Labotatórios de SO - I

Arnon Vieira Silva Renato Tsutomu Tamaki Wesley Dal Col

3 de junho de 2010

RESUMO

Este texto é baseado nas aulas de laboratório da disciplina *Sistemas Operacionais I - CC4211* do curso de Ciência da Computação do Centro Universitário da FEI. Nele são abordados alguns dos principais experimentos realizados durante o curso, tendo como principal objetivo servir de apoio aos alunos que vierem a cursá-lo.

SUMÁRIO

1 LAB 01 - CHAMADAS DE SISTEMA (SYSTEM CALLS) p. 5

1.1 **Objetivos** . . . . . p. 5

1.2 **Exemplos de Chamadas de Sistema** . . . . . p. 5

1.2.1 getpid . . . . . p. 5

1.2.2 nanosleep . . . . . p. 6

1.2.3 kill . . . . . p. 7

1.2.4 read e write . . . . . p. 10

1.3 **Conclusão** . . . . . p. 11

2 LAB 02 - MINISHELL p. 12

2.1 **Objetivos** . . . . . p. 12

2.2 **Requisitos** . . . . . p. 12

2.3 **Implementação** . . . . . p. 13

2.4 **Conclusão** . . . . . p. 16

3 LAB 03 - INTERPRETADOR DE COMANDOS p. 17

3.1 **Objetivos** . . . . . p. 17

3.2 **Requisitos** . . . . . p. 17

3.3 **Implementação** . . . . . p. 18

3.4 **Conclusão** . . . . . p. 22

4 LAB 04 - BOOTSECTOR p. 23

4.1 **Objetivos** . . . . . p. 23

4.2 **Requisitos** . . . . . p. 23

4.3 **Implementação** . . . . . p. 24

4.4 **Conclusão** . . . . . p. 26

REFERÊNCIAS p. 27

1 LAB 01 - CHAMADAS DE SISTEMA (SYSTEM CALLS)

As Chamadas de Sistema são o conjunto de funções que compõe a interface entre o sistema operacional e os programas de usuário. Elas permitem o acesso a recursos e dispositivos da máquina através de uma API padronizada, de forma que o programador não precise se preocupar com detalhes do hardware.

Essa API permite acesso aos recursos internos do SO, de forma que se por um lado ela é essencial para o desenvolvimento de aplicações de baixo nível, por outro seu uso indevido pode ser potencialmente danoso ao sistema inteiro (MITCHELL ALEX SAMUEL, 2001)

1.1 Objetivos

O objetivo deste laboratório é familiarizar o aluno com o uso de algumas chamadas de sistema simples, como getpid, read, write, etc.

Utilizaremos para isso alguns exemplos de códigos escritos em linguagem C.

1.2 Exemplos de Chamadas de Sistema

1.2.1 getpid

```
pid_t getpid(void);
```

A chamada getpid retorna o id do processo corrente (pid - process ID)(WATSON, 2010). Um pid é uma estrutura de dados arbitrária que alguns sistemas operacionais utilizam para identificar unicamente um processo. No Unix o pid é um número inteiro sem sinal.

Para obtermos o pid do processo atual utilizando linguagem C podemos compilar e executar o código abaixo:

```
//---------------------------------------------------------------------
// getpid.c
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    printf("Current process ID: %d\n", getpid());
    return EXIT_SUCCESS;
}
//---------------------------------------------------------------------
```

Salve este código em um arquivo chamado getpid.c, compile-o e execute-o com os comandos:

```
gcc -o getpid getpid.c
./getpid
Current process ID: 8015
./getpid
Current process ID: 8017
```

Note que a cada execução, o pid muda. Isso acontece porque a cada nova execução, é criado um novo processo com um novo e exclusivo pid.

1.2.2 nanosleep

```
int nanosleep(const struct timespec *req, struct timespec *rem);
```

A chamada nanosleep suspende a execução da thread atual até que se passe o tempo indicado pelos parâmetros(WATSON, 2010). A thread literalmente dorme por um período determinado. Vamos ao código:

```
//---------------------------------------------------------------------
// nanosleep.c
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    struct timespec tv;
    tv.tv_sec  = (time_t)(2);           // Configurando timer para 2
    tv.tv_nsec = (long)(0.5 * 1e+9);    // segundos e meio

    printf("0.0 sec\n");
    nanosleep(&tv, &tv);
    printf("2.5 sec\n");

    return EXIT_SUCCESS;
}
//---------------------------------------------------------------------
```

Novamente salve o código em um arquivo chamado nanosleep.c, compile e execute com:

```
gcc -o nanosleep nanosleep.c
./nanosleep
0.0 sec
2.5 sec
```

Ele mostra o texto `0.0 sec`, aguarda dois segundos e meio, mostra o texto `2.5 sec` e então encerra.

