Using GNU Compiler and Binutils by Example

Hao-Ran Liu

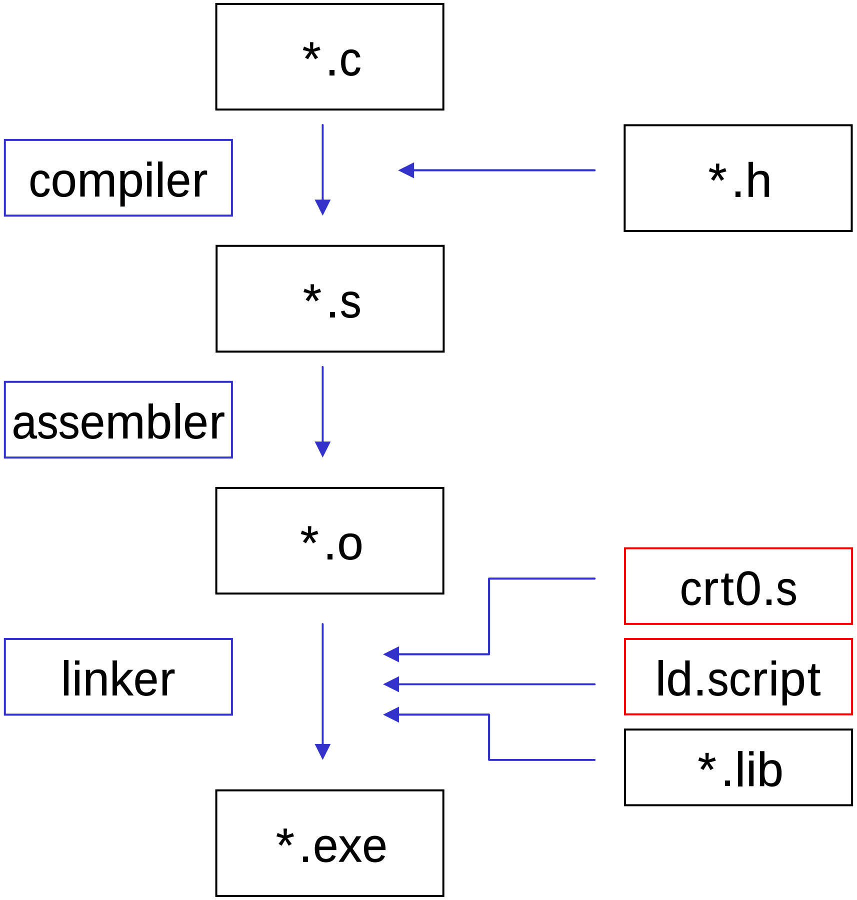
Goals of this tutorial

- To familiar with the building process of the image of Linux kernel
- To learn the know-how of building an embedded platform

GNU toolchains

- GNU toolchain includes:
  - GNU Compiler Collection (GCC)
  - GNU Debugger (GDB)
  - GNU C Library (GLIBC)
    - Newlib for embedded systems
  - GNU Binary Utilities (BINUTILS)
    - Includes LD, AS, OBJCOPY, OBJDUMP, GPROF, STRIP, READELF, NM, SIZE…

Compiling procedure



*.c

compiler

*.h

*.s

assembler

*.o

crt0.s

linker

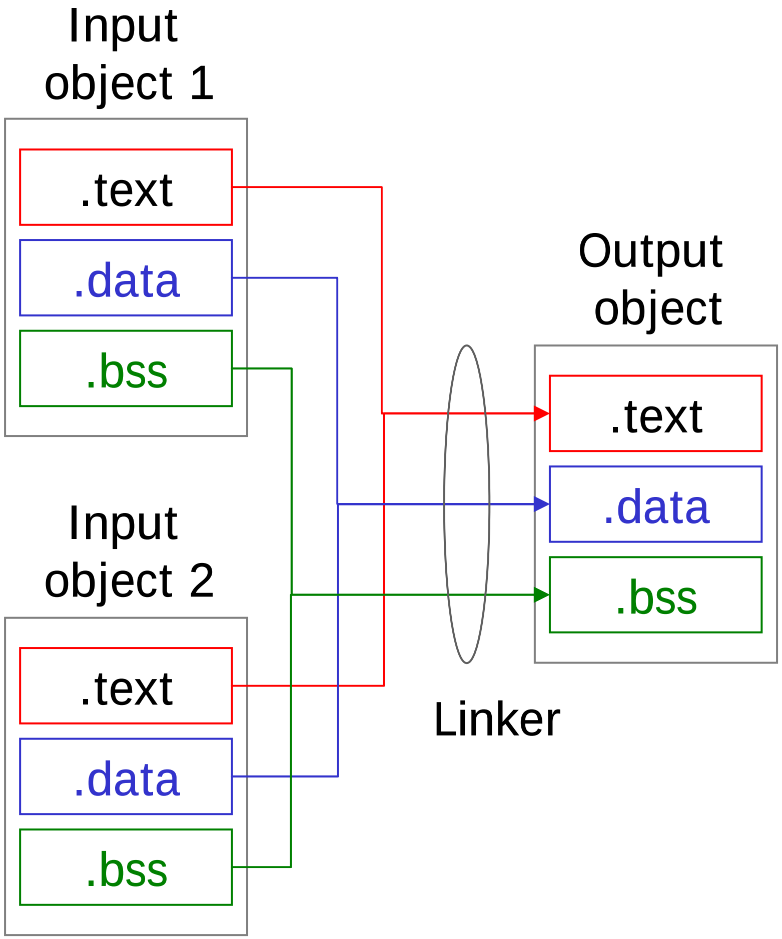
ld.script

*.lib

*.exe

Linker overview

- Linker combines input objects into a single output object
- Each input object has two table and a list of sections.
- Linker use the two table to:
  - Symbol table: resolved the address of undefined symbol in a object
  - Relocation table:
    - Translate ‘relative addresses’ to ‘absolute address’

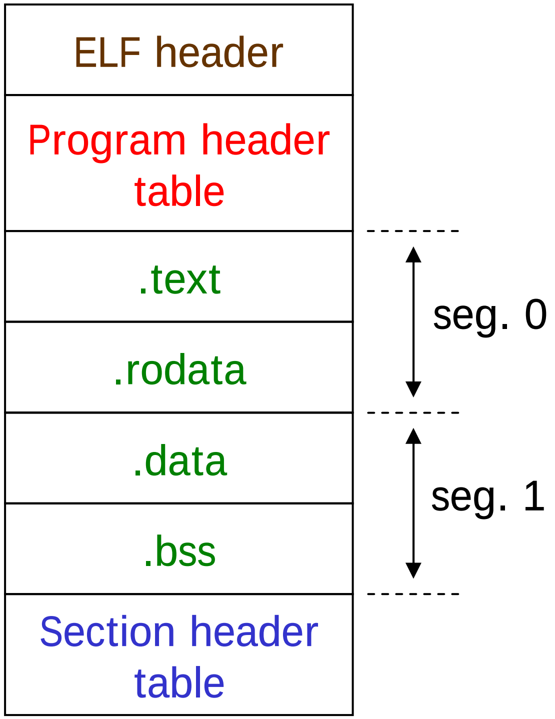


The roles of crt0.s and ld.script in embedded development environment

- crt0.s
  - The real entry point: _start()
  - Initialize .bss sections
  - Initialize stack pointer
  - Call main()
- Linker script
  - Control memory layout of a output object
  - How input objects are mapped into a output object
  - Defaule linker script: run ld --verbose

