ECE 498 – Linux Assembly Language Lecture 1

Vince Weaver

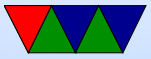
http://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

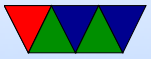
13 November 2012

Assembly Language: What's it good for?

- Understanding at a low-level what your computer is doing. It separates the Computer Engineers from the CS majors.
- Shown when using a debugger
- Compiler writers must translate higher languages to assembly
- Operating system writers (some things not expressible in C)

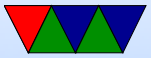


- Embedded systems (code density)
- Research. Computer Architecture. Emulators/Simulators.
- Video games (or other perf critical routines, glibc, kernel, etc.)



Benefits of Using Linux

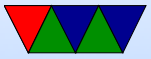
- Tools! In-place debugging tools
- Flat 32-bit address space
- I/O, timing, device drivers, all handled for you
- The O/S acts as a cross-platform abstraction layer



Other OSes

If the architecture is the same, the assembly language is the same.
All that changes is how you call the O/S.

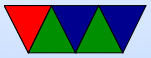
- DOS
- Windows
- BSD/OSX/Other UNIX



Tools

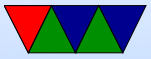
- assembler: GNU Assembler as (others: tasm, nasm, masm, etc.)
  creates object files

- linker: ld
  creates executable files. resolves addresses of symbols. shared libraries.

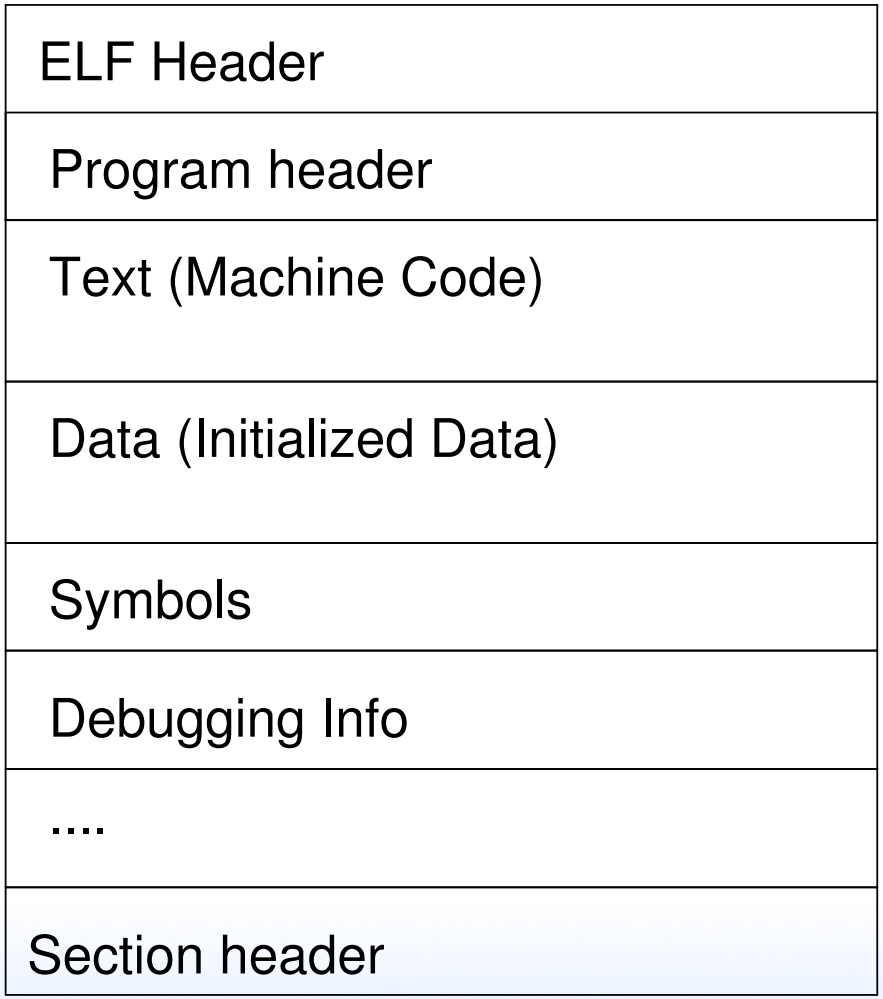


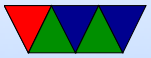
Executable Format

- ELF – Linux
  header, then header table describing chunks and where they go
- Other executable formats: a.out, COFF



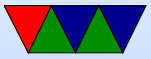
ELF Layout


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



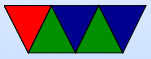
ELF Background

- Executable and Linker Format
- ELF Header includes a "magic number" saying it's ELF, architecture type, OS type etc.
- Program Header has info telling the OS what parts to load and where
- Program Data follows: machine code, initialized data
- Other data: things like symbol names, debugging info



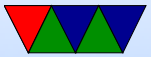
(DWARF), etc.