A chamada nanosleep é utilizada para a criação de timers de alta resolução e possui mais algumas capacidades que não serão exploradas no momento, como a habilidade de retornar o tempo que falta até o timer acabar, caso ela seja interrompida antes do planejado.

1.2.3 kill

```
int kill(pid_t pid, int sig);
```

A chamada kill é utilizada para enviar sinais a qualquer processo(WATSON, 2010). Um sinal é uma forma de comunicação entre o kernel e os processos e entre processos, uma “espécie de mensagem”. Dentre esses sinais estão comandos para que o processo espere, continue a sua execução ou termine, por exemplo.

Neste exercício vamos utilizar as chamadas anteriores para um processo que executa um laço infinito e então vamos criar um programa que força a finalização do outro processo com uma chamada kill.

No primeiro programa, *loop*, usamos getpid para capturar o pid do processo. Em seguida, fazemos um while infinito mostrando uma mensagem na tela a cada 5 segundos. A pausa é feita utilizanod a chamada nanosleep.

No segundo programa, *terminator*, capturamos o pid do processo que está em loop passado como argumento, e então enviamos o sinal *SIGTERM*, que significa o programa que o recebeu deve terminar imediatamente.

Veja os códigos abaixo:

```
//---------------------------------------------------------------------
// loop.c
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    struct timespec tv;
    tv.tv_sec  = (time_t)(5);           // Configurando timer para 5
    tv.tv_nsec = (long)(0);             // segundos

    printf("Current process ID: %d\n", getpid());

    while (1) {
        printf("Aguardando 5 sec\n");
        nanosleep(&tv, &tv);
    }
    return EXIT_SUCCESS;
}
//---------------------------------------------------------------------
```

```
//---------------------------------------------------------------------
// terminator.c
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char** argv) {
    pid_t pid_to_kill;

    if (argc < 2) {
        printf("Preciso de um pid para exterminar\n");
        return EXIT_FAILURE;
    }

    pid_to_kill = (pid_t)abs(atoi(argv[1]));

    printf("Hasta la vista %d\n", pid_to_kill);
    kill(pid_to_kill, SIGTERM);

    return EXIT_SUCCESS;
}
//---------------------------------------------------------------------
```

Salve os arquivos com os nomes indicados e compile-os com os comandos:

```
gcc -o loop loop.c
gcc -o terminator teminator.c
```

Para realizar a experiência, vamos executar o programa loop em segundo plano. Em Unix isso é feito acrescentando um & logo após o nome do programa ao invocá-lo. Não se preocupe com isso agora, esse truque é apenas para que tenhamos acesso novamente ao terminal, já que o programa em loop não termina e precisamos executar outro programa.

```
./loop &
Current process ID: 10515
Aguardando 5 sec
[1] 10515
Aguardando 5 sec
Aguardando 5 sec
Aguardando 5 sec
...
```

Depois de invocado, o programa loop vai mostrar o seu pid e a cada 5 segundos vai escrever uma mensagem na tela. Para encerrá-lo, vamos invocar o programa terminator e passar o pid do

loop como parâmetro, de forma que o terminator possa encontrar o processo loop e enviar-lhe um sinal (SIGTERM).

```
...
Aguardando 5 sec
./terminator 10515
Hasta la vista 10515
[1]+  Finalizado              ./loop
```

1.2.4 read e write

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

As funções read e write são duas das chamadas de função capazes de acessar dispositivos de entrada e saída.

Enquanto a função read é capaz de ler dados de um determinado dispositivo, a sua contraparte write é capaz de gravar dados em dispositivos.

