CS334-OS-Project

> 12110817 张展玮
>
> 12110813 刘圣鼎
>
> 12110714 谢嘉楠

How to port Linux

modify Makefile.header

```
TARGET := x86_64
UNAME := $(shell uname)
ifeq ($(TARGET), x86_64)
    ifeq ($(UNAME), Linux)
    AS  = as --64
    LD  = ld
    LDFLAGS = -m elf_x86_64
    CC  = gcc
    CFLAGS  = -g -m64 -fno-builtin -fno-stack-protector -fomit-frame-pointer -
fstrength-reduce -D__X64__ #-Wall

    CPP = cpp -nostdinc -D__X64__
    AR  = ar
    STRIP = strip
    OBJCOPY = objcopy
    # we should use -fno-stack-protector with gcc 4.3
    gcc_version=$(shell ls -l `which gcc` | tr '-' '\n' | tail -1)
    endif
else
    ifeq ($(UNAME), Linux)
        AS  = as --32
        LD  = ld
        #LDFLAGS = -m elf_i386 -x
        LDFLAGS = -m elf_i386
        CC  = gcc
        CFLAGS  = -g -m32 -fno-builtin -fno-stack-protector -fomit-frame-pointer
-fstrength-reduce -D__X86__ #-Wall

        CPP = cpp -nostdinc -D__X86__
        AR  = ar
        STRIP = strip
        OBJCOPY = objcopy
        # we should use -fno-stack-protector with gcc 4.3
        gcc_version=$(shell ls -l `which gcc` | tr '-' '\n' | tail -1)
    endif
```

The target platform (x86 or x86_64) is first specified via the TARGET variable. Then use the `$(shell)` function to get the current system information, stored in the UNAME variable.

Next, set the corresponding compiler variables according to the different values of the TARGET and UNAME variables.

When TARGET is x86_64, AS, LD, CC, CPP, AR, STRIP and OBJCOPY variables are set according to the UNAME system for Linux; When the TARGET is x86, the corresponding variable is set according to the UNAME system for Linux.

After setting these compiler variables, some compiler parameters and flags (such as compiler options -g, -fno-stack-protector, and so on) are set in turn. Finally, different gcc versions will be set according to different UNAME variables.

Modify Makefile

source:

```
LDFLAGS += -Ttext 0 -e startup_32
CFLAGS  += $(RAMDISK) -Iinclude
CPP += -Iinclude
```

modified code:

```
# TODO64
ifeq ($(TARGET), x86)
    LDFLAGS += -Ttext 0 -e startup_32
    CFLAGS  += $(RAMDISK) -Iinclude
    CPP     += -Iinclude
else
    LDFLAGS += -T system64.ld -e startup_64
    CFLAGS  += $(RAMDISK) -Iinclude
    CPP     += -Iinclude
endif
```

This code is a conditional statement directive that makes conditional judgments based on the value of the `TARGET` variable. If the value of the variable `TARGET` is equal to `x86`, the code block after if is executed; If not equal to `x86`, the code block after else is executed.

In the if block, the value of the LDFLAGS variable is set to `-Ttext 0 -e startup_32`, which is a connector option for 32-bit architectures. THE CFLAGS VARIABLE IS SET TO $(RAMDISK) -Iinclude, which represents the compiler flag, where $(RAMDISK) represents the RAM DISK specified by the link; The CPP variable is set to -Iinclude, which indicates that the C++ compiler searches for header file paths.

In the else block, the value of the LDFLAGS variable is set to `-T system64.ld -e startup_64`, which is a connector option for 64-bit architectures. The values of the CFLAGS and CPP variables are the same as the values in the if block.

The purpose of this code is to set different connector options for different types of architectures (32-bit or 64-bit) for proper linking. This is a common technique in operating system kernel development. The use of variables makes it easy to switch between different architectures, improving code maintainability and extensibility.

