SYSTEM SOFTWARE
OBJECT FILE

Nguyen Huu Duc

School of Information and Communication Technology
Hanoi University of Science and Technology

What is an object file?

- Object files are binary files that is produced by an assembler or compiler
- Object files can be combined (by the linker) to generate a single object
- Object files can be loaded on memory (by the loader) to execute or be used by other programs.

Objective and content of the lesson

- Objective:
  - Understand the role of object files
  - Understand structure of an object file and its organization mechanism
    - under limitation of hardware
    - in the allocation of memory
    - in the relocation
- Content
  - Study structure of a common object file.
    - MS-DOS COM
    - MS-DOS EXE
    - UNIX a.out
    - UNIX ELF
  - Study allocation of memory for object files
  - Study techniques for relocation of object files

Classification of object file

- Linkable
  - Linkable with other object files
  - Used as input to the linker
  - Contain global symbols and relocation information
- Executable
  - Loadable on memory and executable
  - Usually contain page code for easily mapping on address space
  - Not contain symbol
  - Not have (very little) relocation information
- Loadable
  - Be loadable onto memory and used by other programs (libraries)
  - Static linking: Not contain symbols
  - Dynamic linking: contain symbols and relocation information

Content of an object file

- Header: General information about the whole object file
  - Size of each segment
  - Name of the object file
  - Creation date
- Object code
  - Execution code
  - Data
- Relocation information
  - Positions in object code need to be re-aligned when the linker changes the original address of that snippet code.
- Symbol
  - Global symbols are defined in module (export)
  - Symbols refer to the outside of module (import)
  - Symbols are defined by the linker
- Debug information
  - Content of source file and row index.
  - Symbols that are locally used in module
  - Data structures are used in object code

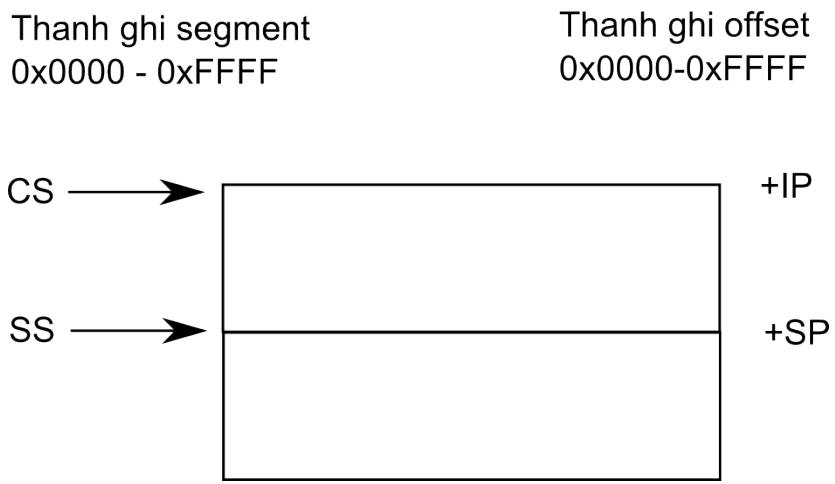
Forms of object code

- MS-DOS COM (Null Object Format)
- MS-DOS EXE (MZ)
- UNIX a.out
- UNIX ELF (Executable and Linking Format)

Memory management in 8086 processor

- 8086 processor uses 16-bit registers to address for program
  - E.g. Instruction pointer IP, Stack pointer SP
  - 64KB memory limit
- To use the memory space greater than 64KB, it needs to combine with 16-bit segment registers
  - Segment register for program code CS
  - Segment register for stack SS
  - Segment register for data DS, ES
- Address conversion
  - (segment:offset) $\rightarrow$ absolute address = (segment lsh 4) + offset
  - Access up to 1 MB of memory

