1044414: Advanced Operating Systems and Virtualization **Spring 2018**

Lecture 13: April 20

Lecturer: Alessandro Pellegrini *Scribe: Anxhelo Xhebraj*

We continue studying the ELF file format. An ELF file can be a relocatable file which is an intermediate representation of the program that cannot be executed until linked.

13.1 User space process management

When the linker is gluing together relocatable files it must know where symbols are declared and used. When some file uses a variable that is defined in another module the compiler cannot produce really the assembly since it doesn't have that symbol.

ELF must have a set of bookmarks that tells for each instruction if there is need to be resolution of the address and how. Symbol table allows to identify position of objects in a given relocatable file. The relocation table is what was called bookmarks: one entry for each instruction that needs to be relocated and points to the specific symbol in the symbol table allowing the linker to know what address to put.

This was the static relocation table. The dynamic relocation table is used at runtime for shared libraries etc.

System V ABI document for more information about shared libraries, operating systems etc

Two different organizations: rel and rela. In the latter there is an additional member: addend. Rela was added later . Rela is related to the way according to which the x86 arc deals with targets of assembly instructions. Tells how to transform `movl $1, i` into right byte code. `c7 <displacement> 1 (immediate)`. `call foo` becomes `e8 <displacement>`. The two cases should be treated differently. In the former what is put in displacement is basically the virtual address while the latter needs into displacement is the difference from the call to the definition of the code. On call the cpu will do `RIP = RIP + disp`. The assembler writes `e8` and leaves some space in the displacement to then be filled by the linker. In the former it will find the virtual address of the variable and write in the space. In the latter it has to compute the difference between the current position and where the function was defined. Since displacement has 4 bytes of size and the firmware will have RIP at byte + 4bytes the linker must take into account the 4 bytes of offset. Addend carries the number of bytes to be added.

On legacy systems the linker instead had to look in the bookmark left by the assembler to see how many bytes it had to add.

Traditional moves had in the displacement just the virtual address. There is also an addressing mode in 64 bit systems that does mov through the difference from rip like `movl $1, i(%rip)`. (displacement always 4 bytes therefore 32 bit.)

Segments are fewer than sections. Normally only the three segment of pg 131 that must be put in the program header. The ALLOCATE tells that the section will occupy memory when loaded and must be put in a segment therefore.

To reduce the number of segments wrt sections data and bss sections are glued together. BSs does not occupy space on disk, just on ram.

13.2 Program Header

One entry for each segment in the program header. Is similar to the section header. Segment file offset tells where the segment starts on disk. Segment size in file and memory entries since as we said they can be different. Flags tell for which arc the program was compiled. Segment virtual address and segment physical address. The VA is the virtual address in ram where the program segment will be loaded in RAM. Physical address has nothing to do with what the OS has to do with virtual to physical translation. This field takes into account embedded systems to know where to find it in ROM (pa) and load it in RAM (va).

The OS has a linked list of binary programs handlers. The handler will look at the program header to know how to map that program in memory. Segment flags tell whether the segment is read only/executable etc.

When a program was too big to fit in Main Memory on legacy systems. There was a piece of code called overlay manager that on call checked whether some function was into main memory and if it wasn't it loaded it. ask?

The linker reads the elf headers of the relocatable files and puts sections of the same type together.

(pp 134) Green boxes are instructions positions. Offset within a section in the relocatable file. While in the final executable there are the virtual addresses of the positions. The kernel has to honour the mapping that the linker has decided at link time. ask value?

In blue function addresses while in red data address.

How to inform the linker where to put some specific portions of code? Through linker script directives. Location counter is represented by `.`. By setting it to some value then the following deifinitions are put starting from that address. Second line is telling that the section is formed of all sections of the same type in the order passed when compiling.

When compiling `ld -T` specifies the script. Otherwise `gcc -X-linker -T foo.c`.

```
objdump -x example-program
```

Dynamic section tells that there is need of some shared library.