ELF format

- What we load is partially defined by ELF
- **E**xecutable and **L**inkable **F**ormat
- Four major parts in an ELF file
    - ELF header – roadmap
    - Program headers describe segments directly related to program loading
    - Section headers describe contents of the file
    - The data itself



ELF header

Program header table

.text

.rodata

.data

.bss

Section header table

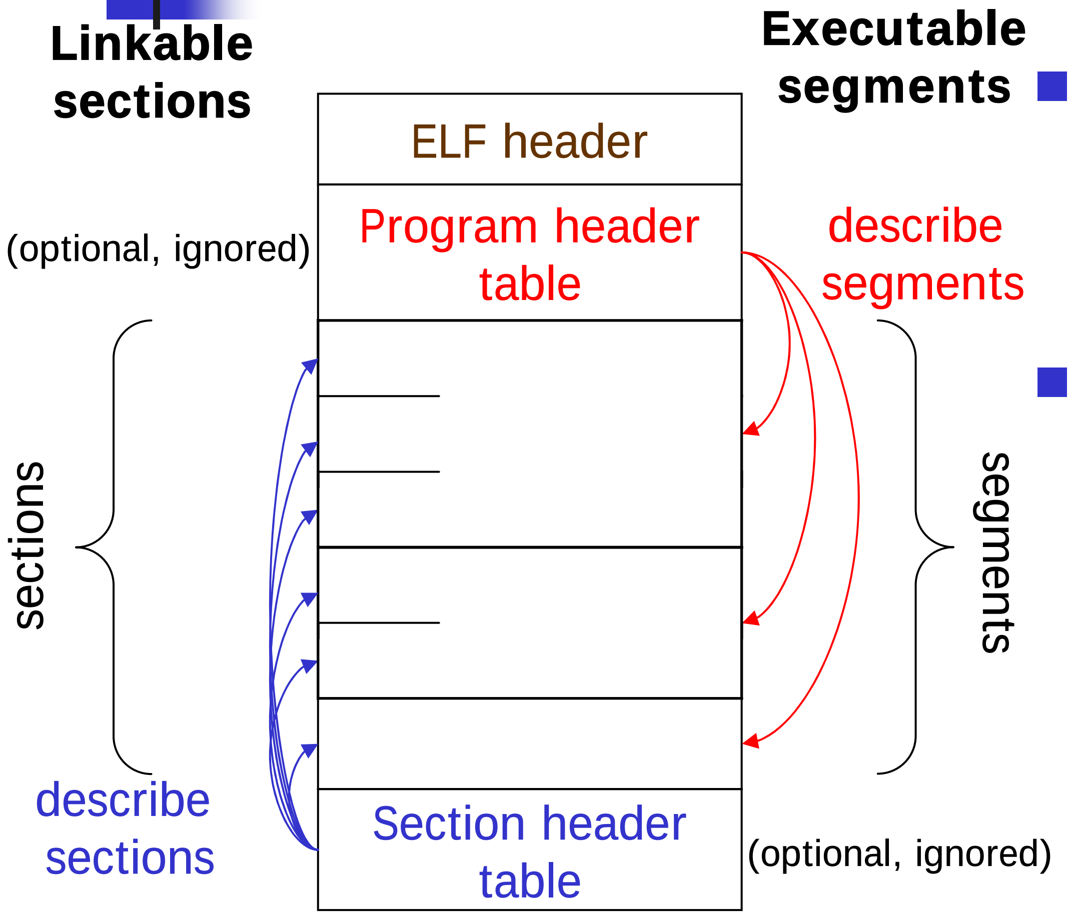
seg. 0

seg. 1

3 types of ELF

- relocatable(*.o) for linker
- Executable(*.exe) for loader
- shared object for both(*.so) (dynamic linking)

Two views of an ELF file



Linkable sections

Executable segments

ELF header

(optional, ignored)

Program header table

describe segments

sections

segments

describe sections

Section header table

(optional, ignored)

- Program header is for ELF loader in Linux kernel
- Section header is for linker

	Program header	Section header
relocatable		V
executable	V	V
Shared object	V	V

ELF header

```
typedef struct {
   char magic[4] = "\177ELF"; // magic number
   char class;      // address size, 1 = 32 bit, 2 = 64 bit
   char byteorder; // 1 = little-endian, 2 = big-endian
   char hversion;  // header version, always 1
   char pad[9];
   short filetype; // file type: 1 = relocatable, 2 = executable,
                   // 3 = shared object, 4 = core image
   short archtype; // 2 = SPARC, 3 = x86, 4 = 68K, etc.
   int fversion;   // file version, always 1
   int entry;      // entry point if executable
   int phdrpos;    // file position of program header or 0
   int shdrpos;    // file position of section header or 0
   int flags;      // architecture specific flags, usually 0
   short hdrsize;  // size of this ELF header
   short phdrent;  // size of an entry in program header
   short phdrcnt;  // number of entries in program header or 0
   short shdrent;  // size of an entry in section header
   short phdrcnt;  // number of entries in section header or 0
   short strsec;   // section number that contains section name strings
} Elf32_Ehdr;
```

ELF section header

```
typedef struct {
    int sh_name;    // name, index into the string table
    int sh_type;    // section type (PROGBITS, NOBITS, SYMTAB, …)
    int sh_flags;   // flag bits (ALLOC,WRITE, EXECINSTR)
    int sh_addr;    // base memory address(VMA), if loadable, or zero
    int sh_offset;  // file position of beginning of section
    int sh_size;    // size in bytes
    int sh_link;    // section number with related info or zero
    int sh_info;    // more section-specific info
    int sh_align;   // alignment granularity if section is moved
    int sh_entsize; // size of entries if section is an array
} Elf32_Shdr;
```

ELF program header

```
typedef struct {
   int type;      // loadable code or data, dynamic linking info, etc.
   int offset;    // file offset of segment
   int virtaddr;  // virtual address to map segment (VMA)
   int physaddr;  // physical address (LMA)
   int filesize;  // size of segment in file
   int memsize;   // size of segment in memory (bigger if contains bss)
   int flags;     // Read, Write, Execute bits
   int align;     // required alignment, invariably hardware page size
} Elf32_Phdr;
```

Section header of a executable

```
$ objdump –h vmkernel

vmkernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000130  00200000  00200000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000049  00200140  00200140  00001140  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00000044  0020018c  0020018c  0000118c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          00002020  002001e0  002001e0  000011e0  2**5
                  ALLOC
  4 .comment      00000033  00000000  00000000  000011e0  2**0
                  CONTENTS, READONLY
```