Segment address



Example

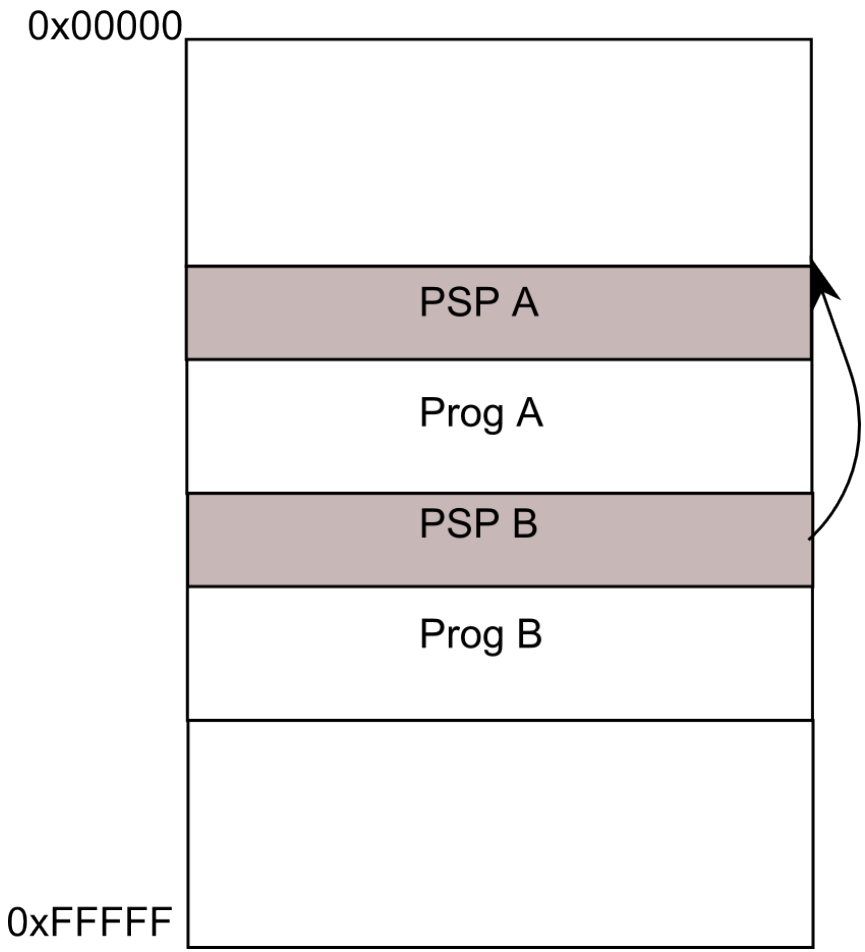
```
CS:IP points to instruction address
CS=0x1234, IP=0x2140
Absolute address: 0x12340 + 0x2140 = 0x14480

SS:SP points to the top of stack
SS=0x8000, SP=0xFFFF
Absolute address: 0x80000 + 0xFFFF = 0x8FFFF
```

Memory management in MS-DOS

- All programs are loaded onto the same address space
- When a program is loaded
  - A new memory region is allocated immediately after the memory region was used, on the top of freedom memory
  - PSP(program segment prefix, 256 bytes) is created on the top of this new memory region
    - Contain information about size of the allocated memory region, point to caller's PSP, etc.
  - Load and execute the program
- The program is not loaded at a fixed address.

