Driver para um display i2c

1st Igor dos Santos Soares Pinto

2nd Luis Fernando Segalla

***Abstract*—Este projeto tinha como objetivo usar dos conhecimentos adquiridos na disciplina de Sistemas Operacionais para implementar um driver para a comunicação entre um display LCD e um Raspberry Pi modelo 3B usando o protocolo de comunicação i2c. Serão mostrados os passos realizados pela dupla para a realização do trabalho, assim como suas maiores dificuldades e conhecimentos adquiridos com o projeto.**

***Index Terms*—Sistemas Operacionais, Raspberry Pi, i2c, display LCD**

I. Introdução

Os drivers de dispositivos são softwares que permitem a comunicação entre determinado sistema operacional e um dispositivo conectado a ele. Esses softwares são importantes, pois permitem realizar uma abstração do funcionamento do dispositivo para o sistema, e também para os usuários, que não precisam conhecer o funcionamento exato do processo por trás da execução desse hardware.

Como parte do conteúdo visto em sala de aula, sabe-se que os dispositivos de Entrada/Saída precisam que haja uma camada de software para realizar a comunicação desse hardware entre o nível do kernel e o nível de usuário, o driver de dispositivo, de modo a possibilitar ao usuário executar diversas operações sobre o aparelho de forma simplificada e sem a preocupação de entender seu funcionamento interno. Foi escolhido como tema para este projeto a implementação de um driver de dispositivo para um display LCD de 16 colunas x 2 linhas, que se enquadra em um dispositivo de saída do tipo de caractere. Esse tipo de dispositivo envia e recebe seus dados caractere a caractere, em oposição aos hardwares de bloco, que trabalham com dados em blocos de maior tamanho.

A partir da escolha do tema, o objetivo principal do projeto foi desenvolver um software(driver) que permitisse ao usuário fazer todas as operações básicas sobre o display, como inicialização, escrita, finalização e outras, com uma interface mais amigável a ele, sem a necessidade da realização de chamadas de sistema para a realização de comandos sobre o display, por exemplo.

Ao término do projeto, o driver implementado permite ao usuário executar operações como iniciar e finalizar o display, ligar seu backlight, acionar o cursor e piscá-lo, limpar a tela, e principalmente escrever textos.

Na seção a seguir, serão descritos os materiais e métodos empregados para a realização desse projeto.

II. Materiais e métodos

Para a realização do trabalho foram usados um Raspberry Pi 3, um Display LCD 16×2 e um módulo PCF8574T que serve como um expansor GPIO para i2c. O módulo foi usado para facilitar o trabalho na hora de ter que fazer a comunicação, polpando os alunos de terem que trabalhar diretamente no display e provendo uma ”interface” mais direta para a comunicação i2c.

III. Implementação

Nesta seção iremos discutir como o driver foi implementado mostrando detalhes e discutindo as decisões de projeto.

A. i2cDriverSO_t

O primeiro passo em nossa implementação consiste em criar uma estrutura de dados que represente cada dispositivo, isso será útil pois é um método de se reunir várias informações sobre nosso dispositivo todas em um mesmo ”lugar”. A implementação da estrutura pode ser vista na imagem a seguir:

```
typedef struct i2cDriverSO
{
    struct i2c_client *meuCliente;
    struct mutex       meuMutex;
    struct cdev        meuCdev;

    u8                 regs_cntrl;
    u8                 linha;
    u8                 coluna;
}i2cDriverSO_t;
```

Fig. 1. Estrutura i2cDriverSO_t, usada para facilitar o manejo do driver.

B. Inicializando o módulo

Para inicializar o módulo é preciso criar ao menos duas funções, uma para carregá-lo no kernel e outra para removê-lo de lá. As funções meuInit e meuExit como podem ser vistas na figura a seguir recebem respectivamente as macros __init e __exit. Isso ”diz” ao kernel que elas são usadas somente durante a inicialização do módulo ou então durante sua remoção. Por fim chamamos as macros $module\_init()$ e $module\_exit()$ para definir que nossas funções sejam chamadas na hora que o módulo é carregado no kernel, com a init, e descarregado dele, com a exit.

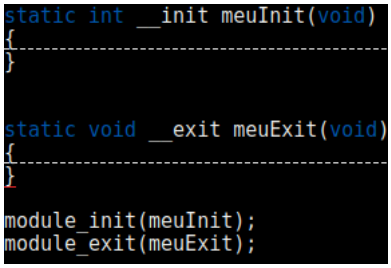

Fig. 2. Funções meuInit e meuExit.

