Talk Outline

1 Review of LINUX basics
- Processes
- File permissions

2 Command interpreters (shell)
- Basic scheme
- Additional features

3 Data structures in files
- Basics
- ELF object and executable files

4 Linking and loading
- Linker: merging files
- Linker: performing relocation
- Loading the executable file

UNIX/LINUX Basics: Processes

A unix process is a program in execution. Features:

- Process ID (pid) - index into system process table.
- Its own (almost) complete address space.
- Open files.
- Signals handling scheme.
- Real and effective user ID (uid).
- Real and effective group ID (gid).

The uid, gid are used to determine process permissions.

UNIX/LINUX Basics:
File permissions

Unix file permissions defined in mode word (see: man chmod).

- Owner (also called user)
- Group
- Others

For each category: rwx bits.

Other bits: suid, sgid, sticky

suid: if set, process effective uid becomes same as file owner's uid.

File permissions determine: result of open().
After open(), can access file regardless of permission bits!

Command Interpreter (Shell)

Command interpreter is a USER program, can be either:
1 Command-line interpreter.
2 Windows point-and click interpreter.

Simple command-line interpreter:

```
while(true) {
    get_line(buf, stdin);
    if(feof(stdin))
        exit(0);
    parse(buf, path, argv, envp);
    if(!(pid=fork()))
        execve(path, argv, envp);
    wait_for_child(pid);
}
```

Shell - continued

Running and loading a program:

execve(path, argv, envp)

This is a UNIX/LINUX system call:

- Maps executable file named path into memory.
- Prepares arguments for:
  main(argc, argv, envp)
- Sets up and jumps to program entry point.

Note: this is the same PROCESS, but executing a NEW executable file. Does NOT return to caller, except in case of error.
(E.g. if doing execve from a shell, this no longer a shell!).

Shell - continued

In order to continue running, shell "clones" itself using fork() system call. Clones receive:

- Complete copy of memory image.
- All open files.

After the fork:

- One instance (child) executes the commanded program.
- Other instance (parent) continues to execute the shell.

Can tell difference using value returned from fork(): ZERO to child, and the child pid to parent.

Shell: Additional Features

Background process (using & in command line): simply omit wait for child process!

UNIX/LINUX shells support redirection of stdin, stdout, stderr

- > path
- < path
- >& path

Can be implemented in shell by using freopen (or close/open).

Pipes. Example:

```
ls -a | tee list | wc
```

- Run two (or more) programs.
- Connect stdout of first program to stdin of second program, etc. using pipe() system call.

Shell: Additional Features

Shell allows for SCRIPTS.

Simplest version:
Simply reading command lines from a file (instead of stdin).

Improvements: control structures

- if
- while

Numerous other features:

- History mechanism and command line editing.
- Autocompletion
- Spelling and corrections
- Artificial intelligence...

Data Structures in Files (Basics)

Usage: persistent storage of a data structure.

Observe: usually cannot just store memory contents - POINTERS no longer valid!

Use offset in file instead (begin counting from byte 0 in file).

Typicaly use several bytes in file (say 4 bytes) as an unsigned integer offset.

Representation is critical (little endian vs. big endian).

Data Structures in Files (Basics)

Accessing data structures (after opening the file), either:

1 As a normal file (using lseek and then read/write).

2 In memory, following the mmap system call.

Access data in a normal file:

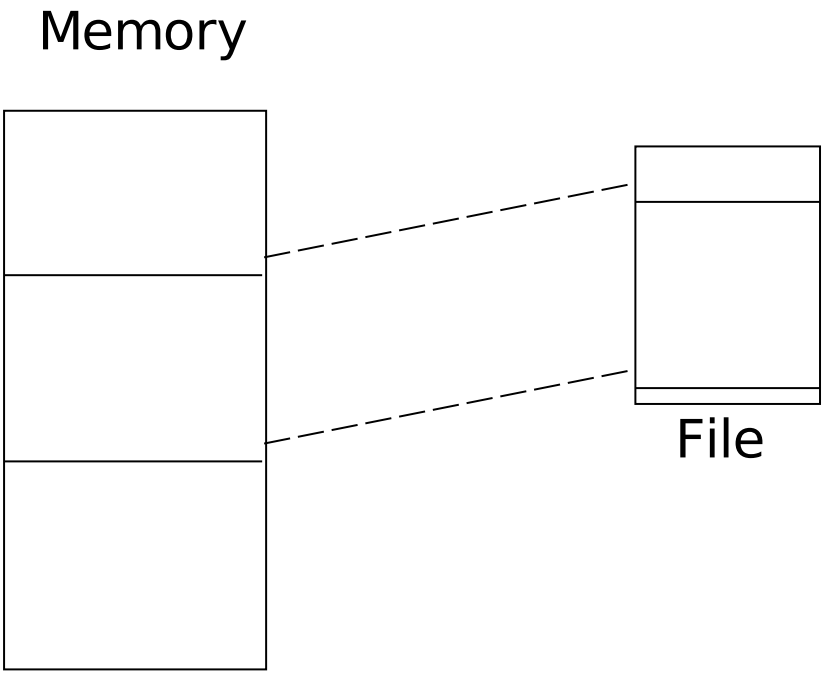
```
structure in_ mem;
unsigned int offset;

lseek(fd, POINTER_OFFSET, SEEK_SET);
read( fd, &offset, 4);
lseek(fd, offset, SEEK_SET);
read( fd, &in_ mem, size_of(structure));

in_ mem.member = modified_val;

lseek(fd, offset, SEEK_SET);
write( fd, &in_ mem, size_of(structure));
```

Data Structures in Files (Basics)

Using memory mapped files:

s = mmap(s, len, prot, flgs, fd, ofs);

- s: where in memory
- len: how many bytes
- prot: read, write, execute
- flgs: private, shared, fixed
- fd: file descriptor, of course...
- ofs: where in file to start



Data Structures in Files (Basics)

After mmap, simple(?) memory access of data.

Caveat: behaviour of "pointers"...

```
void *start;
unsigned int p_off;
structure * stp;

p_off = (*((unsigned int *)
            (start+POINTER_OFFSET)));

stp = (structure *)(start+p_off);

stp->member = modified_val;
```

Data Structures in Files
ELF Files

Executable Linkable Format:
(file) data structure for object, executable, and library files.

Contents of ELF file (simplified):

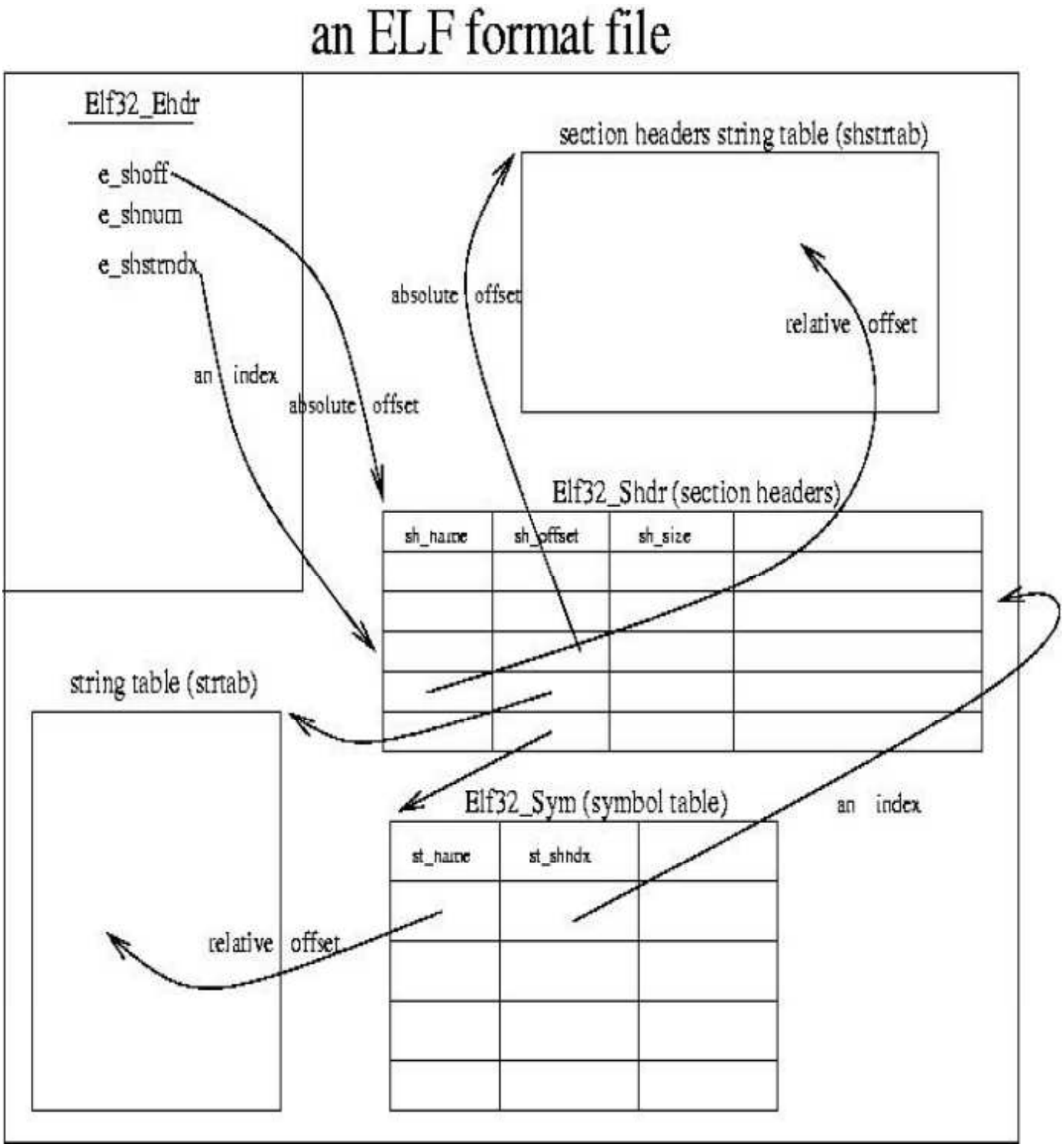
- ELF header.
- Program header table.
- Section header table.
- Sections (special: section header string table - shstrtab).