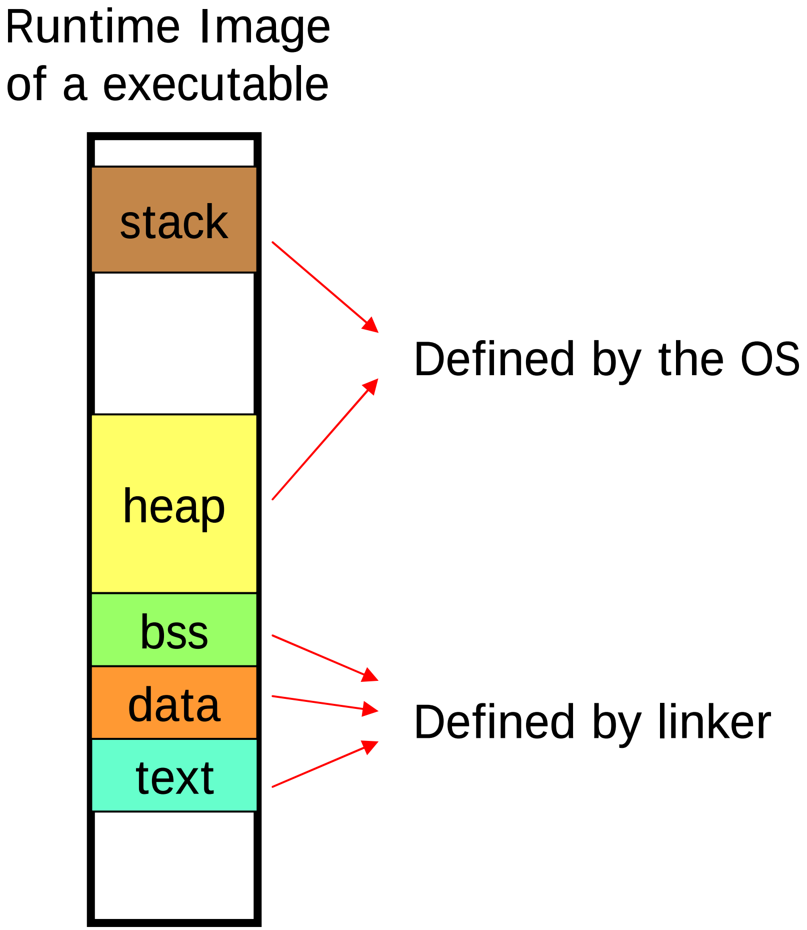
Program header of a executable

```
$ objdump -p vmkernel

vmkernel:     file format elf32-i386

Program Header:
  LOAD off    0x00001000 vaddr 0x00200000 paddr 0x00200000 align 2**12
       filesz 0x000001d0 memsz 0x00002200 flags rwx
```

- Note: memsz > filesz
  - The difference (.bss section) is zero-inited by the operating system

Where do we load?

- A program's address space is defined by linker and operating system together.



Where do variables go?

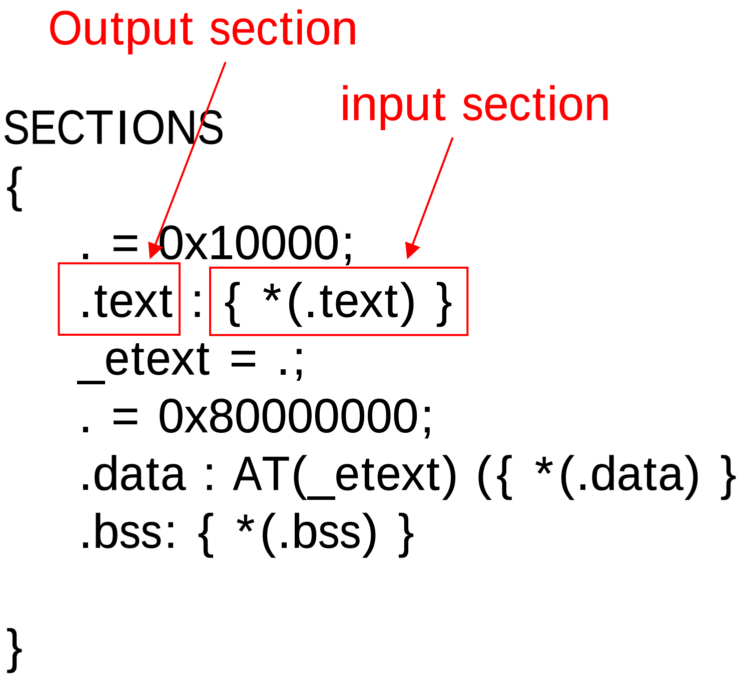
			.text	.rodata	.data	.bss	stack
global	static	initialized			v		
		non-initialized				v	
	non-static	initialized			v		
		non-initialized				v	
	const			v			
local	static	initialized			v		
		non-initialized				v	
	non-static	initialized					v
		non-initialized					v
	const		v				
Immediate value			v				

Basic linker script concepts

- Linker scripts are often used to serve the following goals:
  - Change the way input sections are mapped to the output sections
  - Change LMA or VMA address of a section
    - LMA (load memory address): the address at which a section will be loaded
    - VMA (virtual memory address): the runtime address of a section
    - In most cases the two addresses will be the same. An example of when they might be different is when a data section is loaded into ROM, and then copied into RAM when the program starts up
  - Define additional symbols for use in your code

An example of linker script

- SECTIONS command defines a list of output sections
- ‘.’ is the VMA location counter, which always refer to the current location in a output object
- ‘*’ is a wildcard to match the filenames of all input objects
- ‘_etext’ is a symbol of the value of current location counter
- AT command change the LMA of .data section
  - Only .data section has different addresses for LMA and VMA



```
SECTIONS
{
      . = 0x10000;
      .text : { *(.text) }
      _etext = .;
      . = 0x80000000;
      .data : AT(_etext) ({ *(.data) }
      .bss: { *(.bss) }

}
```

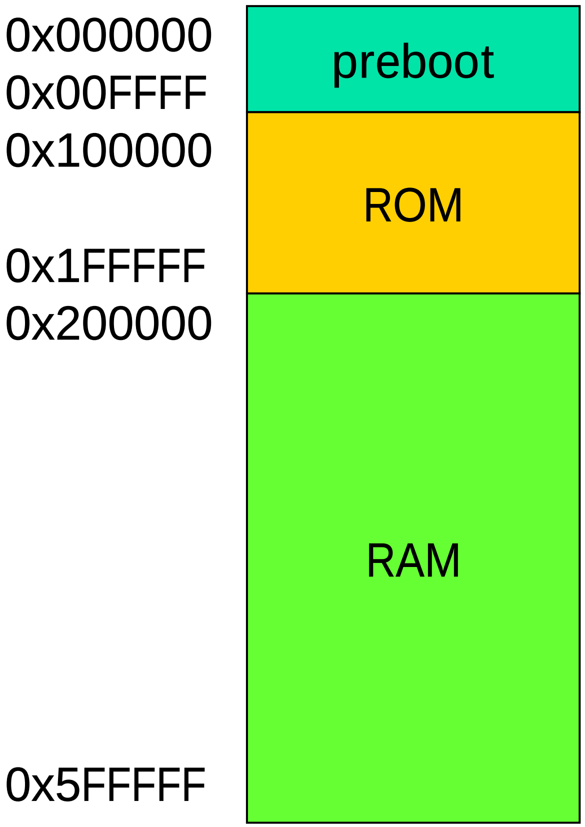
A mini-example demonstrating development of embedded system

- Runs in Linux user level for 3 reasons:
  - Learn most of the essential concepts without real hardware
  - Verify runtime memory contents with the help of GNU debugger
  - Avoid writing machine dependent code (switch into 32-bit protected mode on x86); besides, GCC cannot generate 16-bit code

