UNERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

Sistemas Operacionais I
Profª Regina Helena Carlucci Santana

1º Trabalho Prático - 1º Semestre 2011

Chamadas ao sistema do LINUX

Fábio Alexandrino 6792537
Rafael Sartori 6792861
Fernando Maia 6792652

Resumo

Esse trabalho tem como objetivo testar primitivas de chamada ao sistema Linux. Foram implementados pequenos programas para testar três chamadas de sistema de cada uma das seguintes categorias: gerenciamento de memória, processos, E/S (entrada e saída) e arquivos.

Sumário

Introdução

O Sistema operacional é um software que cuida da interface entre o programa e a máquina. Um de seus objetivos é fazer com que os aplicativos do usuário funcionem apropriadamente.

Um aplicativo pode precisar de alguns recursos para seu funcionamento, como alocar memória, ou manipulação de arquivos. Para tal, dispõem das chamadas de sistema (ou syscalls) - que são a forma dos aplicativos do usuário trocarem mensagens com o Sistema Operacional.

Revisão das chamadas ao sistema

As chamadas de sistema são normalmalmente escritas em linguagem de montagem ou em bibliotecas em C.

Estaremos testando as primitivas das funcionalidades mais comuns do SO: gerenciamento de memória, controle de processos, entrada e saída e arquivos.

Os processos são programas em execução, sendo que somente um pode ocupar o processador por vez. Para o usuário ter uma sensação de paralelismo, os processos ficam sendo trocados dentro do processador por certos intervalos de tempo. Um exemplo de syscall de processo seria o de matar um processo (primitiva fork).

Assim no gerenciamento de memória, podem ocorrer alocações e liberações de memória. Em arquivos, um arquivo pode ser aberto ou fechado. E por sua vez, em entrada e saída, o syscall pode mandar ler o teclado do usuário, escrevendo o conteúdo no prompt de comando.

Apresentação do código-fonte implementado

1) processos.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

/* ------------------------------------------------
Programa que testa 4 syscalls de processo:

pid_t getpid(void);
pid_t getppid(void);
pid_t fork(void);
_exit();
------------------------------------------------ */

int main (void) {
        int x;

        printf("#### System Calls de processos ####");
        printf("\n\nCapturando a identificação do processo atual...\n");
        printf("ID do processo atual: %d", getpid()); /* syscall que captura a ID do
processo atual */
        printf("\n\n");

        printf("Capturando a identificação do processo pai do atual...\n");
        printf("ID do processo pai: %d", getppid()); /* syscall que captura a ID do
processo do pai */
        printf("\n\n");

        printf("Criando um novo processo...\n");
```

```
	x = fork();

	if(x == -1) /* erro no fork() */
		printf("Falha ao executar system call fork");
	else
		printf("ID do novo processo: %d\n", fork()); /* syscall que cria novo
processo */

	printf("\n\n");
	_exit(0); /* syscall para sair do programa atual */

	return 0;
}
```

2) memoria.c

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/* ------------------------------------------------
Programa que testa syscalls de memoria indiretamente:

malloc()
free()
calloc()
------------------------------------------------ */

int main(void) {
	int i, *vetor;

	printf("#### System Calls de Memoria ####");
	printf("\n\nAlocando memoria com malloc... \n");
	/* Alocando memoria para o vetor de inteiros */
	vetor = malloc(3*sizeof(int));
```

```
	vetor[0] = 1;
	vetor[1] = 2;
	vetor[2] = 3;


	for(i=0; i < 3; i++) {
		printf("Valor de vetor[%d]: : %d\n", i, vetor[i]);
		printf("Endereco de vetor[%d]: : %p\n\n", i, &(vetor[i]));
	}

	/* Libera memoria alocada para o vetor */
	printf("Liberando memoria... \n");
	free(vetor);

	for(i=0; i < 3; i++) {
		printf("Valor de vetor[%d] apos liberar memoria: %d\n", i, vetor[i]);
		printf("Endereco de vetor[%d] apos liberar memoria: %p\n\n", i,
&(vetor[i]));
	}

	printf("\n");

	printf("Alocando memoria com calloc... \n");
	/* Alocando memoria para o vetor de inteiros */
	vetor = calloc(5, sizeof(int));
	vetor[0] = 5;
	vetor[1] = 4;
	vetor[2] = 3;
	vetor[3] = 2;
	vetor[4] = 1;

	for(i=0; i < 5; i++) {
		printf("Valor de vetor[%d]: : %d\n", i, vetor[i]);
		printf("Endereco de vetor[%d]: : %p\n\n", i, &(vetor[i]));
	}
```

```
	return 0;
}
```