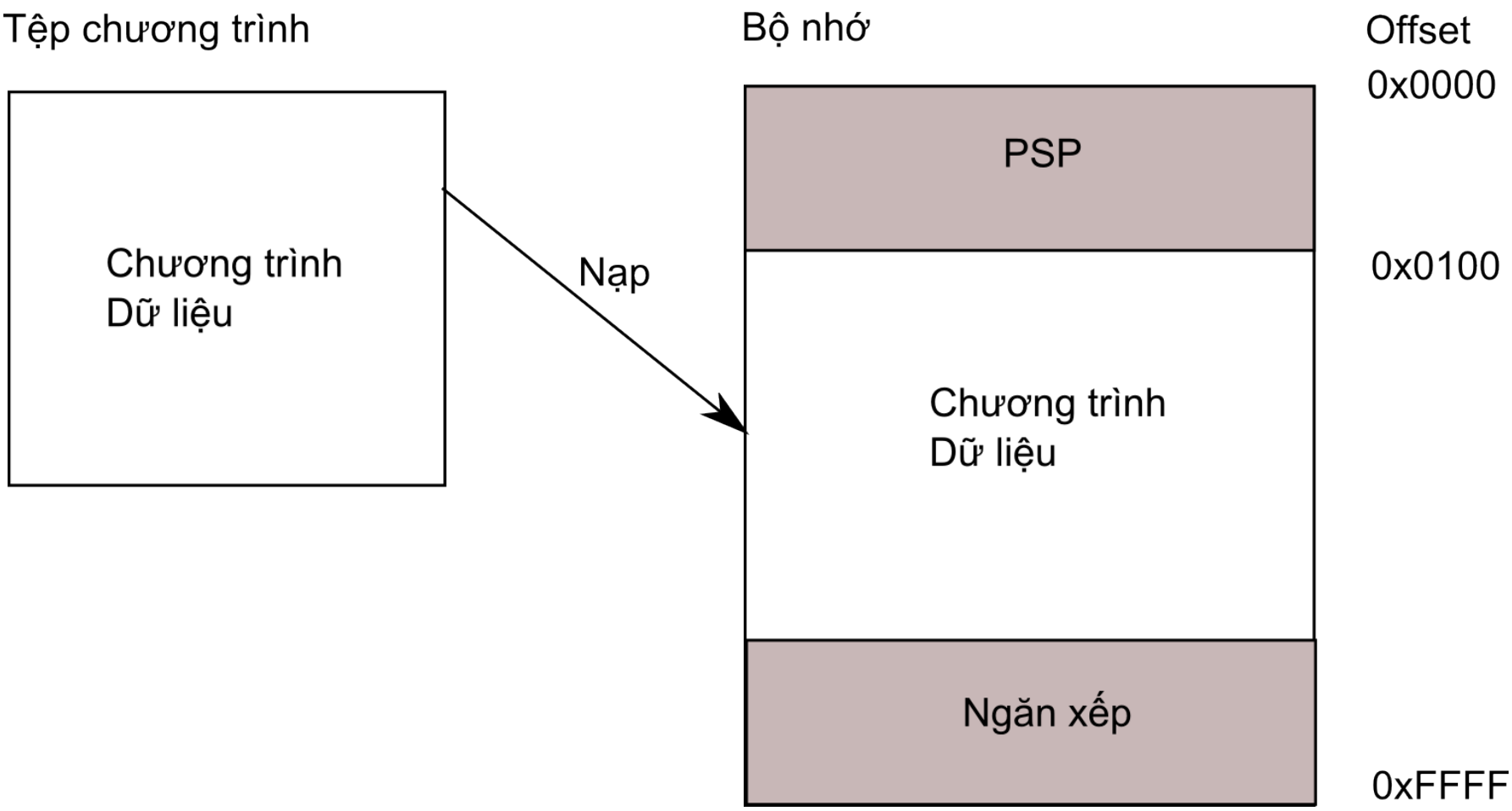
Memory management in MS-DOS



The object file MS-DOS COM (Null Object Format)

- Characteristic
  - Simple
  - Only contain program code (execution code, data), not contain anything else
- When a program is loaded
  - A new memory region is allocated immediately after the memory region was used, on the top of freedom memory
  - PSP(program segment prefix, 256 bytes) is created on the top of this new memory region
    - Contain information about size of the allocated memory region, point to caller's PSP, etc.
  - Load and execute the program
- The program is not loaded at a fixed address.