Mini-example component overview

- preboot
  - preboot.c
  - This stage doesn't exist on real system. It is a helper loader to load boot image into ROM area
- Boot
  - head.s, boot.c
  - The boot loader, copy kernel from ROM to RAM
- Kernel
  - head.s, main.c
  - This is the kernel ☺

Memory layout



0x000000
0x00FFFF
0x100000
0x1FFFFF
0x200000
0x5FFFFF

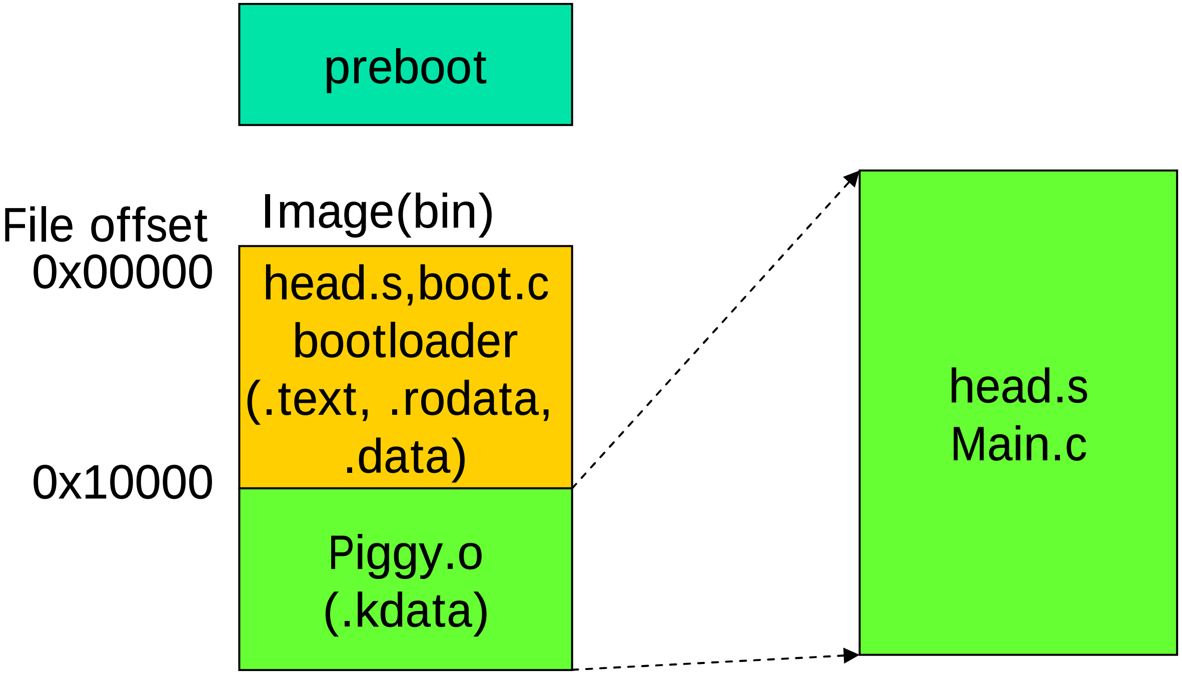
preboot

ROM

RAM

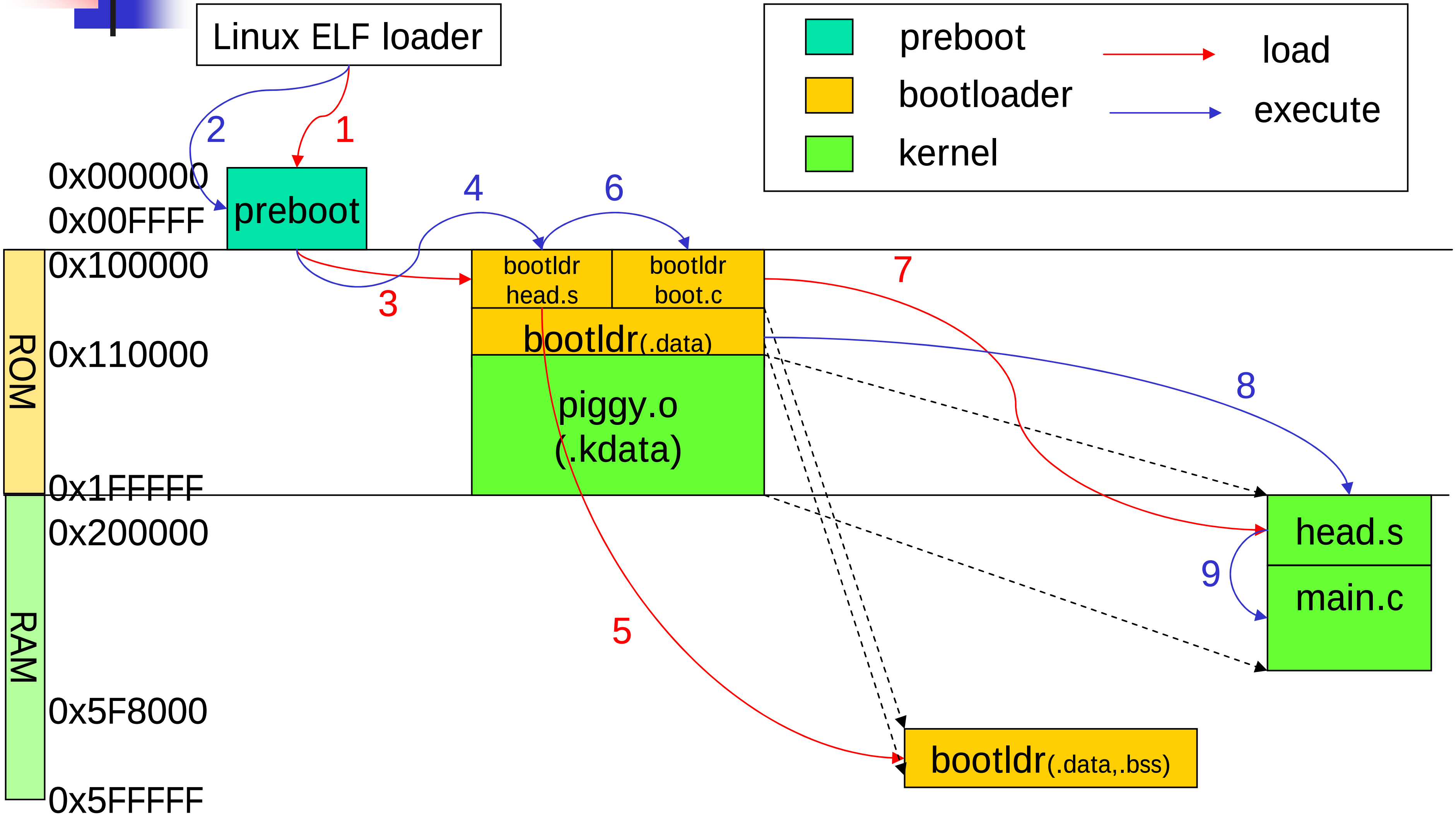
Mini-example file layout

- There are 2 runnable files in this example
  - Preboot
  - Boot image
    - Boot loader
    - kernel



