Sistemas Operacionais
Roteiro de Laboratório 1

1 Objetivos

O objetivo deste laboratório é ter um contato prático com as *system calls* do Linux.

2 *System Calls* no Linux

Vamos mencionar aqui alguns pontos já discutidos em aula e introduzir novos conceitos e informações úteis.

2.1 Aonde fica o *kernel* do SO?

Na maioria dos sistemas operacionais, o *kernel* é carregado no espaço de endereçamento virtual de todos os programas em execução. Por exemplo, o Linux em uma arquitetura x86 32-bits é mapeado no gigabyte (GB) mais "alto" do espaço de endereçamento, começando no endereço `0xf0000000`.

Note que o espaço de endereçamento virtual de um processador 32-bits é $2^{32} = 4$ GB, o que leva a um espaço de endereçamento virtual efetivo de 3 GB para a aplicação em si e 1 GB para o *kernel*.

Então como o *kernel* evita que uma aplicação reescreva as estruturas do *kernel* ou chame as funções do *kernel* diretamente? Isso é tarefa do mecanismo de mapeamento de memória, que permite ao SO especificar em qual *ring* a CPU deve estar executando para poder acessar uma dada região de memória.

2.2 *Protection rings*

A CPU x86 possui quatro *rings*, ou níveis de privilégio. Entretanto, a maioria dos OSes usa somente dois *rings*: *ring* 0 (*kernel mode*) e *ring* 3 (*user mode*). Os *rings* de numeração mais alta são mais restritos, indicando que eles não podem executar certas instruções privilegiadas, tais como instruções que vão interagir diretamente com o *hardware*. De forma similar, os mecanismos de proteção de páginas de memória, que serão estudados adiante no curso, conseguem diferenciar permissões de acesso dependendo do *ring* atual em que a CPU está executando.

2.3 Trocando de *rings*

Como a CPU sai de um *ring* para outro?

Em geral, uma vez que a CPU entrou no *ring* 3 (*user mode*), o único jeito de retornar ao *kernel mode* é através de uma *interrupção*. Uma interrupção pode ser um evento de *hardware*, tal como um disco sinalizando a conclusão de uma operação de leitura/escrita; ou pode ser também uma *exceção*, tal como uma divisão por zero; ou ainda um *trap*, aonde o *software* intencionalmente levanta uma interrupção.

Na arquitetura x86, interrupções são associadas com um valor 8-bits específico. Por exemplo, a exceção de divisão por zero recebe o número de interrupção 0. Este valor serve como um índice

na *interrupt descriptor table (IDT)*, onde o *kernel* instala um *handler* (função) que é chamado quando uma interrupção dispara.

A IDT também especifica em qual *ring* o *handler* deve executar; em geral, o *ring* é zero. Assim, qualquer *software* que pode causar alguma interrupção vai levar a CPU a trocar para o *ring* zero e começar a executar o *handler* específico.

Alguns números de interrupção são designados pelo desenvolvedor do *hardware*. A Intel reserva as interrupções 0–31 para exceções, e por convenção, as 16 seguintes são tipicamente utilizadas para interrupções de dispositivos.

Os outros 212 códigos de interrupção restantes ficam sob controle do *kernel*. O uso mais comum de um *handler* de interrupções é tratar as *traps* (ou *system calls*) de uma aplicação. Por exemplo, o Linux utiliza 0x80, ou 128 em decimal, para a sua interrupção de *system call*. O Windows, por outro lado, utiliza 0x2e, ou 46 em decimal. Essa escolha é totalmente arbitrária.

E como isso fica no código? Se você fizer um *disassemble* de um binário 32-bits antigo que faz uma chamada de sistema, você deve ver uma linha contendo `int $0x80`. A instrução `int` levanta uma interrupção de *software* que leva a um salto na execução para a função especificada como o *handler* da interrupção 0x80, que roda no *ring* 0. O *kernel* retorna o controle para a aplicação por meio da instrução `iret`, que restaura os registradores da aplicação e retorna para *ring* 3.

Importante: `int $0x80` é um código legado e deve ser evitado, pois não está mais disponível em CPUs 64-bits. (Ele só foi utilizado como um exemplo.) O método atual de entrar em *kernel mode* em arquiteturas x86 64-bits é com a instrução `syscall`.