source:

```
start:
	@qemu -m 16M -boot a -fda Image -hda $(HDA_IMG)

bochs-start:
	@$(BOCHS) -q -f tools/bochs/bochsrc/bochsrc-hd.bxrc

debug:
	@qemu -m 16M -boot a -fda Image -hda $(HDA_IMG) -s -S -nographic -serial
'/dev/ttyS0'
```

modified code:

```
start:
ifeq ($(TARGET), x86)
	@qemu-system-i386 -m 16M -boot a -drive format=raw,file=Image,if=floppy -
drive format=raw,file=hdc-0.11.img,index=0,media=disk
else
	@qemu-system-x86_64 -m 4G -boot a -drive format=raw,file=Image,if=floppy -
drive format=raw,file=hdc-0.11.img,index=0,media=disk
endif

debug:
ifeq ($(TARGET), x86)
	@qemu-system-i386 -m 16M -boot a -drive format=raw,file=Image,if=floppy -
drive format=raw,file=hdc-0.11.img,index=0,media=disk -s -S
else
	@qemu-system-x86_64 -m 4G -boot a -drive format=raw,file=Image,if=floppy -
drive format=raw,file=hdc-0.11.img,index=0,media=disk -s -S
endif
```

This code is a Makefile rule that starts the virtual machine to run the operating system kernel.

In the Makefile rule, `start` and `debug` are the names of the directives, which can be run separately using the `make start` or `make debug` commands. These instructions are used to start the operating system kernel in a virtual machine.

`ifeq ($(TARGET), x86)` in the directive determines whether the `TARGET` variable is equal to `x86`, and if so, executes the qemu-system-i386 command in the if block to start the 32-bit virtual machine; Otherwise, execute the qemu-system-x86_64 command in the else block to start the 64-bit virtual machine. The qemu-system-i386 command and the qemu-system-x86_64 command are used here to start the corresponding virtual machine, the `-m` flag is used to set the virtual machine memory size, `-boot a` is used to make the virtual machine boot from the floppy disk, and the `-drive` is used to add virtual disks to the virtual machine, the Image file and the hdc-0.11.img file are added here; the `-s` and `-s` commands are used to connect GDB when the virtual machine starts Debugger.

The `start` directive is used to start a healthy virtual machine, and the `debug` directive is used to start the virtual machine and connect the GDB debugger for debugging.

The purpose of this code is to start the corresponding virtual machine on a different architecture (32-bit or 64-bit) and connect a GDB debugger to the virtual machine for debugging and testing by operating system kernel developers.

Added code:

```
lldb-as:
    @echo "gdb 1234\ncommand script import ./tools/lldb_helper.py" > .lldbinit
    @lldb --local-lldbinit

lldb-src:
    @echo "target create ./tools/system\ngdb 1234" > .lldbinit
    @lldb --local-lldbinit
```

This code is a Makefile rule that starts the LLDB debugger.

'lldb-as' and 'lldb-src' in the directive are the names of the directives, which can be run using the `make lldb-as` or `make lldb-src` commands, respectively. These instructions are mainly used to debug assembly and source code of the operating system kernel during development.

In the 'lldb-as' directive, use the `@lldb --local-lldbinit` command to start the LLDB debugger and load the `local-lldbinit` script during startup. This script is used to initialize LLDB and set up the assembly debugging environment for the operating system kernel.

In the `lldb-src` directive, use the `@lldb --local-lldbinit tools/system` command to start the LLDB debugger and load the `local-lldbinit` script in the `tools/system` directory during startup. This script is used to initialize LLDB and set up the source debugging environment of the operating system kernel.

The purpose of this code is to start the LLDB debugger and provide a convenient debugging environment for developers to quickly identify and fix problems at the assembly or source level.

Added code:

```
.PHONY: all clean info distclean backup start debug lldb-as lldb-src help sofar
sofard sofar_image
```

This code defines some pseudo-targets (PHONY) that can be followed by running the `make`command. Specifically:

- `all`: The default target, which performs all operations, compiles and links all files.
- `clean`: Delete all generated target files and executables.
- `info`: Displays some information related to this system.
- `distclean`: Delete all generated target files, executables, and backup files.
- `backup`: Create a backup file.
- `start`: Start the operating system.
- `debug`: Use gdb to debug the executable.
- `lldb-as`: Start the lldb debugger in assembly mode.
- `lldb-src`: Start the lldb debugger in source mode.
- `help`: List all available phoney targets.
- `sofar`: Compile and run the current system.
- `sofard`: Compile and run the current system in debug mode.
- `sofar_image`: Write the operating system to a disk image file for loading in the virtual machine

Added code:

```
sofar:
@make clean
@make sofar_image

sofard:
@make sofar
@make debug

sofar_image: init/main.o
@make -C boot
@make head.o -C boot/
@$(LD) $(LDFLAGS) -o tools/system boot/head.o init/main.o
@nm tools/system | grep -v '(compiled)|(.o$$)|( [aU] )|(..ng$$)|(LASH[RL]DI)'|
sort > System.map
@cp -f tools/system system.tmp
@$(STRIP) system.tmp
@$(OBJCOPY) -O binary -R .note -R .comment system.tmp tools/kernel
@tools/build.sh boot/bootsect boot/setup tools/kernel Image $(ROOT_DEV)
@rm system.tmp
@rm -f tools/kernel
@sync
```

This code defines three goals, where `sofar` and `sofard` are pseudo-targets, while `sofar_image`is a real goal.