Podemos demonstrar o seu uso básico com um programa bem simples que captura a entrada do teclado e a imprime na tela:

```
//------------------------------------------------------------------
// read_write.c
#include <unistd.h>
#include <stdlib.h>

#define BUFFSIZE    1024
#define STDIN       0
#define STDOUT      1
#define STDERR      2

int main(void){
    char buffer[BUFFSIZE];
    int read_chars = 0;

    while((read_chars = read(STDIN, buffer, BUFFSIZE)) > 0) {
        write(STDOUT, buffer, read_chars);
    }

    return EXIT_SUCCESS;
}
//------------------------------------------------------------------
```

Salve com o nome indicado, compile e execute com:

```
gcc -o read_write read_write.c
./read_write
```

O programa vai ficar esperando que você digite algum texto no teclado. Ao pressionar ENTER, o texto é capturado da entrada padrão pela chamada read e impresso na saída padrão pela chamada write.

1.3 Conclusão

As chamadas de sistema fornecem uma forma simples de acessar recursos e dispositivos do sistema como timers, sinais, teclado, vídeo, entre outros. entretanto devem ser utilizadas com cautela pos podem até mesmo forçar o encerramento de um outro processo.

Em geral, em uma camada acima das chamadas de sistema são construídas bibliotecas que encapsulam as funcionalidades, adicionando facilidades e/ou mecanismos de controle para um acesso ainda mais simples e seguro.

2 LAB 02 - MINISHELL

Um shell (casca, concha) um programa que executa acima do sistema operacional, responsável por criar a interface entre o usuário e a máquina.

O shell deve receber as interações do usuário e transformar essas acões em algo compreensível para o computador. Analogamente ele precisa receber as respostas do kernel e traduzir isso para uma linguagem compreensível para humanos.

O shell também deve ser capaz de disparar a execução dos programs de usuário conforme forem requisitados.

2.1 Objetivos

Neste laboratório vamos mostrar como consntruir um minishell bem simples utilizando somente chamadas de sistema. Vamos apresentar também algumas chamdas novas, capazes de executar programas externos.

2.2 Requisitos

O nosso minishel fará a interface entre o usuário e a máquina. Para isso ele precisará fornecer mecanismos para que o usuário possa requisitar algo e receber ma resposta, que pode ser uma requisição bem sucedida ou não.

Para que o usuário possa interagir com a máquina, forneceremos o acesso ao teclado, dessa forma nosso minishell precisará ler dados do teclado. Já sabemos como fazer isso utilizando a chamada read.

Analogamente para que o usuário obtenha uma resposta, precisamos ser capazes de mostrar uma mensagem na tela. Para isso utilizaremos a chamada write.

O próximo requisito é que o minishell seja capaz de executar comandos digitados pelo

usuário.

Para cumprir esse requisito nosso minishel precisa ser capaz de fazer duas coisas: a primeira é criar um novo processo e a segunda é utilizar esse processo para executar um novo programa.

Apresentamos então as chamadas *fork* e *execve*.

A chamada fork cria uma cópia do processo atual, ou seja, um novo processo a partir do processo corrente. O novo processo é chamado processo-filho (child) do processo que o originou, enquanto que o processo original é chamado de pai (parent). Eles mantem alguams relações interessantes entre si, mas não abordaremos aqui.

Para que o novo processo recém-criado seja capaz de executar um programa externo, a chamada execve substitui a imagem do processo (código e dados) pela imagem de um programa especificado.

O nosso minishel deverá então ser capaz de criar um novo processo e substituir esse processo por um programa externo qualquer.

2.3 Implementação

Conforme a sofisticação das aplicações vão aumentando, torna-se cada vez mais trabalhoso trabalhar diretamente com as chamadas de sistema. Por isso que foram criadas as bibliotecas de mais lato nível.

Para começar, temos que definir na mão labels para tamanho máximo de buffer, stdin, stdout, entre outros. Mostrar algo mais elaborado na tela também vai ficando complicado.

Toda vez que lemos algo do teclado, precisamos estar atentos ao fato de que o último caracter digitado foi o ENTER, logo ele sempre estará no final do buffer. precisamos removê-lo pois senão a função excve não vai encontrar os comandos que digitarmos e portanto não vai conseguir executá-los.

