Lab 1 HOWTO - A Step-By-Step Walkthrough from Scratch

Overview

There are three tasks in Lab 1:

1. **Design and implement a firmware on MSIM to load and execute the code from MBR**.
2. **Design and implement a bootloader on MSIM to load and execute the kernel residing on the second MBR partition, regardless of the booting status of that partition.**
3. **Design and implement a kernel to initialize hardware and print a hello world message, then do whatever you like.**

The requirement is *to design a scalable, extensible framework for kernel to allow convenient support for heterogeneous platform, hardware, and/or architecture.* This is really a software engineering problem existing in all software projects.

Environment Prerequisites

1. MIPS toolchain.
    - Grab one from Sourcery Codebench Lite if you don't have one in official repository.
2. MSIM simulator.

This lab *does not require Linux*, you can do experiments under Cygwin or MinGW on Windows, as long as you have a working toolchain under Cygwin (which is supported by Sourcery Codebench Lite, consult the installation instructions for details). MSIM simulator is runnable under Cygwin and MinGW.

A Linux is always preferred, though.

Knowledge Prerequisites

This lab assumes that you have basic knowledge of MIPS assembly (you could get familiar with it in Lab 0), and basic C programming.

Also, it assumes that you're familiar with using `man(1)`, and sometimes `info(1)`.

- Make sure you have installed POSIX man pages if you're using Cygwin.

This lab will teach you from writing assembly, C code, to writing Makefiles, linker scripts, up to framework designs.

Shipped files

Make sure you have the following files:

- `headers` directory
  - `addrspace.h`
  - `asm.h`
  - `cp0regdef.h`
  - `elf.h`
  - `io.h`
  - `irq.h`
  - `mipsregs.h`
  - `regdef.h`
  - `stdarg.h`
  - `stddef.h`
  - `stdio.h`
  - `string.h`
  - `sys/types.h`
- `src` directory
  - `memcpy.c`
  - `memset.c`
  - `snprintf.c`

Note that we'll (almost surely) move the headers and sources to elsewhere.

Examples and exercises

This document contains different examples along with step-by-step exercises. It's highly recommended to stick to the exercises to complete the lab.

What firmware should do

The firmware, or BIOS as is called on x86 machines, contains the first piece of code the machine runs after startup.

On x86 machines, the BIOS does the following:

1. Initializes essential hardware for loading MBR, printing message, accepting keyboard input, etc.
2. Expose a series of standardized interfaces for disk access, displaying, keyboard input, etc., via software interrupts.

3. Loads the first sector on the main external storage, a.k.a. MBR, and execute the code there.

Unfortunately, there's no standardized interfaces as in x86 machines. So we'll probably implement BIOS and MBR however we like.

Firmware code design

Writing everything in assembly is *awful*. If you're writing code for `printf(3)` in assembly, you'd probably find yourself ~~dead~~ mad. Therefore, we'd like to:

1. Bootstrap the program in assembly, setting up necessary environment, and
2. Leave everything but the most low-level ones to C.

MIPS register guide in hardware

MIPS treats all except 2 of the registers equally on hardware. The exceptions are:

1. The 0th register `$0`, or more commonly known, the *wired zero register* (`zero`), which always returns 0 when read, and always ignores writes.
2. The 31st register `$31`, or more commonly known, the *return address register* (`ra`).

IA32 and IA64 architectures define separate instructions for stack operations, explicitly reserving two registers (`ESP` and `EBP`) for that purpose. However, RISC architectures such as ARM and MIPS don't assume a stack.

The return address register

`ra` stores the return address when a `BAL` instruction or `JAL` instruction is executed. That means, you can write assembly *functions* and assembly *function calls*, like the following assembly code:

```
        # Code snippet 1
        jal     func            # (1)
        addu    s0, s1          # (2)
        ...
func:   ...                     # function body
        jr      ra              # (3)
```

The `JAL` instruction at (1) makes an assembly function call, storing the return address, that is, the address of (2) into `ra`, and jumps to the label `func`. The function `func` then proceeds, and returns by the jump-to-register instruction `jr ra` at (3). The program then continues execution from (2). The overall effect is that (1) made a function call at `func`.

Problem: nested assembly calls However, consider the following code snippet:

```
        # Code snippet 2
        jal     func            # (1)
        addu    s0, s1          # (2)
        ...
func:   ...
        jal     func2           # (3)
        subu    s3, s4          # (4)
        ...
        jr      ra              # (5)
        ...
func2:  ...
        jr      ra              # (6)
```

We can trace the code above manually: when executing (1), the processor stores the return address, that is, the address of (2), to `ra`, and jumps to `func`. After some execution, the processor finds (3), stores the address of (4) to `ra`, and jumps to `func2`.

The problem is, if the return address of `func`, which is the address of (2) and the value of `ra` before the execution of (3), is not preserved, then `ra` will be overwritten by instruction (3). After returning from `func2`, and the function `func` finishes its code, it'll never be able to return to instruction (2) to continue its work.

The register resource is usually scarce, so the return address register is usually preserved on a *stack*, which is a part of the main memory. The *stack pointer register*, `sp`, points to the non-empty stack top, as a convention of MIPS ABI. Moreover, the stack grows *downward*: allocation on stack decreases `sp`, while deallocation increases `sp`.

Assuming `func` in Code Snippet 2 has nothing to preserve other than the return address register, and `func2` won't make any further calls, we would rewrite the code, including allocation from stack during function entry, and deallocation on stack during function exit:

```
        # Code snippet 3
        jal     func            # (1)
```

```
        addu    s0, s1          # (2)
        ...
func:   addiu   sp, -4          # allocate 4 bytes from stack
        sw      ra, (sp)        # preserve ra onto stack
        ...
        jal     func2           # (3)
        subu    s3, s4          # (4)
        ...
        lw      ra, (sp)        # restore ra from stack
        addiu   sp, 4           # return the 4-byte space back to stack
        jr      ra              # (5)
        ...
func2:  ...
        jr      ra              # (6)
```

We can see that managing the stack by handwritten assembly needs extra care, which is one of the reasons we want to write most code in C, since the compiler would manage the stack for us.

Writing assembly functions

`asm.h` in the `headers` directory provides three macros for writing assembly functions.

- `LEAF(symbol)` is to define a leaf assembly function, which does not make calls, hence the return address register `ra` is not saved on stack since its value probably won't change.
- `NESTED(symbol, framesize, rareg)` is to define an assembly function which will make further calls. As nested calls semantically require a stack, the allocation size for preserving return address as well as other local variables is specified at `framesize` parameter. The return address register `rareg` parameter is usually `ra`.
- Both kinds of function should be ended with `END(symbol)`.

By using these macros, you can expose assembler functions to C code.

A rewrite of Code Snippet 1 with the three macros, assuming `func` does not make further calls:

```
        # Code Snippet 4
        jal     func            # (1)
        addu    s0, s1          # (2)
        ...
LEAF(func)
```

```
        ...                     # function body
        jr      ra              # (3)
END(func)
```

A rewrite of Code Snippet 3, under the same assumption:

```
        # Code Snippet 4
        jal     func            # (1)
        addu    s0, s1          # (2)
        ...
NESTED(func, 4, ra)             # tell the assembler that we're allocating
                                # 4-bytes from stack to preserve ra
        addiu   sp, -4          # allocate 4 bytes from stack
        sw      ra, (sp)        # preserve ra onto stack
        ...
        jal     func2           # (3)
        subu    s3, s4          # (4)
        ...
        lw      ra, (sp)        # restore ra from stack
        addiu   sp, 4           # return the 4-byte space back to stack
        jr      ra              # (5)
END(func)
        ...
LEAF(func2)
        ...
        jr      ra
END(func2)
```

Side note: source code organization

We should keep the goal of scalability in mind: we're designing a codebase which should support heterogeneous architecture and hardware.

The using of `LEAF`, `NESTED` and `END`, as well as other things in `asm.h` all indicates that this is a MIPS-specific header file. Therefore, we shall separate those headers with other more generic, less hardware-dependent headers.

A possible solution is to make a directory called

```
./include
```

to store all the headers. Then, make a directory

```
./include/arch/mips/asm
```

to store all MIPS-specific headers. The source files as well as C files should include the MIPS-specific headers like

```
#include <asm/asm.h>
```

The `asm/` prefix suggests that this is an architecture-specific header.

Move `asm.h` to the MIPS-specific header directory.

Such organization allows later developers to extend supported architectures by adding the specific headers to somewhere like

```
./include/arch/armv7a-le/asm
./include/arch/i386/asm
```

Source Management Exercise Guess which headers in the `headers` directory are MIPS-specific, and which are hardware-independent headers. Move the MIPS-specific to the MIPS-specific header directory, and the generic headers to `include` directory.

Bootstrapping Environment for C code

The first thing is to write a function for *reset*, which is, after startup.

Clearing `ERL`

First, we should disable the `ERL` bit in `STATUS` register, to indicate that we're executing normally:

```
/* firmware-entry.S */
#include <asm/asm.h>

LEAF(__reset)
        mfc0    a0, CP0_STATUS
        ori     a0, ST_ERL
        xori    a0, ST_ERL
        mtc0    a0, CP0_STATUS
END(__reset)
```

The names for general purpose registers are in `regdef.h` in `headers` directory, which you should have moved to the MIPS-specific header directory by now as it's MIPS-specific.

The names for CP0 registers as well as the macros for their bitfields are in `cp0regdef.h` in `headers` directory, which is yet another MIPS-specific header.

Therefore, we should include the two headers:

```
/* firmware-entry.S */
#include <asm/asm.h>
#include <asm/regdef.h>
#include <asm/cp0regdef.h>

LEAF(__reset)
        mfc0    a0, CP0_STATUS
        ori     a0, ST_ERL
        xori    a0, ST_ERL
        mtc0    a0, CP0_STATUS
END(__reset)
```

Initializing stack pointer `sp`

The second thing to do is to initialize the stack pointer register `sp`. Apparently it should point to somewhere in RAM. Theoretically, anywhere is fine. However, make sure that

- Loading MBR and kernel won't interfere with the stack.
- The address stored in `sp` is *virtual* address. CPU always operates on virtual addresses, while devices usually operate on *physical* ones.
- The corresponding physical address of `sp` is indeed inside the RAM.

```
/* firmware-entry.S */
#include <asm/asm.h>
#include <asm/regdef.h>
#include <asm/cp0regdef.h>

LEAF(__reset)
        mfc0    a0, CP0_STATUS
        ori     a0, ST_ERL
        xori    a0, ST_ERL
        mtc0    a0, CP0_STATUS
        li      sp, 0x8f000000  /* assuming RAM size > 0x0f000000 */
END(__reset)
```

Paper Exercise Think of the reason why CPU should operate on virtual adresses, and why devices usually operate on physical addresses.

Pseudo-instructions, reordered branch-delays, and hardware instructions GNU assembler provide some *pseudo-instructions* for the most common operations, easing the assembly development. See this Wikipedia article or See MIPS Run, Chapter 8 for a list of instructions and pseudo-instructions, and the MIPS Architecture Volume II for a list of hardware instructions, thereby discriminating the pseudo-instructions.