- `sofar`: First call the `make clean` command to clean all generated object files and executables, and then call the `make sofar_image` command to compile and write the operating system to the disk image file.

- `sofard`: Build the operating system by calling the `make sofar` command to compile and link all files, and then start the system by running the executable in debug mode.
- `sofar_image`: Compile the code and write the operating system to a disk image file. First compile the code in the boot directory with `make -C boot`, then compile the `init/main.o` file, and use the compiler to link the compiled `head.o` and `init/main.o` files into an executable `tools/system`. Next, use the `nm` command to print out the executable, filter out unnecessary output, and output it to a `System.map` file. Then, copy an uncompressed executable file into a temporary file, use the `strip` command to get a compressed version, output it to a binary file via the `objcopy` command, and finally write it to a disk image file.

boot directory

Modify Makefile

```
include ../Makefile.header

LDFLAGS += -Ttext 0

all: bootsect setup

bootsect: bootsect.s
	@if [ "${TARGET}" = "x86" ]; then\
		$(AS) -o bootsect.o bootsect.s;\
		$(LD) $(LDFLAGS) -o bootsect bootsect.o;\
	else\
		as --32 -o bootsect.o bootsect64.s;\
		ld -m elf_i386 -Ttext 0 -o bootsect bootsect.o;\
	fi
	@$(OBJCOPY) -R .pdr -R .comment -R.note -S -O binary bootsect



# NOTE for target x86_64: we are still in 32-bit protected mode in setup64.s.
# So assemble and link it in 32 mode.
setup: setup.s setup64.s
	@if [ "${TARGET}" = "x86" ]; then\
		$(AS) -o setup.o setup.s;\
		$(LD) $(LDFLAGS) -o setup setup.o;\
	else\
		as --32 -o setup.o setup64.s;\
		ld -m elf_i386 -Ttext 0 -o setup setup.o;\
	fi
	@$(OBJCOPY) -R .pdr -R .comment -R.note -S -O binary setup

head.o: head.s head64.s
	@if [ "${TARGET}" = "x86" ]; then\
		$(AS) -o head.o head.s;\
	else\
		$(AS) -o head.o head64.s;\
	fi
clean:
	@rm -f bootsect bootsect.o setup setup.o head.o
```

The Makefile file under boot is used to compile the boot sector and install the program. We have added `bootsect64.s` and `setup64.s` assembly code files to the original `bootsect.s` and `setup.s` files.

Through conditional compilation in makefile, select the appropriate assembler and linker on both x86 and x64 platforms to process different code files and generate corresponding executable object files.

The all target will compile the bootsect and setup targets, both of which are generated by the compilation of assembly code files, where bootsect is required for the boot sector and setup is required for the installer.

In the `bootsect` target, use conditional compilation to select the appropriate assembler and linker, assemble `bootsect.s` into a relocatable object file `bootsect.o`, compile `bootsect.s` and `bootsect64.s` files into corresponding object files, and then link them into bootsect executable object files. Finally, the objcopy tool is used to convert the bootsect executable object file into a binary program and use it to generate a disk image file.

The `setup`target is similar to the `bootsect`target. Select the correct assembler and linker according to the target platform, compile the `setup.s` and `setup64.s` files into corresponding object files, and then link them into setup executable object files, and convert them into binary programs using the objcopy tool.

`head.o` is a header file in 32-bit protected mode that is linked into a kernel image along with the boot sector and installer. Some start address and kernel information are defined so that other program information can be obtained, such as the size of the kernel and the entry point.

