CS302 Assignment 2

何泽安 12011323

2023.3.9

1. The usage of each parameters in the following command:

```
qemu-system-riscv64 \
-machine virt \
-nographic \
-bios default \
-device loader,file=bin/ucore.bin,addr=0x80200000
```

- `qemu-system-riscv64` is the executable file under `path/to/qemu/installation/riscv64-softmmu`, as we have added this path into the environment variable `PATH`, we can directly run the riscv64 VM by calling its name.
- `-machine virt` is the command that can select the emulated machine, and the param `virt` is to use the RESC-V VirtIO board.

```
heza12011323@VM-8-14-ubuntu:~/qemu-5.0.0/riscv64-softmmu$ qemu-system-riscv64 -h | grep machine
-machine [type=]name[,prop[=value][,...]]
                selects emulated machine ('-machine help' for list)
                specify machine UUID
                configure or create an on-board (or machine default) NIC and
--preconfig     pause QEMU before machine is initialized (experimental)
                        xenpv machine type).
heza12011323@VM-8-14-ubuntu:~/qemu-5.0.0/riscv64-softmmu$ qemu-system-riscv64 -machine help
Supported machines are:
none                 empty machine
sifive_e             RISC-V Board compatible with SiFive E SDK
sifive_u             RISC-V Board compatible with SiFive U SDK
spike                RISC-V Spike Board (default)
spike_v1.10          RISC-V Spike Board (Privileged ISA v1.10)
spike_v1.9.1         RISC-V Spike Board (Privileged ISA v1.9.1)
virt                 RISC-V VirtIO board
```

- `-nographic` specify the VM to disable graphical output and redirect serial I/Os to the console.
- `-bios default` sets the filename (default) for the BIOS
- `-device loader,file=bin/ucore.bin,addr=0x80200000` is adding a device (using the *Generic Loader*), specifying the file `bin/ucore.bin` as the minimal OS kernel binary file (ELF), and specifying that the ELF should be loaded to the base address `0x80200000`, which will be executed after start up.

2. Explain the meaning of codes in `kernel.ld`: (as marked in the comment)

```
OUTPUT_ARCH(riscv)  /* specify the output/target's CPU architecture is RISCV */
ENTRY(kern_entry)  /* specify the entry point of the target is kern_entry */

BASE_ADDRESS = 0xFFFFFFFFC0200000;  /* define the base address of location counter
*/

SECTIONS  /* start describing the memory layout of the output */
```

```
{
    /* Load the kernel at this address: "." means the current address
     * this acturally specify the starting address of the
     * `.text` section of the output file
     */
    . = BASE_ADDRESS;

    /* define the sequence of items in the `.text` section of output */
    .text : {
        /* the `.text` of output will be filled with the below elements in sequence
         * first find the kern_entry, then is the `.text`, `.stub` section,
         * then fill all block entries under the `.text` section, finally
         * is the gun linkonce for inline assembler text section
         */
        *(.text.kern_entry .text .stub .text.* .gnu.linkonce.t.*)
    }

    /* using the PROVIDE keyword can define a symbol, when we need to reference it
     * but is not defined. this can prevent the confliction with function name in C
     * this will act as the `extern` variables mentioned below
     */
  PROVIDE(etext = .);  /* Define the 'etext' symbol to `.` (location counter
here)*/

    /* define the sequence of items in the `.rodata` (read only) section of output
*/
    .rodata : {
        /* the `.rodata` of output will be filled with the below elements in
sequence
         * first is all files' `.rodata` section, then are all the blocks under
         * the `.rodata` section, then is the gun linkonce for inline assembler
         * rodata section
         */
        *(.rodata .rodata.* .gnu.linkonce.r.*)
    }

    /* Adjust the address for the data segment to the next page,
     * as the page size of RISCV is 4kb = 0x1000byte
     */
    . = ALIGN(0x1000);

    /* define the sequence of items in the `.data` segment of output */
    .data : {
        *(.data)  /* first fills with all files' `.data` section */
        *(.data.*)  /* then all blocks entry under `.data` section */
    }

    /* define the sequence of items in the `.sdata` (static data) segment of output
*/
```

```
    .sdata : {
        *(.sdata)  /* first fills with all files' `.sdata` section */
        *(.sdata.*)  /* then all blocks entry under `.sdata` section */
    }

    PROVIDE(edata = .);  /* Define the 'edata' symbol to `.` (location counter)*/

    /* define the sequence of items in the `.bss` (block starting symbol) segment
of output */
    .bss : {
        *(.bss)  /* first is all files' `.bss` section */
        *(.bss.*)  /* then all blocks entry under `.bss` section */
        *(.sbss*)  /* finally, all files' `.sbss*` (static bss) section */
    }

    PROVIDE(end = .);  /* Define the 'end' symbol to `.` (location counter)*/

    /* specify which sections to be discarded, aka, these sections in the input
files
     * will not be included in the output file
    */
    /DISCARD/ : {
        *(.eh_frame .note.GNU-stack)  /* discard `.eh_frame` and `.note.GNU-stack`
sections */
    }
}
```