GNU assembler also automatically resolves branch delays by inserting `NOP`s and rearranging instructions. So one could write assembly sources as if there's no branch delays. However, one should always take branch delays into account when computing instruction addresses, disassembling, or doing anything other than *writing assemblies yourself*.

Jump to C code

Supposing that the C entry function for firmware is `main(void)` which does not (or should not return) a value, a simple `JAL` instruction would suffice. As we're executing a `JAL` instruction, it's recommended to change the `LEAF` macro to `NESTED` macro.

Note that you don't need any additional header for referencing the function `main`, as opposed to C; the assembler would automatically recognize it as a function call, leaving the actual symbol resolution work to the linker `ld(1)`:

```
/* firmware-entry.S */
#include <asm/asm.h>
#include <asm/regdef.h>
#include <asm/cp0regdef.h>

NESTED(__reset, 0, ra)
        mfc0    a0, CP0_STATUS
        ori     a0, ST_ERL
        xori    a0, ST_ERL
        mtc0    a0, CP0_STATUS
        li      sp, 0x8f000000
        jal     main
END(__reset)
```

We can see that:

> *Assemblies can call a `void` C function without parameters by* `JAL`.

Ordinarily, we should preserve `ra` into memory before `JAL`. However, this is OK here because we already knew that `main` function here *would never return*, thereby negating the necessity of preserving `ra`.

Reserving spaces for exception handler

MIPS specification requires that, when `BEV` in `STATUS` register is set (which is the case after startup),

- TLB refill exception handler should be placed at `0xbfc00200`.
- Cache error handler should be placed at `0xbfc00300`.
- Generic exception handler should be placed at `0xbfc00380`.

If we don't reserve places for the exception handler, even if we won't do anything with them, the C code we'll write later would probably be put inside the exception handler entry addresses, and in case of an exception, the processor jumps to execute from *the middle of your C code*, thereby wreaking havoc.

However, manually flooding the code with `NOP` is certainly not an option, as it would make the code extremely ugly.

The `.align` assembler directive GNU assemblers provide a `.align` directive for conveniently arranging instructions by alignments.

Apparently `__reset` begins at virtual address `0xbfc00000`, the following code inserts an instruction at address `0xbfc00200`, which is the entry of TLB refill exception handler (passed to generic handler now since we're not dealing with memory management):

```
/* firmware-entry.S */
#include <asm/asm.h>
#include <asm/regdef.h>
#include <asm/cp0regdef.h>

NESTED(__reset, 0, ra)          /* 0xbfc00000 */
        mfc0    a0, CP0_STATUS
        ori     a0, ST_ERL
        xori    a0, ST_ERL
        mtc0    a0, CP0_STATUS
        li      sp, 0x8f000000
        jal     main
END(__reset)

        .align  9
NESTED(__tlbrefill, 0, ra)      /* 0xbfc00200 */
        jal     __generic_exception
END(__tlbrefill)
```

The `.align 9` directive instructs the assembler to put the next instructions at the address greater than the previous *hardware instruction*, with the lower 9 bits

cleared. Therefore, the instruction succeeding the `.align 9` directive resides at `0xbfc00200`.

Programming Exercise Write `__cacheerror` function and `__generic_exception` function, whose entry should match the corresponding virtual addresses.

Note that you should write a forever loop in `__generic_exception` to catch all exceptions ~~and die~~.

Trap: consecutive `.align` directives Consider the following code snippet:

```
/* firmware-entry.S */
#include <asm/asm.h>
#include <asm/regdef.h>
#include <asm/cp0regdef.h>

NESTED(__reset, 0, ra)          /* 0xbfc00000 */
        mfc0    a0, CP0_STATUS
        ori     a0, ST_ERL
        xori    a0, ST_ERL
        mtc0    a0, CP0_STATUS
        li      sp, 0x8f000000
        jal     main
END(__reset)

        .align  9
LEAF(__tlbrefill)               /* 0xbfc00200 */
        /* Note that this function is empty */
END(__tlbrefill)

        .align  8
LEAF(__cacheerror)              /* ??? */
        jal     __generic_exception
END(__cacheerror)
```

As we stated, `.align` is only a directive (hence not an instruction, either pseudo-ones or hardware ones). As there's no instruction between `.align 9` and `.align 8`, the entry of `__cacheerror` is still `0xbfc00200` instead of the intended `0xbfc00300`!

Hello world from BIOS in C

Finally we're turning to C.

The first thing for our C code is to implement a `main` function, to connect with the assembly, as well as printing the message:

```
/* firmware-main.c */

int main(void)
{
        kputs("Hello world from BIOS in C!\n");
        for (;;)
                /* nothing */;
}
```

Accessing printer from C

The problem is how to implement the printer driver in C. As device registers in MSIM are mapped to physical memory addresses, the question is now how to access these addresses.

The answer is to use *C pointers*, which holds virtual addresses. By assigning them certain addresses, we could easily associate the pointers with the device registers.

Assuming the output register for printer is mapped to physical address `0x1f000010`, we could write the following code:

```
/* firmware-main.c */

void kputs(const char *str)
{
        unsigned char *printer = (unsigned char *)phys_to_virt(0x1f000010);
        for (; *str != '\0'; ++str)
                *printer = *(unsigned char *)str;
}

int main(void)
{
        kputs("Hello world from BIOS in C!\n");
        for (;;)
                /* nothing */;
}
```

How to translate between physical and virtual address? One problem unresolved is that how to translate from physical address to virtual ones on MIPS, i.e. how to implement `phys_to_virt` function above.

Before that, we need to learn the convention of MIPS physical address layout. The layout is rather simple:

- Physical addresses from `0x00000000` to `0x0fffffff` correspond to RAM.
- Physical addresses from `0x10000000` to `0x1fffffff` correspond to hardware devices.
  - Specially, `0x1fc00000` is the start of firmware address space. The actual size of address space depends on manufacturing.
- The physical address layout above `0x20000000` is left entirely to the manufacturer.

Translating physical addresses lower than `0x20000000` is simple, depending on whether you want to, and be able to, employ caches.

- If caches are available and you want to use caches, adding `0x80000000` to the physical address yields the virtual address.
  - Ideal for accessing RAM, since RAM contents are usually fully controlled by the processor only.
  - Therefore, the virtual address indicating the start of RAM is `0x80000000`.
- If you're unable to use caches, or you don't want to use caches, then adding `0xa0000000` to the physical address yields the virtual address.
  - Hardware devices *MUST* be accessed in this manner. The reason is that on some MIPS processors, the consistency between CPU cache and hardware devices are not maintained by hardware.
  - MSIM simulates a transparent, perfect cache, so that external changes in hardware devices can be immediately pushed back into caches.
  - Loongson CPUs maintain consistency by hardware, so you don't need to worry about inconsistency, either.

Translating virtual addresses between `0x80000000` and `0xbfffffff` is therefore easy as well: just take the lower 29-bit and its done.

There's no simple method compared to above to deal with physical addresses higher than `0x20000000`. We'll deal with it in later labs. For now, physical address `0x00000000` to `0x1fffffff` is sufficient.

Paper exercise Consider, that we have an MSIM keyboard at physical address `0x1f000010`, and the CPU has a cache, whose consistency with the keyboard is not maintained by hardware, i.e. the changes at devices could not be reflected back to caches automatically.

Each time a key is pressed, the program executes the following function, in which the keyboard is accessed through caches (by adding `0x80000000` to the physical address to obtain the virtual address):

```
void keypress(void)
{
        volatile char key = *(volatile char *)0x9f000010;
        /* ... */
}
```

1. What's the value of `key` when `keypress()` is executed the first time, after key `a` is pressed?
2. What's the value of `key` when `keypress()` is executed the second time, after key `b` is pressed, if the cache still keeps the content of address `0x9f000010`?

Trap: optimization

GCC is an optimizing compiler. When compiling the code above, GCC would regard the loop in `kputs` *useless*, as it (seemingly) does no change to the outside world, thereby removing the loop as a whole.

To prevent this, we have to declare the `printer` pointer as `volatile`, which means that the value may change *from outside our code*, thus preventing the compiler to optimize our loop away:

```
/* firmware-main.c */

void kputs(const char *str)
{
        volatile unsigned char *printer = (volatile unsigned char *)0xbf000010;
        for (; *str != '\0'; ++str)
                *printer = *(unsigned char *)str;
}

int main(void)
{
        kputs("Hello world from BIOS in C!\n");
        for (;;)
                /* nothing */;
}
```

We're all set for a greeting BIOS. The problem is now how to compile these sources, how to link them together, and how to obtain the binary with everything unnecessary stripped away for loading into MSIM.

Programming Exercise

1. Implement a `kprintf` function, which is a cousin of `printf(3)`. more specifically, you should do the following to send the result to printer:
   - Output the result to be printed into a local buffer, whose size is `BUFSIZ` (defined in `stddef.h`, which is a generic header, and should be already moved to `include` directory accordingly). You should use the function `snprintf` declared in `stdio.h` (yet another generic header), and implemented in `snprintf.c`.
   - Send the result to printer via `kputs`.
2. Print the current hour, minute, and second via your `kprintf` implementation. You'll have to access the *real time clock* (RTC) driver provided by MSIM, whose behavior is described in the reference manual.
3. *(Optional)* Print the current year, month, and day via your `kprintf` implementation. You'll see why you'd prefer to work in C rather than assembly.

Your `kprintf` will be a starting point for BIOS and MBR *static* debugging.

Compiling with GCC

Assuming that you have latest GCC MIPS toolchain (you can get one at Sourcery Mentor, and be sure to have a 32-bit C library installed), you can compile the sources like:

```
mips-linux-gnu-gcc -EL -nostdinc -nostdlib -mabi=32 -mips32 -fno-pic -g \
        -mno-abicalls -I. -I<your-generic-header-directory> \
        -I<your-architecture-specific-header-directory-without-asm> \
        -c -o <filename>.o <filename>.[S|c]
```

Assuming that you strictly followed the steps above, you can compile both sources into object files by:

```
mips-linux-gnu-gcc -EL -nostdinc -nostdlib -mabi=32 -mips32 -fno-pic -g \
        -mno-abicalls -I. -I./include -I./include/arch/mips \
        -c -o firmware-entry.o firmware-entry.S

mips-linux-gnu-gcc -EL -nostdinc -nostdlib -mabi=32 -mips32 -fno-pic -g \
        -mno-abicalls -I. -I./include -I./include/arch/mips \
        -c -o firmware-main.o firmware-main.c
```

Note: you should also compile `snprintf.c` in the same way if you implemented date & time printing.

Brief explanations to compiler options

You can always refer to `gcc(1)`.

1. `-EL`: generate little-endian code, as MIPS can run in both big-endian and little-endian.
2. `-nostdinc`: do not search headers from standard include directory, which is usually `/usr/include`
3. `-nostdlib`: do not use routines from standard system libraries.
4. `-mabi=32`: choose MIPS o32 ABI. When playing with Loongson you'll have to change this option to `-mabi=64` for MIPS n64 ABI.
5. `-mips32`: use MIPS32 instruction set. You'll have to change this to `-mips64r2` when playing with Loongson.
6. `-fno-pic`: do not generate position independent code (which complicates everything when developing kernels)
7. `-g`: generate debugging information
8. `-mno-abicalls`: do not generate dynamic objects (which again complicates everything)
9. `-I`: search header files in the given directory.
10. `-c`: generate an object file, and do not run the linker.
11. `-o`: specifies the output file name.