The `clean` target cleans up the generated target files and executables.

head.s VS head64.s

```
pg_dir:
PML4:
    .quad 0x101007  # PDPT_0
    .fill 255,8,0
    .quad 0x101007
    .fill 255,8,0
PDPT_0: # Not all processors that support IA-32e paging support 1 pages
    .quad 0x102007 # PD_0
    .rept 0x1ff
        .quad 0
    .endr
PD_0:
    .set i, 0
    .rept 64        # 64*2=128MB
        .quad (i << 21)+0x87
        .set i, (i+1)
    .endr
    .rept 512-64
        .quad 0
    .endr

.org 0x5000
```

```
.globl startup_64
startup_64:
    movl $0x10,%eax
    mov %ax,%ds
    mov %ax,%es
    mov %ax,%fs
    mov %ax,%gs
    mov %ax,%ss
    mov stack_start(%rip),%rsp

    mov turn_HHK(%rip), %rax
    pushq $0x08
    pushq %rax
    lretq

turn_HHK:
 .quad post_HHK # 0xFFFF8000XXXXXXXX after ld

post_HHK:
    movq $0, PML4(%rip)
resetup_gdt:
    lgdt gdt_descr(%rip)
    movl $0x10,%eax
    mov %ax,%ds
    mov %ax,%es
    mov %ax,%fs
    mov %ax,%gs
    mov %ax,%ss
    mov stack_start(%rip),%rsp
```

1. Introduction of new page table structures: In the old code, the page table structures used were for a 32-bit system. However, in the new code, new page table structures are introduced, namely PML4 (Page Map Level 4), PDPT_0 (Page Directory Pointer Table), and PD_0 (Page Directory). These new table structures are critical components for managing page tables in a 64-bit system. By introducing the new page table structures, support for larger physical memory spaces is enabled, allowing the system to correctly address and map memory.
2. Modification of stack setup: There are some differences in stack setup between 32-bit and 64-bit systems. In the old code, 32-bit registers and instructions were used to set up the stack. In the new code, 64-bit registers and instructions are used to set up the stack. This is because in a 64-bit system, using wider registers and instructions better accommodates stack operations in 64-bit mode.
3. Reloading the Global Descriptor Table (GDT) and segment registers: In the old code, segment registers' values were loaded based on the needs of the 32-bit system. In the new code, it is necessary to reload the Global Descriptor Table (GDT) and segment registers to ensure that after switching from 32-bit mode to 64-bit mode, the segment registers have the correct values. This ensures that the system can properly access and protect different segments and supports memory management in 64-bit mode.

These modifications are made to adapt the migration from a 32-bit system to a 64-bit system. By introducing new page table structures, modifying stack setup, and reloading the GDT and segment registers, the system can run properly in 64-bit mode and take full advantage of the larger physical memory space. These modifications ensure that the system can operate correctly on the new architecture and utilize the features and advantages of a 64-bit system effectively.

```
setup_idt:
    lea ignore_int(%rip),%rdx
    movl $0x00080000,%eax /* selector = 0x0008 = cs */
    movw %dx,%ax
    movw $0x8E00,%dx    /* interrupt gate - dpl=0, present */
    movq $0x00000000FFFF8000, %rbx
    lea idt(%rip), %rdi
    mov $256,%ecx
rp_sidt:
    movl %eax,(%rdi)
    movl %edx,4(%rdi)
    movq %rbx,8(%rdi)
# movl %ebx,12(%rdi)
    addq $16,%rdi
    dec %ecx
    jne rp_sidt
    lidt idt_descr(%rip)

setup_tss:
    leaq tss_table(%rip), %rdx
    xorq %rax, %rax
    xorq %rcx, %rcx
    movq $0x89, %rax # present, 64bit TSS
    shlq $40, %rax
    movl %edx, %ecx
    shrl $24, %ecx
    shlq $56, %rcx
    addq %rcx, %rax
    xorq %rcx, %rcx
    movl %edx, %ecx
    andl $0xffffff, %ecx
    shlq $16, %rcx
    addq %rcx, %rax
    addq $0x67, %rax # limit
    leaq gdt(%rip), %rdi
    movq %rax, 24(%rdi) # gdt_idx = 3
    shrq $32, %rdx
    movq %rdx, 32(%rdi)
# load tss
    mov $0x18, %ax # 11000b, gdt_idx = 3
    ltr %ax

# long return to main
    lea main(%rip), %rax
    pushq   $0x08
    pushq %rax
    lretq
```