3. Explain the memset operation in kern_init:

```
extern char edata[], end[];  // two pointers from kernel.ld
memset(edata, 0, end - edata);
```

   As we defined the `edata` and `end` symbol in `kernel.ld` (explained above), that between `edata` and `end` is the `bss` section. From the function prototype that `memset(void* dest, int ch, size_t count)`, this instruction is to fill all the memory bytes between the address `edata` and `end` point to, aka. the `bss` segment (that should be initially set to zero, which satisfies the requirements of program and OS).

4. Explain how `cputs` prints a string via sci:

```
// kern/libs/stdio.c
int cputs(const char *str) {
    int cnt = 0;
    char c;
    while ((c = *str ++) != '\0') {
        cputch(c, &cnt);  //<----
    }
    cputch('\n', &cnt);
```

```
    return cnt;
}

static void cputch(int c, int *cnt) {
    cons_putc(c);  //<----
    (*cnt) ++;
}

// kern/driver/console.c
void cons_putc(int c) {
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        sbi_console_putchar((unsigned char)c);  //<----
    }
    local_intr_restore(intr_flag);
}

// libs/sbi.h

#define SBI_CALL(which, arg0, arg1, arg2) ({              \
  register uintptr_t a0 asm ("a0") = (uintptr_t)(arg0);   \
  register uintptr_t a1 asm ("a1") = (uintptr_t)(arg1);   \
  register uintptr_t a2 asm ("a2") = (uintptr_t)(arg2);   \
  register uintptr_t a7 asm ("a7") = (uintptr_t)(which);  \
  asm volatile ("ecall"                                   \
          : "+r" (a0)                                     \
          : "r" (a1), "r" (a2), "r" (a7)                  \
          : "memory");                                    \
  a0;                                                     \
})

#define SBI_CALL_1(which, arg0) SBI_CALL(which, arg0, 0, 0)
#define SBI_CONSOLE_PUTCHAR 1

static inline void sbi_console_putchar(int ch)
{
  SBI_CALL_1(SBI_CONSOLE_PUTCHAR, ch);  //<----
}
```

The full call stack is listed above, where a C-string is first passed into the `cputs` function, which processes the string char-by-char (upcast to int). `cputch` then handle each char (in the C-string, and an additional `\n`): since our target is to display the char in the console, it passes the char (upcasted to int) into the `cons_putc` to do the display job, and it also manages a counter that helps `cputs` keep tracking of the number of chars it put to the console.

Diving down to the `cons_putc` function, one must notice the `local_intr_save` and `local_intr_restore`, these operations first disables interrupt delivery on the current processor after saving the current interrupt state into `intr_flag` and later restores that state which was stored into `intr_flag` by `local_intr_save` and ensures that during sbi call the system won't be trapped. Finally, the execution of `sbi_console_putchar` utilize the macro `SBI_CALL_1` which further dive into the `SBI_CALL` that consists of inline assembly and uses `ecall` to make a request to the supporting execution environment and `register` to pass data.