3) arquivos.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

/* ------------------------------------------------
Programa que testa 3 syscalls de Arquivos:

int mkdir(const char *pathname, mode_t mode);
int chdir(const char *path);
int rmdir(const char *pathname);
------------------------------------------------ */

int main(){
	int i, novaPasta;
	int dif, dif2, dif3;

	printf("#### System Calls de Arquivos ####\n\n");

	/* Criando um diretorio */
	printf("Criando novo diretorio...\n");
	dif = mkdir("novaPasta", S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
	if(dif == 0)
		printf("Diretorio 'novaPasta' criado corretamente.\n");
	if(dif == -1)
		printf("Falhar ao criar diretorio.\n\n");
```

```
	/* Acessando um diretorio */
	printf("Acessando o diretorio criado...\n");
	dif2 = chdir("novaPasta");

	if(dif2 == 0)
		printf("Novo diretorio 'novaPasta' acessado corretamente.\n");
	if(dif2 == -1)
		printf("Falha ao acessar novo diretorio.\n");


	/* Acessando um diretorio */
	printf("Retornando para diretorio raiz...\n");
	dif2 = chdir("../");

	if(dif2 == 0)
		printf("Retornou ao diretorio raiz corretamente.\n");
	if(dif2 == -1)
		printf("Falha ao retornar ao diretorio raiz.\n");


	/* Removendo um diretorio */
	printf("Removendo diretorio 'novaPasta'...");
	dif3 = rmdir("novaPasta");
	if(dif3 == 0)
		printf("Diretorio 'novaPasta' foi removido corretamente.\n");
	if(dif3 == -1)
		printf("Falha ao remover diretorio.\n");


	return 0;
}
```

4) entradaSaida.c

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

/* ------------------------------------------------
Programa que testa 4 syscalls de entrada/saída:

int open(const char *pathname, int flags);
ssize_t read(int fd, void *vuf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int close(int fd);
------------------------------------------------ */

int main(void) {
	int file, flagclose, flagwrite, flagread;

	printf("#### System Calls de Entrada e Saída ####");
	/* Abrindo arquivo para escrita e leitura */
	file = open("Teste.txt", O_CREAT|O_RDWR);

	printf("\n\n");
	printf("Abrindo arquivo ...\n");

	if(file == -1)
		printf("Falha ao abrir o arquivo.\n");
	else
		printf("O arquivo foi aberto corretamente.\n");


	printf("\n\n");
	printf("Escrevendo no arquivo ...\n");

	/* Escrevendo no arquivo */
	flagwrite = write(file, "Trabalho de SO 1 - Entrada e Saída", 50);

	if(flagwrite == -1)
```

```
            printf("Falha na escrita do arquivo.\n");
        else
            printf("O arquivo foi escrito corretamente.\n");


        /* Fechando o arquivo */
        printf("\n\n");
        printf("Fechando o arquivo...\n");
        flagclose = close(file);

        if(flagclose != 0)
            printf("Falha ao fechar o arquivo.\n");
        else
            printf("Arquivo foi fechado corretamente.\n");

        return 0;
}
```

Apresentação dos testes e dos resultados obtidos

Para testar as syscalls, utilizamos o sistema operacional Linux e o código foi escrito na linguagem C.

1) **processos.c**

Neste arquivo foram utilizadas as seguintes primitivas:

- getpid: retorna o ID do processo corrente,
- getppid: retorna o ID do processo pai do corrente,
- fork: duplica o processo corrente gerando um novo processo.