1. Modification in `setup_idt`:
    - The original code used 32-bit registers `%edx` and `%edi` to manipulate the IDT (Interrupt Descriptor Table) and load it using 32-bit instructions.
    - The modified code uses 64-bit registers `%rdx` and `%rdi` to manipulate the IDT and load it using 64-bit instructions.
    - The new code also adds setting of the attributes field for each gate in the IDT (using 64-bit register `%rbx`), providing more precise definition of the characteristics of the interrupt gates.
2. Modification in `setup_tss`:
    - The new code introduces a new function `setup_tss` for setting up the TSS (Task State Segment).
    - The modified code uses 64-bit registers `%rdx` and `%rdi` to manipulate the TSS and load it using 64-bit instructions.
    - The new code calculates and sets various fields of the TSS, including the type, segment limit, base address, etc.
    - Finally, the TSS is loaded into the task register TR using the `ltr` instruction.

```
.align 4
.word 0
idt_descr:
    .word 256*16-1      # idt contains 256 entries
    .quad idt

.align 4
.word 0
gdt_descr:
    .word 256*8-1       # so does gdt (not that that's any
    .quad gdt                   # magic number, but it works for me :^)

# In 64-bit processor, an entry in idt is 16B long.
idt:    .fill 256*2,8,0     # idt is uninitialized

gdt:
    .quad   0                       # dummy
    .long 0, 0x00209a00 # code readable in long mode
    .long 0, 0x00209200 # data readable in long mode
    .fill 504,8,0            # space for LDT's and TSS's etc (252*2=504)

tss_table:
    .fill 26,4,0 # 26 * 4 = 104 (64bit TSS)

stack_start:
    .quad 0xFFFF800000200000
```

1. Alignment modification: Changed `".align 2"` to `".align 4"` to ensure data alignment to a 4-byte boundary. This is done to comply with the memory alignment requirements of 64-bit systems.
2. Descriptor size modification: Changed `".word 256*8-1"` to `".word 256*16-1"` to accommodate the 16-byte size of each IDT (Interrupt Descriptor Table) entry in a 64-bit

system. Similarly, the descriptor size of the GDT (Global Descriptor Table) was modified accordingly.

3. IDT and GDT padding modification: Changed `".fill 256,8,0"` to `".fill 256*2,8,0"` to ensure that the padding size of the IDT matches the modified size of each entry.
4. Other padding modifications: Corresponding modifications were made to the padding size of tables such as GDT and TSS to accommodate a 64-bit system.

lib/string.c

For `lib/string.c`, navigate to the `lib` directory and run `make` to enter the compilation process. By analyzing the error messages, we can identify the areas that require modification.

For 64-bit systems, we introduced macro definitions and corresponding instructions to ensure compatibility with the 64-bit instruction set. During the modification process, certain instructions used in 32-bit systems were replaced to meet the requirements of the 64-bit system. For example, the `movl` instruction was replaced with the `movq` instruction, and the `decl` instruction was replaced with the `movq` instruction.

Modify the file keyboard.S

```
#ifdef __X86__
/* these are the offsets into the read/write buffer structures */
rs_addr = 0
head = 4
tail = 8
proc_list = 12
buf = 16
#elif __X64__
rs_addr = 0
head = 4
tail = 8
proc_list = 16
buf = 20
#enfif
```

This code defines the offsets of some variables. The variable offsets are different for different architectures (X86 and X64). If the compiler detects that the X86 architecture is being used, then the offset of rsaddr is 0, the offset of head is 4, the offset of tail is 8, the offset of proc_list is 12, and the offset of buf is 16. If the compiler detects that the X64 architecture is being used, then the offset of rs_addr is 0, the offset of head is 4, the offset of tail is 8, the offset of proc_list is 16, and the offset of buf is 20. These offsets can be used to access the values of the corresponding variables in a structure.

```
keyboard_interrupt:
#ifdef X64
push %rax
push %rbx # TODO64: we can remove %rbx if we use %rax to replace %ebx:%eax
push %rcx
push %rdx
push %rdi
push %rsi
push %r8
push %r9
push %r10
push %r11
#elif X86
push %eax
push %ebx
push %ecx
push %edx
push %ds
push %es
```

This code is a keyboard interrupt handling function that uses different instructions for processing depending on the target architecture (X64 or X86).