5. Implement the `shutdown` function.

   As the `cputs` function is a wrapper of `cons_putc` defined in `console.h`, comparing the functionality that the `shutdown` function fits the `console` more than `stdio`, I choose to define the function and implement it in `console.h/c`

```
// kern/driver/console.h
void shutdown(void);

// kern/driver/console.c
void shutdown(void) {
    sbi_shutdown();
}
```

   Similar to `cons_putc` that makes a sbi call, our `shutdown` also use a sbi call, that was already defineded in `sbi.h`, but it's implementation was just added by me:

```
// libs/sbi.h
void sbi_shutdown(void);

// libs/sbi.c
uint64_t SBI_SHUTDOWN = 8;  // this const was already here

void sbi_shutdown() {
    sbi_call(SBI_SHUTDOWN, 0, 0, 0);
}
```

   Finally, we make the function call in `init.c`

```
// kern/init/init.c

int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);

    const char *message = "os is loading ...\n";
    cputs(message);

    // ----------start-------------
    cputs("The system will close.\n");
```

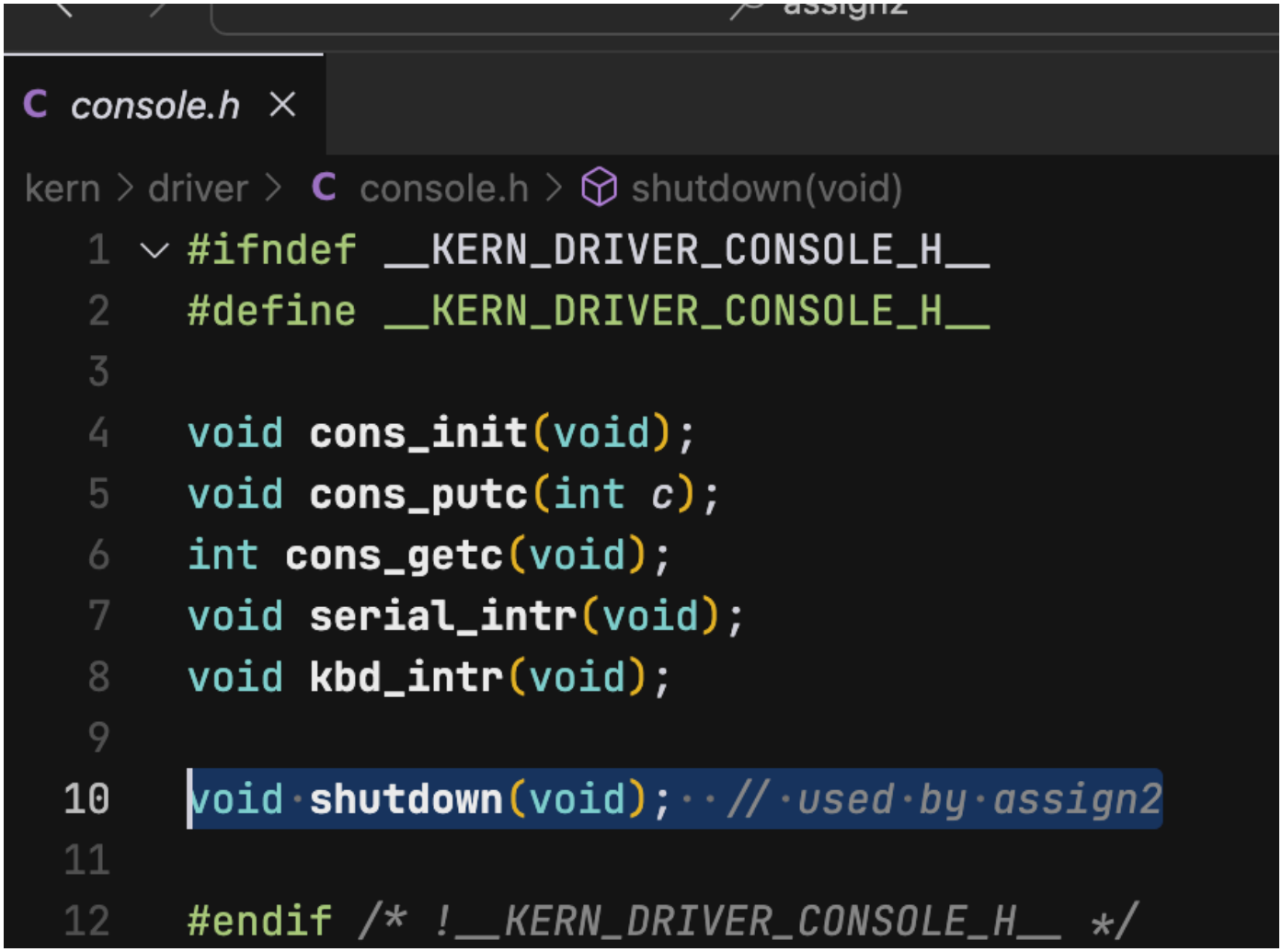
```
    shutdown();
    // -----------end--------------

    while (1) ;
}
```