* Kernel (vmkernel.bin) is embedded inside piggy.o as a .kdata section

Mini-example loading: step by step



The design of preboot.c

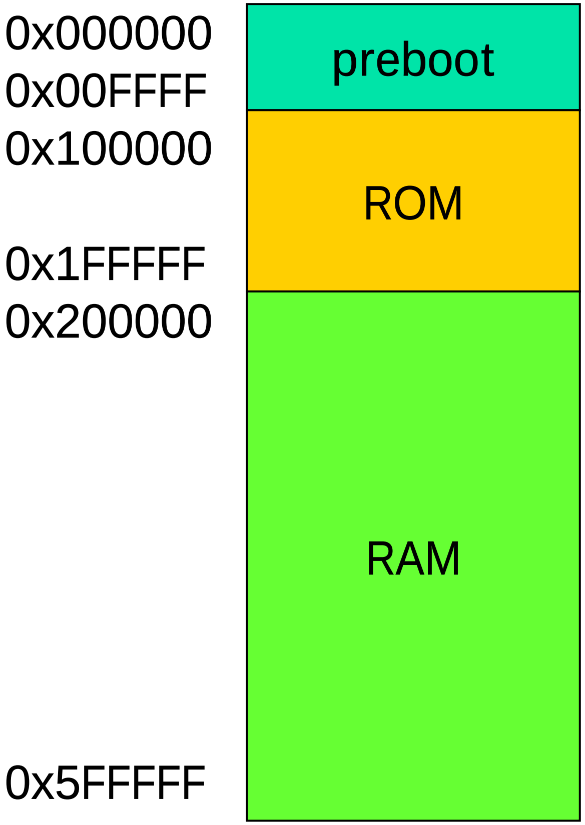
- To simplify the layout of runtime memory, no C library function, only system calls!
- _start() is the entry point! call _exit() system call to end the function
- Loaded at 0x0000 by Linux and then:
  - brk() to increase the boundary of data segment to 0x00600000
  - open() file "Image" and copy it to 0x100000
  - Jump to 0x100000

Simplified preboot.c

```
#define READSIZE 1024
#define IMG_FILENAME "Image"
const unsigned long brkptr = 0x00600000;
const unsigned long boot_start = 0x00100000;

void _start() {
    int imgfd, i, byte_read;
    char *ptr = (char *)boot_start;

        write(STDOUT_FILENO, pbmsg, sizeof(pbmsg));


    /* get more space to copy Image to ROM */
    brk(brkptr);

    imgfd = open("Image", O_RDONLY, 0));

    /* copy Image contents to ROM */
    i = 0;
    while (1) {
         byte_read = read(imgfd, ptr+i, READSIZE);
          if (byte_read < READSIZE) break;
    }
    /* jump to boot loader */
    ((void (*)())boot_start)();
}
```

System calls

Memory layout

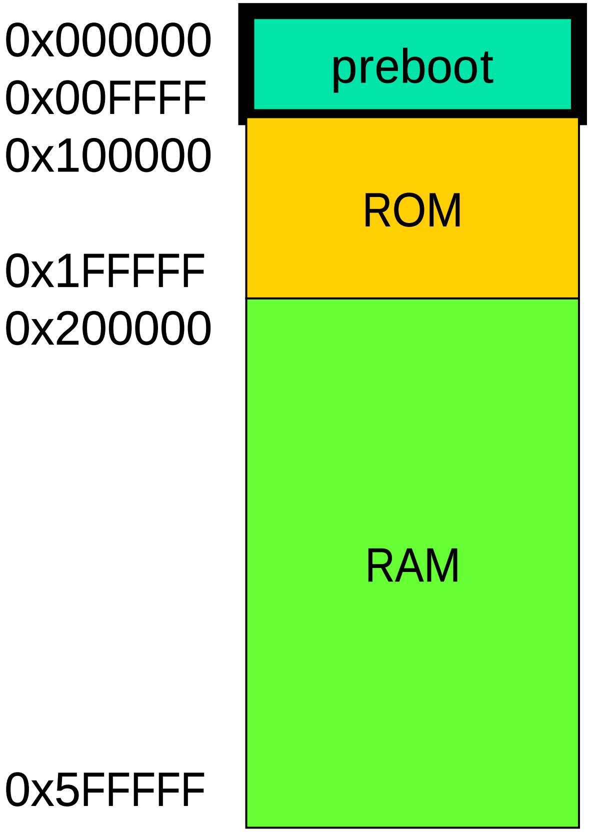


Makefile related to preboot

```
# pre-boot loader address map
PREBOOT_TEXT = 0x00000000  1
PREBOOT_DATA = 0x00002000
PREBOOT_LDFLAGS = -Ttext $(PREBOOT_TEXT) \
                  -Tdata $(PREBOOT_DATA)

2 %.o: %.c
          $(CC) $(CFLAGS) -c $< 3

# preboot is in ELF format. run it to load Image
preboot: preboot.o
          $(LD) $(PREBOOT_LDFLAGS) -o $@ $<
                                       4
```

1. Tell linker the runtime start address of text and data sections
2. Generic pattern rules to make a .o file from a .c
   ex: preboot.o -> preboot.c
3. The name of first prerequisite. That is %.c
4. The filename of the target of the rule

Memory layout



Let’s check preboot memory layout - objdump & readelf

objdump

```
start address 0x00000000

Program Header:
    LOAD off    0x00001000 1vaddr 0x00000000 paddr 0x00000000 align 2**12
         filesz 0x00000487 memsz 0x00000487 flags r-x
    LOAD off    0x00002000 vaddr 0x00002000 paddr 0x00002000 align 2**12
         filesz 0x00000000 memsz 0x00000000 flags rw-

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000322  00000000  00000000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  00002000  00002000  00002000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .rodata       00000147  00000340  00000340  00001340  2**5
                  CONTENTS,2ALLOC, LOAD, READONLY, DATA
  3 .bss          00000000  00002000 3 00002000 00002000  2**2
                  ALLOC
```