Vamos então analisar com calma o código abaixo:

```
//-------------------------------------------------------------------
// minishell.c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

#define BUFFSIZE    1024
#define STDIN       0
#define STDOUT      1
#define STDERR      2

int main(int argc, char** argv) {
    char buffer[BUFFSIZE];
    pid_t pid;
    int status, read_chars;

    while (1) {
        // prompt
        write(STDOUT, "% ", 2);

        if ((read_chars = read(STDIN, buffer, BUFFSIZE)) <= 0) break;

        buffer[--read_chars] = '\0';
        if (!read_chars) continue;

        if((pid = fork()) < 0) {
            write(STDERR, "fork error\n", 11);
        }
        else if (pid == 0){
            // Child
            execve(buffer, NULL, NULL);
            write(STDERR, "couldn't execute: ", 18);
            write(STDERR, buffer, read_chars);
            write(STDERR, "\n", 1);
            return 127;
        }

        // parent
        if((pid = waitpid(pid, &status, 0)) < 0) {
            write(STDERR, "waitpid error\n", 14);
        }
    }

    write(STDOUT, "\n", 1);
    return EXIT_SUCCESS;
}
//-------------------------------------------------------------------
```

Nas primeiras linhas, fazemos includes dos headers que vamos utilizar e criamos alguns labels para referenciar stdin, stdout, etc.

Dentro do main, declaramos algumas variáveis e inciamos o loop principal do minishel. Esse loop se encerra na ocorrência de algum erro ou quando o usuário digitar Ctrl+D no teclado.

Em seguida escrevemos na tela o prompt (sinal %) e esperamos dados da entrada padrão. quando o usuário entra com algum texto e tecla ENTER, a chamada read coloca esse texto no buffer e então limpamos o ENTER do último caracter simplesmente sobrescrevendo-o com o caracter nulo (terminador de string).

Agora nosso buffer está pronto e podemos partir para a execução do fork.

Conforme comentamos anteriormente, a chamada fork cria uma cópia do processo atual. ela é invocada no processo pai uma única vez, mas por causa da duplicação dos processos, ela retorna duas vezes! Ela retorna no processo pai **E** retorna também no processo filho. A diferença é que quando ela retorna no processo pai, seu valor de retorno é o pid do processo recém-criado, enquanto que ao retornar no processo filho, seu valor de retorno é zero.

Utilizamos essa informação para executar diferentes rotinas depois do fork, dependendo se estivermos no processo pai ou no processo filho.

Quando estivermos no processo filho devemos tentar executar o comando externo que foi passado para o buffer. Isso é feito através da chamada execve, que tenta interpretar o buffer como o caminho completo para um programa. Se ela conseguir localizar o programa, ela substitui a imagem do processo corrente (filho) pela imagem do programa encontrado, o comando que digitamos no prompt do minishell. A partir daí o programa externo começa sua execução tomando o lugar do processo filho.

Note que a partir do momento em que a execve foi executada com sucesso, nada do processo filho existe mais, nem dados, nem código, apenas a "casca" do processo criado pelo sistema operacional permanece, para que o novo programa execute. Portanto, a execve nunca retorna, pois ela não existe mais. Se a execve retornar, significa que ela falhou em substituir a imagem do processo atual pela imagem do programa externo.

Analogamente, quando no processo pai, executamos ações distintas. No processo pai, esperamos o processo filho encerrar para que os recursos alocados por ele pssam ser liberados. Para isso utilizamos outra chamada de sistema, a *waitpid*.

A chamada waitpid faz com que o processo atual aguarde uma alteração no estado do processo filho, que geralmente (mas não necessariamente) é o processo terminar.

Se tudo ocorreu satisfatoriamente, recomeçamos o loop aguardando um novo comando do usuário.

Note também que nosso minishell só consegue executar comandos quando passamos o caminho completo para eles. A execve não tenta procurá-los em variáveis de ambiente ou diretórios padrão, ela simplesmente tenta executar exatamente o que passamos para ela. Chamadas de sistema tem essa característica de serem o mais diretas possível. As facilidades e comodidades devem ser implementadas pelas bibliotecas de mais alto nível, como a libc por exemplo.

Em nosso próximo projeto vamos adicionar um pouco mais de comodidade ao usuário, utilizando funções da libc.

2.4 Conclusão

Utilizando apenas chamadas de sistema fomos capzaes de implementar um pequeno shell bastante simples. Entretanto já começamos a nos defrontar com dificuldades geradas pela necessidade de sofisticação da aplicação enquanto que as chamadas de sistema fornecem apenas funcionalidades mais cruas.

Após esse projeto, o aluno deverá ser capaz de entender o básico do funcionamento de um shell, bem como das chamadas fork, execve e waitpid.

3 LAB 03 - INTERPRETADOR DE COMANDOS

Um interpretador de programas é uma camada de interface com o usuário que roda acima ou dentro de um shell, oferecendo algumas facilidades que um shell cru não não entrega.

