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
- *.h
- *.s
- *.o
- *.exe
- crt0.s
- ld.script
- *.lib
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
  - Default linker script: run ld --verbose
ELF format

- What we load is partially defined by ELF
- Executable and Linkable Format
- 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
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

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

<table>
<thead>
<tr>
<th></th>
<th>Program header</th>
<th>Section header</th>
</tr>
</thead>
<tbody>
<tr>
<td>relocatable</td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>executable</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Shared object</td>
<td>V</td>
<td>V</td>
</tr>
</tbody>
</table>

- ELF header
- Program header table
- ELF header
- Program header table
- Section header table
- Section header table
- sections
  - (optional, ignored)
- segments
  - (optional, ignored)
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 phdrcnt;  // size of an entry in program header
    short phdrcnt;  // number of entries in program header or 0
    short shdrcnt;  // size of an entry in section header
    short shdrcnt;  // number of entries in section header or 0
    short strsec;  // section number that contains section name strings
} Elf32_Ehdr;
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;
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:

<table>
<thead>
<tr>
<th>Idx</th>
<th>Name</th>
<th>Size</th>
<th>VMA</th>
<th>LMA</th>
<th>File off</th>
<th>Algn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.text</td>
<td>00000130</td>
<td>00200000</td>
<td>00200000</td>
<td>00001000</td>
<td>2**2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CONTENTS, ALLOC, LOAD, READONLY, CODE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>.rodata</td>
<td>00000049</td>
<td>00200140</td>
<td>00200140</td>
<td>00001140</td>
<td>2**5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CONTENTS, ALLOC, LOAD, READONLY, DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>.data</td>
<td>00000044</td>
<td>0020018c</td>
<td>0020018c</td>
<td>0000118c</td>
<td>2**2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CONTENTS, ALLOC, LOAD, DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>.bss</td>
<td>00002020</td>
<td>002001e0</td>
<td>002001e0</td>
<td>000011e0</td>
<td>2**5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALLOC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>.comment</td>
<td>00000033</td>
<td>00000000</td>
<td>00000000</td>
<td>000011e0</td>
<td>2**0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CONTENTS, READONLY</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
$ 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-init by the operating system
Where do we load?

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

  ![Runtime Image of a executable](image)

  - Defined by the OS
  - Defined by linker
# Where do variables go?

<table>
<thead>
<tr>
<th></th>
<th>.text</th>
<th>.rodata</th>
<th>.data</th>
<th>.bss</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>global</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>static</td>
<td>initialized</td>
<td></td>
<td>v</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>non-initialized</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>non-static</td>
<td>initialized</td>
<td></td>
<td>v</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>non-initialized</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>const</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v</td>
</tr>
<tr>
<td>local</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>static</td>
<td>initialized</td>
<td></td>
<td>v</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>non-initialized</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>non-static</td>
<td>initialized</td>
<td></td>
<td>v</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>non-initialized</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>const</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v</td>
</tr>
<tr>
<td>Immediate value</td>
<td></td>
<td></td>
<td>v</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
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 😊
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

Linux ELF loader

0x000000
0x00FFFF
0x100000
0x110000
0x1FFFFF
0x200000
0x5F8000
0x5FFFFF

preboot
RAM

bootldr (.data)
head.s
bootldr boot.c
piggy.o (.kdata)

bootldr (.data, .bss)

head.s
main.c
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 0x00000 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
```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 (*)(void))boot_start)();
}
```
Makefile related to preboot

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

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

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

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
Let's check preboot memory layout
- objdump & readelf

**objdump**

start address 0x00000000

Program Header:

<table>
<thead>
<tr>
<th>Type</th>
<th>Offset</th>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Alignment</th>
<th>Size</th>
<th>File Size</th>
<th>Memory Size</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>00001000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>2**12</td>
<td>0x00000487</td>
<td>0x00000487</td>
<td>r-x</td>
<td></td>
</tr>
<tr>
<td>LOAD</td>
<td>00002000</td>
<td>0x00002000</td>
<td>0x00002000</td>
<td>2**12</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>rw-</td>
<td></td>
</tr>
</tbody>
</table>

Sections:

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Size</th>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>File Offset</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.text</td>
<td>322B</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x0001000</td>
<td>2**2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CONTENTS, ALLOC, LOAD, READONLY, CODE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>.data</td>
<td>0B</td>
<td>0x00002000</td>
<td>0x00002000</td>
<td>0x0002000</td>
<td>2**2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CONTENTS, ALLOC, LOAD, DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>.rodata</td>
<td>147B</td>
<td>0x000340</td>
<td>0x0000340</td>
<td>0x0001340</td>
<td>2**5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CONTENTS, ALLOC, LOAD, READONLY, DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>.bss</td>
<td>0B</td>
<td>0x0002000</td>
<td>0x0002000</td>
<td>0x0002000</td>
<td>2**2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALLOC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**readelf**

Section to Segment mapping:

Segment Sections...

00 .text .rodata
01

1 = 2 + 3
Preboot memory layout
- verify from procfs

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

```bash
> 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
bffffe000-c0000000 rwxp ffffff000 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

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

```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 = __bss_start - _end;
    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

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. 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

```
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)
  $(OBJCOPY) -O binary $(SYSTEM) vmkernel.bin
  $(LD) -o $@ -r --format binary --oformat elf32-i386 vmkernel.bin -T piggy.lds
  rm -f vmkernel.bin

Image: head.o boot.o piggy.o
  $(LD) $(IMAGE_LDFLAGS) -o $@.elf $^
  $(OBJCOPY) -O binary $@.elf $@
```
The design of Image file
- piggy.lds, boot.lds

Image.elf memory map:

<table>
<thead>
<tr>
<th>section</th>
<th>Image.elf File offset</th>
<th>LMA</th>
<th>VMA</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>0x00001000</td>
<td>0x00100000</td>
<td>0x00100000</td>
<td>0x253</td>
</tr>
<tr>
<td>.rodata</td>
<td>0x00001260</td>
<td>0x00100260</td>
<td>0x00100260</td>
<td>0x10F</td>
</tr>
<tr>
<td>.data</td>
<td>0x00002000</td>
<td>0x0010036F</td>
<td>0x005F8000</td>
<td>0x050</td>
</tr>
<tr>
<td>.bss</td>
<td>N/A</td>
<td>N/A</td>
<td>0x005F8050</td>
<td>0x004</td>
</tr>
<tr>
<td>.kdata</td>
<td>0x00003000</td>
<td>0x00110000</td>
<td>0x00110000</td>
<td>0x1D0</td>
</tr>
</tbody>
</table>

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.

Sections in ELF are aligned to specific boundaries for demand paging operation.
The complete picture of building the ‘Image’ file

Compilation:
- `head.s` → `head.o`
- `boot.c` → `boot.o`
- `main.c` → `main.o`
- `head.s` → `head.o`
- `main.c` → `main.o`

Linking:
- `head.o` + `boot.o` + `main.o` + `vmkernel.lds` → `vmkernel`
- `vmkernel` → `vmkernel.bin`
- `piggy.o` + `piggy.lds` → `piggy.o`

Packaging:
- `Image.elf`
- `Image`
- `objcopy`
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