1 = 2 + 3

readelf

```
 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata
   01
```

Preboot memory layout - verify from procfs

- /proc/<pid>/maps
  - This shows runtime memory map

```
>cat /proc/3167/maps
00000000-00001000 r-xp 00001000 08:02 303766     /home/josephl/embed_example/boot/preboot
00002000-00600000 rwxp 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

brk()

The design of the boot loader

- Typical boot loader does:
  - Initialize CPU DRAM register
  - Setup stack register
  - Copy .data from ROM to RAM and zero-init .bss on RAM
  - Copy kernel to RAM
- The only difference here
  - No DRAM register init!
- NOTE
  - bootloader executes on the ROM!
  - If there is no global or static variable => we don’t need to copy .data or zero-init .bss
  - Here, copy of .data and zero-init of .bss can be written in C!

Simplified boot loader - head.s and boot.c

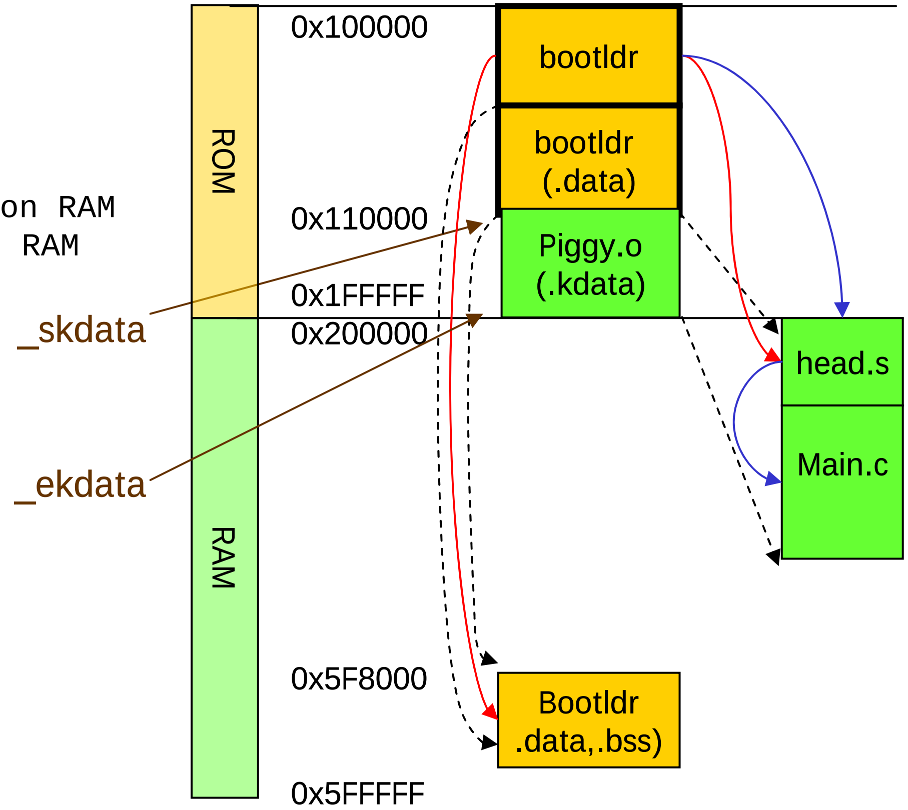
head.s

```
.text
     .globl startup_32

startup_32:
     movl $0x600000,%esp # setup stack pointer
     call clear_bss      # zero-init bss section on RAM
     call init_data      # copy .data from ROM to RAM
     call start_boot
```

defined in piggy.lds

boot.c

```
extern char _skdata[], _ekdata[];
const unsigned long kernel_start = 0x00200000;

void start_boot() {
     int i, copylen;

     /* copy kernel to RAM */
     copylen = _ekdata - _skdata;
     for (i=0; i < copylen; i++)
          *(char *)(kernel_start+i) = _skdata[i];

     /* jump to kernel */
     write(STDOUT_FILENO, bootjump, sizeof(bootjump));
     ((void (*)())kernel_start)();
}
```

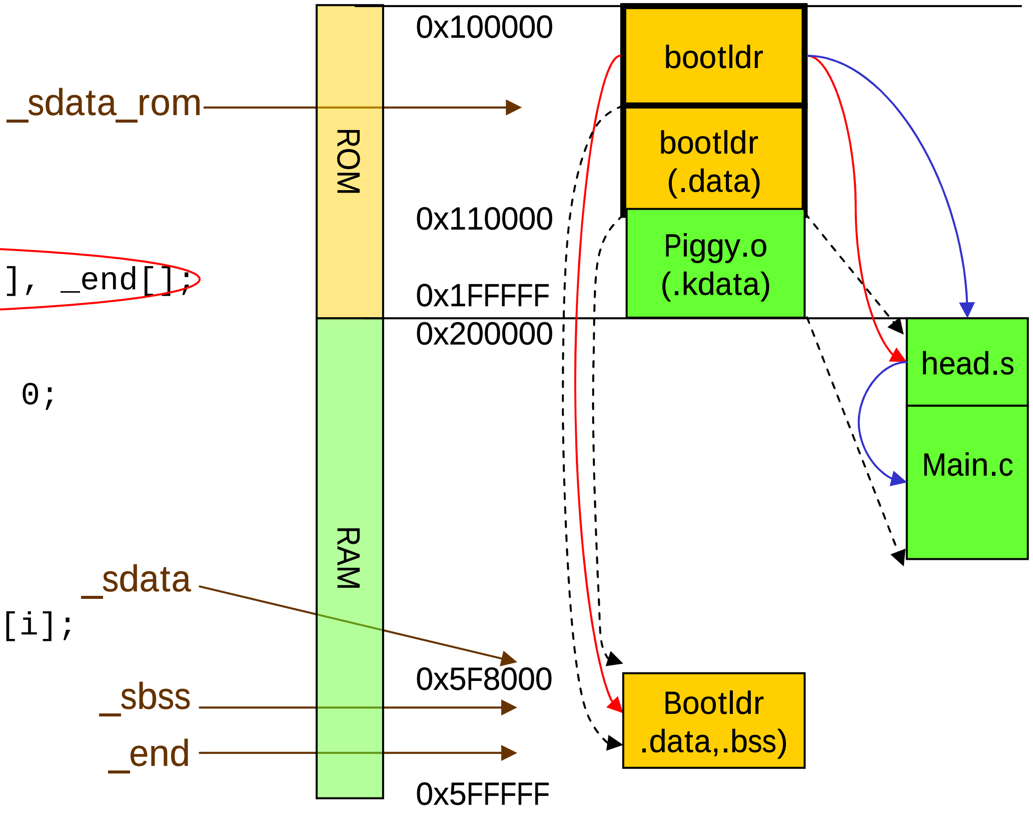


Simplified boot loader code - boot.c

Those ***symbols*** are defined in boot.lds

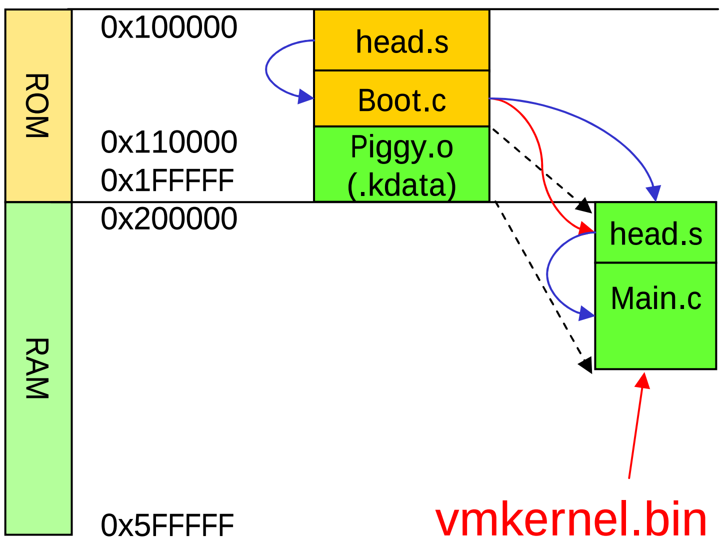
boot.c