Geralmente, shell e interpretador de comandos são o mesmo programa, que além de fornecer acesso a programas externos, também possui algums comandos inernos, chamados de *built-in*.

3.1 Objetivos

Neste laboratóorio vamos mostrar como criar um mini interpretador de comandos utilizando algumas das funções da biblioteca C padrão.

Note que a complexidade aumentaria em muito se usássemos somente as chamadas de sistema assim como no laboratório anterior.

3.2 Requisitos

Nosso mini interpretador de comandos deverá ser capaz de interpretar uma lista de comandos definida previamente, repassando as chamadas para os programas disponíveis no sistema operacional e implementando alguns comandos internos.

A lista de comandos aceitas deve conter:

- **indice**: Mostra o conteúdo do diretório corrente (ls);
- **local**: Mostra qual é o diretório corrente (pwd);
- **mostra**: Mostra o conteúdo de um arquivo (cat);
- **escreve**: Cria e/ou edita um arquivo (nano);

- **muda**: Muda o diretório corrente (cd);
- **ajuda**: Mostra a lista de comandos disponíveis;

3.3 Implementação

O código do mini interpretador de comandos é dividido em várias partes, implementadas como funções em linguagem C. O seu loop principal chama algumas dessas funções conforme descreveremos a seguir.

Dentro da função main ocorre o loop principal do programa. Esse loop repete a sequência de chamadas de função, até que o usuário digite Ctrl+D para encerrar. Essa sequência consiste em mostrar o prompt (%), ler um comando e seus parâmetros diretamente do teclado com a função `read_cmd()`, executar o comando com `execute()` e fazer a limpeza que for necessária com `cleanup()`.

A função `read_cmd()` lê a entrada do teclado, coloca-a em um buffer e em seguida processa esse buffer para extrair o comando, os parâmetros e então gerar um array conforme a função `execv()` exige. Para isso ela utiliza alocação dinâmica de memória com `realloc()`, implicando na necessidade de se chamar `cleanup()` para liberar essa memória com `free()` após a execução do comando.

A função `execute()`, através da função `real_cmd()` faz o mapeamento entre os comandos suportados e os comandos reais no sistema. Em seguida ela verifica se o comando digitado é um comando interno ou se é uma chamada a um programa do SO. Caso seja um comando interno ela mesma executa subrotinas que realizam a tarefa, senão ela usa `fork()`, `excev()` e `waitpid()` para executar o rograma externo.

No final do loop é chamada a função `cleanup()` para liberar a memória dinamicamente alocada, se for o caso.