The object file MS-DOS COM

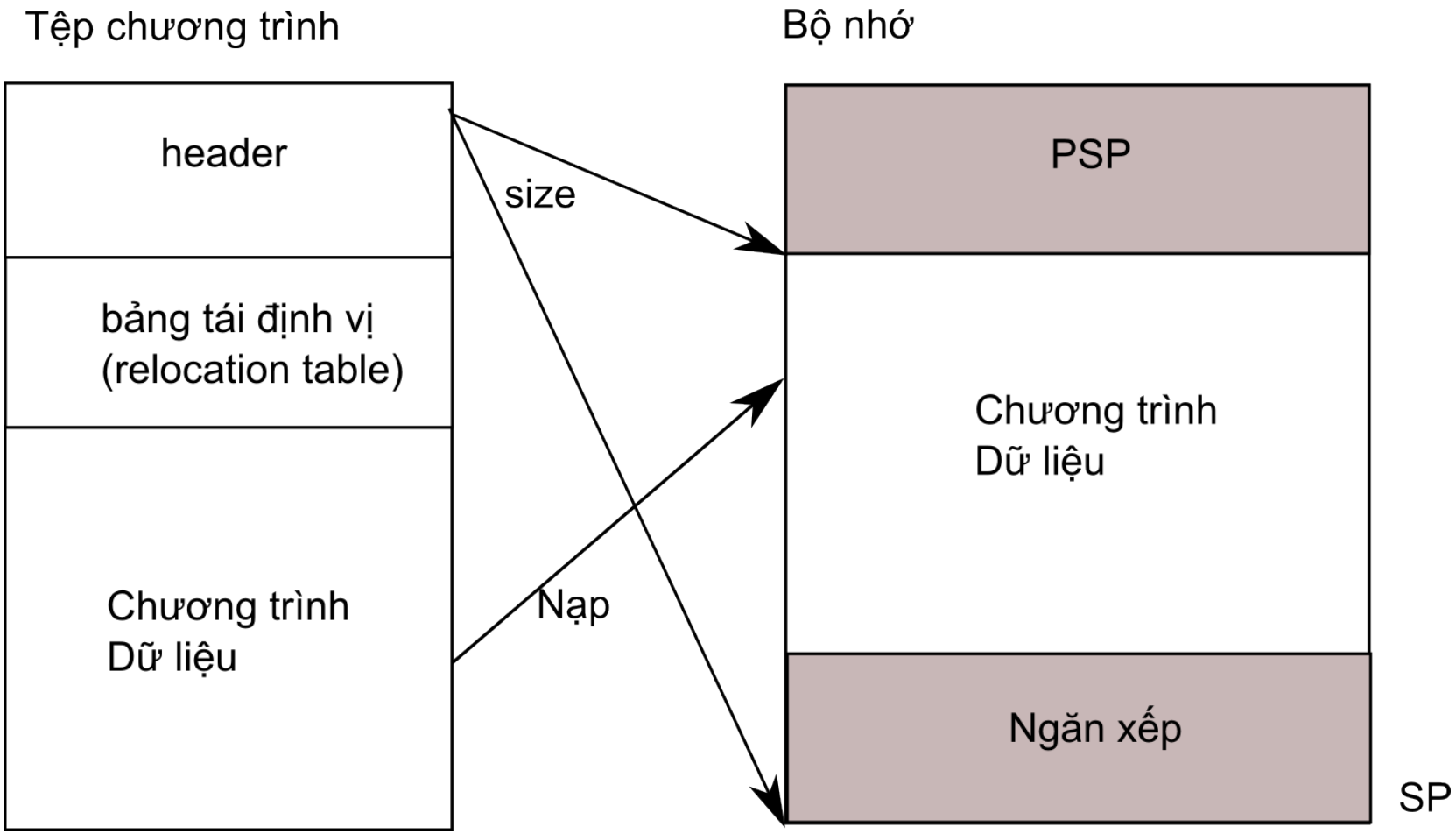


The object file MS-DOS COM

- Load and execute (by operating system)
  - Determine the loading address of program
  - Create PSP on memory
  - Load the program starting from the offset 0x0100
  - Initialize segment registers and stack pointer (SP)
  - Execute the program from the address 0x0100
- A program can be loaded on any memory region (that is determined by operating system via setting segment registers)
- 64KB memory size limit