Dessas três, a única que possibilita erros é a fork, por lidar com tempo e memória para realizar a duplicação.

```
#### System Calls de processos ####

Capturando a identificação do processo atual...
ID do processo atual: 32198

Capturando a identificação do processo pai do atual...
ID do processo pai: 3103

Criando um novo processo...
ID do novo processo: 32200


ID do novo processo: 32201


ID do novo processo: 0


ID do novo processo: 0
```

2) memoria.c

Aqui utilizamos funções mais convencionais de memória do C: malloc, free, calloc que trazem consigo o uso da primitiva mmap (que dá um nível de acesso completo ao mapeamento da memória).

```
#### System Calls de Memoria ####

Alocando memoria com malloc...
Valor de vetor[0]: : 1
Endereco de vetor[0]: : 0x98dc008

Valor de vetor[1]: : 2
Endereco de vetor[1]: : 0x98dc00c

Valor de vetor[2]: : 3
Endereco de vetor[2]: : 0x98dc010

Liberando memoria...
Valor de vetor[0] apos liberar memoria: 0
Endereco de vetor[0] apos liberar memoria: 0x98dc008

Valor de vetor[1] apos liberar memoria: 2
Endereco de vetor[1] apos liberar memoria: 0x98dc00c

Valor de vetor[2] apos liberar memoria: 3
Endereco de vetor[2] apos liberar memoria: 0x98dc010


Alocando memoria com calloc...
Valor de vetor[0]: : 5
Endereco de vetor[0]: : 0x98dc018

Valor de vetor[1]: : 4
Endereco de vetor[1]: : 0x98dc01c

Valor de vetor[2]: : 3
Endereco de vetor[2]: : 0x98dc020

Valor de vetor[3]: : 2
Endereco de vetor[3]: : 0x98dc024

Valor de vetor[4]: : 1
Endereco de vetor[4]: : 0x98dc028
```

3) arquivos.c

As primitivas usadas por esse arquivo foram:

- mkdir: cria um diretório
- chdir: acessa um diretório
- rmdir: remove um diretório, o qual deve estar vazio.

```
#### System Calls de Arquivos ####

Criando novo diretorio...
Diretorio 'novaPasta' criado corretamente.
Acessando o diretorio criado...
Novo diretorio 'novaPasta' acessado corretamente.
Retornando para diretorio raiz...
Retornou ao diretorio raiz corretamente.
Removendo diretorio 'novaPasta'...Diretorio 'novaPasta' foi removido corretamente.
```

4- entradaSaida.c

Neste arquivo foram utilizadas as seguintes primitivas:

- open: abre um arquivo
- read: realiza a leitura de um arquivo
- write: escreve em em um arquivo
- close: fecha o arquivo previamente aberto

```
#### System Calls de Entrada e Saída ####

Abrindo arquivo ...
O arquivo foi aberto corretamente.


Escrevendo no arquivo ...
O arquivo foi escrito corretamente.


Fechando o arquivo...
Arquivo foi fechado corretamente.
```

Conclusão

Através dos testes nestas quatro categorias, conseguimos visualizar essa hierarquia no controle do sistema operacional (que vem sido abordada desde o início do curso), onde as primitivas se encontram no nível mais elevado - tendo poder suficiente para estabelecer de fato a conexão entre programas de usuário (cujos acessos à recursos da máquina são restritos e/ou indiretos) e os dispositivos físicos.

Bibliografia

Linux Man Pages: http://www.linuxmanpages.com

Wikipedia: http://pt.wikipedia.com

Digilife:http://www.digilife.be/quickreferences/QRC/LINUX%20System%20Call%20Quick%20Reference.pdf

TANEMBAUM, Andrew S., WOODHULL, Albert S.; Sistemas Operacionais, projeto e implementação; 2ª edição, 2002.

Slides e anotações das Aulas de Sistemas Operacionais – 2011 – profª Regina Helena