Confira o código completo logo abixo:

```
//---------------------------------------------------------------------
// command.c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define MAXLINE 1024

char* my_commands[] = {
    "/bin/ls",
    "/bin/pwd",
    "/bin/cat",
    "/bin/nano",
    "muda",
    "ajuda"
};

char* my_index[] = {
    "indice",
    "local",
    "mostra",
    "escreve",
    "muda",
    "ajuda"
};

const int num_cmd = sizeof(my_commands) / sizeof(char*);

void prompt() {
    fprintf(stdout, "%% ");
}

char* real_cmd(char* fake) {
    int i;

    if (fake) {
        for (i = 0; i < num_cmd; i++) {
            if (!strcmp(fake, my_index[i])) {
                return my_commands[i];
            }
        }
    }

    return NULL;
}
```

```
void read_cmd(char*** cmd, int* len) {
    char buffer[MAXLINE];
    char* token = NULL;
    int token_len = 0;

    if (fgets(buffer, MAXLINE, stdin) != NULL) {
        buffer[strlen(buffer) - 1] = 0;
        *len = 0;
        *cmd = 0;
        token = strtok(buffer, " ");
        while (token) {
            (*len)++;
            token_len = strlen(token);

            (*cmd) = (char**)realloc((*cmd), (*len) * sizeof(char*));
            (*cmd)[(*len) - 1] = (char*)malloc((token_len + 1) * sizeof(char));
            strncpy(((*cmd)[(*len) - 1]), token, (token_len + 1));

            token = strtok(NULL, " ");
        }

        (*cmd) = (char**)realloc((*cmd), ((*len) + 1) * sizeof(char*));
        (*cmd)[(*len)] = (char*)malloc(sizeof(char));
        ((*cmd)[(*len)]) = NULL;
    }
    else *len = -1;
}

void cleanup(char*** cmd, int len) {
    int i;

    if (*cmd) {
        for (i = 0; i <= len; i++) free((*cmd)[i]);

        free((*cmd));
        (*cmd) = NULL;
    }
}

void muda(char** argv, int len) {
    char destdir[MAXLINE];

    if (len <= 1) strncpy(destdir, ".", 2);
    else strncpy(destdir, argv[1], MAXLINE);

    if (chdir(destdir)) printf("Cannot change\n");
}
```

```
void ajuda() {
    int i;

    for (i = 0; i < num_cmd; i++) printf("%s\n", my_index[i]);
}

int execute(char** argv, int len) {
    pid_t pid;
    int status;

    if (len > 0) {
        char* command = real_cmd(argv[0]);

        if (!command) {
            fprintf(stdout, "Comando inválido\n");
            return 1;
        }

        if (strcmp(command, "muda") == 0) {
            muda(argv, len);
            return 1;
        }

        if (strcmp(command, "ajuda") == 0) {
            ajuda();
            return 1;
        }

        if ((pid = fork()) < 0){
            fprintf(stderr, "fork error\n");
        }
        else if (pid == 0) {
            char* command = real_cmd(argv[0]);
            execv(command, argv);

            fprintf(stderr, "couldn't execute: %s\n", argv[0]);
        }

        if ((pid = waitpid(pid, &status, 0)) < 0){
            fprintf(stderr, "waitpid error\n");
        }
        else {
            return 1;
        }
    }

    return 0;
}
```

```
int main(int argc, char** argv) {
    char** cmd = NULL;
    int len = 0;
    int repeat = 1;

    while (repeat) {
        prompt();
        read_cmd(&cmd, &len);
        repeat = execute(cmd, len);
        cleanup(&cmd, len);
    }

    fprintf(stdout, "\n");

    return EXIT_SUCCESS;
}
//------------------------------------------------------------------------
```

3.4 Conclusão

Combinando chamadas de sistema e funções da biblioteca C padrão podemos criar aplicações cada vez mais sofisticadas, de forma a facilitar o uso do sistema operacional por parte dos usuários.

Após este laboratório o aluno deverá ser capaz de entender melhor o funcionamento de um interpretador de comandos, bem como ser capaz de combinar os diversos recursos de programação fornecidos pelo SO para implementar aplicações cada vez mais sofisticadas.

4 LAB 04 - BOOTSECTOR

Quando o computador inicializa, ele começa a executar instruções eu uma área pré-determnada de memória, usualmente onde começa a BIOS. Então são feitas algumas checagens como o POST (power-on self test) e finalmente a bios procura o setor de boot de algum dispositivo presente. Se encontrado, esse código é executado e toma controle da máquina, geralmente fazendo a carga do SO.

Em PCs, o setor de boot é o primeiro setor de dados do dispositivo, tendo apenas 512 bytes de tamanho. Essa área ainda é subdividida em alguns pedaços, como a marcação de partições, assinatura do disco, entre outros, restando somente 440 bytes para código executável.

Os últimos dois bytes (0x55 e 0xaa)são a assinatura da MBR(PAL, 2010).

4.1 Objetivos

O objetivo deste laboratório é mostrar como criar um código em linguagem assembly capaz de tornar um dispositivo de aramzenamento (tipicamente discos rígidos, disquetes e pendrives) inicializável.

Uma abordagem semelhante pode ser aplicada no desenvolvimento de carregadores de sistemas operacionais (bootloaders).

4.2 Requisitos

Os códigos apresentados aqui estão em linguagem assembly compatível com a arquitetura Intel x86, no entanto familiaridade com a linguagem não é mandatória, embora facilite o entendimento do exemplo.

Precisamos também de um pendrive comum com pelo menos 512 bytes de capacidade total de armazenamento e um computador capaz de ser inicializado através de alguma de suas portas USB.

Este exercício foi desenvolvido e testado utilizando Ubuntu 10.04. Vamos precisar dos programas as, ld (do pacote binutils) e dd (do pacote coreutils).

4.3 Implementação

A implementação consiste em dois passos. O primeiro é a compilação do código de boot em assembly e o segundo é a instalação do executável no pendrive.