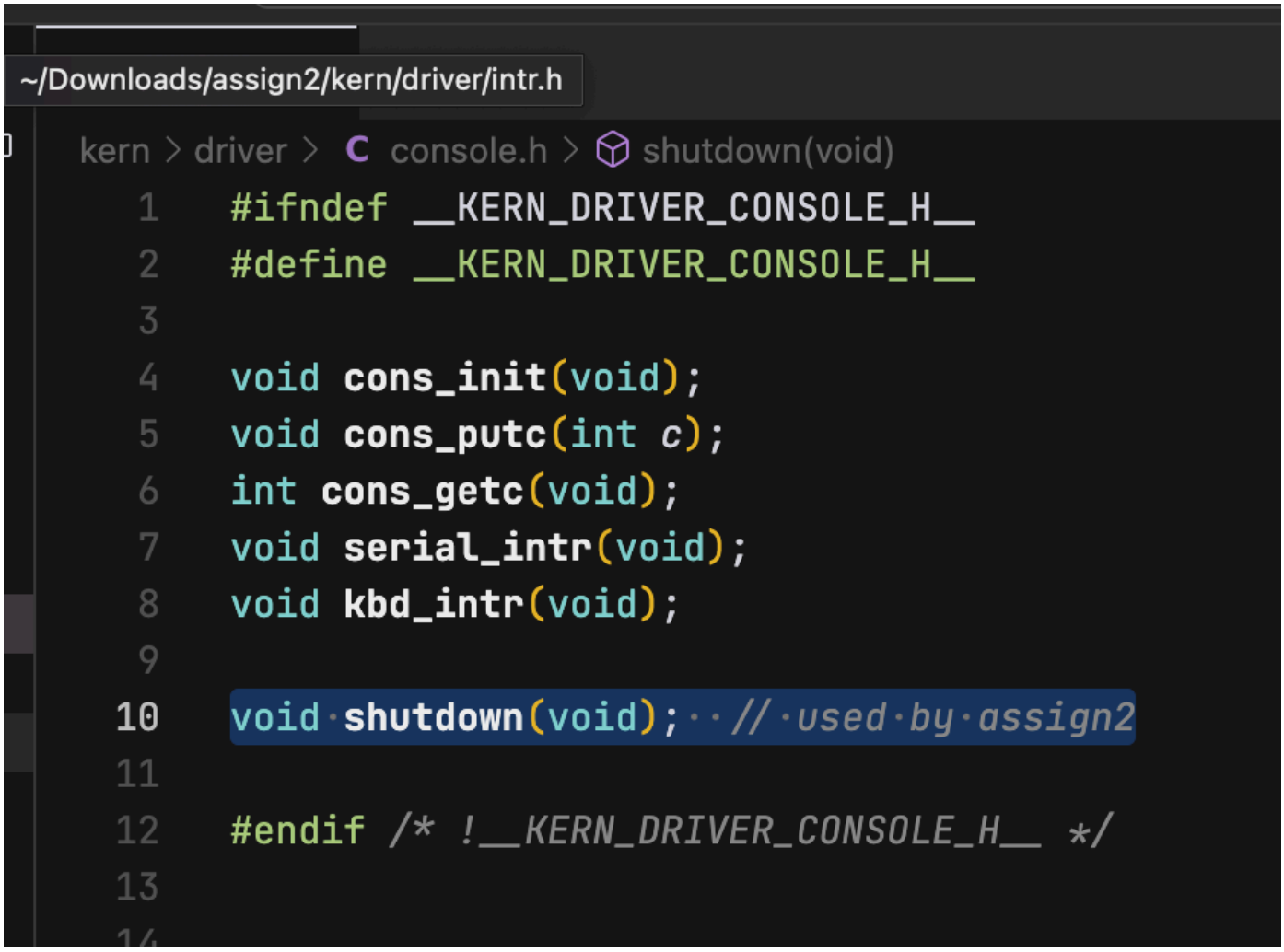
And the result is as below:

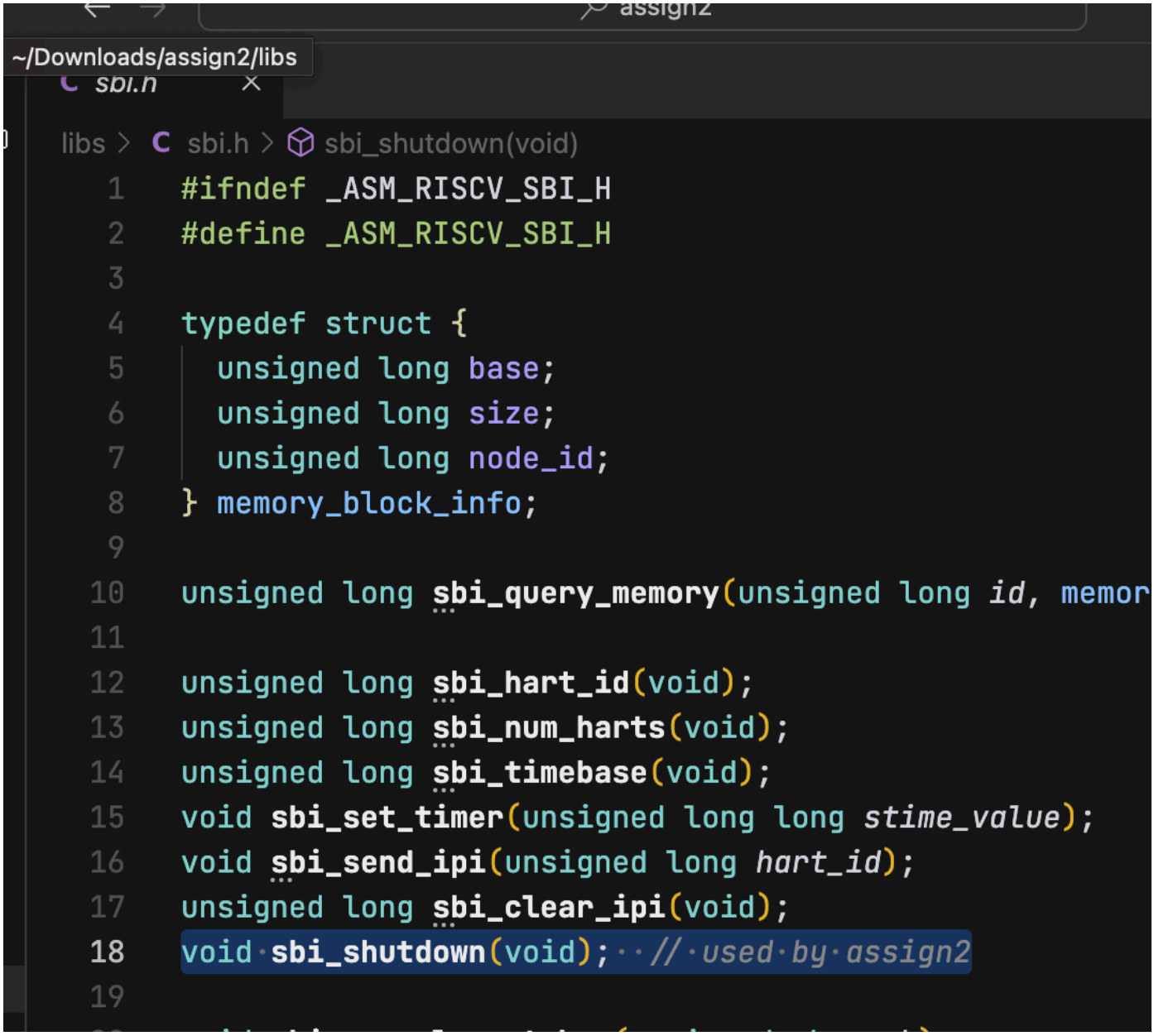
```
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.bin

OpenSBI v0.6
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 120 KB
Runtime SBI Version    : 0.2

MIDELEG : 0x0000000000000222
MEDELEG : 0x000000000000b109
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
os is loading ...

The system will close.

heza12011323@VM-8-14-ubuntu:~/assign2$
```