The object file MS-DOS EXE (MZ)

- Contain relocation table
- Its size can be greater than 64KB

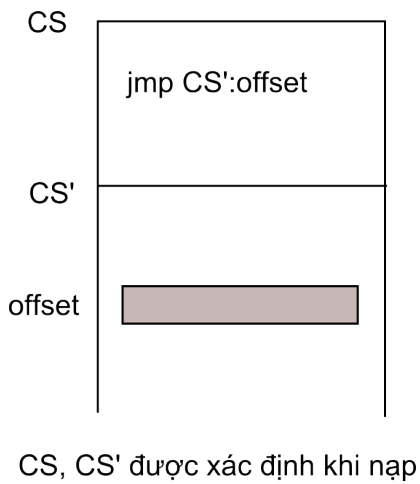


The object file MS-DOS EXE

- Header
  - Size (header, relocation table, program, data)
  - Initial values for SP and IP
  - `minalloc` stores size of BBS
  - `maxalloc`
  - ...
- Program and data
  - Be combined together
  - BBS region is particularly determined by `minalloc`

The object file MS-DOS EXE

- Relocation table
  - Program loading address is not specified when compiling
  - The compiler generates code for the program starting from the address 0x0000
  - Jump instruction (across the border of a segment) is set by real address (generated by compiler)



The object file MS-DOS EXE

- Load and execute (by operating system)
  - Read the header and determine size of memory allocating for program
  - Create PSP on memory
  - Load program (`size = nblocks * 512 + lastsize`)
  - Relocation
    - Position: `relocpos`
    - Amount: `nreloc`
  - Rewrite operands of program instructions (Especially, segment address) described in relocation table
  - Initialize SP, IP and execute the program

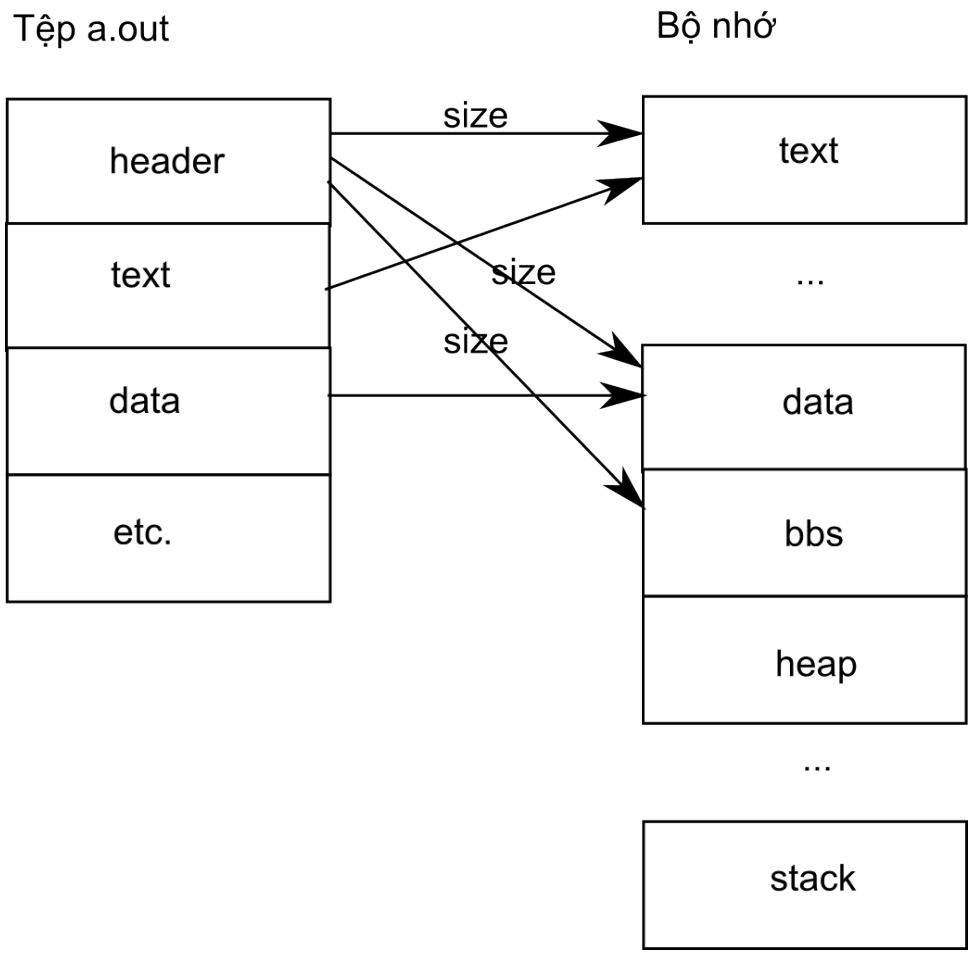
MS-DOS EXE header

```
char signature[2] = “MZ”;// magic number
short lastsize;          // size of data in final block
short nblocks;           // number of 512-byte-block
short nreloc;            // number of relocation entry
short hdrsize;           // size of file header(in 16byte praragraph)
short minalloc;          // minimum size of additional memory
short maxalloc;          // maximum size of additional memory
void far *sp;            // initial stack pointer
short checksum;          // 1’s complement of sum of file content
void far *ip;            // initial instruction pointer
short relocpos;          // location of fixup-table for relocation
short noverlay;          // number for overlay(0 for normal program)
char extra[];            // extra for overlay
void far *relocs[];      // pointer to relocation ently
```