Vamos ao código de boot:

```
##---------------------------------------------------------------------
## boot.s
.code16
.section .text
.globl _start
_start:
    mov $0xb800, %ax
    mov %ax, %ds
    movb $'X', 0
    movb $0x1e, 1
idle:
    jmp idle
##---------------------------------------------------------------------
```

O comando `.code16` instrui o assembler (as) a trabalhar em modo 16 bits. A linha `.section .text` declara a seção de texto, ou seção de código. A linha `.globl _start` declara o símbolo `_start` , que é o ponto de entrada do programa. As linhas logo abaixo *definem* o símbolo `_start` e as intruções a serem executadas.

As instruções `mov $0xb800, %ax` e `mov %ax, %ds` carregam para o registrador de segmento o valor 0xb800, que é um dos endereços onde a memória de vídeo é mapeada, significando que a partir desse ponto, cada endereço relativo de memória revere-se a esse segmento.

Em seguida escrevemos um caractere na memória, que é automaticamente repassado para o vídeo.

Nesse modo de vídeo, cada caracter na tela é representado por dois bytes, sendo o primeiro o dado e o segundo os atributos desse caracter na tela (corese de frente e fundo). Assim as duas linhas seguintes `movb $'X', 0` e `movb $0x1e, 1` carregam para a primeira e seguda posição da memória de vídeo o caracter X com oa atributos cor de fundo azul e cor de frente amarelo, respectivamente.

A linha seguinte faz o programa entrar em um laço infinito.

Salve o cóodigo acima em um arquivo chamado boot.s e gere o executável com os comandos:

```
as -o boot.o boot.s
ld --oformat binary -o boot.com boot.o
```

Em seguida, vamos testar o executável. Para isso podemos utilizar o emulador DOSEMU. Instale-o se precisar e simplesmente execute o comando:

```
dosemu boot.com
```

Se tudo deu certo no canto superior esquerdo da tela do dosemu, deve aparecer o caracter X com fundo azul e frente amarelo. Aproveite para testar outros caracteres e cores.

Agora vamos à parte mais crítica do projeto que a a gravação do executável no pendrive. Preste bastante atenção e vá com bastante cuidado, pois um erro neste passo pode destruir os dados no HD do seu computador.

Primeiro vamos descobrir para qual device o SO mapeia o pendrive. Para isso, insira o pendrive no computador e espere ele ser detectado. Agora execute o comando df para mostrar todos os filesystems montados:

```
df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda1              3876364   2807484    871968  77% /
none                    504748       300    504448   1% /dev
none                    508968       184    508784   1% /dev/shm
none                    508968        88    508880   1% /var/run
none                    508968         0    508968   0% /var/lock
none                    508968         0    508968   0% /lib/init/rw
/dev/sdc1              1959376    847196   1012648  46% /home
/dev/sdd1              7800336   1418892   6381444  19% /media/D400-841A
```

O pendrive foi detectado e associado ao device /dev/sdd (/dev/sdd é o pendrive e /dev/sdd1 é a partição montada).

Vamos então copiar o executável de boot com os comandos abaixo. Note que você precisará de permissão de administrador para acessar o dispositivo.

Lembrando: um erro aqui e você pode destruir os dados no dispositivo errado.

```
dd if=boot.com of=/dev/sdd
echo -ne "\x55\xaa" | dd seek=510 bs=1 of=/dev/sdd
```

O primeiro comando copia o executável boot.com, de apenas 17 bytes de tamanho, para o início do pendrive. O segundo comando grava a assinatura da MBR nas posições 510 e 511 do pendrive, finalizando o setor de boot.

Agora é só dar boot por esse pendrive e ver o caracter aparecendo na tela.

4.4 Conclusão

Quando o sistema operacional ainda não está carregado não temos disponíveis syscalls ou biliotecas. Entretanto, ainda assim, utilizando diretamente linguagem assembly podemos criar programas para acessar e controlar os dispositivos.

Essas técnicas são utilizadas por exemplo para criar os programas que carregam o SO ou ainda na criação de programas que rodam em dispositivos que não possuem ou não podem ter um sistema operacional.

REFERÊNCIAS

MITCHELL ALEX SAMUEL, J. O. M. L. *Advanced Linux Programming*. [S.l.]: Sams, 2001.

PAL, S. *Susam Pal's website*. 2010. Disponível em: <`http://susam.in/articles/boot-sector-code`>.

WATSON, C. *Unix Online Manual*. [S.l.], Fevereiro 2010.