The basic idea of the code is to push the CPU's context information (register contents and status flags, etc.) onto the stack to save the state before the interrupt occurred. Specifically:

- If the compilation target is the X64 architecture, the values of the registers %rax, %rbx, %rcx, %rdx, %rdi, %rsi, %r8, %r9, %r10, and %r11 are pushed onto the stack in order. This part of the code uses the "push" instruction to push the values of the respective registers onto the stack.
- If the compilation target is the X86 architecture, the values of the registers %eax, %ebx, %ecx, %edx, %ds, and %es, are pushed onto the stack in order. This part of the code similarly uses the "push" instruction to push the values of the respective registers onto the stack.

Thus, when the interrupt processing is completed, the saved context information can be popped from the stack to restore the CPU state before the interrupt occurred, and continue executing the code after the interrupt processing.

```
#ifdef __X86__
    pushl $0
    call do_tty_interrupt
    addl $4,%esp
    pop %es
    pop %ds
    popl %edx
    popl %ecx
    popl %ebx
    popl %eax
#elif __X64__
    pushq $0
    call do_tty_interrupt
    addq $8,%rsp
    pop %r11
    pop %r10
    pop %r9
    pop %r8
    pop %rsi
    pop %rdi
    pop %rdx
    pop %rcx
    pop %rbx
    pop %rax
#endif
```

This code is a tty interrupt handling function that uses different instructions for processing depending on the target architecture during compilation.

Specifically, if the compilation target is the X86 architecture, the function pushes the parameter 0 onto the stack, calls the do_tty_interrupt function, and then restores the values of the %es, %ds, %edx, %ecx, %ebx, and %eax registers in order by poping them from the stack.
If the compilation target is the X64 architecture, the function pushes the parameter 0 onto the stack, calls the do_tty_interrupt function, and then restores the values of the %r11, %r10, %r9, %r8, %rsi, %rdi, %rdx, %rcx, %rbx, and %rax registers in order by poping them from the stack.

Thus, the function hands over the control of keyboard input interrupt processing to the do_tty_interrupt function, and restores the state of the registers from the stack before continuing the execution from the interrupt processing location after the call returns.

```
put_queue:
#ifdef __X86__
    push %ecx
    push %edx
#elif __X64__
    #push %rcx
    #push %rdx
#endif
```

This code is a conditionally compiled block of code that performs different operations depending on the target architecture of the compilation machine.

Specifically:

- If the compilation target is the X86 architecture, the code pushes the values of the %ecx and %edx registers onto the stack, saving their current values.
- If the compilation target is the X64 architecture, the code comments out the two push instructions using the '#' symbol to avoid generating unrecognizable code.

This code block should be an important step in pushing character data into the system queue, but specific implementation details may need to be supplemented by other parts of the code.

```
#elif __X64__
    push %rax
    #push %rdi
    #push %rsi
    #push %rdx
    push %rcx
    #push %r8
    #push %r9
    #push %r10
    #push %r11
    # call show_stat # TODO64
    #pop %r11
    #pop %r10
    #pop %r9
    #pop %r8
    pop %rcx
    #pop %rdx
    #pop %rsi
    #pop %rdi
    pop %rax
#endif
```

This code segment is processed with conditional compilation instructions. When the macro definition **X64** is defined, the code between #elif and #endif will be compiled; otherwise, this code segment will be ignored.

This code segment is implemented using assembly language, and its main purpose is to save the values of registers in the stack, call the show_stat function, and store the return value in the register. Then, it restores the values of the registers from the saved values in the stack.

Specifically, this code uses the push instruction to save the values of registers %rax and %rcx in the stack. Before calling the show_stat function, the push instruction, which is commented out, saves the values of the remaining registers in the stack. The commented call instruction then calls the show_stat function, for which the implementation is not provided here. Finally, the pop instruction restores the values of the registers from the saved values in the stack.

In conclusion, the above assembly code is an interrupt handler for the keyboard driver in the Linux kernel. Its main purpose is to read the keyboard scan codes and convert them into corresponding ASCII characters. During the processing, it adjusts the keyboard's function modes, such as SHIFT, CAPS LOCK, or NUM LOCK, and the LED light states.

There are some preprocessor directives in the code that set variable values based on the different architectures, such as 'X86' or 'X64'. The code also uses some assembly instructions, such as 'push', 'pop', 'ret', and 'call', which respectively represent pushing registers onto the stack, popping registers off the stack, returning from a function, and calling a function. Some of these instructions also have conditional jump instructions, such as addition comparison using the 'cmp' instruction.