The object file a.out (NMAGIC)

- Is a traditional form of Unix, using widely from 1990
- Program and data are organized independently
  - Program can be divided into read-only pages or read/write pages. Ensuring safety.
  - Easy to share
    - `fork`: share program
    - `thread`: share program and data, stack is not shared
    - `vthread`: share everything
- Entry of each program is definability due to supporting virtual memory mode (Hardware does relocation)

The object file UNIX a.out



a.out header

```
int a_magic; // magic number
int a_text; // size of text segment
int a_data; // size of initialized data
int a_bss; // size of uninitialized data
int a_syms; // size of symbol table
int a_entry; // entry point
int a_trsize; // size of relocation information of text
int a_drsize; // size of relocation information of data
```

Memory segments in the file a.out

- Program segment (text) : contain program code
- Data segment (data): Contain initialized data
- Other segment (not necessary for the execution)
  - Relocation table for program
  - Relocation table for data
  - Symbol table
  - Name table

Load and execution the object file a.out

- Read the header and calculate data size
- Check program code to see if it is shared by other processes
  - If so, mapping to the shared memory
  - If not, allocate a new memory region for the code.
- Prepare the data area that is initialized and not initialized
  - Read and copy the initialized data.
  - Erase the un-initialized data to 0
- Create stack
- Set value for registers and pass the control to the beginning of program.

Derivative forms of the object file a.out

- ZMAGIC : designed for paging mechanism
  - Format is in Page (4KB)
  - Operating system can store/restore a raw memory image to disk
- QMAGIC
  - Eliminate redundancy in the header and segments
  - To avoid NULL pointer, operating system does not map the page 0

Relocation form of a.out

- Relocation form of a.out (relocatable a.out) is a form of object file supporting linking feature (file .o)
- Contain the following information:
  - Relocation information for program code
  - Relocation information for data
  - Symbol table
  - Name table
- Not support
  - Dynamic linking
  - Object-oriented language (E.g. C++)