A implementação da função meuInit consiste basicamente dos seguintes passos:

- Alocam-se os Major e Minor Numbers com $alloc\_chrdev\_region$
- Pede acesso ao barramento $i2c-1$ com a função $i2c\_get\_adapter$
- Usamos a função $i2c\_new\_device$ para instanciar um novo dispositivo. Seus campos são inicializados com a struct $i2c\_board\_info$
- Registramos o driver no subsistema usando a função $i2c\_add\_driver$

Como nosso driver precisa de um dispositivo escravo para que haja a comunicação é necessário mais um passo para que possamos inicializá-lo de forma completa. Após a $i2c\_new\_device$, quando um novo dispositivo é instanciado, a função $meuProbe()$ é chamada. A função tem como objetivo:

- Checar se o dispositivo que foi instanciado está entre aqueles que o driver suporta (faz isso checando se seu nome e endereço estão na tabela definida pela struct $i2c\_device\_id$).
- Checar se seu barramento suporta as funcionalidades requeridas pelo driver (usando a função $i2c\_check\_functionality$).
- Aloca memória e inicializa o dispositivo usando a função $devm\_kzalloc()$.
- Adicionar o Char Device ao sistema com a função $cdev\_add$.

C. file_operations

Para que possamos usar as funções implementadas é necessário criar uma struct do tipo $file\_operations$. Ela consiste, basicamente, de ponteiros para funções definidas pelo driver para realizar operações sobre si mesmo através de chamadas de sistema. Resultando em ações sendo executadas sobre um arquivo no diretório /dev, que representa um dispositivo que utiliza o driver. Nossa implementação da struct $file\_operations$ pode ser vista na imagem a seguir:

- **.owner**: módulo para o qual as funções são definidas.
- **.write**: envia dados para o dispositivo escravo.
- **.read**: recebe dados do dispositivo escravo. Por questões de tempo não a implementamos.
- **.open**: função para abertura do arquivo que representa nosso dispositivo.

```
struct file_operations meuFile_operations =
{
    .owner                  = THIS_MODULE,
    .write                  = meuWrite,
    .read                   = meuRead,
    .open                   = meuOpen,
    .release                = meuRelease,
    .unlocked_ioctl         = meuIoctl,
};
```

Fig. 3. Funções presentes dentro da struct $file\_operations$.

- **.release**: função para ”liberar” o arquivo depois que terminar de usá-lo.
- **.ioctl**: usada para implementação de comandos específicos para o dispositivo, no nosso caso a tela.

D. Escrevendo na tela

Como nosso projeto consistia em fazer com que o dispositivo escravo (a tela) mostrasse o conteúdo enviado pelo mestre (Raspberry) as funções de escrita foram as mais importantes do trabalho e com certeza as que mais geraram dificuldade. Dividimos esta função em duas: o comando meuWrite usado para fazer as chamadas de sistema com a struct $file\_operations$ e uma outra função chamada $write\_command$.

A implementação da função meuWrite é relativamente simples. Percorre-se a string enviada como parâmetro pelo usuário chamando $write\_command$ para cada caractere. Um cuidado especial precisa ser tomado aqui relacionado ao numero de linhas e colunas do display. Este controle era mantido com uma instância da struct $i2cDriverSO\_t$ declarada como static no começo do nosso código. O retorno da função meuWrite é o tamanho da string escrita na tela.

```
static ssize_t meuWrite(struct file *filp, const char __user *buff, size_t count, loff_t *offp)
{
    ssize_t tamSaida = 0;
    int i;
    printk(KERN_ALERT "DENTRO DA FUNÇÃO meuWrite.\n");

    if(!buff)//confere se o buffer não está vazio
    {
        printk(KERN_ALERT "Buffer vazio.\n");
        return -ERESTARTSYS;
    }

     mutex_lock(&display->meuMutex);

     for(i = 0; i < strlen(buff); i++)
     {
        write_command(display->meuCliente,buff[i],1); // 1 no terceiro argumento para dados

        display->coluna++; //incremento minha contagem das colunas
        if(display->coluna > 15) //confiro se estou na ultima coluna
        {
            display->coluna = 0;

            if(display->linha == 0) //confiro se estou na ultima linha
            {
                display->linha = 1;
                write_command(display->meuCliente,LCD_LINE1,0); // Para instruções -> Segundo bit RS do display

            }
            else
            {
                display->linha = 0;
                write_command(display->meuCliente,LCD_LINE0,0); // Para instruções -> Segundo bit RS do display

            }
        }
    }

    mutex_unlock(&display->meuMutex);
    tamSaida = strlen(buff);

    return tamSaida;
}
```

Fig. 4. Nossa implementação para a chamada de sistema write.