Programming exercise Adapt the commands above to fit your work.

Writing Linker Scripts

The only thing remaining is how to direct the linker to combine the object files together to produce a single executable.

Technically, one can instruct the linker by providing options, but using the *linker scripts* is more flexible.

Basics

The linker script should start with three lines, indicating the output format, output architecture, and entry point.

```
/* firmware.ld */

/*
 * You can view a list of supported formats by looking for the "supported
 * targets" output by "mips-linux-gnu-ld --help"
 */
```

```
OUTPUT_FORMAT("elf32-tradlittlemips")
/* Apparently the architecture should be mips */
OUTPUT_ARCH("mips")
/*
 * Theoretically the entry point could be anything as long as 0xbfc00000
 * points to the reset entry.
 */
ENTRY(__reset)
```

`SECTIONS` Command

The next thing we should do is to specify the arrangement of each sections from each object files.

A *section* is basically a binary block for a specific purpose. The most important sections generated by GNU GCC are:

`.text` Contains the code.

`.data` Contains the global data and variables with initial values other than 0.

`.bss` Contains the global variables with no/zero initial values. This section doesn't take physical space in object file as there's no need to reserve space for them anyway.

`.rodata` Contains the read-only global data, e.g. constant strings.

Preliminary `SECTIONS` command syntax Here's a sample linker script containing an example of `SECTIONS` command:

```
/* firmware.ld */
OUTPUT_FORMAT("elf32-tradlittlemips")
OUTPUT_ARCH("mips")
ENTRY(__reset)

SECTIONS {
        .text : {
                *(.text);
        }
        .rodata : {
                *(.rodata);
        }
```

```
        .data : {
                *(.data);
        }
        .bss : {
                *(.bss);
        }
}
```

The linker script tells the linker to put `.text` section first, containing `.text` sections from all object files. Then, the linker should put `.rodata`, which contains `.rodata` sections from all the object files, after `.text` section. Likewise, `.data` follows `.rodata`, and `.bss` follows `.data`.

Each section in `SECTIONS` command is specified with a list of sections from the object files in the following format:

```
<filename-pattern>(<section-name>)
```

The pattern can match multiple (or all) files given to `ld(1)`, which will arrange the sections in the same order as the file parameters given to `ld(1)`.

Manually specify ordering Of course, you can explicitly specify the order of object files in each section, like

```
        .text : {
                firmware-entry.o(.text);
                firmware-main.o(.text);
        }
```

Or combine wildcards with specific names:

```
        .text : {
                firmware-entry.o(.text);        /* firmware-entry.o first */
                firmware*(.text);       /* others starting with "firmware" next */
                *(.text);               /* put the rest then */
        }
```

You can also mix different kinds of sections if you like:

```
        .text : {
                *(.text);
                *(.rodata);
        }
```

You can even define your own sections:

```
.firmware.entry.text : {
        firmware-entry.o(.text);
}
```

Using linker scripts

Linking object files could be done by

```
mips-linux-gnu-ld -T <your-linker-script> -o <your-program-file> \
        <your-object-file-list>
```

Assuming that your firmware linker script is `firmware.ld`, the command becomes (moreover, assuming that you're strictly following the guide above, and you'd like to output the final program to file `firmware`):

```
mips-linux-gnu-ld -T firmware.ld -o firmware firmware-entry.o \
        firmware-main.o
```

Note: you should add `snprintf.o` to the object file list if you implemented date & time printing.

Disassembling built programs

`objdump(1)` is a program for disassembling programs and binary files according to given or guessed architecture and format.

A typical usage of disassembling a program is

```
mips-linux-gnu-objdump -EL -S <your-program-file>
```

Programming exercise Disassemble the firmware program you've built. You should notice that there's something wrong in your program file.

Linker scripts: advanced

Location counter .

The program by default starts at virtual address 0x0, which is not desirable.

To address this issue, we could use the *location counter*, which is written as a dot (`.`). The location counter automatically points to current location, and could be assigned.

We could rewrite the linker script, specifying the beginning address:

```
/* firmware.ld */
OUTPUT_FORMAT("elf32-tradlittlemips")
OUTPUT_ARCH("mips")
ENTRY(__reset)

SECTIONS {
        . = 0xbfc00000;
        .text : {
                *(.text);
        }
        .rodata : {
                *(.rodata);
        }
        .data : {
                *(.data);
        }
        .bss : {
                *(.bss);
        }
}
```

Programming exercise Change the order of `firmware-entry.o` and `firmware-main.o` in the linking command. Disassemble the product and see the difference.

You should see that the object file containing `__reset` should be the first object file, or the first instruction executed by the CPU won't be that in `__reset`.

Make some changes in the linker script, such that changing the order of object files in linking command still works.

- Hint: one additional line in the linker script is enough.

Stripping program headers

If you open the program file via some hex editor like `xxd(1)` or `hexedit(1)`, you'll notice that the content is different from the output of disassembly. The reason is that the program file contains additional informations such as ELF header, program headers, section headers (see `elf(5)`), as well as lots of garbage sections.

`objcopy(1)` can strip the headers and garbage sections away, by specifying the sections you want, as well as the output format:

```
mips-linux-gnu-objcopy -O <format> [-j <section> [-j <section> [...]]] \
        <input-file> <output-file>
```

For example, to output a firmware binary file only containing code, enter (assuming you're strictly following all steps above)

```
mips-linux-gnu-objcopy -O binary -j .text firmware firmware.bin
```

Programming exercise Change the command above to introduce the three data sections into the binary file.

Running Firmware

As we've obtained the binary block, we're ready for running the firmware.

MSIM configuration file

MSIM configuration file is basically a list of commands with annotations starts with `#`. The annotations are simply ignored.

Be sure to read the reference to fully understand the meaning of each command and how to read the contents of CPU, storage and devices. Moreover, consult the reference to see how to enable instruction-level tracing at startup.

```
# firmware.conf
add dcpu cpu0
add rwm ram 0x00000000
ram generic 256M
add rom fw 0x1fc00000
fw generic 1M
fw load "firmware.bin"
add dprinter lp0 0x1f000010
```

Running MSIM

Assuming that you had stored the configuration file in `firmware.conf`, you can run MSIM using the configuration file:

```
msim -c firmware.conf
```

See if the firmware is greeting you! And if you had implemented the function for printing date & time, see if the time is correct.

Writing a Makefile

You'd probably want to automate the building process, as typing `gcc` and `ld` in compiler each time you want to build the program is painful, especially when there's a ton of object files.

`make(1)` and Makefiles are good tools to automate the compiling and building process. However, it's often hard to understand or write a Makefile at start.

It's possible to write a shell script instead of a Makefile for automatic build, however Makefiles are often lighter.

Basics

The main body of a Makefile is a rule set, with the following format:

```
# Makefile
<target>: <dependency-1> <dependency-2> ...
        <list-of-commands>
```

The silliest Makefile for building the program above is (the following examples are without `snprintf.c`, and you should add the related commands, rules or modify certain variables if you implemented date & time printing):

```
# Makefile
# Add snprintf.c if you have date & time printing
firmware.bin: firmware-main.c firmware-entry.S
        mips-linux-gnu-gcc -EL -nostdinc -nostdlib -mabi=32 -mips32 -fno-pic \
                -g -mno-abicalls -I. -I./include -I./include/arch/mips \
                -c -o firmware-entry.o firmware-entry.S
        mips-linux-gnu-gcc -EL -nostdinc -nostdlib -mabi=32 -mips32 -fno-pic \
                -g -mno-abicalls -I. -I./include -I./include/arch/mips \
                -c -o firmware-main.o firmware-main.S
        mips-linux-gnu-ld -T firmware.ld -o firmware firmware-entry.o \
                firmware-main.o
        mips-linux-gnu-objcopy -O binary -j .text <more-options> firmware \
                firmware.bin
```

NOTE: you should replace the spaces with tabs.

Typing `make` is equal to `make all`. So you'll have to add a rule for this target. You can make `all` depend on `firmware`:

```
# Makefile
# Add snprintf.c if you have date & time printing
```

```
firmware: firmware-main.c firmware-entry.S
        mips-linux-gnu-gcc -EL -nostdinc -nostdlib -mabi=32 -mips32 -fno-pic \
                -g -mno-abicalls -I. -I./include -I./include/arch/mips \
                -c -o firmware-entry.o firmware-entry.S
        mips-linux-gnu-gcc -EL -nostdinc -nostdlib -mabi=32 -mips32 -fno-pic \
                -g -mno-abicalls -I. -I./include -I./include/arch/mips \
                -c -o firmware-main.o firmware-main.S
        mips-linux-gnu-ld -T firmware.ld -o firmware firmware-entry.o \
                firmware-main.o
        mips-linux-gnu-objcopy -O binary -j .text <more-options> firmware \
                firmware.bin

all: firmware.bin
```

But this Makefile is no different from a shell script.

You can give finer dependency hierarchy by making the program to be dependent on object files, which in turn depend on source files, like this:

```
# Makefile
# Add snprintf.c if you have date & time printing
firmware.bin: firmware
        mips-linux-gnu-objcopy -O binary -j .text <more-options> firmware \
                firmware.bin

firmware: firmware-main.o firmware-entry.o
        mips-linux-gnu-ld -T firmware.ld -o firmware firmware-entry.o \
                firmware-main.o

firmware-main.o:
        mips-linux-gnu-gcc -EL -nostdinc -nostdlib -mabi=32 -mips32 -fno-pic \
                -g -mno-abicalls -I. -I./include -I./include/arch/mips \
                -c -o firmware-main.o firmware-main.c

firmware-entry.o:
        mips-linux-gnu-gcc -EL -nostdinc -nostdlib -mabi=32 -mips32 -fno-pic \
                -g -mno-abicalls -I. -I./include -I./include/arch/mips \
                -c -o firmware-entry.o firmware-entry.S

all: firmware.bin
```

Programming exercise Try both Makefiles, make modifications to one of the sources, and see the difference. That difference is one of the reasons why we prefer to use `make(1)` rather than shell scripts.

Variables

Variables can be declared to write configurable, cleaner Makefiles, and change the default behaviors.

One can assign values to variables in the following format:

```
<VARIABLE> = <value>
```

Or, one can append to variables by

```
<VARIABLE> += <value>
```

The variables can be referenced later by `$(<VARIABLE>)`

Default behavior

If `make(1)` can't find the rule for building the object file target, it'll try to guess what it should do based on a huge set of implicit rules.

Most implicit rules are based on Makefile variables. For example, when `make(1)` can't find the rule of an object file, it tries to guess what the corresponding source file is, by replacing the extension `.o` with various source code extensions such as `.c`, `.S`, `.cpp` etc. If it found a corresponding C file, it tries to generate the object file by invoking the following command

```
$(CC) $(CFLAGS) -c -o <target-name>.o <target-name>.c
```

where `CC` and `CFLAGS` are both Makefile variables, the former having a default value `cc`.

See *Implicit Rules* section in `info make`, or the reference manual on the web, for a full list of implicit rules.

Therefore, one can change the compiler name and compiler options, thereby changing the default behavior by assigning `CC` and `CFLAGS` with new values.