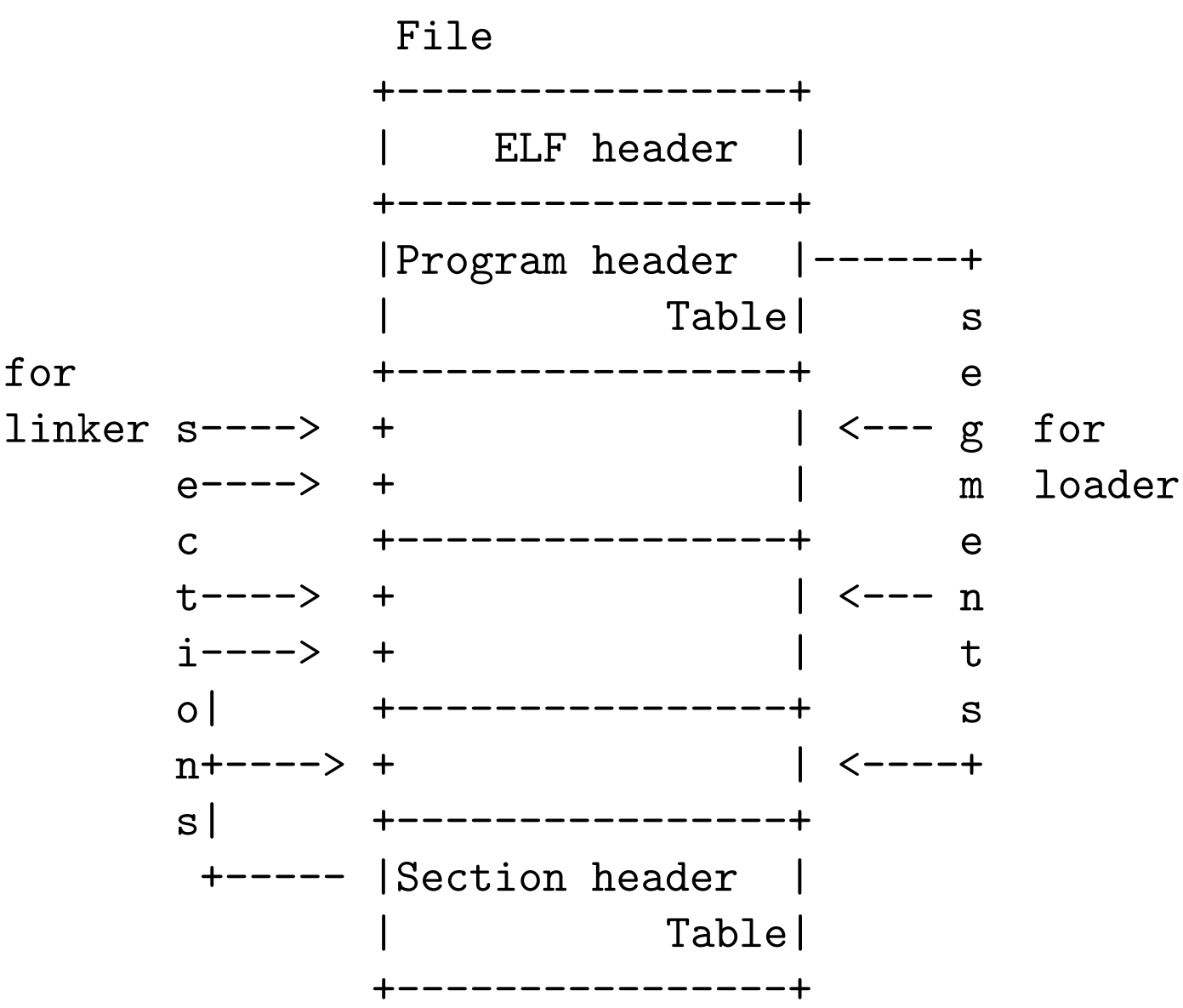
Relocation form a.out

```
file
+--------+
| header |
+--------+
|text    |
+--------+
|data    |
+--------+
|reloc.  | ... relocation information
|for text|     (only for undefined symbols)
+--------+         - address (offset of text/data seg.)
|reloc.  |         - text seg.? or data seg.?
|for data|         - pointer to symbol table
+--------+
|symbol  | ... Information about relocation symbols (function/variable)
|   table|         - address (offset in string table)
+--------+         - type (global or local)
|string  |
|   table| ... Label of symbol (function/variable)
+--------+
```