A escrita propriamente dita é feita dentro da função $write\_command$. Esta função é interessante pois exigiu uma leitura mais profunda do datasheet (referência 5) e alguns conhecimentos das aulas de microcontroladores.

O display recebe mensagens de 8bits e quebra ela em duas partes de 4 bits, uma com os MSB e outra com os bits LSB. Isso é necessário porque internamente ele multiplexa estes dois valores e através de uma tabela encontra o caractere que deve ser mostrado na tela.

O display só é capaz de enviar as mensagens de seu registrador de dados para sua memória relacionada à exibição dos dados durante um pequeno pulso no pino ENABLE, por isso é preciso também que simulemos este comportamento, enviando um bit em alto, fazendo uma pequena pausa (delay) e logo em seguida enviando um bit em baixo. Isso precisa ser feito para as duas partes do byte divididas anteriormente.

Um esquema de como funciona esta parte da comunicação pode ser visto a seguir:

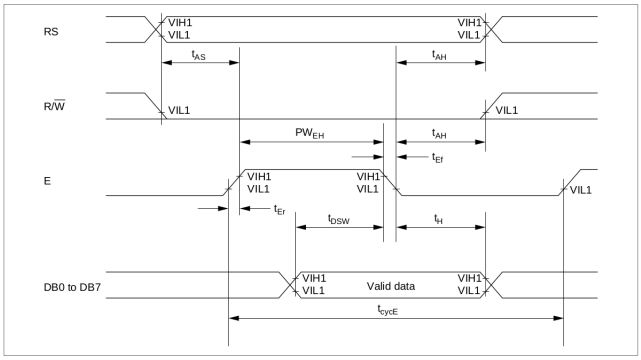

Fig. 5. Esquema do timing de envio de dados.

```c
void write_command(struct i2c_client *client, u8 data, int instr)
{
    u8 up,low;

    up  = LCD_DATA_MSB(data); // separando 8 bits em duas partes de 4, pelo modo como é feito o envio
    low = LCD_DATA_LSB(data);

    if(instr == 0){ // se for instrução, não caracteres
        up  = LCD_RS_INSTR(up); // Avisando que é instrução
        low = LCD_RS_INSTR(low);
    }else{
        up  = LCD_RS_DATA(up); // Avisando que é dados(não instrução)
        low =LCD_RS_DATA(low);
    }

    up = up | LCD_BACKLIGHT; // display ativo
    low = low | LCD_BACKLIGHT;

    // Parte upper bits
    i2c_smbus_write_byte(client,up); // Mandando a parte upper dos 8 bits
    udelay(500);

    i2c_smbus_write_byte(client,LCD_E_HI(up)); // Enable high
    udelay(500);
    i2c_smbus_write_byte(client,LCD_E_LOW(up)); // Enable low -> Para poder enviar os bits, pois precisa de uma troca de borda no enable para envio
    udelay(500);

    // Parte lower bits

    i2c_smbus_write_byte(client,low); // Mandando a parte lower dos 8 bits
    udelay(500);

    i2c_smbus_write_byte(client,LCD_E_HI(low)); // Enable high
    udelay(500);
    i2c_smbus_write_byte(client,LCD_E_LOW(low)); // Enable low -> Para poder enviar os bits, pois precisa de uma troca de borda no enable para envio
    udelay(500);

    return;
}
```

Fig. 6. Implementação da função que faz a comunicação entre a tela e o raspberry.

E. ioctl

O campo ioctl na struct $file\_operations$ tem como objetivo a implementação de algumas funções mais específicas para o dispositivo em questão. Como estávamos trabalhando com um display, era interessante podermos interagir com ele de outras formas além da escrita. Foram implementados então os seguintes métodos:

- **LIMPA_TELA**: Apaga todo o conteúdo da tela e volta o cursor para a primeira posição (linha e coluna em zero).
- **BACKLIGHT**: Apaga ou acende a luz de fundo da tela de acordo com o comando passado pelo usuário.
- **CURSOR**: Mostra ou deixa de mostrar a posição atual do cursor.
- **BLINK**: Faz com que o cursor fique piscando onde está.

```c
static long meuIoctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    printk(KERN_ALERT "CHAMANDO A FUNÇÃO meuIoctl.\n");
    mutex_lock(&display->meuMutex);

    switch(cmd)
    {
        case LIMPA_TELA://MODIFIQUEI AQUI
            write_command(display->meuCliente,LCD_CLEAR,0);
            display->linha = display->coluna = 0;
            break;

        case BACKLIGHT:
            setBacklight(display,arg);
            break;

        case CURSOR:
            display_cursor(display,arg);
            break;

        case BLINK:
            display_blink(display,arg);
            break;

        default:
            printk(KERN_ALERT "Comando inválido.\n");

    }

    mutex_unlock(&display->meuMutex);
    return 0;
}
```

Fig. 7. Implementação da ioctl. De acordo com o comando uma ação específica é tomada e dependendo dela usa-se o campo arg da chamada de função.

IV. Funções em Nível de Usuário

Dar ao usuário acesso a chamadas de sistema e a uma interação mais direta com o kernel não costuma ser uma boa ideia. Com isso em mente decidimos abstrair toda essa parte do usuário. Para isso implementamos uma biblioteca chamada ”userLevelFunctions” a ideia é que o usuário tenha acesso apenas a uma interface construída por nós para que ele use o display. É permitido usar as funções da ioctl, assim como escrever e apagar a tela, tudo isso através de um menu que pode ser visto na imagem a seguir:

```
Digite seu comando mestre:
0 - Sair
1 - Escreve na tela
2 - Apagar o display
3 - Acende o display
4 - Limpar o display
5 - Ativa o cursor
6 - Desativa o cursor
7 - Pisca o cursor
8 - Para de piscar o cursor
```

Fig. 8. Menu com o qual o usuário pode interagir.

```c
#include "userLevelFunctions.h"

int main()
{
    Display();

    return 0;
}
```

Fig. 9. O que o usuário tem acesso quando usa nossa interface.

```c
void Display()
{
    int in;
    int fd = OpenDisplay();
    char *text;
    if(fd < 0)
    {
        printf("Erro ao abrir o arquivo.\n");
        return;
    }

    while( in != 0 )
    {
        printf("Digite seu comando mestre: \n0 - Sair\n1 - Escreve na tela\n2 - Apagar o display\n3
        scanf("%d", &in);

        switch(in)
        {
            case 1:
                scanf("%s",text);
                write(fd,text,0);
                break;

            case 2:
                ioctl(fd,BACKLIGHT,0);
                break;

            case 3:
                ioctl(fd,BACKLIGHT,1);
                break;

            case 4:
                ioctl(fd,LIMPA_TELA,0);
                break;

            case 5:
                ioctl(fd,CURSOR,1);
                break;

            case 6:
                ioctl(fd,CURSOR,0);
                break;

            case 7:
                ioctl(fd,BLINK,1);
                break;

            case 8:
                ioctl(fd,BLINK,0);
                break;

            default:
                printf("COMANDO INVÁLIDO.\n");
                break;
        }
    }
}
```

Fig. 10. Implementação por trás do menu. Usuário não tem acesso a esta parte.

V. Versão inicial da biblioteca para o nível de usuário

Após a apresentação do projeto ao professor, foram feitas algumas ressalvas em relação às formas de implementação e outros detalhes construtivos. Como a principal observação, foi destacado que as funções de nível de usuário poderiam ter sido implementadas de outra forma, permitindo, por exemplo, o usuário efetuar a chamada de cada função diretamente, ao invés de escolher comandos numéricos para a realização das operações.

