memory Layout on Linux

- 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.



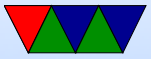
Program Layout

- Kernel: is mapped into top of address space, for performance reasons
  DANGER: MELTDOWN
- Command Line arguments, Environment, AUX vectors, etc., available above stack
- For security reasons "ASLR" (Address Space Layout Randomization) is often enabled. From run to run the exact addresses of all the sections is randomized, to make it harder for hackers to compromise your system.



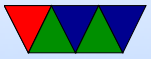
Loader

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



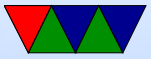
Static vs Dynamic Libraries

- 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

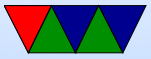


How Dynamic Linking Works

- 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

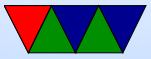


stub that calls into linker. Only resolves the link if you actually use it. Why is this a benefit (faster startup, not load things not need). Does add indirection every time you call.

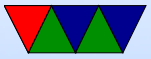


How a Program is Loaded on Linux

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



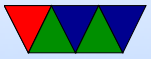
- Loader loads it. Clears out BSS. Sets up stack. Jumps to entry address (specified by executable)
- Program runs until complete.
- Parent process returned to if waiting. Otherwise, init.



UCLinux

Linux typically relies on MMU (virtual memory). You can run it on systems w/o virtual memory, this version is called ucLinux (micro-controller Linux).

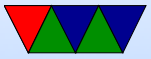
Our OS in the homework is similar in design to this.



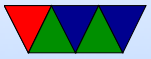
Flat File Format

- http://retired.beyondlogic.org/uClinux/bflt.htm
- bFLT or 0x62, 0x46, 0x4C, 0x54
- ```
struct flat_hdr {
    char magic[4];
    unsigned long rev;   /* version */
    unsigned long entry;   /* Offset of first executable instruction
                                   with text segment from beginning of file */
    unsigned long data_start;   /* Offset of data segment from beginning of
                                   file */
    unsigned long data_end;   /* Offset of end of data segment
                                   from beginning of file */
    unsigned long bss_end;   /* Offset of end of bss segment from beginning
                                   of file */

    /* (It is assumed that data_end through bss_end forms the bss segment.) */
```

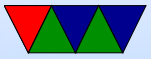

```

```
    unsigned long stack_size;   /* Size of stack, in bytes */
    unsigned long reloc_start;  /* Offset of relocation records from
                                   beginning of file */
    unsigned long reloc_count;  /* Number of relocation records */
    unsigned long flags;
    unsigned long filler[6];    /* Reserved, set to zero */
};
```

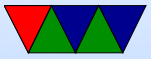


Figuring out how it actually works

- Spec isn't worth much
  Your best bet is various Wikis and blog postings (TI-nspire?)
- Actual code more useful
- `fs/binfmt_flat.c` in kernel source.
- Making the binaries hard. Not just a simple matter of telling gcc or linker (no one has bothered yet). Most



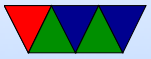
people use “elf2flt” but not-standard and hard to even find which code repository to use.



Loading a flat binary

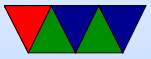
- load_flat_binary()

- adjust stack space for arguments (argv and envp)
  - loading header. Uses `ntohl()`. Why?
    Endian issues.
  - check for bFLT magic
  - check version
  - check rlimits() [stack, etc]
  - setup_new_exec()

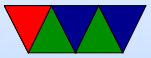


- allocate mem for our binary (separately handle XIP and compressed format)
- read_code()
- put all of our values in mm struct (Start/stop of all sections)
- RELOCATION – fix up any symbols that changed due to being moved. (HOW DOES THIS WORK)
- flush_icache()
- zero the BSS and STACK areas

• setup shared libraries

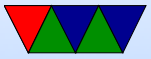


- install_exec_creds()
- set_binfmt()
- actually copy command line args, etc, at front of stack
- put stack pointer in mm structure
- start_thread()



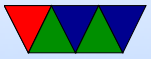
PIC/PIE

- Position independent code
- Instead of loading from absolute address, uses an offset, usually in a register or PC-relative.
- gcc has an option `-fPIC` to generate



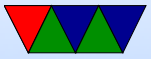
Relocation

- List of offsets to pointers
- PIC compiles things with zero offset
- At load time the pointers are fixed up to have the load address
- Separate relocation for GOT (global offset table) which is a list of pointers at the beginning of the data segment, ending with -1



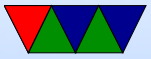
Flat Shared Libraries

- Like mini executables, can have up to 256 of them
- Libraries loaded in place, then the callsites are fixed up to have the right address.
- Also at start time the various library init routines are called



Execute in Place

- Want our text in ROM. Why? Save space, save copying. Why bad? ROM often slow, more complicated binaries (data not follow text)



RAM Disk

- How to load our code?
- Can we load from disk? No driver yet.
- We can create a RAM disk, will be loaded by our bootloader right after. Sometimes called an initrd.