Section header: init, fini (constructors and destructors). Hash is used to navigate the content of the file.

Symbol table: A lot of stuff has to do with crd0 object which is linked to any compiled program to glue the system to the code.

Bss start and end is put by the linker. df column has some flags: o tells that it is an object (variable), f tells that is a function, the next colum tells the section which can be `*UND*` which means undefined. Second column g global, l local, w weak. A weak symbol states that if there is another symbol in the module with the same signature then the latter should be used. Local means that only the c module in which the function is defined can see it (`static`).

default means global, hidden means local, protected means weak. Internal tells that the function cannot be accessed even by function pointers. Crash at runtime.

`__attribute__` can be used to define the visibility and other stuff that do not belong to the standard C.

`#pragma` to talk directly to the compiler and make less painful to do visibility stuff.

The kernel takes the image of the program, loads it using the elf header. The actual activation is not issued by giving control to `_start` but instead it launches the dynamic

loader. One can also ask to use a different dynamic linker by specifying it in the `.interp` section. If no dynamic linker is specified control is given at address specified in `e_entry`.

The linker initializes internal data structures, loads shared libraries, resolves the relocations and eventually transfer control to the application. Two main tables are used to find out where specific functions offered by shared libraries in the virtual address space cache information (`__dl_lookup`). PLT and GOT are the tables.

Dynamic libraries have 3 sections. dynsym is a symbol table used by the linker to perform relocation. Hash is a table used to find a symbol in the dynsym table. dynstr string table that allows to map symbols to their names.

Finally we have to give access to the program to global variables and other variables. The population is done upon need, is resolved only after the first call/access (*lazy binding*).

The PLT has except for the first one, one entry for each function that has to be solved. In the PLT there is a jump to the GOT entry. pp. 148

Code has a call to an entry of the PLT that was statically associated by the linker to that function. The entry of the plt has a jump to an entry of the GOT. In the got entry there is the address at system startup that jumps to the address immediately after in the PLT (in the first call). The resolver will use the dynsym and hash to find at what point of the address space there is the implementation of the function. Once found the address is written in the got entry. The first time is the resolver that jumps to the function. While in the second call the jump to the address written in the got entry will be directly to the function.

Once setup the libraries needed the linker jumps to `_start`. The code we're going to see is 32 bit. The first instruction is a xor to the `ebp` as suggeste by ABI to mark the outermost frame (initial frame of the program). The second instruction is a pop into esi which moves the argc to esi. The pop will have changed the stack pointer and then we move argv into ecx. Align the stack to 16 bytes and then push some stuff to then call `__libc_start_main`. This function will never return since the last instruction performed is `exit(main(...))`. The `hlt` ensures that if there is a misuse of the environment provided by the standard library it gets crashed.

Parameters of libc start main

destructor of the dynamic linker rtld fini.

Before activating main libc launches `libc_init_first`. It finds the environment variables and sets the global variable `_environ`. envp is found after argv and there is also one more table that is after envp and is called ELF Auxiliary table.

By setting `LD_SHOW_AUXV=1` when launching a program the ELF Auxiliary Table will be printed.

TLS keeps thread local variables. The standard variable will put fs to the address of TLS. Therefore TLS is not available without the standard library.

`_init()`: program constructor. Functions that are called before main. Constructors can be implemented also into static libraries. It checks for an entry in the plt callde `gmon_start` which is an entry tells if the program is being profiled etc. That entry contains the address of the function to setup profiling. init then calls `frame_dummy` which setups information about frame activation windows on stack used for exceptions in C++. Finally the constructors are called through `_do_global_ctors_aux`.

A loop calling function pointers taken from the address `__CTOR_END__` that is a symbol inserted by the compiler pointing to the table in section `.ctor` that is not present in the

program header.

Constructors can be inserted through the attribute facility. `libc_csu_init` calls a set of initial functions that also have arguments (constructors do not have parameters). Again these are defined through the attribute facility.

Exit calls also additional functions (is not the system call but the library exit function).

References