- Section Header (tells other tools what sections are where)



Loader

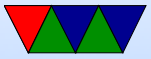
- /lib/ld-linux.so.2
- loads the executable



Static vs Dynamic Libraries

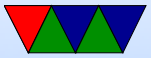
- Static: includes all code in one binary.
  Large binaries, need to recompile to update library code, self-contained

- Dynamic: library routines linked at load time.
  Smaller binaries, share code across system, automatically links against newer/bugfixes



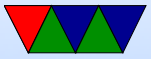
How a Program is Loaded

- Kernel Boots
- `init` started
- `init` calls `fork()`
- child calls `exec()`
- Kernel checks if valid ELF. Passes to loader
- Loader loads it. Clears out BSS. Sets up stack. Jumps

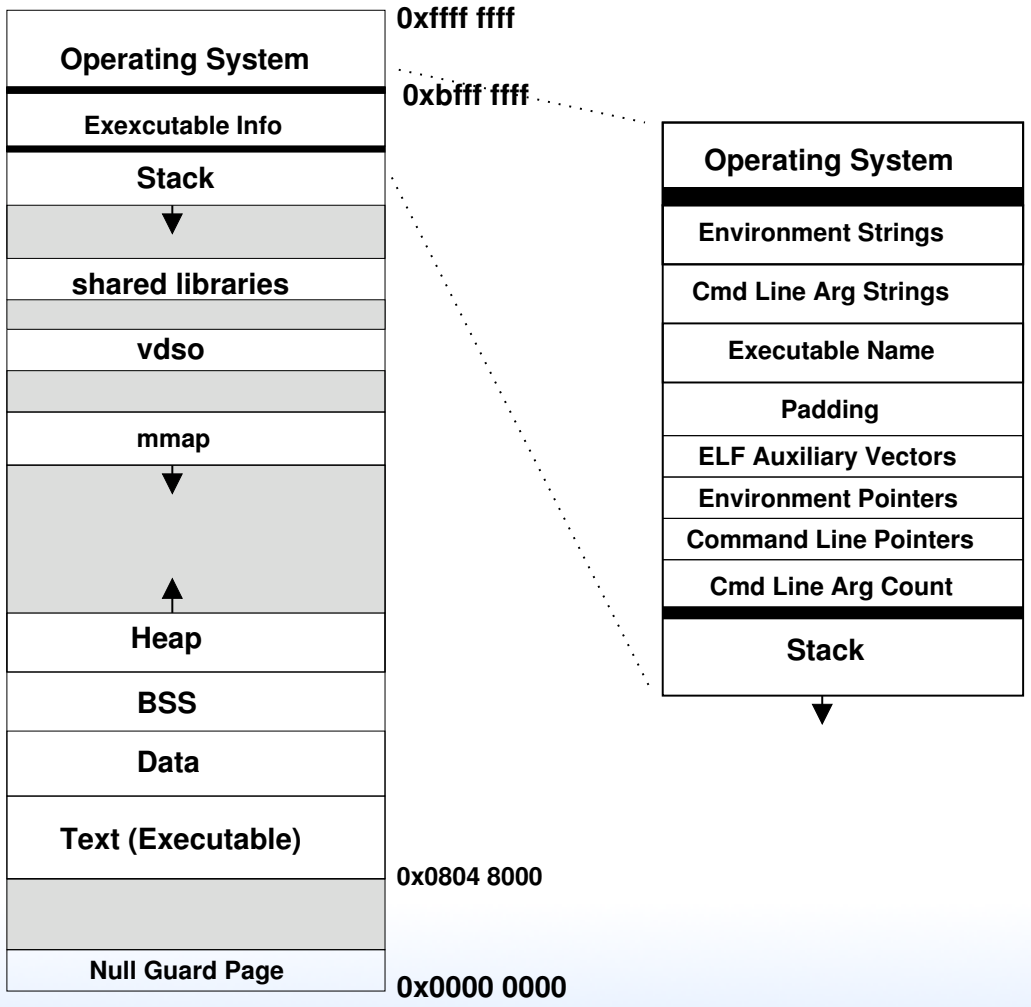


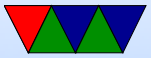
to entry address (specified by executable)

- Program runs until complete.
- Parent process returned to if waiting. Otherwise, init.



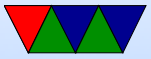
What a program's memory layout looks like





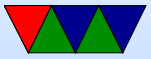
What a program's memory layout looks like

- 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.



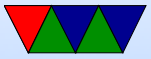
Program Layout

- Kernel: is mapped into top of address space, for performance reasons
- 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.



