

# Labotatórios de SO - I

Arnon Vieira Silva

Renato Tsutomu Tamaki

Wesley Dal Col

30 de maio de 2010

# SUMÁRIO

<b>1</b>	<b>LAB 01 - CHAMADAS DE SISTEMA (SYSTEM CALLS)</b>	p. 3
1.1	Objetivos . . . . .	p. 3
1.2	Exemplos de Chamadas de Sistema . . . . .	p. 3
1.2.1	getpid . . . . .	p. 3
1.2.2	nanosleep . . . . .	p. 4
1.2.3	kill . . . . .	p. 5
1.2.4	read e write . . . . .	p. 8
1.3	Conclusão . . . . .	p. 9
<b>2</b>	<b>LAB 02 - MINISHELL</b>	p. 10
2.1	Objetivos . . . . .	p. 10
2.2	Requisitos . . . . .	p. 10
2.3	Implementação . . . . .	p. 11
2.4	Conclusão . . . . .	p. 14

## 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.

### 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). 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. 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. 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 utilizando 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 terminator.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 pois 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 ações 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 programas de usuário conforme forem requisitados.

### **2.1 Objetivos**

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

### **2.2 Requisitos**

O nosso minishell fará a interface entre o usuário e a máquina. Para isso ele precisará fornecer mecanismos para que o usuário possa requisitar algo e receber a 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 algumas 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 character digitado foi o ENTER, logo ele sempre estará no final do buffer. precisamos removê-lo pois senão a função *execve* 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 iniciamos o loop principal do minishell. 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 possam 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 capazes 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`.