The code also contains some labels, such as 'caps', 'leds', 'ctrl', etc., which represent state variables and keyboard key values. By testing these variables, the program can detect whether a certain mode is on (e.g. whether the CAPS LOCK key is pressed) or when to trigger certain operations when a specific key is pressed (e.g. CTRL-ALT-DEL for reboot).

In summary, this program is an important kernel component that enables the Linux kernel to properly respond to keyboard inputs.

Modify the file rs_io.S

```
#ifdef __X86__
/* these are the offsets into the read/write buffer structures */
rs_addr = 0
head = 4
tail = 8
proc_list = 12
buf = 16
#elif __X64__
rs_addr = 0
head = 4
```

```
tail = 8
proc_list = 16
buf = 20
#enfif
```

This code defines the offsets of some variables. The variable offsets are different for different architectures (X86 and X64). If the compiler detects that the X86 architecture is being used, then the offset of rsaddr is 0, the offset of head is 4, the offset of tail is 8, the offset of proc_list is 12, and the offset of buf is 16. If the compiler detects that the X64 architecture is being used, then the offset of rs_addr is 0, the offset of head is 4, the offset of tail is 8, the offset of proc_list is 16, and the offset of buf is 20. These offsets can be used to access the values of the corresponding variables in a structure.

```
#ifdef __X86__
    pushl $table_list+tail
    jmp rs_int
.align 2
rs2_interrupt:
    pushl $table_list+buf
rs_int:
#ifdef __X86__
    pushl %edx
    pushl %ecx
    pushl %ebx
    pushl %eax
    push %es
    push %ds        /* as this is an interrupt, we cannot */
    mov $0x10, %ax      /* know that bs is ok. Load it */
    mov %ax, %ds
    mov %ax, %es
#elif __X64__
    pushq %rdx
    pushq %rcx
    pushq %rbx
    pushq %rax
    pushq %rdi
    pushq %rsi
    pushq %r8
    pushq %r9
    pushq %r10
    pushq %r11
#endif
```

This code implements the saving and setting of register states during interrupt handling on different architectures (X86 and X64). Specifically:

- For the X86 architecture, the "pushl" instruction is used to push registers eax, ebx, ecx, edx as well as data segment register ds and additional segment register es onto the stack to save their values. Then, the "mov" instruction is used to set the values of data segment register ds and es to 0x10, allowing the kernel data segment to be used. Finally, the code jumps to either rs_int or rs2_interrupt for further processing.
- For the X64 architecture, the "pushq" instruction is used to push registers rax, rbx, rcx, rdx, rsi, rdi, r8, r9, r10, r11 onto the stack to save their values. Then, the code jumps to rs_int for

further processing.

In short, the purpose of this code is to save register states during interrupt handling to ensure that the program can correctly restore the context after interrupt processing is completed.

```
#ifdef __X86__
    pop %ds
    pop %es
    popl %eax
    popl %ebx
    popl %ecx
    popl %edx
    addl $4,%esp        # jump over _table_list entry
#elif __X64__
    pop %r11
    pop %r10
    pop %r9
    pop %r8
    pop %rsi
    pop %rdi
    pop %rax
    pop %rbx
    pop %rcx
    pop %rdx
#endif
```

This code is performing the restoration of register states in different architectures (X86 and X64) while handling interrupts. Specifically:

- For X86 architecture, the addl instruction is used to add 4 to the stack pointer esp to jump over the _table_list entry. Then, the popl instruction is used to pop the values of registers edx, ecx, ebx, eax, ds and es in the order they were pushed to the stack earlier to restore their states.
- For X64 architecture, the popq instruction is used to pop the values of registers rdx, rcx, rbx, rax, rdi, rsi, r8, r9, r10, and r11 from the stack in the order they were pushed, to restore their states.

In summary, the purpose of this code is to restore the saved register states after the interrupt handling is completed, to ensure the program can resume execution correctly.

Here's a detailed comparison of interrupt handling code for x86 and x64 architectures.

