User-Level Memory Mapping

```c
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- `len` bytes
- `start` (or address chosen by kernel)
- `offset` (bytes)

Disk file specified by file descriptor `fd`
Process virtual memory
```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

void mmapcopy(int fd, int size);

int main(int argc, char **argv) {
    struct stat stat;
    int fd;
    /* Check for required cmd line arg */
    if (argc != 2) {
        printf("usage: %s <filename>
");
        exit(0);
    }
    /* Copy input file to stdout */
    if ((fd = open(argv[1], O_RDONLY, 0)) < 0)
        perror("open");
    fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
    close(fd);
    return 0;
}

void mmapcopy(int fd, int size) {
    /* Ptr to memory mapped area */
    char *bufp;
    bufp = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
    write(STDOUT_FILENO, bufp, size);
    return;
}
```
// boot_readseg(dst, src_sect, filesz, memsz)
737475
//    Load an ELF segment at virtual address 'dst' from the IDE disk's sector
77// 'src_sect'. Copies 'filesz' bytes into memory at 'dst' from sectors
78// 'src_sect' and up, then clears memory in the range
79// '[dst+filesz, dst+memsz)'.
80static void boot_readseg(uintptr_t ptr, uint32_t src_sect, size_t filesz, size_t memsz) {
81    uintptr_t end_ptr = ptr + filesz;
82    uintptr_t end_ptr = ptr + filesz;
83    memsz += ptr;
84    // round down to sector boundary
85    ptr &= ~(SECTORSIZE - 1);
86    // read sectors
87    for (; end_ptr < memsz; ++end_ptr) {
88        *(uint8_t*) end_ptr = 0;
89    }
90    // clear bss segment
91    for (; end_ptr < memsz; ++end_ptr) {
92        *(uint8_t*) end_ptr = 0;
93    }
94
95    /* boot.c */
96
97    #include "x86-64.h"
98
99    #include "elf.h"
100
101    // boot.c
102
103    // WeensyOS boot loader. Loads the kernel at address 0x40000 from
104    // the first IDE hard disk.
105    // A BOOT LOADER is a tiny program that loads an operating system into
106    // memory. It has to be tiny because it can contain no more than 510 bytes
107    // of instructions: it is stored in the disk's first 512-byte sector.
108    // When the CPU boots it loads the BIOS into memory and executes it. The
109    // BIOS initializes devices and CPU state, reads the first 512-byte sector of
110    // the boot device (hard drive) into memory at address 0x7C00, and jumps to
111    // that address.
112    // The boot loader is contained in bootstart.S and boot.c. Control starts
113    // in bootstart.S, which initializes the CPU and sets up a stack, then
114    // transfers here. This code reads in the kernel image and calls the
115    // kernel.
116    // The main kernel is stored as an ELF executable image starting in the
117    // disk's sector 1.
118    #define SECTORSIZE 512
119    #define ELFHDR ((elf_header*) 0x10000) // scratch space
120    void boot (void) __attribute__((noreturn));
121    static void boot_readsect(uintptr_t dst, uint32_t src_sect);
122    static void boot_readseg(uintptr_t dst, uint32_t src_sect, size_t filesz, size_t memsz);
123
124    // boot
125    // Load the kernel and jump to it.
126    void boot (void) {
127        // read 1st page off disk (should include programs as well as header)
128        // and check validity
129        boot_readseg((uintptr_t) ELFHDR, 1, PAGESIZE, PAGESIZE);
130        while (ELFHDR->e_magic != ELF_MAGIC) {
131            /* do nothing */
132        }
133        // load each program segment
134        elf_program* ph = (elf_program*) ((uint8_t*) ELFHDR + ELFHDR->e_phoff);
135        elf_program* eph = ph + ELFHDR->e_phnum;
136        for (; ph < eph; ++ph) {
137            boot_readseg((ph->p_vma, ph->p_offset / SECTORSIZE + 1,
138            ph->p_filesz, ph->p_memsz);
139        }
140        // jump to the kernel
141        typedef void (*kernel_entry_t)(void) __attribute__((noreturn));
142        kernel_entry_t kernel_entry = (kernel_entry_t) ELFHDR->e_entry;
143        kernel_entry();
144    }
145
146
2. Two more examples of I/O instructions

(a) Reading keyboard input

The code below is an excerpt from WeensyOS’s k−hardware.c

This reads a character typed at the keyboard (which shows up on the
"keyboard data port" (KEYBOARD_DATAREG)).

/* Excerpt from WeensyOS x86−64.h */

// Keyboard programmed I/O

#define KEYBOARD_STATUSREG      0x64
#define KEYBOARD_STATUS_READY   0x01
#define KEYBOARD_DATAREG        0x60

int keyboard_readc(void) {
    static uint8_t modifiers;
    static uint8_t last_escape;

    if ((inb(KEYBOARD_STATUSREG) & KEYBOARD_STATUS_READY) == 0) {
        return −1;
    }

    uint8_t data = inb(KEYBOARD_DATAREG);
    uint8_t escape = last_escape;
    last_escape = 0;

    if (data == 0xE0) {         // mode shift
        last_escape = 0x80;
        return 0;
    } else if (data & 0x80) {   // key release: matters only for modifier keys
        int ch = keymap[(data & 0x7F) | escape];
        if (ch >= KEY_SHIFT && ch < KEY_CAPSLOCK) {
            modifiers &= ~(1 << (ch − KEY_SHIFT));
        } else if (ch >= 'a' && ch <= 'z') {
            if (modifiers & MOD_CONTROL) {
                ch −= 0x60;
            } else if (!(modifiers & MOD_SHIFT) != !(modifiers & MOD_CAPSLOCK)) {
                ch −= 0x20;
            } else if (ch >= KEY_CAPSLOCK) {
                modifiers ^= 1 << (ch − KEY_SHIFT);
                ch = 0;
            } else if (ch >= CKEY(0) && ch <= CKEY(21)) {
                ch = complex_keymap[ch − CKEY(0)].map[modifiers & 3];
            } else if (ch < 0x80 && (modifiers & MOD_CONTROL)) {
                ch = 0;
            }
        } else if (ch >= 'A' && ch <= 'Z') {
            modifiers |= 1 << (ch − KEY_SHIFT);
            ch = 0;
        }

        return ch;
    }
}

(b) Setting the cursor position

The code below is also excerpted from WeensyOS’s k−hardware.c. It
uses I/O instructions to set a blinking cursor somewhere on a 25 x 80
screen.

// console_show_cursor(cpos)
//    Move the console cursor to position 'cpos', which should be between 0
//    and 80 * 25.

void console_show_cursor(int cpos) {
    if (cpos < 0 || cpos > CONSOLE_ROWS * CONSOLE_COLUMNS) {
        cpos = 0;
    }
    outb(0x3D4, 14);       // Command 14 = upper byte of position
    outb(0x3D5, cpos / 256);
    outb(0x3D4, 15);       // Command 15 = lower byte of position
    outb(0x3D5, cpos % 256);
}

Nov 06, 2023 12:46 Page 4/5
3. Memory-mapped I/O

a. Here is a 32-bit PC’s physical memory map:

```
+----------------+   <= 0xFFFFFFFF (4GB)
| 32-bit         |
| memory mapped  |
| devices        |
/\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\| Unused | <= depends on amount of RAM
+----------------+   <= 0x0D100000 (1MB)
| BIOS ROM       |
+----------------+   <= 0x000F0000 (960KB)
| 16-bit devices,|
| expansion ROMs |
+----------------+   <= 0x000C0000 (768KB)
| VGA Display    |
+----------------+   <= 0x000A0000 (640KB)
| Low Memory     |
+----------------+   <= 0x00000000
```

[Credit to Frans Kaashoek, Robert Morris, and Nickolai Zeldovich for this picture]

b. Loads and stores to the device memory *go to hardware*.

An example is in the console printing code from WeensyOS. Here is an excerpt from link/shared.ld:

```c
/* Compare the address below to the map above. */
PROVIDE(console = 0xB8000);

/* prints a character to the console at the specified */
/* cursor position in the specified color. */
/* Question: what is going on in the check */
/* if (c == '\n') */
/* ? */
/* Hint: '\n' is "C* for "newline" (the user pressed enter). */

static void console_putchar(printer* p, unsigned char c, int color) {
    console_printer* cp = (console_printer*) p;
    if (cp->cursor >= console + CONSOLE_ROWS * CONSOLE_COLUMNS) {
        cp->cursor = console;
    }
    if (c == '\n') {
        int pos = (cp->cursor - console) % 80;
        for (; pos != 80; pos++) {
            *cp->cursor++ = ' ' | color;
        }
    } else {
        *cp->cursor++ = c | color;
    }
}
```