```
extern char _sdata_rom[], _sdata[], _sbss[], _end[];
void clear_bss() {
          int i, len = _end - _sbss;
          for (i=0; i<len; i++) _sbss[i] = 0;
}
void init_data() {
          int i, len = _sbss - _sdata;

          for (i=0; i<len; i++)
                    _sdata[i] = _sdata_rom[i];
}
```



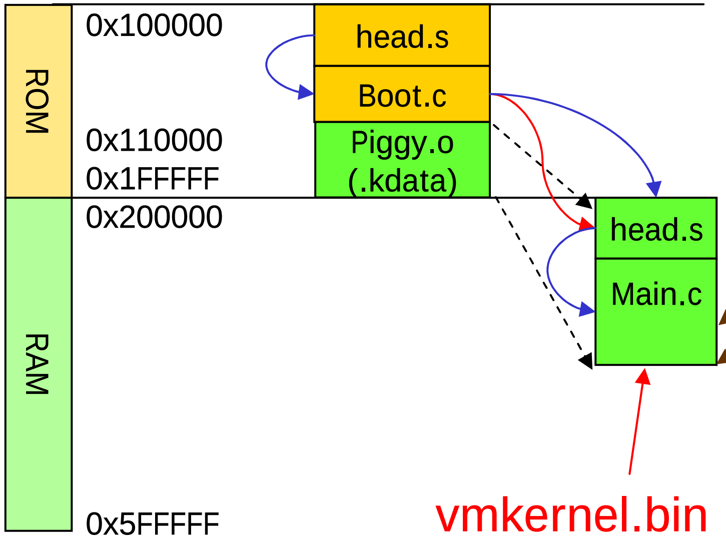
The design of the kernel

- Loading the kernel
  - The actual kernel got loaded is in binary format and can be run directly once it is copied to the RAM.
  - vmkernel.bin is the runtime image of the kernel with .text and .data sections ready!
  - Kernel needs to initialize .bss and stack pointer by itself.



Simplified kernel code - head.s, main.c

- stack area is inside the .bss section



head.s

```
.text
          .globl startup_32
startup_32:
          movl stack_start,%esp # setup stack pointer
          call start_kernel
```

main.c

```
#define STACK_SIZE 8192
char stack_space[STACK_SIZE]; /* in .bss section */
char *stack_start = &stack_space[STACK_SIZE];
extern char __bss_start[];
extern char _end[];

void clear_bss() {
          int i, len = _end - __bss_start;
          for (i=0; i<len; i++)
                    __bss_start[i] = 0;
}

start_kernel() {
          clear_bss();
          for(;;);
}
```

clear_bss() in start_kernel() makes start_kernel() unable to return, but this is not an issue since kernel never returns.

Building the kernel
- vmkernel.lds, Makefile

- Output object format: ELF

Makefile

```
LDFLAGS = -T vmkernel.lds   1

all: $(SYSTEM)   2

%.o: %.S
        $(CC) $(CFLAGS) -c $<
%.o: %.c
        $(CC) $(CFLAGS) -c $<
$(SYSTEM): head.o main.o
        $(LD) $(LDFLAGS) -o $@ $^
```

vmkernel.lds

```
3  ENTRY(startup_32)
   SECTIONS {
       . = 0x00200000;  4
       .text : { *(.text) }
       .rodata : { *(.rodata) }
       .data : { *(.data) }
       .bss : {
           __bss_start = .;
           *(.bss)
       }
       _end = .;
   }
```

1. Specify a linker script to be used
2. The filename of kernel in ELF format
3. Entry point of the program
4. Start address of the kernel

Building the Image file - Makefile

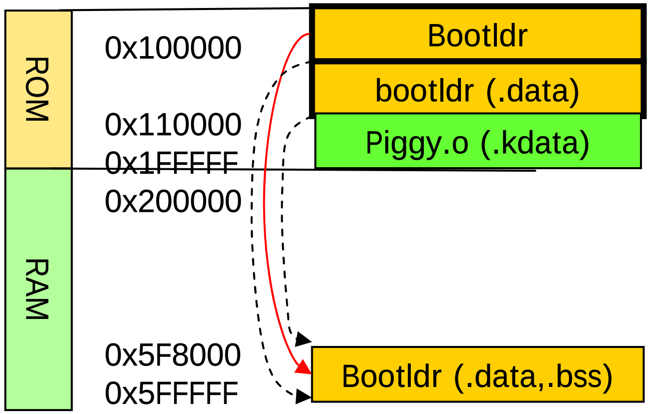
- Output object format: binary

```
                                  1
IMAGE_LDFLAGS = -T boot.lds

# generic pattern rules to compile a .c to a .o
%.o: %.c
          $(CC) $(CFLAGS) -c $<
%.o: %.S
          $(CC) $(CFLAGS) -c $<

# kernel must be in binary format since ELF loader is not available
piggy.o: $(SYSTEM)  2
                     3
          $(OBJCOPY) -O binary $(SYSTEM) vmkernel.bin                   1
          $(LD) -o $@ -r --format binary --oformat elf32-i386 vmkernel.bin -T piggy.lds
          rm -f vmkernel.bin           4             5

Image: head.o boot.o piggy.o
          $(LD) $(IMAGE_LDFLAGS) -o $@.elf $^
          $(OBJCOPY) -O binary $@.elf $@   6
```

1. Specify a linker script to be used
2. The filename of kernel in ELF format
3. Generate a relocateable output
4. The format of input object (vmkernel.bin)
5. The format of output object (piggy.o)
6. Make a binary object from a ELF one

The design of Image file - piggy.lds, boot.lds

- Image.elf memory map:



section name	Image.elf File offset	LMA	VMA	size
.text	0x00001000	0x00100000	0x00100000	0x253
.rodata	0x00001260	0x00100260	0x00100260	0x10F
.data	0x00002000	0x0010036F	0x005F8000	0x050
.bss	N / A	N / A	0x005F8050	0x004
.kdata	0x00003000	0x00110000	0x00110000	0x1D0

piggy.lds

```
SECTIONS
{
  .kdata : {
      _skdata = .;
      *(.data)
      _ekdata = .;
  }
}
```

boot.lds

```
ENTRY(startup_32)
SECTIONS {
  . = 0x00100000;
  .text : {*(.text) }
  .rodata : { *(.rodata) }
  _sdata_rom = .;
  . = 0x00110000;
  .kdata : { *(.kdata) }
  . = 0x005F8000;
  _sdata = .;
  .data : AT(_sdata_rom) { *(.data) }