Inicialmente, nossa implementação seguia essa linha de raciocínio. Havia uma struct, de nome LCD_t, que encapsulava informações relativas ao display, como as posições da linha e coluna do cursor no momento, um booleano verificador de inicialização, além de um inteiro relativo ao descritor de arquivo vinculado ao dispositivo. Esta struct estava contida em um arquivo de nome LCDlib.h, junto dos protótipos das funções disponíveis para a biblioteca. A imagem a seguir mostra a organização desse arquivo inicial:

```c
#ifndef LCD_LIB_H
#define LCD_LIB_H

#include <unistd.h>
#include <sys/ioctl.h> /* ioctl */
#include <errno.h>
#include <fcntl.h>

#define LCD_CLEAR           0x01
#define LCD_HOME            0x02
#define INICIA_TELA         10  // Valores para ioctl (switch-case)
#define BACKLIGHT           11
#define CURSOR              12
#define BLINK               13

// Biblioteca utilizada para os comandos para o display HD44780 a nivel de usuário
// Autores: Luis Fernando Segalla e Igor dos Santos

typedef struct lcd{ // struct relativa à abstração do dispositivo utilizando o driver do display

    int pos_linha; // posicao atual representando cursor para coluna
    int pos_coluna; // postcao atual representando cursor para coluna
    int lin_num; // num total colunas tela (se suporte para mais de um tipo de tela com o driver)
    int col_num; // num total colunas tela (se suporte para mais de um tipo de tela com o driver)
    int fd_display; // inteiro que será o file descriptor
    bool inicializado; // 0 - desativado ; 1 inicializado (Estado display)

}LCD_t;

// ---------------------------------------- Protótipos funções biblioteca usuário display 16x02 ----------------------------------------//

LCD_t* LCD_Init(); // retorna o ponteiro para a struct relativa à abstração do dispositivo utilizando o driver do display
void LCD_EscreveTexto(LCD_t *display, char* msg); // Escreve a mensagem msg na tela, iniciando na posicao linha = 0, coluna = 0
void LCD_LimpaTela(LCD_t *display); // Limpa a tela do display
void LCD_Backlight(LCD_t *display, int value); // Seleciona backlight ativo ou não, de acordo com o valor de value(0 apagado, 1 ligado)
void LCD_CursorShow(LCD_t *display, int value); // Exibe ou não o cursor da tela, de acordo com o valor passado em value
void LCD_BlinkCursor(LCD_t *display, int value); // Pisca ou não o cursor, de acordo com o valor passado para value(0 -> Não pisca, 1 -> pisca)
void LCD_Finaliza(LCD_t **display); // Finaliza o uso do display e libera recursos alocados

#endif
```

Fig. 11. Arquivo .h da biblioteca para o usuário

Na parte do arquivo de código fonte da biblioteca, de nome LCD_lib.cpp, foram implementadas as funções que podem ser chamadas pelo usuário, incluindo funções para iniciar, finalizar e escrever na tela, além das de manipulação do cursor e backlight, limpeza da tela e do efeito de piscar do cursor presente no display. Abaixo estão as imagens ilustrando algumas das funções implementadas para essa biblioteca.

```c
#include <iostream>
#include <errno.h>
#include "LCDLib.h"

// Implementações das funções de nivel de usuário para tela 16x02 HD44780

// Retorna um ponteiro para a struct LCD_t, que faz uma abstração do display para o usuário

LCD_t* LCD_Init(){

    LCD_t* disp = new LCD_t; // Alocando struct que representa de forma abstraida o display, para o usuario
    disp->pos_linha = 0;
    disp->pos_coluna = 0;
    disp->col_num = 16; // display 16x02
    disp->lin_num = 2;

    char* dir_device = "/dev/DriverSDi2c"; //Diretório no qual o device está representado em /dev

    disp->fd_display = open(dir_device, O_RDWR); // Abertura do arquivo em /dev para possibilitar a posterior comunicação

    if(disp->fd_display < 0 ){
            std::cerr << "Falha na inicialização do dispositivo. Verifique se o módulo(driver) para este device está instalado!\n";
            return nullptr;
    }

    LCD_LimpaTela(disp); // setirando possiveis valores salvos nos regs de dados do display
    for(int i=0;i < 10000;i++){} // delay

    LCD_Backlight(disp,1); // Acionando backlight
    for(int i=0;i < 10000;i++){} // delay

    return disp;
}
```

Fig. 12. Função de inicialização do display na biblioteca

```c
// Recebe um ponteiro para o display e a string da mensagem como parametros
// Escreve a mensagem msg no display representado pelo  ponteiro para a struct LCD_t *display
void LCD_EscreveTexto(LCD_t *display, char* msg){

    if(display->inicializado){ // Device válido
        write(display->fd_display,msg,0); // Envio para função write da struct file operations, relativo as funções disponíveis para o device
    }else{
        std::cerr << "Display nao inicializado!\n";
    }
    return;
}

// Limpa a tela do display
// Recebe o ponteiro para a struct do display como parâmetro
void LCD_LimpaTela(LCD_t* display){

    if(display->inicializado){ // Se foi previamente inicializado o display
        ioctl(display->fd_display,LCD_CLEAR,0);
    }else{
        std::cerr << ("Display não inicializado!\n");
    }
    return;
}

// Ativa ou desativa o backlight do display, de acordo com o valor de value(1 = aceso, 0 = apagado)
// Recebe o ponteiro para a struct do display como parâmetro, e o valor referente a opção desejada em value
void LCD_Backlight(LCD_t* display,int value){


    if(display->inicializado){ // Se foi previamente inicializado o display
        ioctl(display->fd_display,BACKLIGHT,value);
    }else{
        std::cerr << ("Display não inicializado!\n");
    }
    return;
}
```