In our example, we can rewrite Makefile to

```
# Makefile

CC              = mips-linux-gnu-gcc
CFLAGS          = -EL -nostdinc -nostdlib -mabi=32 -mips32 -fno-pic -g \
                  -mno-abicalls -I. -I./include -I./include/arch/mips

firmware.bin: firmware
```

```
        mips-linux-gnu-objcopy -O binary -j .text <more-options> firmware \
                firmware.bin

# Add snprintf.o to the list if you have date & time printing
firmware: firmware-main.o firmware-entry.o
        mips-linux-gnu-ld -T firmware.ld -o firmware firmware-entry.o \
                firmware-main.o

# firmware-main.o is omitted and being processed by default rules.
# And so is snprintf.o

# The lengthy compiler names and compiler options are replaced by the
# variables.
firmware-entry.o:
        $(CC) $(CFLAGS) -c -o firmware-entry.o firmware-entry.S

all: firmware.bin
```

Programming exercise

1. Figure out the implicit rule for `.S` files, and make changes to the Makefile by removing rule for `firmware-entry.o` and adding/modifying variables. Your Makefile should still work.

Writing Configurable, Scalable, Extensible Makefiles

As your project grows, more and more source files would be added. Changing or adding rules in Makefiles in a *per file* basis is inconvenient.

Moreover, other people may want to build from your code, and not everyone's MIPS GCC compiler is named `mips-linux-gnu-gcc`.

Therefore, writing a flexible Makefile is important.

Let's see how we could improve the Makefile above.

1. `gcc`, `ld`, `objcopy` are all toolchain programs which share the same prefix, so we could generalize them by introducing a new variable `CROSS_COMPILE`:

```
# Makefile
CROSS_COMPILE   = mips-linux-gnu-

CC              = $(CROSS_COMPILE)gcc
LD              = $(CROSS_COMPILE)ld
OBJCOPY         = $(CROSS_COMPILE)objcopy
```

```
CFLAGS          = -EL -nostdinc -nostdlib -mabi=32 -mips32 -fno-pic -g \
                  -mno-abicalls -I. -I./include -I./include/arch/mips

# See if you can generalize your own variables.

firmware.bin: firmware
        $(OBJCOPY) -O binary -j .text <more-options> firmware firmware.bin

# Add snprintf.o to the list if you have date & time printing
firmware: firmware-main.o firmware-entry.o
        $(LD) -T firmware.ld -o firmware firmware-entry.o firmware-main.o

# firmware-main.o is omitted and being processed by default rules.
# And so is snprintf.o

# You should have already removed firmware-entry.o rule by now according to
# the above exercise.

all: firmware.bin
```

2. We can further extract files or file lists to independent variables:

```
# Makefile
CROSS_COMPILE   = mips-linux-gnu-

CC              = $(CROSS_COMPILE)gcc
LD              = $(CROSS_COMPILE)ld
OBJCOPY         = $(CROSS_COMPILE)objcopy

CFLAGS          = -EL -nostdinc -nostdlib -mabi=32 -mips32 -fno-pic -g \
                  -mno-abicalls -I. -I./include -I./include/arch/mips

LDSCRIPT        = firmware.ld

ELF             = firmware
BINARY          = firmware.bin
# Take caution of the order, if you didn't make the required changes
# in linker script in previous exercise.
# Add snprintf.o to the list if you have date & time printing
OBJS            = firmware-entry.o \
                  firmware-main.o

# See if you can make further changes to your own variables.
```

```
$(BINARY): $(ELF)
        $(OBJCOPY) -O binary -j .text <more-options> $(ELF) $(BINARY)

# Add snprintf.o to the list if you have date & time printing
$(ELF): $(OBJS)
        $(LD) -T $(LDSCRIPT) -o $(ELF) $(OBJS)

# firmware-main.o is omitted and being processed by default rules.
# And so is snprintf.o

# You should have already removed firmware-entry.o rule by now according to
# the above exercise.

all: $(BINARY)
```

3. We can even separate the options for include directories in `CFLAGS` out, as a separate variable `INCS`. Later header addition/modification only needs to take care of `INCS`, rather than `CFLAGS`.

```
# Makefile
CROSS_COMPILE   = mips-linux-gnu-

CC              = $(CROSS_COMPILE)gcc
LD              = $(CROSS_COMPILE)ld
OBJCOPY         = $(CROSS_COMPILE)objcopy

INCS            = -I. -I./include -I./include/arch/mips

CFLAGS          = -EL -nostdinc -nostdlib -mabi=32 -mips32 -fno-pic -g \
                  -mno-abicalls $(INCS)

LDSCRIPT        = firmware.ld

ELF             = firmware
BINARY          = firmware.bin
# Take caution of the order, if you didn't make the required changes
# in linker script in previous exercise.
# Add snprintf.o to the list if you have date & time printing
OBJS            = firmware-entry.o \
                  firmware-main.o

# See if you can make further changes to your own variables.

$(BINARY): $(ELF)
        $(OBJCOPY) -O binary -j .text <more-options> $(ELF) $(BINARY)
```

```
$(ELF): $(OBJS)
        $(LD) -T $(LDSCRIPT) -o $(ELF) $(OBJS)

# firmware-main.o is omitted and being processed by default rules.
# And so is snprintf.o

# You should have already removed firmware-entry.o rule by now according to
# the above exercise.

all: $(BINARY)
```

Programming exercise Write a rule for `make clean`, to remove all `make`-generated stuff including binaries, programs, object files, etc. With these variables, writing new rules would become substantially easier.

.PHONY

If you accidentally created a file named `clean` in the same directory as the Makefile, invoking `make clean` would give you a ridiculous message:

```
make: 'clean' is up to date.
```

The reason is that, when `make clean` is invoked, `make` is actually trying to build the `clean` target. And if `make` found that a file named `clean` already exists, it'll assume that the target `clean` is already built, doing nothing.

`.PHONY` is a special target which instructs `make` to always build the specified dependencies regardless of the modification time, file existence, or anything else.

A typical usage of `.PHONY` targets is those command-like targets such as `all`, `clean`, `install`, etc.

```
# Makefile
INCS            = -I. -I./include -I./include/arch/mips

CFLAGS          = -EL -nostdinc -nostdlib -mabi=32 -mips32 -fno-pic -g \
                  -mno-abicalls $(INCS)

LDSCRIPT        = firmware.ld

ELF             = firmware
BINARY          = firmware.bin
# Take caution of the order, if you didn't make the required changes
# in linker script in previous exercise.
```

```
# Add snprintf.o to the list if you have date & time printing
OBJS            = firmware-entry.o \
                  firmware-main.o

# See if you can make further changes to your own variables.

$(BINARY): $(ELF)
        $(OBJCOPY) -O binary -j .text <more-options> $(ELF) $(BINARY)

$(ELF): $(OBJS)
        $(LD) -T $(LDSCRIPT) -o $(ELF) $(OBJS)

# firmware-main.o is omitted and being processed by default rules.
# And so is snprintf.o

# You should have already removed firmware-entry.o rule by now according to
# the above exercise.

all: $(BINARY)

# Your rule for "make clean" goes...

.PHONY: all clean
```

Organizing firmware source files

This exercise requires us to write three things: the firmware, the bootloader, and the kernel.

They are three separate programs, so merely mixing the sources together in a single directory is almost certainly not preferable.

In general, you should organize the source files (including C code, assembly code, Makefiles, linker scripts) into a hierarchy.

The top level is *purpose*. We should make at least four directories:

1. `firmware`.
2. `boot`: for MBR code and Makefile.
3. `kern`: for kernel source files, which will be further divided by modules, functionality, etc.
4. `lib`: for code shared between programs, and are not specific to either program or module.
    - Probably a `libc` subdirectory should be made there to hold the C library function equivalents such as `snprintf.c`. As the project

goes on, userspace programs would also use the source code in `lib` directory.

Source Management Exercise Move the sources you've written, as well as the source files in `src` directory according to the hierarchy.

Change your Makefile (and possibly your linker script) according to your arrangement.

Writing a top-level Makefile

Since we have moved firmware source files to `firmware` directory, a top-level Makefile is needed, because it's more favorable to invoke `make` once at the top directory, letting the utility to take care of everything, rather than manually changing into each directory and build things there.

It's easy to write a top-level Makefile, provided that you've already written the Makefiles at the next level (e.g. the Makefile in `firmware` directory).

```
# Makefile (top-level)
all:
        $(MAKE) -C firmware
        # more subdirectories...

clean:
        $(MAKE) -C firmware clean
        # more subdirectories...

.PHONY: all clean
```

See `make(1)` for the meaning of `-C` option.

The reason we use `$(MAKE)` rather than `make` here is *platform portability*. Some operating systems, such as OpenBSD, provides two versions of `make`: a BSD `make` and a GNU `gmake` as a port package, and you should allow others building your project using things like OpenBSD to conveniently choose which `make` they prefer.

The default `$(MAKE)` value is the `make` program or its variant processing this Makefile, so you don't need to assign it a value manually.

Exporting variables top-down

Top-level Makefiles can "export" variables to Makefiles at lower-levels, meaning that lower-level Makefiles can use variables defined at upper-levels.

The `export` command in Makefile does this job.

For example, we can move the toolchain configuration variables from `firmware/Makefile` to the top-level `Makefile`:

```
# Makefile (top-level)
CROSS_COMPILE   = mips-linux-gnu-

CC              = $(CROSS_COMPILE)gcc
LD              = $(CROSS_COMPILE)ld
OBJCOPY         = $(CROSS_COMPILE)objcopy

# Export the variables all the way to bottom
export

all:
        $(MAKE) -C firmware
        # more subdirectories...

clean:
        $(MAKE) -C firmware clean
        # more subdirectories...

.PHONY: all clean
```

And the Makefile in `firmware` directory becomes

```
# firmware/Makefile
# Note that
# 1. Since we have moved the Makefile to firmware directory, the
#    include directories should be changed accordingly.
# 2. Toolchain variables such as CC and LD are received from upper-level,
#    so we don't need to assign them here.
INCS            = -I.. -I../include -I../include/arch/mips

CFLAGS          = -EL -nostdinc -nostdlib -mabi=32 -mips32 -fno-pic -g \
                  -mno-abicalls $(INCS)

LDSCRIPT        = firmware.ld

ELF             = firmware
BINARY          = firmware.bin
# Take caution of the order, if you didn't make the required changes
# in linker script in previous exercise.
# Add snprintf.o to the list if you have date & time printing
OBJS            = firmware-entry.o \
```

```
                    firmware-main.o

# See if you can make further changes to your own variables.

$(BINARY): $(ELF)
        $(OBJCOPY) -O binary -j .text <more-options> $(ELF) $(BINARY)

$(ELF): $(OBJS)
        $(LD) -T $(LDSCRIPT) -o $(ELF) $(OBJS)

# firmware-main.o is omitted and being processed by default rules.
# And so is snprintf.o

# You should have already removed firmware-entry.o rule by now according to
# the above exercise.

all: $(BINARY)

# Your rule for "make clean" goes...

.PHONY: all clean
```

The reason is that the top-level Makefile enables us to configure toolchains, as well as other global settings, there, and `make` can apply the settings *globally*, passing the settings all the way to bottom.