  _sbss = .;
  .bss : { *(.bss) }
  _end = .;
}
```

The design of Image file - objcopy

- objcopy consults the **LMA address** of each section in the input object when making a binary object. It reorders sections by their LMA addresses in ascending order and copy those sections in that arranged order to the output object, starting from the first LMA address. If there is any gap between two sections, it fill the gap with zeros.
- 'AT' keyword moves .data section from 0x5f8000 to the space between .rodata and .kdata sections. This largely reduces the size of the Image file from ~5MBytes to ~64KBytes.

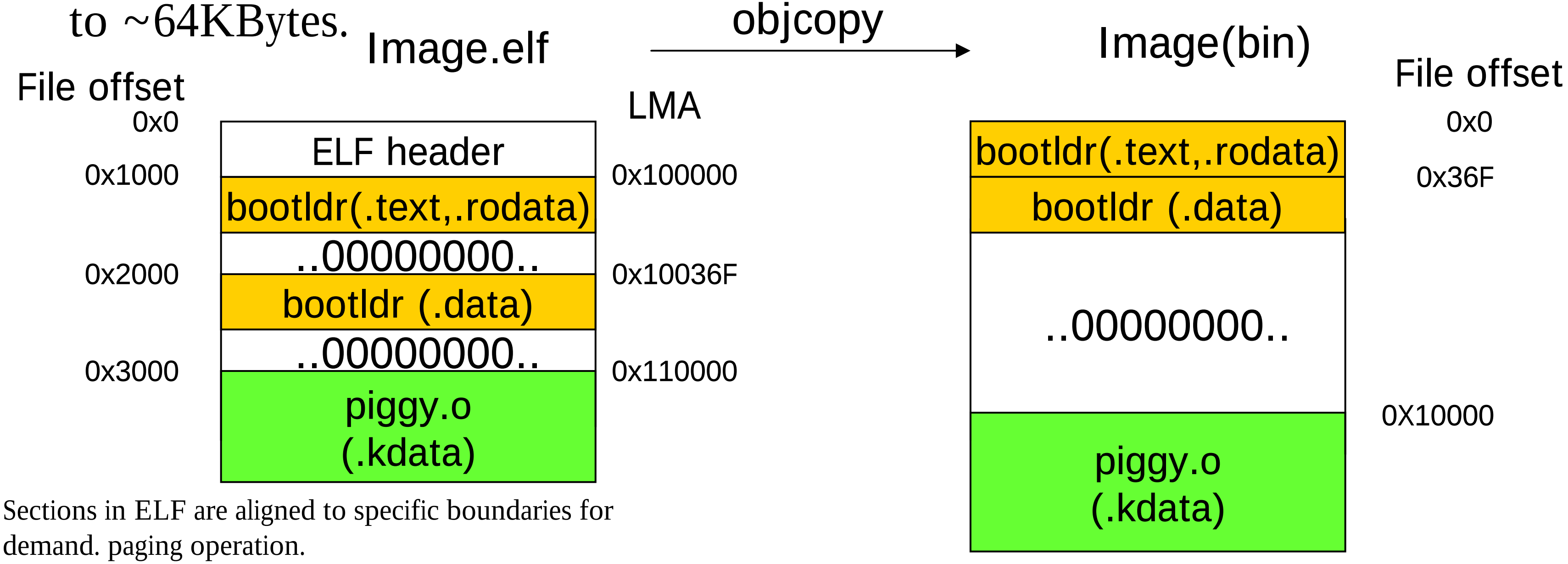


Image.elf

objcopy

Image(bin)

File offset

LMA

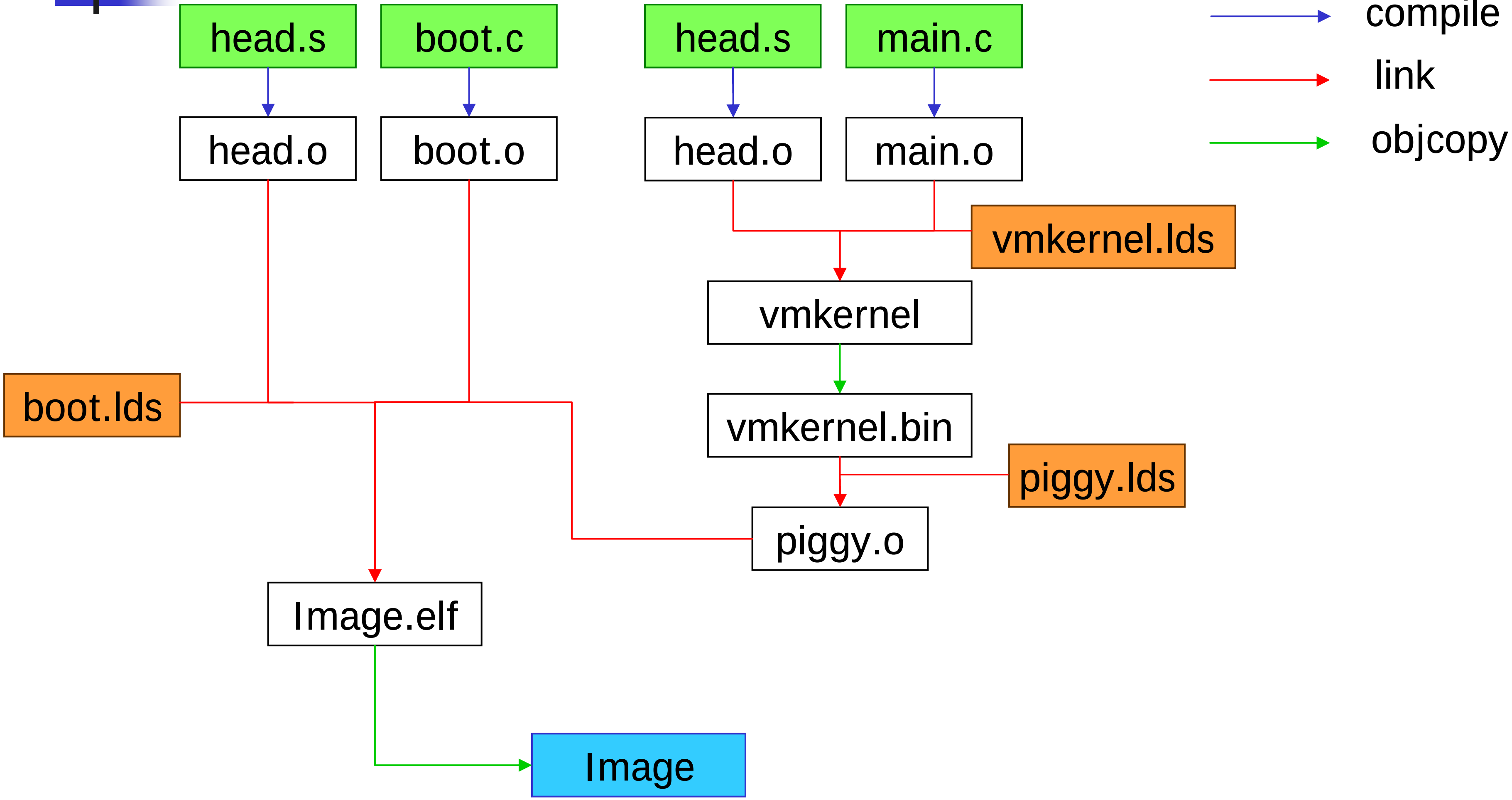
File offset

File offset	Image.elf	LMA
0x0	ELF header	
0x1000	bootldr(.text,.rodata)	0x100000
	..00000000..	
0x2000	bootldr (.data)	0x10036F
	..00000000..	
0x3000	piggy.o (.kdata)	0x110000

Image(bin)	File offset
bootldr(.text,.rodata)	0x0
bootldr (.data)	0x36F
..00000000..	
piggy.o (.kdata)	0X10000

Sections in ELF are aligned to specific boundaries for demand. paging operation.

The complete picture of building the 'Image' file



Use objdump and gdb to verify the design

- objdump
  - Verify Image.elf section header and symbol table
  - Disassemble Image.elf
- gdb
  - dump runtime memory contents to a file
    - Use ‘hexdump –C’ to compare the file contents with the corresponding section in the Image.elf

Reference

- John R. Levine, Linkers and Loaders, Morgan Kaufmann, 2000
- Using ld, Free Software Foundation, 2000
- Linux 2.4 kernel source