Fig. 13. Implementação de algumas das funções disponíveis na biblioteca

No entanto, após fazermos alguns testes com essa biblioteca implementada, percebeu-se que comportamentos inesperados ocorriam, e em alguns casos também ocorriam falhas de segmentação. Devido a isso, decidiu-se partir para a implementação das funções de usuário exibidas na seção anterior, a fim de evitar possíveis erros em sua execução. Caso a equipe tivesse mais tempo, essa biblioteca seria ajustada e seria utilizada como interface para o usuário operar sobre o display.

VI. Conclusão

Embora relativamente simples de ser implementado, o trabalho provou-se um desafio para a dupla. O que foi de grande ajuda foi a grande quantidade de materiais sobre o assunto tanto em livros quanto na internet. Os dois livros usados (referências 1 e 2) foram de extrema importância para que entendêssemos a teoria e todos os passos necessários para a construção de um char driver e as configurações necessárias para um driver específico de i2c. Além dos livros pesquisamos por trabalhos já existentes similares ao nosso, assim encontramos o trabalho dos alunos Jesuino Vieira e Lucas Camargo (que fizeram sobre o mesmo tema no semestre de 2018.2)

e uma implementação no github do usuário Telecnatron, os trabalhos ajudaram a termos ideias de implementações e sair de momentos de impasse.

De forma geral as maiores dificuldades da dupla foram no início, quando não entendiamos exatamente como funcionava, ou deveria, funcionar o driver e como fazer a relação software/hardware. Porém conforme íamos testando as versões iniciais do módulo a teoria começou a fazer mais sentido. A parte de escrita também gerou uma certa dificuldade, tivemos dois maiores problemas com ela. O primeiro foi todo o contexto da quebra da mensagem de 8 bit em duas de 4 bit, inicialmente tentávamos mandar a mensagem inteira usando a função $i2c\_smbus\_write\_byte$, o que fazia com que nada aparecesse na tela. Com a ajuda dos códigos encontrados na internet e do datasheet do display encontramos o erro e o corrigimos. A segunda dificuldade foi a questão do pulso no pino enable. Foram necessárias várias leituras do datasheet e análise de outros códigos para que entendêssemos realmente o funcionamento e a necessidade desta operação.

Se tivéssemos mais tempo para o projeto nossa ideia era usar o segundo barramento i2c do Raspberry Pi para conectar algum tipo de sensor, ou qualquer dispositivo escravo que recebesse dados. Por falta de tempo e por não ser o foco do projeto não implementamos uma função para que o dispositivo mestre conseguisse ler dos escravos. A ideia seria usar o segundo barramento para ler algo (temperatura, velocidade, intensidade luminosa, etc...), tratar estes dados e então mandar mostrá-los no display. Isso seria interessante pois mostraria uma interação mais intrincada da comunicação i2c, além de exigir ainda mais estudo da dupla.

Após a apresentação para o professor também foram sugeridas algumas melhorias e mudanças a serem implementadas no trabalho. A mais gritante é com certeza uma melhor implementação de thread safety, que conforme testado pelo professor, não ocorre no trabalho. Mas também era necessário que trabalhássemos melhor nosso makefile que em certos casos de operação podia levar a erros, assim como também oferecer um arquivo Read_me com instruções para o usuário.

De forma geral, o trabalho foi muito positivo para os alunos. A experiência de se trabalhar com o sistema operacional em um nível de desenvolvimento como este agregou muito aos alunos, além de proporcionar entendimento mais completo de conceitos vistos em sala.

REFERENCES

[1] J. Corbet, A. Rubini e G. Kroah-Hartman ”LINUX DEVICE DRIVERS”
[2] John Madieu ”Linux Device Drivers Development: Develop customized drivers for embedded Linux”
[3] Jesuino Vieiera e Lucas Camargo ”https://github.com/jesuinovieira/os”
[4] Telecnatron ”https://github.com/telecnatron/RPI_PCF8574_I2C_LCD”
[5] Datasheet display ”https://www.sparkfun.com/datasheets/LCD/HD44780.pdf”