Brief overview of Virtual Memory

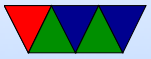
- Each program gets a flat 4GB (on 32-bit) memory space. The CPU and Operating system work together to provide this, even if not that much RAM is available and even though different processes seem to be using the same addresses.
- Physical vs Virtual Memory
- OS/CPU deal with "pages", usually 4kB chunks of memory.



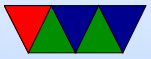
- Every mem access has to be translated. The operating system looks in the "page table" to see which physical address your virtual address maps to.
  This is slow. That's where TLB comes in; it caches pagetable translations. As long as you don't run out of TLB entries this goes fast.

- Demand paging: the OS doesn't have to load pages into memory until the first time you actually load/store them.

- Context Switch: when you switch to a new program,

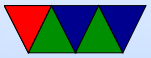


the TLB is flushed and a different page table is used to provide the new program its own view of memory.



What you have at entry

- Registers
- Instruction pointer at beginning
- Stack
- command line arguments, aux, environment variables
- Large contiguous VM space

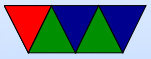


RISC / CISC / VLIW

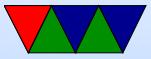
- RISC: Reduced Instruction Set Computer
  Small set of instructions to make processor design simpler. Usually fixed-length instructions, load/store

- CISC: Complex Instruction Set Computer
  Wide ranging complicated instructions; have complicated CPU decode circuitry. Often variable length instructions. Often allow operating on memory directly.

- VLIW: Very Long Instruction Word



Instructions come in long “bundles”, often 3 at a time. Cannot have dependencies; may have to fill with “nops”. Allows compiler to exploit inherit parallelism in code (most modern CPUs do this in hardware instead, VLIW puts this complexity in software).

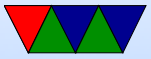


CISC/RISC/VLIW Examples

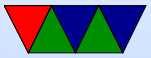
- MIPS is RISC: roughly only 40 integer instructions , (more if you include FP)

- x86 is CISC: hundreds of complicated instructions, including ones that access memory, auto-increment registers, have complex shift/add address modes

- Hybrid: ARM or Power started out RISC but have accumulated more complicated instructions over time



- x86, while CISC externally, internally decodes to a RISC-like code before executing

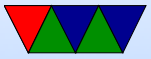


Converting Assembly to Machine Language

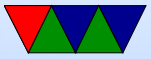
MIPS - 4 bytes

```
oooooooss ssstttttt dddddaaa aafffffff
```

- o6: Instruction opcode
- s5: First source register
- t5: Second source register
- d5: Destination register



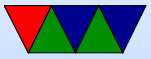
- a5: Shift amount, for shift instructions.
- f6: Function.



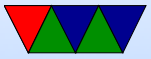
Converting Assembly to Machine Language

x86: 1 - 15 bytes

- LOCK prefix
- REP prefix
- SEGMENT prefix
- 16/32/64 bit (REX) prefix
- Opcode

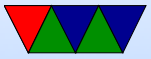


- VEX/XOP byte
- MODRM (register or memory operand)
- SIB scaling factor
- displacement, scaling factor



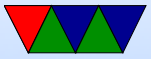
Things to determine when writing assembly

- Can find this in reference books. Manuals online for free Intel 64 and IA-32 Architectures Developer's Manual, huge. Similar from AMD. ARM ARM.
- Instruction names (and what they do)
- Number of registers, Register names (or aliases), orthogonality of registers.
- ABI: Registers uses (which to save, which are parameters,

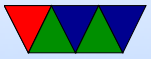


stack pointer, instruction pointer, zero pointer, multiply results, floating point, return address)

- Endianess
- How to make a syscall
- if opcode takes 2 or 3 arguments
- Load/store vs otherwise
- Flags / Condition Codes
- instruction width

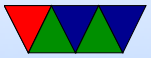


- Addressing Modes

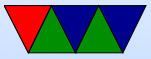


Weirder features to watch for

- Register windows
- branch delay slot
- predication
- unaligned loads
- byte access (alpha)
- HW divide

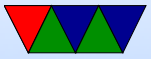


- zero register
- sign-extend (especially x86 vs x86_64)



Types of Instructions

- ALU: add/addu/sub/subu/addi/addiu/mult/multu/div/div
- Memory: lw,lh,lhu,lb,lbu,sw,sh,sb,lui,mfc,mtc,mfhi,mflo
- Logical: and,andi,or,ori,xor,nor,slt,slti
- Shift: sll,srl,sra
- Cond Branch: beq,bne
- Uncond Jump: j, jr, jal



- HW Specific (System Calls, cache flush, TLB, perf counter, etc)