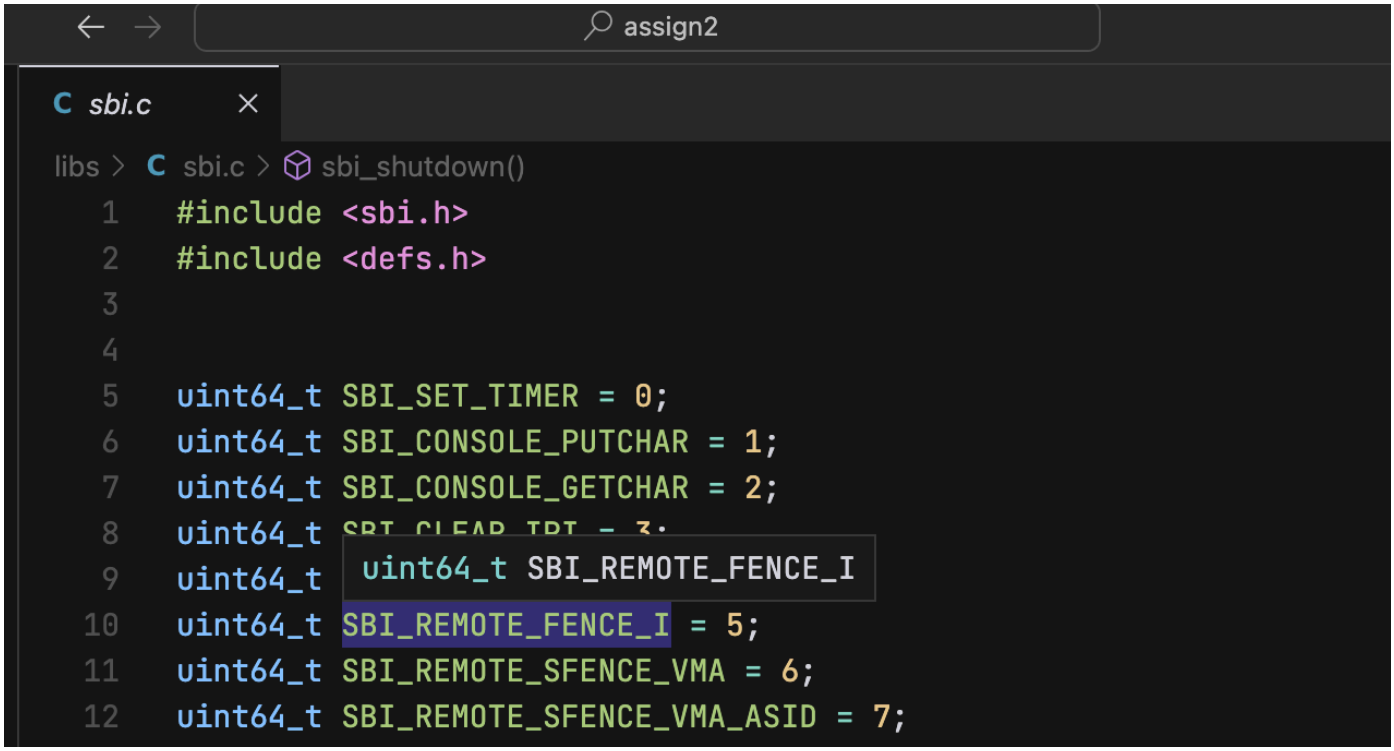
Required screenshots:

```c
int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);

    const char *message = "os is loading ...\n";
    cputs(message);




    // clock_init();
    // ----------start-------------

    cputs("The system will close.\n");
    shutdown();

    // -----------end--------------

    while (1)
        ;
}
```