The Object file ELF (Executable and Linkable Format)

- Linkable, executable, and loadable
- Support dynamic linking
- Support object-oriented languages
- header has rich information
  - Architecture
  - Byte order
  - 32bit/64bit

Structure of the object file ELF



ELF header

```
char magic[4] = "\177ELF";  // Magic number
char class;       // address size (1:32bit, 2:64bit)
char byteorder;   // byte order (1:little endian, 2:big endian)
char hversion;    // header's version(= 1)
char pad[9];
short filetype;   // 1:relocatable,2:executable,3:shared object,4:core image
short archtype;   // architecture(2:SPARC, 3:x86, 4:68K, ...)
int fversion;     // file version(= 1)
int entry;        // starting point in case of executable file
int phdrpos;      // location of program header, or 0
int shdrpos;      // location of section header, or 0
int flags;        // flag set for a specific architecture (normally set to 0)
short hdrsize;    // size of ELF header
short phdrent;    // size of one item in program header
short phdrcnt;    // the number of item in program header, or 0
short shdrent;    // size of one item in section header
short shdrcnt;    // the number of item in section header, or 0
short strsec;     // number of section containing name of sections
```

Partition in the object file ELF

- Program and data are partitioned into multiple section, which is convenient for the phase of linking
  - `.text`: program code section
  - `.init`: program code section for initializing program
  - `.fini`: program code section for finalizing program
  - `.data`: data section
  - `.bbs`: un-initialized data section
  - `.rodata`: read-only data section
  - `.rel .text`: relocation table for program code section
  - `.rel .rodata`: relocation table for read-only data section
  - `.rel .data`: relocation table for data section
  - `.symtab`: Symbol table
  - `.dynsym`: Symbol table for dynamic linking
  - `.strtab`: Name table
  - `.dynstr`: Name table for dynamic linking symbols

Section header

```
int sh_name;      // Name (index to the name table)
int sh_type;      // section type
int sh_flags;     // flag
int sh_addr;      // Base address of section (if loaded), or 0
int sh_offset;    // Starting location of section in the file
int sh_size;      // section size
int sh_link;      // number of assosiated section, or 0
int sh_info;      // additional information for each kind of section
int sh_align;     // Value need to be aligned when shifting section
int sh_entsize;   // size of item if the section is an array
```

Partition in the object file ELF

- Progarm and data are divided into multiple segments, which is convenient for the execution
  - Segment for program: E.g. `.text`, `.init`, `.fini`
  - Segment for read-write data `.data`
  - Segment for read-only data `.rodata`

Program header

```
int type;       // Type of program segment (code, data,...)
int offset;     // offset addres of program segment in the file
int virtaddr;   // virtual address for mapping segment
int physaddr;   // physical address (not used)
int filesize;   // segment size in the file
int memsize;    // segment size on the memory
int flags;      // flag bit (Read, Write, Execute)
int align;      // Segment alignment information (size of a page)
```

Symbol table in the object file ELF

- The same to the symbol table in text sextion
  - name: Pointer to string containing the name in string table
  - value: symbol's address (in program section, or data)
  - size: memory size for a symbol
  - type: symbol type (data, function, section)
  - bind: scope (global/local)
  - sect: number of section or flag showing that the object is not defined yet

Symbol table

```
int name;       // name (index in the name table)
int value;      // Value (Relative for relocation file
                          Absolute for executable file)
int size;       // size of object or function
char type:4;    // Type (object, function, section, or special symbol)
char bind:4;    // scope (local, global, weak)
char other;     // reserve
short sect;     // number of segment, ABS, COMMON, or UNDEF
```