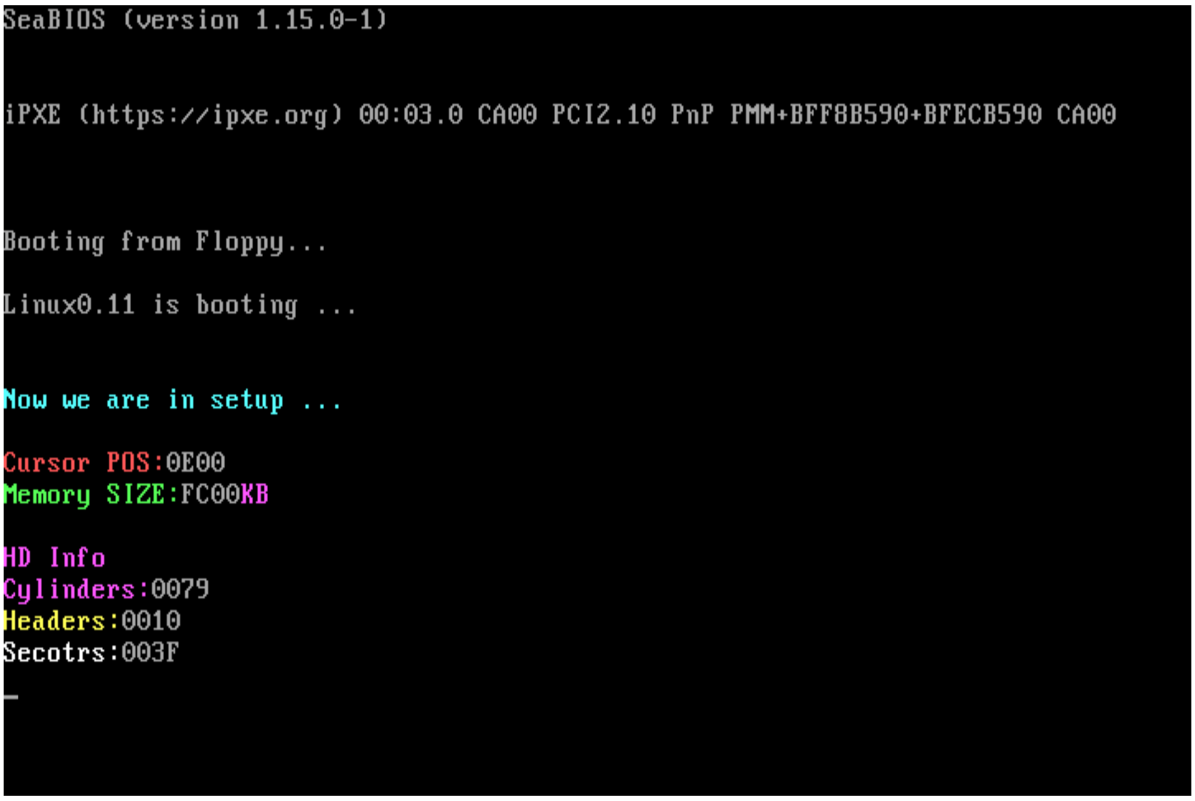
1. Registers:
   x86 interrupt handlers typically use registers eax, ebx, ecx, edx, esi, and edi to save program state. x64 uses registers rax, rbx, rcx, rdx, rsi, rdi, rip, rsp, and so on. Both x86 and x64 use esp to address the stack during interrupt handling.
2. Memory operations:
   The memory read and write operations differ slightly between x86 and x64. x86 uses movl, pushl, popl, addl, subl, incl, decl, and, orl, xorl, jne, and so on. x64, on the other hand, uses movq, pushq, popq, addq, subq, incq, decq, andq, orq, xorq, jne, and so on. Because of the register width restriction in x86, more instructions are needed to compute the address than in x64, which uses more straightforward address computation.

3. Stack:
   x86 interrupt handlers need to put the return address in the stack at the beginning of the handler. However, x64 has already put the return address in the rdi register before calling the interrupt handler.
4. Others:
   x86 uses the iret instruction to return to the main program, whereas x64 uses iretq because it requires an 8-byte return address. x86 interrupt handlers use an array called "table_list" to reference the buffer chain, while x64 uses a variable named "rs_addr."

In conclusion, x64 uses wider registers and simpler memory operations, making its interrupt handling program much simpler and more efficient than x86's program. However, x86 is still widely used for compatibility with older hardware and some applications.

How to run

```
git clone https://github.com/ITBillZ/Linux-0.11.git
cd Linux-0.11
make clean
make sofar
make start
```



```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+BFF8B590+BFECB590 CA00



Booting from Floppy...

Linux0.11 is booting ...


Now we are in setup ...

Cursor POS:0E00
Memory SIZE:FC00KB

HD Info
Cylinders:0079
Headers:0010
Secotrs:003F
_
```