Visible using readelf utility .

Object files contain sections with:

- Code and data
- Symbol table(s)
- Relocation table(s)
- String table(s)

ELF Object File Sections



ELF Executable Files

Executable file contains:

- Code and data
- Program headers
- Entry point

May (optionally) contain sections.

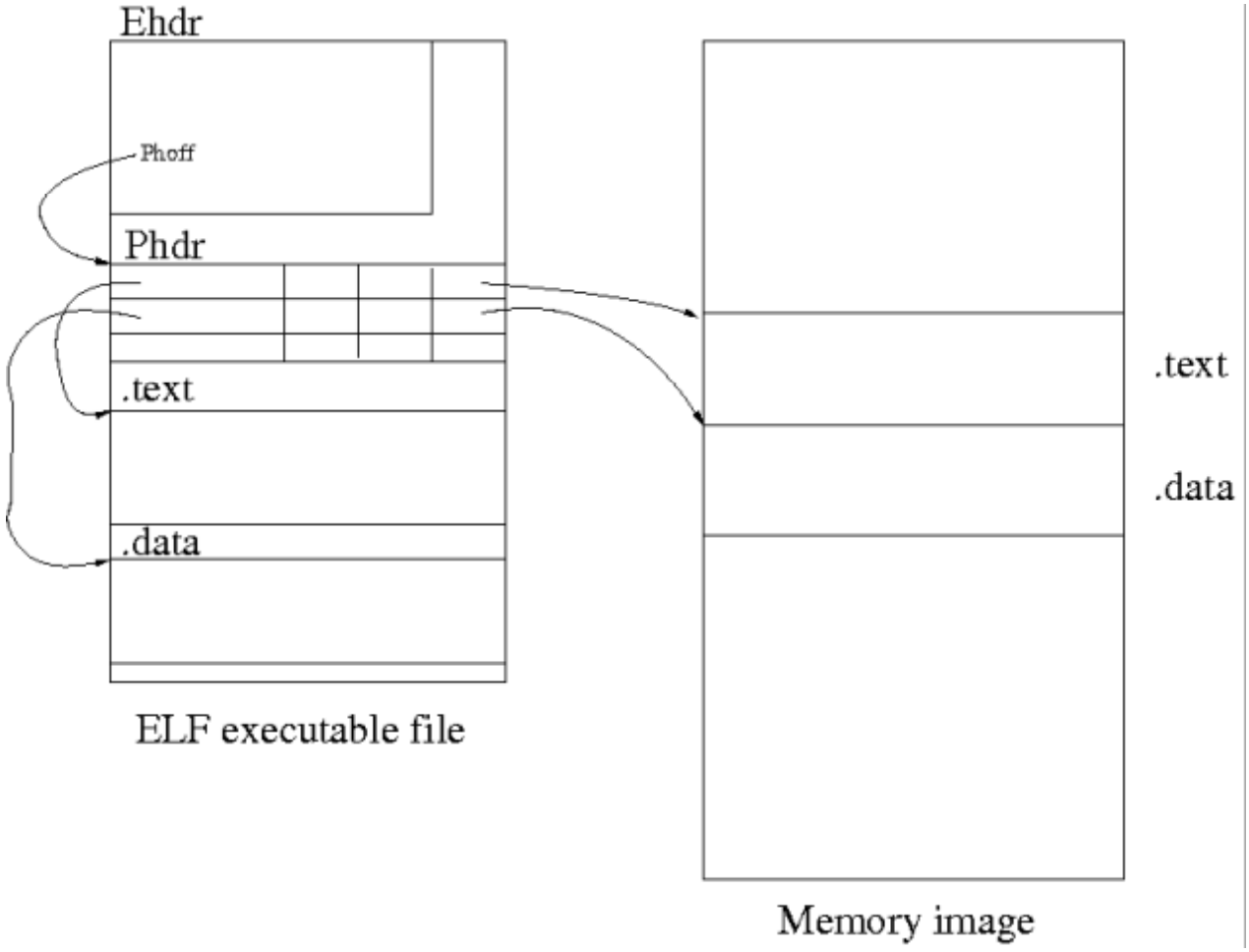
ELF header specifies:

- Offest of program headers.
- Number of program headers.
- Entry point (memory address).

Program header specifies:

- File offset + memory address
- Size in file + size in memory
- Flags (type, permissions)

ELF Executable Structure



Linking and Loading

Linking and loading process:
1 Compiler generates object files.
2 Linker links object files and libraries into executables.
3 Load and run executable.

Note: object code incomplete - cannot determine final memory address at compile time!

SYMBOL TABLE contains symbol names and properties.

RELOCATION TABLE specifies where to fix code/data.

Linking: Pass I (Merging)

Scan all files, to ensure that all symbols are UNIQUELY DEFINED in some file.

Method:
For each file f.o
  For each symbol s UNDEFined
    Find DEFINED s in another
    symbol table.

Can be done efficiently by first scanning all symbol tables and keeping them in memory.

Merge sections from files
(code, data, symbols, relocation)

Linking: Pass II (Relocation)

Process has several steps:

1 Assign memory locations.
2 Finalize symbol values.
3 Perform relocations (fixups).
4 Create program headers.
5 Establish entry point and flags to make executable.

Linking: Pass II
Memory Locations

From initial address (0x8048000), plan memory assignments:

- Code sections (.text)
- Read-only data (.rodata)
- Initialized data sections (.data)
- Uninitialized data (.bss)

Example:
section .text size 0x5000, starts in memory at 0x8048000 and ends at:

```
    0x8048000
  +     0x5000
-----------------------
    0x804D000
```

Section .rodata can start there.
(Complication: blocks...)

Linking: Pass II
Finalizing Symbol Values

Now we know where sections will be in memory at runtime!

Symbol information:

- Section in which symbol is defined.
- Offset inside section.

Can compute final symbol value:

```
    section memory address
+   offset inside section
-----------------------------------------
     final memory address
```

Linking: Pass II
Actual Relocation (Fixups)

Relocation table entry contains:

- Offset inside section (what do we need to fix?)
- Symbol on which this depends.
- Relocation type and size.

We will examine two types:

1 Fixed address

2 Relative address

Both types can use a "trick" where some meaningful value already stored at fixup location.

Linking: Pass II
Actual Relocation (Fixups)

Fixed address: what should be there is an absolute address of something (data or code).
Therefore: write symbol value
(+ value already at fixup location).

Relative address: what should be there is the "distance" (in bytes) of the symbol value from the "current" address.
Therefore: write symbol value MINUS memory address of fixup (+ value already at fixup location).