Maintaining Global Data in ROM

Sometimes, ROM programs may have to maintain global data, possibly altering them during execution, which leads to new problems.

Programming exercise Add the following logic to `firmware-main.c`:

```c
int g = 6;

void change_global_data(void)
{
        /* You should have implemented kprintf() earlier */
        kprintf("g = %d\n", g);
        g = 8;
        kprintf("g = %d\n", g);
}
```

Execute that logic and check if it works. Try to infer the reason. What's the address of global variable `g`? Use `readelf(1)` to find the address of global symbols.

Unsuccessful attempt: solely changing location counter

A direct thought is to change the location counter at the start of `.data` section in the linker script, like

```
/* firmware/firmware.ld */
OUTPUT_FORMAT("elf32-tradlittlemips")
OUTPUT_ARCH("mips")
ENTRY(__reset)

SECTIONS {
        . = 0xbfc00000;
        .text : {
                *(.text);
        }
        .rodata : {
                *(.rodata);
        }
        . = 0x80000000;         /* RAM physical address */
        .data : {
                *(.data);
        }
        .bss : {
                *(.bss);
        }
}
```

Unfortunately, this modification would result in an extremely large firmware binary (~1GB). You can invoke `ls -l` to see the size of generated firmware binary.

Paper exercise `ls -l` shows that the binary is about 1GB large. However, another program, `du(1)`, for querying disk usage of a file or a set of files, reports with a much less space (usually in KB). Check the disk usage of your binary by

```
du -h <your-binary>
```

How could this difference happen? Search on the web for answer.

Reason: translation from ELF to binary

ELF defined two addresses for each section, one called the *load address* (LMA), where the initial data is stored, and another one called *virtual address* (VMA, don't be confused with the virtual address concept in memory management!), where the code performs accesses.

`objcopy(1)` translates ELF to binary by copying each section to file offset *linearly mapped* from its LMA.

In most cases, LMA and VMA are identical, and LMA are rarely used in daily development.

The corner case where LMA kicks in is ROM development, which, unfortunately, is the situation we're facing now. Putting `.data` section in ROM addresses won't work, while changing VMA while keeping LMA identical to VMA does no help because

1. Practically, it'll result in a much larger binary.
2. Logically, the initial data should be stored in ROM anyway (you can't store anything in RAM before powering on).

The only solution is to make LMA different from VMA: VMA points to RAM, making code perform accesses according to that address, while LMA continues to point to ROM, where the initial data holds. Prior to accessing the global data, the code must *relocate*, or copy the data from LMA to VMA.

Linker script: `AT` directive

The linker script uses an `AT` directive at the beginning of section declaration in `SECTIONS` command. An example of usage is given below:

```
/* firmware/firmware.ld */
OUTPUT_FORMAT("elf32-tradlittlemips")
OUTPUT_ARCH("mips")
ENTRY(__reset)

SECTIONS {
        . = 0xbfc00000;
        .text : {
                *(.text);
        }
        .rodata : {
                *(.rodata);
        }
        . = 0x80000000;
```

```
        .data : AT(0xbfc10000) {        /* LMA inside the parentheses */
                *(.data);
        }
        .bss : {
                *(.bss);
        }
}
```

You don't need to specify LMA for later sections; `ld(1)` will automatically put them one after another. `ld(1)` does check LMAs for possible overlaps to prevent some sections overwriting other sections, though.

Linker script: providing symbols

We can see that directly assigning LMA with a concrete number is inflexible by the example above because:

1. It wastes space if `.text` and `.rodata` sections are small.
2. `.data` may be overwritten or overwrite `.text` and/or `.rodata` if the sections are large.

Linker scripts can instruct `ld(1)` to emit symbols, which could be referenced by both linker script itself and C/assembly code, at given addresses. The following example emits a symbol pointing to the ending address of `.rodata` section by assigning the location counter to that symbol.

```
/* firmware/firmware.ld */
OUTPUT_FORMAT("elf32-tradlittlemips")
OUTPUT_ARCH("mips")
ENTRY(__reset)

SECTIONS {
        . = 0xbfc00000;
        .text : {
                *(.text);
        }
        .rodata : {
                *(.rodata);
                __rodata_end__ = .;     /* Points to end of .rodata */
        }
        . = 0x80000000;
        .data : AT(__rodata_end__) {    /* LMA given by symbol */
                *(.data);
        }
```

```
        .bss : {
                *(.bss);
        }
}
```

Programming exercise Modify the linker script to emit four symbols, pointing to the beginning and the end of `.data` section and `.bss` section.

If you run into problems that `.data` or `.bss` overlaps with something else, put the other section into `/DISCARD/` section like:

```
        /DISCARD/ : {
                *(.reginfo);
        }
```

Linker script: `MEMORY` command

A slightly more complex but more structured way to handcraft section arrangement is to use the `MEMORY` command, which is to specify different memory regions.

The following linker script snippet declares two memory regions, the former for ROM, the latter for RAM.

```
MEMORY {
        rom (RX) : ORIGIN = 0xbfc00000, LENGTH = 1M
        ram (RW) : ORIGIN = 0x80000000, LENGTH = 256M
}
```

`rom` and `ram` are names of each memory region. `RWX` characters inside the parentheses defines the access control, i.e. whether it's Readable, Writable, and/or eXecutable. Other tokens are pretty self-descriptive.

After writing the `MEMORY` command, `SECTIONS` command could be rewritten like follows:

```
/* firmware/firmware.ld */
OUTPUT_FORMAT("elf32-tradlittlemips")
OUTPUT_ARCH("mips")
ENTRY(__reset)

MEMORY {
        rom (RX) : ORIGIN = 0xbfc00000, LENGTH = 1M
        ram (RW) : ORIGIN = 0x80000000, LENGTH = 256M
}
```

```
SECTIONS {
        .text : {
                *(.text);
        } >rom                          /* Put into rom region */
        .rodata : {
                *(.rodata);
                __rodata_end__ = .;     /* Points to end of .rodata */
        } >rom                          /* Put into rom region after .text */
        .data : AT(__rodata_end__) {    /* LMA given by symbol */
                *(.data);
        } >ram                          /* Put into ram region */
        .bss : {
                *(.bss);
        } >ram                          /* Put into ram region after .data */
}
```

Relocating `.data` and `.bss` sections

As noted above, we have to copy the initial data from ROM to RAM before making any global data accesses.

To do this, we need to know where the `.data` section is initially stored (`__rodata_end__`), where we should copy `.data` section into, and how much are we copying. Also, we need to know where we should fill the `.bss` section with 0.

We should provide the symbols indicating the range of `.data` section and `.bss` section first (the following code should really be your work, but I'll post the answer here regardless):

```
/* firmware/firmware.ld */
OUTPUT_FORMAT("elf32-tradlittlemips")
OUTPUT_ARCH("mips")
ENTRY(__reset)

MEMORY {
        rom (RX) : ORIGIN = 0xbfc00000, LENGTH = 1M
        ram (RW) : ORIGIN = 0x80000000, LENGTH = 256M
}

SECTIONS {
        .text : {
                *(.text);
        } >rom
        .rodata : {
                *(.rodata);
```

```
                __rodata_end__ = .;
        } >rom
        .data : AT(__rodata_end__) {
                __data_begin__ = .;
                *(.data);
                __data_end__ = .;
        } >ram
        .bss : {
                __bss_begin__ = .;
                *(.bss);
                __bss_end__ = .;
        } >ram
}
```

Then, at the very first part of C code, copy `.data` section from ROM to RAM, and initialize `.bss`:

```
extern char __rodata_end__[];   /* provided by linker */
extern char __data_begin__[], __data_end__[];
extern char __bss_begin__[], __bss_end__[];

void reloc_data_bss(void)
{
        memcpy(__data_begin__, __rodata_end__, __data_end__ - __data_begin__);
        memset(__bss_begin__, 0, __bss_end__ - __bss_begin__);
}
```

Programming exercise Relocate `.data` and `.bss` sections, and see if the `change_global_data` logic above works.

Read and write a sector from Disk Controller

The disk controller provided by MSIM defined four 32-bit registers. Assuming the physical address is `PA`:

1. The DMA buffer *physical* address register is at `PA + 0x0`. The device directly put data read from disk into the main memory there, or get the data there to write to disk.
    - Do NOT put virtual address in that register. The device have no access to the processor, and therefore has no idea how to translate virtual address to physical address.
2. The sector number register, indicating where to read/write data, is at `PA + 0x4`. The number starts from 0.

3. The register at `PA + 0x8` has different meaning when read/written.
4. The read-only register at `PA + 0xc` returns the size of disk in bytes when read.

See the reference manual for details.

Bad practice

One may tend to let the disk controller to directly read or write to the target memory. This is usually a *bad idea*, as it may easily corrupt memory or disk storage, unless with careful design (which is unlikely). The exercise below demonstrates such situation.

The firmware or kernel should always allocate a separate, static buffer for disk controller to perform operations, and copy the needed content to the target buffer after the disk had finished its command.

Paper/programming exercise Consider the following code snippet:

```
/*
 * Read from sector @sector_num, offsetting @offset bytes, storing the
 * data to @buf with size @len.
 */
void readdisk(size_t sector_num, ssize_t offset, void *buf, size_t len);

/*
 * Write to sector @sector_num, offsetting @offset bytes, storing the
 * data to @buf with size @len.
 */
void writedisk(size_t sector_num, ssize_t offset, void *buf, size_t len);

void write_data_to_disk(void)
{
        unsigned int a = 0x76543210, b = 0xfedcba98;
        writedisk(0, 0, &a, sizeof(a));
}

void read_data_from_disk(void)
{
        unsigned int c = 0x0, d = 0x12345678;
        readdisk(0, 0, &c, sizeof(c));
        kprintf("c = %08x, d = %08x\n", c, d);
}
```

```
int main(void)
{
        write_data_to_disk();
        read_data_from_disk();
        for (;;)
                /* nothing */;
}
```

Assuming that the implementation of `readdisk()` and `writedisk()` directly gives the disk controller the physical address of `buf`.

1. Is it possible to actually implement `readdisk()` and `writedisk()`? If possible,
2. What's the output of `main()` with your implementation?

Programming exercise

Implement the `readdisk` function above.

Are all the four arguments necessary? If so, give the reason. If not, simplify the function prototype and implement your own version of `readdisk`.

Manipulating hard disk images

You have to create a hard disk image for MSIM to play with.

Paper/programming exercise Read the reference manual to see how to make MSIM load the disk image from file. Change the configuration file accordingly if you had written one.

`dd(1)`

`dd(1)` is a utility with strange command syntax for copying files, or parts of files. You can use it to create a blank file, copy a file into the middle of file, extract a part of file into another file, etc.

Since hard disks, as well as partitions of each hard disk, are exposed as special files (see `sd(4)`), `dd(1)` can do more ~~exciting~~ stuff such as saving/restoring MBR, saving/restoring partition tables, etc.

See the man page for full usage.

Programming exercise Use `dd(1)` to create a blank 512M hard disk image.

A hint: read `zero(4)`.

Partitioning hard disk image

The next thing we should do is to make partitions on your disk image. You can do it in command line or in GUI.

Either way, the partitioning utility would make an MBR for you.

`fdisk(8)` `fdisk(8)` is a command line utility for partitioning hard disk.

