

Laboratório de Sistemas Operacionais

Prof. André Leon S. Gradvohl, Dr.

gradvohl@ft.unicamp.br

31 de Março de 2019

Conteúdo

1	Introdução	2
2	Interação com o Sistema Operacional	4
2.1	Interagindo com o sistema operacional	4
2.2	Exercício	4
2.3	Obtendo informações sobre processos	4
2.4	Exercício	5
3	Obtendo informações sobre os processos	6
3.1	Obtendo informações sobre o processo, via programa	6
3.2	Exercício	8
4	Tratamento de Sinais	9
4.1	Exercício	10
5	Disparando vários processos	11
5.1	Exercício	12
5.2	Criando processos zumbis	12
5.3	Exercício	13
5.4	Processos pai e filho diferentes	13
5.5	Exercício	16
6	Compartilhamento de memória	17
6.1	Primitivas para compartilhamento de memória	17
6.2	Exercício	20
7	Programação <i>Multithread</i>	21
7.1	Exercício	23
8	Problema do Produtor-Consumidor	24
8.1	Problema do Produtor-Consumidor com <i>multithreads</i> e semáforos	24
8.2	Exercício	27

Capítulo 1

Introdução

O objetivo deste texto é descrever os exercícios usados no laboratório da disciplina Sistemas Operacionais. Essa disciplina é oferecida na Faculdade de Tecnologia da Universidade Estadual de Campinas (FT/UNICAMP) para os cursos Bacharelado em Sistemas de Informação e Tecnologia em Análise e Desenvolvimento de Sistemas.

Esse material pode ser utilizado por qualquer pessoa, de qualquer curso ou instituição, desde que respeitadas as condições da licença CC-BY-4.0, descrita na página 28. Informações de como obter o material também estão nessa página.

Supõe-se que o sistema operacional utilizado será o Linux. Portanto, todos os comandos descritos neste texto são para o Linux. Recomenda-se que o leitor navegue sequencialmente pelo texto. Assim, terá melhor aproveitamento do laboratório.

Alguns comandos básicos para o sistema Linux estão na Tabela 1.1 a seguir:








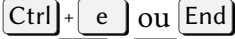

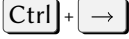
Tabela 1.1: Lista de comandos comuns no Linux.

Comando	Significado
<code>ls</code>	Lista os arquivos locais.
<code>cd dir</code>	Muda para o diretório <code>dir</code> .
<code>gedit arquivo &</code>	Abre o <code>arquivo</code> no editor de textos e libera o terminal para outros comandos.
<code>unzip arq.zip</code>	Descompacta o arquivo <code>arq.zip</code> .

Todos os comandos que serão utilizados nesse tutorial serão executados no interpretador da linha de comandos (*shell*), também chamado de terminal.

Há ainda algumas dicas de teclas para os usuários iniciantes no *bash* (o interpretador de comandos padrão no Linux). Elas estão resumidas na Tabela 1.2 a seguir.

Tabela 1.2: Teclas úteis no *bash*.

Tecla	Significado
	Repete o último comando.
	Repete o próximo comando.
	Envia um sinal de término para o processo.
	Completa o nome do comando ou do arquivo.
	Apaga a próxima palavra a frente do cursor.
	Apaga do cursor até o final da linha.
	Navega para o início da linha.
	Navega para o final da linha.
	Navega para a palavra anterior o cursor.
	Navega para a próxima palavra a frente do cursor.

Capítulo 2

Interação com o Sistema Operacional

2.1 Interagindo com o sistema operacional

O comando básico para obter informações sobre o sistema operacional é o

`uname -a`

Observe a saída desse comando:

```
Linux grid1.cna.unicamp.br 2.4.20-8 #1 Thu Mar 13 17:18:24 EST 2003 i686
↪ athlon i386 GNU/Linux
```

Entre as informações presentes na saída desse comando estão:

- o nome do sistema operacional;
- o nome da máquina;
- versão do kernel;
- plataforma de hardware.

2.2 Exercício

Utilize o comando `uname -a` em sua máquina e tente identificar a saída do comando.

2.3 Obtendo informações sobre processos

Existem dois comandos para obtenção de informações sobre processos: `ps` e `top`.

O comando `ps` informa o status dos processos de forma sucinta. As informações que o comando `ps` apresenta são:

- PID: identificador do processo;
- TTY: terminal onde o processo está sendo executado;
- TIME: tempo de processamento;
- CMD: comando instanciado.