3 Códigos de Exemplos

O programa abaixo é o exemplo clássico de *Hello World* implementado em C.

```
#include <stdio.h>

int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

Esse programa faz uso da função `printf` que está definida em `stdio.h`. Esse arquivo define as funções de I/O que estão implementadas na biblioteca padrão do C (`libc`). Para um usuário normal, essa biblioteca provê a interface com as funcionalidades do SO.

Descendo um nível na API, é possível ver que as funções em `stdio.h` utilizam outras funções de mais baixo nível, as chamadas *system call wrappers*, que são funções que preparam a chamada da *system call* real. O programa abaixo utiliza os *wrappers* para reimplementar o programa de *Hello World*, empregando somente a função `write`, que faz parte do padrão POSIX, definido em `unistd.h`.

```
#include <unistd.h>

int main(void) {
    const char *msg = "Hello World!\n";
    write(STDOUT_FILENO, msg, 13);
    return 0;
}
```

Por fim, é possível realizar diretamente as *system calls* do *kernel*, mas para tal é preciso programar diretamente no *assembly* da arquitetura, como ilustrado no programa abaixo.

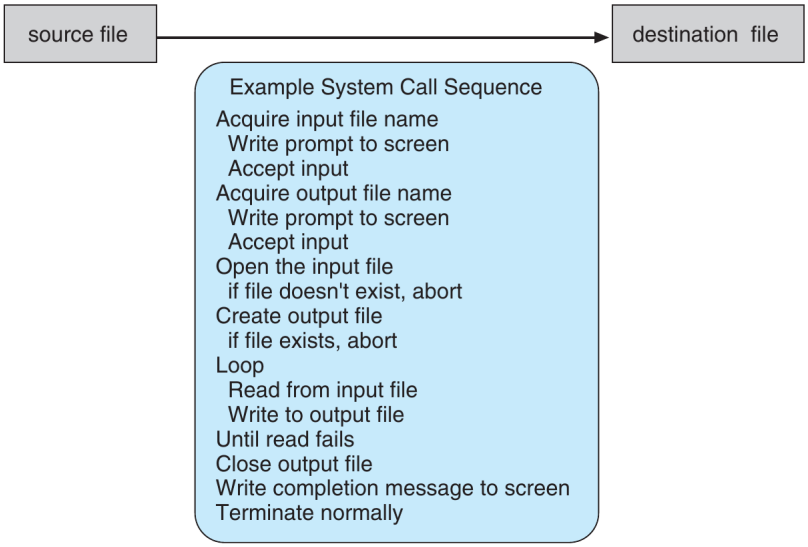
```
# ----------------------------------------------------------------------------
# Writes "Hello World!" to the console using only system calls.
# Runs on 64-bit Linux only.
# To assemble and run:
#     gcc -c hello2.s
#     ld -o hello2 hello2.o
#     ./hello2
# ----------------------------------------------------------------------------

        .global _start

        .text
_start:
        # write(1, message, 13)
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout
        mov     $message, %rsi          # address of string to output
        mov     $13, %rdx               # number of bytes
        syscall                         # invoke operating system to do write

        # exit(0)
        mov     $60, %rax               # system call 60 is exit
        xor     %rdi, %rdi              # we want return code 0
        syscall                         # invoke operating system to exit
message:
        .ascii  "Hello World!\n"
```

O programa acima está escrito em Assembly x86_64, no padrão AT&T, que é o utilizado pelo `as`, o montador do `gcc`. A *system call* que escreve no terminal é invocada pelo comando `syscall`. Esse comando não possui operandos pois cada *system call* tem um número variável de argumentos. Esses argumentos são passados em registradores, que precisam ser preenchidos corretamente antes da chamada. O registrador `rax` sempre deve conter o código da *system call* que deve ser executada. Os demais registradores variam conforme esse código. Uma tabela completa de todas as *system calls* do Linux (com os respectivos registradores) pode ser vista em `http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/`.

4 Tarefa

1. Faça o *download* no AVA dos arquivos de exemplos. Compile-os e execute-os como indicado acima e discutido pelo professor no início da aula.

2. Implemente um programa que realiza a cópia de um arquivo para um outro. O funcionamento deste programa foi discutido em uma aula anterior da disciplina e está resumido na figura abaixo:



Você deve usar somente as funções de sistema do padrão POSIX, que são implementadas como *system call wrappers* para as *system calls* do Linux. Você vai precisar utilizar as funções `read`, `write` e `close`, definidas no arquivo `unistd.h` e a função `open` definida no arquivo `fcntl.h`. É proibido o uso das funções de I/O da biblioteca padrão do C. (Não use nada que esteja em `stdio.h`.)

3. Envie a sua implementação do item 2 pelo AVA na tarefa correspondente. O prazo (inadiável) para submissão é dia 12/09/2017 (terça-feira) às 23:55. Essa tarefa vale um *bit* (1 – fez corretamente dentro do prazo, 0 – caso contrário) na nota dos exercícios de laboratório.