Programming exercise If you're going to make partitions in command line, use `fdisk(8)` to make four DOS partitions, as we're going to put the kernel image at the beginning of the second partition.

GParted You can also do it graphically, using tools like GParted.

First, you need to setup a loop device using utilities such as `losetup(8)`. A *loop device* is a kind of virtual device associated to either real devices or regular files. You can create a loop device with your disk image file by

```
losetup --find --show <your-disk-image-file>
```

`losetup(8)` prints out the device path (like `/dev/loop0`) after successfully created a loop device. Then you can do anything to the loop device as if it's a real device. For example, you can make partitions in GParted, a graphical partitioning utility by

```
gparted /dev/loop0
```

I still recommend using `fdisk(8)` rather than GParted when making partitions, because `fdisk(8)` supports more partition types. However, GParted can make file systems for you (which is of no use since we'll write our own file system).

Transferring control to MBR

On x86 machines, the BIOS loads the first sector, which is called a *master boot record* (MBR), and executes the bootloader there.

Consult the Wikipedia page for MBR to see the structure.

You'll see that there's only 446 bytes (that is, 111 MIPS instructions, probably even less if you include `.rodata`, `.data`, `.bss` sections). So you may have to provide an entrance for bootloader in MBR to read hard disks, as it's usually not possible to fit the disk driver into such small space.

In fact, this is exactly how MBR reads the hard disk. MBR invokes BIOS services by *software interrupts*, which is some kind of entrance. That is, MBR calls a routine inside BIOS, executing code in BIOS to read the hard disk. We will simulate this behavior by simply passing the address of your `readdisk` function as an argument to the bootloader entry function.

You'll have to use *function pointers* to do so.

Why MBR?

The computer should ultimatel/y locate the kernel on a storage, e.g. hard disk, CD-ROM, flash disk, etc. However, the firmware can't directly find the kernel because of different partition schemes, different file systems, multiple kernels on different locations, etc.

There are several methods to resolve this issue:

1. Make the kernel's location another programmable attribute of firmware, just like system time, firmware password, boot sequence, etc. Loongson's original firmware, PMON, does so. This is usually a bad idea because
   - The firmware should support file systems.
   - It's hard to support multiple operating systems (dual-boot).
2. Make it a convention that the firmware always executes some code somewhere on the storage. The content on storage is freely changeable by the user, unlike firmware, and therefore enjoys more flexibility. MBR and today's UEFI fall into this category, the former assumes that the code on the first sector would always be executed, the latter assumes that the code is stored as files on some special partitions (called the *EFI system partition*).

In our lab, for simplicity, we assumes that the kernel always reside on the *second* MBR partition.

Code logic

Suppose that you had already implemented your `readdisk` function:

```
/* The prototype may be different */
void readdisk(size_t sector_num, ssize_t offset, void *buf, size_t len);
```

First, define a function pointer type with the same *function signature*, that is, the information including return type and argument types:

```c
/* Depend on your implementation */
typedef void (*readdisk_t)(size_t, ssize_t, void *, size_t);
```

Moreover, assuming you're able to read the first sector:

```c
#define SECTOR_SIZE     512

void execute_mbr(void)
{
        unsigned char *mbr = MBR_INITIAL_ADDRESS;
        readdisk(0, 0, mbr, SECTOR_SIZE);
}
```

Obviously one need to pass the address of `readdisk` to the bootloader.

However, during code generation, the compiler and the linker usually have no idea where the MBR would be located in RAM, unless:

1. The bootloader code is compiled and built to be *position-independent*, or
2. The developer explicitly hardwires the address in firmware, and tells the linker to generate code to run at that address.

Either way, you will have to define another function type for bootloader entrance, such as

```c
/* Assuming that you adopt scheme 1 */
typedef void (*bootentry_t)(readdisk_t);
```

Then you'll transfer control to MBR in RAM by

```c
        /*
         * __mbr is the location where MBR in RAM is finally located.
         * Either it's position-independent, or you should specify the initial
         * address in linker script for MBR.
         */
        bootentry_t boot = (bootentry_t)__mbr;
        (*boot)(readdisk);
```

Programming exercise Complete the firmware logic to load and execute MBR.

Coding MBR

You probably need only one C file to finish the MBR, as the logic is simple:

1. Locate the kernel (at the start of second partition), which is stored as an ELF.
2. Load each loadable segment from the ELF to the address specified in the segment header (or program header).
3. Jump to the entry specified in ELF header.

The structure of ELF header and program headers, as well as the relation between them, are described in `elf(5)`. The generic header `elf.h` provides the C definition of those headers. You should have moved the header to the generic header directory.

Portablility: 32-bit and 64-bit ELF

You may notice that the layout of the headers are different between 32-bit and 64-bit ELF. Moreover, whether to process 32-bit or 64-bit in MBR is usually determined during compile time rather than run time.

Therefore, we defined common type names at the end of `elf.h`, such as `elfhdr`, `elf_phdr`, etc. Whether an `elfhdr` is an `elf32hdr` or an `elf64hdr` is determined by macro definition of `ELF32` and `ELF64`.

The requirement of using the common type names is that you can determine whether to process 32-bit or 64-bit ELF at *compile time*. If you can't, you may have to write separate routines for handling each of them.

`gcc(1)` can define macros by compiler option `-D`. For example, compiling `a.c` with `-DELF32` option, such as

```
gcc -DELF32 -c -o a.o a.c
```

is equivalent to adding

```
#define ELF32
```

at the very top of C source file `a.c`.

By giving macro definition options rather than defining them in source, switching code for compilation becomes easier, as you can define a `DEFS` file for all macro definition options, and add the `DEFS` variable to `CFLAGS`, to define the macros for all C source files, allowing you to apply macros to all source files by Makefile rather than writing or changing a configuration header.

Jumping to kernel

The ELF header contains the entry point, where the first instruction of the program is located.

The simplest kernel entry function signature is a void function which receives no parameters:

```
typedef void (*entry_t)(void);
```

Assuming that you had already obtained the kernel entry address at `entry_addr` variable, you can jump to kernel by

```
entry_t entry = (entry_t)entry_addr;
(*entry)();
```

This piece of code much resembles that in firmware.

Programming exercise Write a C source file for MBR bootloader code. Make sure your entry function is of the same signature as the boot entry type in firmware code.

Since the firmware would pass the disk reading function entry as an argument, you can use that argument to read data from the hard disk.

Writing Makefile for MBR

You should probably write a Makefile by yourself now.

Programming exercise Write a Makefile to:

1. Compile your C source file into an object file, and
2. Link the object file into an ELF executable, and
3. Translate the ELF executable into a solid binary.

Disassemble the binary to see if the code would correctly do your job.

Personally, I suggest you to hardwire the MBR address in both firmware and MBR. That is, the firmware loads the MBR into a fixed RAM address, say, `0x80001000`.

```c
#define MBR_INITIAL_ADDRESS     0x80001000

void execute_mbr(void)
{
        char *mbr = (char *)MBR_INITIAL_ADDRESS;
    readdisk(0, 0, mbr, SECTOR_SIZE);
    /* ... */
}
```

Then, we specify the initial address in linker script of MBR to be `0x80001000`:

```
/* mbr.ld */
/* ... */
SECTIONS {
        . = 0x80001000;
        /* ... */
}
```

This way, you won't need to deal with position-independent code, which is rather complicated in our settings.

Testing size & Installing to disk image

Probably you can define a target called `msim`, which automatically installs the bootloader code into your disk image.

Installing is not a problem, since `dd(1)` could do that. The problem is how to check if the bootloader exceeds the 446-byte limit, to prevent the partition entries from being overwritten.

Condition and branches in `bash(1)` This section is a bit tricky.

The program to count characters (or bytes) is `wc(1)`

```
wc -c <file>
```

which has the following output

```
<number-of-characters> <file>
```

So we need to cut off the second field, which could be done by `cut(1)`

```
cut -f <field-number> -d <delimiter>
```

Connecting the two commands with pipe yields the number of bytes only.

```
wc -c <file> | cut -f 1 -d " "
```

`bash(1)` provides an if-else control block to perform branches, and a `[ ... ]` syntax to check if a condition holds. You can find the syntax and details in `bash(1)`, as I'm only giving the final shell script here.

```
        if [ $(wc -c <file> | cut -f 1 -d " ") -ge 446]; then \
                echo Bootloader too large; \
        else \
                <write your dd command invocation here>; \
        fi
```

The backslashes are merely line continuations.

The parentheses preceded by a dollar sign `$` would be replaced by the output of the shell command inside.

In Makefile, the rule body is mostly the same as above:

```
msim: all
        if [ $(shell wc -c <file-or-variable> | cut -f 1 -d " ") -ge 446]; then \
                echo Bootloader too large; \
        else \
                <write your dd command invocation here>; \
        fi
```

We can see that the syntax for shell invocation becomes

```
$(shell <shell-command>)
```

Programming exercise Complete the `msim` rule by replacing the fields above with your own stuff. You can then execute `make msim` to automatically test and install bootloader code to MBR in disk image.

Organizing MBR codes

The source files for MBR should be put inside a separate folder.

Source Organization & Programming exercise

1. Move the source files for MBR into a separate folder.
2. Change your top-level Makefile to build bootloader as well.
3. Add `msim` target to build `msim` target in the subfolders.

Coding Kernel

Our AIMv6 is an operating system kernel designed to be runnable on IA32, ARM and MIPS32/MIPS64 *with one codebase*. Therefore, it's crucial to design a scalable framework, which is easy to extend architecture and hardware support, and is easy to replace one or several components while keeping the kernel's logic unaltered.

Here, the job of kernel is to print out a "Hello world" message.

The kernel source files should be put in `kern` directory, of course.

Kernel designing principles

1. Scalability is absolutely one principle.
2. Another principle is to *make as few assumptions of hardware as possible.*

Bootstrapping C Environment in Kernel

Although we can couple bootloader and firmware together here, coupling kernel and MBR/firmware is not a good idea, because the kernel would then only be able to run on limited families of machines.

Decoupling kernel and firmware means to

1. Initialize the C environment by kernel itself, as we can't assume that the firmware would set up the stack or setting up the processor mode for us.
2. Develop kernel's own driver, instead of using the one from firmware. For example, we can't use the `kprintf()` function or `readdisk()` function provided by firmware.

Assembly code: bootstrapping kernel

We can start writing the assembly for bootstrapping C environment based on the `__reset` function in firmware:

```
/* firmware-entry.S */
#include <asm/asm.h>
#include <asm/regdef.h>
#include <asm/cp0regdef.h>

NESTED(__reset, 0, ra)
        mfc0    a0, CP0_STATUS
        ori     a0, ST_ERL
```

```
        xori    a0, ST_ERL
        mtc0    a0, CP0_STATUS
        li      sp, 0x8f000000
        jal     main
END(__reset)
```

The first thing to take caution is that *we could not hardwire the initial stack pointer here*. Firmware developers, usually being also the manufacturers of the machines (especially ARM/MIPS machines where there's no de facto standard), are allowed to do so, but it's usually bad idea for kernel because:

1. A large constant value assumes a large RAM, which is usually an overly strong assumption.
2. A small constant value could not guarantee the stack to be safe from overwriting.