assign2
console.h
kern > driver > console.h > shutdown(void)
1 #ifndef __KERN_DRIVER_CONSOLE_H__
2 #define __KERN_DRIVER_CONSOLE_H__
3
4 void cons_init(void);
5 void cons_putc(int c);
6 int cons_getc(void);
7 void serial_intr(void);
8 void kbd_intr(void);
9
10 void shutdown(void); // used by assign2
11
12 #endif /* !__KERN_DRIVER_CONSOLE_H__ */


~/Downloads/assign2/kern/driver/intr.h
kern > driver > C console.h > shutdown(void)
#ifndef __KERN_DRIVER_CONSOLE_H__
#define __KERN_DRIVER_CONSOLE_H__
void cons_init(void);
void cons_putc(int c);
int cons_getc(void);
void serial_intr(void);
void kbd_intr(void);
void shutdown(void); // used by assign2
#endif /* !__KERN_DRIVER_CONSOLE_H__ */


assign2
~/Downloads/assign2/libs
sbi.h
libs > sbi.h > sbi_shutdown(void)
#ifndef _ASM_RISCV_SBI_H
#define _ASM_RISCV_SBI_H
typedef struct {
unsigned long base;
unsigned long size;
unsigned long node_id;
} memory_block_info;
unsigned long sbi_query_memory(unsigned long id, memor
unsigned long sbi_hart_id(void);
unsigned long sbi_num_harts(void);
unsigned long sbi_timebase(void);
void sbi_set_timer(unsigned long long stime_value);
void sbi_send_ipi(unsigned long hart_id);
unsigned long sbi_clear_ipi(void);
void sbi_shutdown(void); // used by assign2


assign2
sbi.c
libs > sbi.c > sbi_shutdown()
#include <sbi.h>
#include <defs.h>
uint64_t SBI_SET_TIMER = 0;
uint64_t SBI_CONSOLE_PUTCHAR = 1;
uint64_t SBI_CONSOLE_GETCHAR = 2;
uint64_t
uint64_t SBI_REMOTE_FENCE_I
uint64_t SBI_REMOTE_FENCE_I = 5;
uint64_t SBI_REMOTE_SFENCE_VMA = 6;
uint64_t SBI_REMOTE_SFENCE_VMA_ASID = 7;

```c
uint64_t SBI_SHUTDOWN = 8;

uint64_t sbi_call(uint64_t sbi_type, uint64_t arg0, uint64_t arg1,
    uint64_t ret_val;
    __asm__ volatile (
        "mv x17, %[sbi_type]\n"
        "mv x10, %[arg0]\n"
        "mv x11, %[arg1]\n"
        "mv x12, %[arg2]\n"
        "ecall\n"
        "mv %[ret_val], x10"
        : [ret_val] "=r" (ret_val)
        : [sbi_type] "r" (sbi_type), [arg0] "r" (arg0), [arg1] "r"
        : "memory"
    );
    return ret_val;
}

int sbi_console_getchar(void) {
    return sbi_call(SBI_CONSOLE_GETCHAR, 0, 0, 0);
}
void sbi_console_putchar(unsigned char ch) {
    sbi_call(SBI_CONSOLE_PUTCHAR, ch, 0, 0);
}

void sbi_set_timer(unsigned long long stime_value) {
    sbi_call(SBI_SET_TIMER, stime_value, 0, 0);
}

// assign2
void sbi_shutdown() {
    sbi_call(SBI_SHUTDOWN, 0, 0, 0);
}
```