Instead, we could allocate a global static buffer for kernel stack. The assumption of hardware then becomes that the RAM is able to load the whole kernel, which sounds more reasonable. Also, by specifying the kernel stack size, instead of the kernel stack location, the chance of overwriting other data or being overwritten is much smaller (unless infinite recursion occur, which is a bug anyway).

For example, we declare the global static buffer somewhere:

```
unsigned char cpu_stack[KSTACK_SIZE];
```

and provide external symbol in some header:

```
#define KSTACK_SIZE     8192    /* or some other value */

extern unsigned char cpu_stack[];
```

Then we could change the assembly to

```
/* entry.S */
/*
 * Note: this entry point is for kernel, not firmware.
 * Put this file at a reasonable position.  You can also change the file name
 * according to your flavor.
 */
#include <asm/asm.h>
#include <asm/regdef.h>
#include <asm/cp0regdef.h>
```

```
NESTED(__start, 0, ra)
        mfc0    a0, CP0_STATUS
        ori     a0, ST_ERL
        xori    a0, ST_ERL
        mtc0    a0, CP0_STATUS
        LA      sp, cpu_stack
        ADDIU   sp, KSTACK_SIZE
        jal     main
END(__start)
```

The meaning of `LA` and `ADDIU` are defined in `asm/asm.h`

The reason we use instruction macros `LA` and `ADDIU` instead of concrete instructions `la` and `addiu` is also *platform portability*. We have to take both MIPS32 and MIPS64 into consideration, as we'll migrate our codebase smoothly to Loongson boxes. By using instruction macros, the compiler could choose which instruction to use when compiler flags are given. `LA` would become `la` if `-mips32` flag is given, or `dla` if `-mips64r2` flag is given.

The code can also be extended to support multi-processors, as each CPU needs its own stack to operate on, by defining `cpu_stack` as a two-dimension array.

C logic

The main logic for a hello world kernel is simple:

```
int main(void)
{
        kprintf("Hello world from kernel!\n");
        for (;;)
                /* nothing */;
}
```

You can directly borrow the implementation of `kprintf()` from firmware.

The real problem is how to implement `kputs()`. As the firmware only needs to support *its* output device, the kernel has to support multiple devices at once.

Moreover, as you're not going to write a hundred drivers *yourself*, you'll have to design a scalable framework, to allow other developers (usually the manufacturer) to plug their code in.

Scalable Design: supporting multiple output devices

One can try to make `kputs()` output to every output device available. How should you do that?

Usually, we should check the available device list, and find all line printer devices (e.g. serial console, console VGA, etc.). For each line printer device, we call the corresponding device-specific `puts()` function.

`puts()` can be further devided to parts: (1) string traversal, and (2) character output. It's the character output function that differs between devices.

If we're writing the kernel with object-oriented language (e.g. C++), we can design the logic as follows:

```
void kputs(const char *str)
{
        char *s;
        foreach (device : device_list) {
                if (the device is an available line output device) {
                        lp = (line_printer)device;
                        lp_puts(lp, str);
                }
        }
}

void lp_puts(line_printer lp, const char *s)
{
        for (; *s != '\0'; ++s)
                lp.write_char(*s);
}
```

But since we're writing in C rather than C++, we need to change the pseudo-code into:

```
extern struct device *devs[];                           /* (1) */
extern unsigned int ndevs;

void kputs(const char *str)
{
        char *s;
        unsigned int i;

        for (i = 0; i < ndevs; ++i) {
                if ((is_line_output(devs[i])) &&
                    (devs[i]->status & DEVSTAT_AVAILABLE)) {
                        lp = (struct lp *)devs[i];      /* (2) */
                        lp_puts(lp, str);
                }
        }
}
```

```
void lp_puts(struct lp *lp, const char *s)
{
        for (; *s != '\0'; ++s)
                lp->write_char(lp, *s);                 /* (3) */
}
```

The problem then becomes three sub-problems:

1. How to define `struct lp`.
2. How to define `struct device`.
3. How to implement the device-specific `write_char` function.

Framework design: definition of `struct lp`

Recall statement (2) in the code snippet above:

```
lp = (struct lp *)devs[i];
```

In an object-oriented perspective, we can say that the line printer class (`struct lp`) is inherited from the device class (`struct device`). We can do this in C by declaring the structure like

```
struct lp {
        struct device   dev;
        /* ... */
};
```

Getting the base structure is simply done by retrieving the `dev` member structure. But how could we convert a device structure reference `dev` into a `struct lp` reference?

The answer is to directly perform type casting

```
struct lp *lp = (struct lp *)dev;
```

Of course, to enable this type casting, the `struct device` reference should actually point to a `struct lp`. This is why statement (1) defines the list of device as array of pointers instead of concrete structures.

In terms of object-oriented programming, we can say that the device class is an *abstract* class, while the derived line printer class is a *concrete* subclass.

Of course, as of statement (3), we need to add a "method", whose C equivalent is a member function pointer, to `struct lp`:

```
struct lp {
        struct device   dev;
        void            (*write_char)(struct lp *, unsigned char);
        /* ... */
};
```

Finally, we define a concrete array of `struct lp` to store all line printer devices:

```
extern struct lp lps[];
```

Derivation from `struct lp` The above framework leave a problem unanswered: what if we want to define subclasses derived from `struct lp`? For example, a serial console may have configuration functions for setting size of FIFO, interrupt, while a VGA console may need to configure resolution, font size, etc. Are we going to define a concrete array for each subclass of `struct lp`? If yes, the code would look pretty ugly, and become hard to maintain.

The answer is: *yes here, but no later.*

The ideal solution is to dynamically allocate a concrete device structure, just as `new` keyword in C++/C#/Java, and add the reference pointing to that dynamically-allocated device structure to the device list `devs`.

However, as we're not implementing dynamic memory allocation here, the list of `struct lp` is a *workaround* to roughly maintain the framework. After we implement the `kmalloc` function to dynamically allocate memory, the list of `struct lp` will be removed, and `devs` would be replaced with a linked list.

Peripheral Device detection & registeration Preferably, the kernel should scan the device address space to dynamically detect the devices. This is not possible on MSIM since it provides no such mechanism. On Loongson Boxes, theoretically you could scan the PCI space, but it's just probably easier to make a machine-specific directory and hardwire the configurations there. (Loongson Tech. does this)

Each time we detect a peripheral device, we should initialize it. This is probably not necessary on MSIM as the hardware are too simple. On Loongson Boxes, preferably you should initialize the device you would use *again* in the kernel, despite the previous initialization done by the firmware. But you can also stick to the firmware initialization, doing nothing in your kernel, assuming that the firmware did everything perfectly.

After device initialization, the devices should be registered, i.e. to be added to the device list `devs`.

Since we're only using one line printer, you could probably register the line printer by setting up members in `lps[0]`, then add a pointer to `lps[0]` into the device list `devs`.

Framework design: coarse definition of `struct device`

The sections above stated that `struct device` is an abstract class, containing the common properties (for example, the physical address `phys_addr`) and methods (if any):

```
struct device {
        unsigned long   phys_addr;
        /* ... */
};
```

The following section is the original proposal made by myself. We'll analyze the proposal at the next section after that.

A proposal of implementing `write_char` **IMPORTANT:** this section is only a proposal, which is discarded for reasons in the next section. Those who don't care the reviewing process could as well skip to **Current adopted design** section.

Writing characters to line printer requires reading from or writing to device registers. An intuitive solution is to use an assignment statement to do so:

```
void msim_lp_write_char(struct lp *lp, unsigned char c)
{
        volatile unsigned char *reg;
        reg = (volatile unsigned char *)(lp->dev.phys_addr + MSIM_LP_OUT);
        *reg = c;
}
```

However, this is a *bad idea*, because there may exist other ways to access device registers than accessing memory-mapped registers. A typical example is `in` and `out` instructions on IA32/IA64 for accessing PCI device registers.

So we need to abstract the register writing procedure, that is, to think that "writing to a device register" is a *primitive*, or basic operation, of device accesses.

```
void msim_lp_write_char(struct lp *lp, unsigned char c)
{
        struct device *dev = &lp->dev;
        dev->writer.out8(dev, MSIM_LP_OUT, c);
}
```

We can coarsely define `struct device` to contain the following members:

```
struct device;

struct device_reader {
        /* The members can be null */
        uint8_t         (*in8)(struct device *, uint32_t);
        uint16_t        (*in16)(struct device *, uint32_t);
        uint32_t        (*in32)(struct device *, uint32_t);
        uint64_t        (*in64)(struct device *, uint32_t);
};

struct device_writer {
        void            (*out8)(struct device *, uint32_t, uint8_t);
        void            (*out16)(struct device *, uint32_t, uint16_t);
        void            (*out32)(struct device *, uint32_t, uint32_t);
        void            (*out64)(struct device *, uint32_t, uint64_t);
};

struct device {
        unsigned long           phys_addr;
        struct device_reader    reader;
        struct device_writer    writer;
        /* ... */
};
```

The reason I encapsulated the `inX` primitives in a `device_reader` structure is that we can implement a default implementation set of those `inX` primitives, such as by reading memory-mapped registers. When registering devices, we can simply copy the set to the `reader` member in the `device` structure.

Analysis of proposal above After discussion with some other TAs I found that this proposal is not so good as it seems.

Efficiency Most device register accesses could be implemented as a one-line procedure, for example, when writing to a 1-byte device register with physical address `addr`:

```
void out8(unsigned long addr, uint8_t data)
{
        /* phys_to_virt() converts physical address to virtual one */
        *(uint8_t *)phys_to_virt(addr) = data;
}
```

The function above could be inlined by the compiler

```
inline void out8(unsigned long addr, uint8_t data)
{
        /* phys_to_virt() converts physical address to virtual one */
        *(uint8_t *)phys_to_virt(addr) = data;
}
```

so that the compiler replaces the function call with the content inside if optimization is enabled.

By using function pointers, the possibilities of inlining such simple functions are essentially eliminated. Moreover, accessing a device register now requires several jumps. This is not preferable since device register accesses are so frequent that doing additional jumps for every access would become quite costly.

Wrong assumption about device register accesses Suppose, that an `lp` driver implemented under the framework above looks like:

```
void lp_write_char(struct lp *lp, unsigned char c)
{
        struct device *dev = &lp->dev;
        dev->writer.out8(dev, LP_OUT, c);
}

/* Possible implementation of out8() method */
void mem_out8(struct device *dev, uint32_t port, uint8_t data)
{
        *(uint8_t *)phys_to_virt(dev->phys_addr + port) = data;
}

/* Yet another possible implementation */
/* x86 PCI I/O port access */
void pciio_out8(struct device *dev, uint32_t port, uint8_t data)
{
        asm volatile ("out %0, %1" : /* no output */ : "a"(data), "d"(port));
}
```

The code above basically assumes that the `lp` device supports both memory access and PCI I/O access. In fact, this is usually not the case, and even such devices do exist, the operating system usually picks only one to use.

We should regard `lp` devices with memory access and another device with PCI I/O access as *different, separate* devices, each with its own driver implementation.

Current adopted design For reasons above, we take the simpler, more intuitive implementation of `write_char`:

```
void msim_lp_write_char(struct lp *lp, unsigned char c)
{
        volatile unsigned char *reg;
        reg = (volatile unsigned char *)(lp->dev.phys_addr + MSIM_LP_OUT);
        *reg = c;
}
```

The following code is more readable:

```
#include <asm/io.h>             /* for definition of write8() */

void msim_lp_write_char(struct lp *lp, unsigned char c)
{
        write8(MSIM_LP_REG(lp, MSIM_LP_OUT), c);
}
```

Device register access function naming conventions There are mainly three schemes of accessing device registers:

1. Via direct memory mapping. Device registers are mapped to fixed, machine-specific physical addresses.
   - ARM Zedboard, MSIM, and some of Loongson's devices include UART uses this scheme.
   - The function names are `read8`, `write8`, `read16`, `write16`, etc., up to 64 bit.
2. Via PCI memory access.
   - Loongson's VGA uses this, although we probably won't deal with VGA in our experiments.
   - The function names are `in8`, `out8`, `in16`, `out16`, etc., up to 64 bit.
3. Via PCI I/O access. On x86, this calls for `in` and `out` instructions.
   - Most hardware on QEMU/i386, and IDE disk controller, as well as RTC on Loongson uses this.
   - The function names are `inb`, `outb`, `inw`, `outw`, `inl`, `outl`, `inq`, `outq`, each pair corresponding to reading/writing 8/16/32/64 bit.

Paper exercise Since we used an object-oriented programming scheme here, why don't we directly use C++ to code the kernel? Or, why the Unix-like kernel developers all choose C, rather than C++, for coding?

Programming exercise

1. Implement `write8()`, `write16()`, `write32()` and their `read` cousins. Put them in appropriate header file(s).
2. Implement the macro `MSIM_LP_REG`, and put it in appropriate header file(s) or source file(s).

Demo: continuing framework implementation

Suppose that we've already defined a primitive, `write_char`, for all line printers:

```
struct lp {
        void            (*write_char)(struct lp *, unsigned char);
};
```

As we mentioned above, a line printer is a device:

```
struct device {
        /* ... */
};

struct lp {
        struct device   dev;
        void            (*write_char)(struct lp *, unsigned char);
};
```

`struct device` stores basic information of a device, for example, physical address, as mentioned above:

```
struct device {
        unsigned long   phys_addr;
        /* Other information here... */
};

struct lp {
        struct device   dev;
        void            (*write_char)(struct lp *, unsigned char);
};
```

We had already implemented the `write_char` primitive for MSIM line printer:

```
void msim_lp_write_char(struct lp *lp, unsigned char c)
{
        write8(MSIM_LP_REG(lp, MSIM_LP_OUT), c);
}
```

Before emitting characters, we should initialize it, and register it into the device list, along with its `write_char` implementation.

```
void msim_lp_init(unsigned long paddr)
{
        /*
         * Nothing needed for device, though.
         * So we only need to register the device with its physical address.
         */
        msim_lp_register(paddr);
}
```

We should define a list of devices. After implementing dynamic memory allocation we'll use a linked list for the device list, but here we're going to define a line printer structure array and a device pointer array instead:

```
struct device *devs[MAXDEVS];   /* device pointer list */
int ndevs;                      /* number of devices */
struct lp lps[NR_LPS];          /* list of line printers */
```

Then, we register the device into the device pointer list:

```
void device_register(struct device *dev)
{
        devs[ndevs++] = dev;
}
```

And line printer to the line printer list:

```
void msim_lp_register(unsigned long paddr)
{
        struct device *dev = &lps[0].dev;

        dev->phys_addr = paddr;
        dev->type = DEV_LP;                     /* device type */

        lps[0].write_char = msim_lp_write_char; /* our implementation */

        device_register(dev);                   /* register to device list */
}
```

Finally, you should add the `msim_lp_init()` call into the `main()` routine in kernel. On x86 systems it's done without hardwiring by probing PCI configuration space, but ARM and MIPS board involves more-or-less hardwiring anyway.

Nevertheless, you should hardwire "flexibly", by splitting generic initialization and machine-specific initialization, thereby supporting multiple machines by macro switches:

```
/* This is only a demonstration, the actual code may be different */
void main(void)
{
        /* ... */
        mach_init();
}

/* Meanwhile, in machine-specific source file... */
void mach_init(void)
{
        msim_lp_init(MSIM_LP_BASE);
}
```

Then, we're focusing on implementing `kputs()`, by traversing the device list, checking whether it supports line output, and invoke their implementation of `write_char`, thereby supporting heterogeneous devices:

```
/* The following code will be changed after dynamic memory allocation */
void kputs(char *s)
{
        unsigned int i;
        struct lp *lp;

        for (i = 0; i < ndevs; ++i) {
                /*
                 * Check whether the device supports line printing, by
                 * verifying device type.
                 */
                if (dev_support_lp(devs[i])) {
                        /*
                         * From the code above we can see that the pointer
                         * to device structure in "devs" actually points
                         * to a "struct lp".
                         * This is a kind of inheritance, in terms of
                         * object-oriented programming.
                         */
                        lp = (struct lp *)devs[i];
                        lp_puts(lp, s);
                }
        }
}
```

```c
/* Probably in somewhere else, not necessarily in the same file */
void lp_puts(struct lp *lp, char *s)
{
        for (; *s != '\0'; ++s) {
                lp->write_char(*s);
        }
}
```

Organizing kernel source files

As we mainly focus on kernel development in our labs, how to organize kernel source code is also important.

Not only should all the source files be placed into `kern` directory, but also, the source files should be further divided into architecture-specific, and architecture-independent code. Both would in turn be divided according to modules, sub-modules, etc.

One should always not to write a lot of things inside one C source code file. Think of C source files as small, basic modules. Write C source code files to deal with one thing, and do that well. Gather similarly-functioned, or closely-related modules together inside one directory, combining the directories in a higher-level directory, repeat the process, and build a reasonable source code hierarchy.

A scheme suggestion

All C source code files should take no more than 500 lines. If your file takes more, consider splitting it into smaller modules in separate files.

Architecture-specific source codes should be put inside `kern/arch/mips` directory, as we're developing a kernel running on MIPS. Later, the ARM and IA32 developers can contribute their architecture-specific code into `kern/arch/armv7a-le` and `kern/arch/i386`.

You probably need to divide the code in `kern/arch/mips` directory into smaller modules, sub-modules, etc.

Drivers should be put inside `kern/drivers` directory. Each driver type needs a sub-directory.

Architecture-independent kernel initialization code should be put inside `kern/init` directory.

Programming exercise

1. Refine and fill up the framework. Add, change or remove properties or methods by your choice. The framework design above is only a suggestion, and you can make your own modifications to suit your flavor. Your framework should still be scalable, though.
    - This is a demanding task. Do not hesitate to ask teammates, other developers, or your TA for help & reviews.
2. Implement the kernel logic to print hello world message.
3. (Optional) Implement the kernel logic to detect *CPU cycles per second*. As your code would need to access the RTC in MSIM, design the framework for RTC drivers.
    - Think of scalability, define the primitives, and give your implementation.
    - `CP0_COUNT` register is one of the stuff you need.
    - Sooner or later you'll have to implement this when we're implementing `sleep(2)` system call.
4. Write a linker script to link the object files.
    - The linker script is much simpler than that of firmware, as the section addresses and order won't matter that much anymore, and you won't need to relocate `.data` and `.bss` sections.
5. Write a Makefile for automatically build the kernel.
6. Make changes to your top-level Makefile, so as to build and install the kernel executable, along with bootloader and firmware.
7. Write or change the `install` target in top-level Makefile to install the bootloader and kernel into the disk image. You probably want to make changes in sub-directory Makefiles.
    - A Perl script `tools/install-kernel.pl` is provided for installing the kernel onto a partition of a device/disk image. Execute the script with no arguments to see the usage.
    - Better solution without using the Perl script is available. *Figure this out and earn your bonus!*
8. Write or change the `msim` target in top-level Makefile to start MSIM with the disk image loaded. You should receive the Hello World message.

Turning on optimization: issues again

If we add an optimization flag in the compiler flags:

```
CFLAGS          = <...> -O2
```

Remake the project, and you'll see the firmware crashes.

Programming exercise (optional) If you decide to turn on optimization at some point. Try to figure out what's wrong with the binary.

1. Use `xxd(1)` or other tools to figure out what's missing.
2. Use `readelf(1)` to see which sections GCC has generated now.
3. Modify your Makefile and/or linker script to ensure that nothing is missed.

Weekly Schedule

Milestones:

1. Booting kernel
2. SMP spinup & spinlocks
3. Trap handling
4. Physical memory allocator (Buddy system or First-fit, allocating pages)
5. Virtual memory allocator (SLAB, allocating bytes)
6. Kernel thread management: `fork(2)`, `exit(2)`, `wait(2)`, `kill(2)`
    - Include basic signal handling
7. Userspace thread management, context switches: `exec`
    - No file system involved
8. Scheduler
9. File system: simplified UFS
    - Optional challenge: implement the actual UFS
10. (Optional) Migrate to Loongson using existing codebase
    - Ideally, your existing codebase should change only a little
    - Your work should focus on developing drivers then
11. (Optional) Merge your codebase with other groups' work

We have around 11 weeks. So preferably one should complete one milestone per week.

License

We use GNU Public License v2 in our project.

Include the following license at the top of your source files:

> Copyright (C) {year} {your-name} {your-personal-email}
> {more-copyrights-and-names}
>
> This program is free software; you can redistribute it and/or modify it
> under the terms of the GNU General Public License as published by the
> Free Software Foundation; either version 2 of the License, or (at your option) any later version.

If you borrowed open source implementations elsewhere, put their license before the GPL license above if they are not GPL, or add the contributors before your name if they're GPL.

Notes

Code collaborations are allowed. Just indicate the contributors and their contributions in source code as comments.

- Code contribution is one of the most important factor affecting your grading.
- Evaluating code contribution is mainly subjective, as it's rather hard to develop an actual metric. Here, I'll give grades according to
  - Significance of code contribution
  - Code quality
  - Quality of documentation and comments
  - Contribution size
  - Number of contributors
  - Bonuses below
  - Innovations (they must work, though, or they may backfire)
  - so on so forth...

Always ask TAs for whatever help you can think of (!)

Bonuses

Requests of OS-irrelevant implementations are allowed. If you think some functions or procedures are boring, and not related to operating system design and implementation (for example, C library functions such as `memcpy`, `memset`, `snprintf` etc.), you can inform me and I'll give you the implementation if I confirm that it's indeed OS-irrelevant.

- *Good requests earn you bonuses*, as they show your skill of discriminating code responsibilities, which is important in software development as a system architect or project manager.
- ~~This bonus is not so easy to earn as I usually give you the boring code right at the start.~~

Framework design challenges are allowed. If you have a different, better framework than the one inside the tutorial (and later documents), email me and *you may earn bonus after thorough discussion and confirmation.*

Skeleton requests are allowed. If you have difficulty writing operating system from scratch, you can email me to request a workspace with skeleton code to work with.

- You can send me your workspace (if any) so I can help by giving minor hints or suggestions rather than sending you source files. *This affect your bonuses* (see below).
- *You may earn bonus if you completed your milestone without requested skeleton source files.*
- *You may earn smaller bonus if you completed your milestone with minor skeleton source files.*