O comando **top** é um pouco mais poderoso, pois reporta mais informações. Entre tais informações estão:

- tempo em que o sistema está no ar;
- carga média do sistema;
- informações da CPU:
 - porcentagem de tempo dedicada aos processos do usuário;
 - porcentagem de tempo dedicada aos processos do sistema;
 - porcentagem de tempo sem processamento (*idle*).
- informações sobre a memória:
 - memória total;
 - memória livre;
 - memória compartilhada;
- informações sobre os processos:
 - PID: identificador do processo;
 - USER: nome do usuário;
 - PRI: prioridade;
 - SIZE: tamanho do processo em kbytes;
 - RSS: tamanho total de memória física do processo;
 - SHARE: tamanho total de memória compartilhada;
 - STAT: estado do processo, que pode ser S (*sleeping*) ou R (*running*).

2.4 Exercício

Utilize o comando **top** em sua máquina e tente identificar as informações providas pelo comando.

Capítulo 3

Obtendo informações sobre os processos

Neste capítulo, vamos verificar como obter informações sobre o próprio processo a partir dele mesmo.

3.1 Obtendo informações sobre o processo, via programa

É possível construir programas que interajam com o sistema operacional e obtenham algumas informações. Observe o código do programa a seguir:

```
/**
 * Programa para capturar informacoes sobre um processo.
 * Desenvolvido por:
 *   Prof. Andre Leon S. Gradvohl, Dr.
 *
 * Ultima atualizacao:
 *   26/11/2003
 *
 * Para compilar:
 *   gcc infoProcesso.c -o infoProcesso
 */
#include <stdlib.h>      // Cabecalho de Biblioteca padrao
#include <stdio.h>       // Cabecalho de Biblioteca de I/O padrao
#include <sched.h>       // Cabecalho de Biblioteca de escalonamento
#include <sys/types.h>   // Cabecalho de Biblioteca
#include <sys/utsname.h> // Cabecalho de Biblioteca
#include <unistd.h>      // Cabecalho de Biblioteca

#define Kbyte 1024.
#define Mbyte 1048576. //(1024 Kbytes)
#define Nelem 3

int main(void)
{
    pid_t idProcesso;
    pid_t idProcessoPai;
    uid_t idUsuario;
    gid_t idGrupo;

    long memTotal;
    long memDisp;
    int tamPagina;

    double carga[Nelem];

    char dirTrabalho[100];
    char str[30];

    int n;
    int politicaEscalonamento;
```

```

struct utsname info;

puts("Programa para captura de informacoes sobre o processo.");

// Captura o id desse processo.
idProcesso = getpid();

// Captura o id do processo pai.
idProcessoPai = getppid();

// Captura o id do usuario
idUsuario = getuid();

// Captura o id do grupo
idGrupo = getgid();

// Captura o diretorio de trabalho desse processo
getcwd(dirTrabalho, 99);

puts("Informacoes sobre o processo:");
printf("\t0 identificador do meu processo e: %d\n", idProcesso);
printf("\t0 identificador do meu processo pai e: %d\n", idProcessoPai);
printf("\t0 identificador de usuario desse processo e: %d\n", idUsuario);
printf("\t0 identificador de grupo desse processo e: %d\n", idGrupo);
printf("\t0 diretorio de trabalho eh: %s\n", dirTrabalho);

// Captura o tamanho da pagina
tamPagina = getpagesize();

/* Captura a quantidade de paginas de memoria a multiplica pelo
   tamanho da pagina */
memTotal = sysconf(_SC_PHYS_PAGES) * tamPagina;

/* Captura a quantidade de paginas de memoria disponiveis e
   a multiplica pelo tamanho da pagina */
memDisp = sysconf(_SC_AVPHYS_PAGES) * tamPagina;

puts("Informacoes sobre a memoria:");
printf("\t0 tamanho da pagina e: %d (%.0f Kbytes)\n",
       tamPagina, tamPagina/Kbyte);
printf("\t0 tamanho total da memoria: %ld (%f Mbytes)\n",
       memTotal, memTotal/Mbyte);
printf("\t0 tamanho de memoria disponivel: %ld (%f Mbytes)\n",
       memDisp, memDisp/Mbyte);

// Captura a media de carga do sistema: numero de processos/tempo
n = getloadavg(carga, Nelem);
if (n > -1)
{
    printf("Media de carga: \n");
    printf("\t no ultimo minuto: %f\n", carga[0]);
    printf("\t nos ultimos 5 minutos: %f\n", carga[1]);
    printf("\t nos ultimos 15 minutos: %f\n", carga[2]);
}

// Captura a politica de escalonamento utilizada pelo SO
politicaEscalonamento = sched_getscheduler(idProcesso);

puts("A politica de escalonamento:");
switch(politicaEscalonamento)
{
    case SCHED_FIFO: puts("\tPolitica FIFO");
        break;
    case SCHED_RR: puts("\tPolitica RoundRobin");
        break;
    case SCHED_OTHER: puts("\tPolitica default");
        break;
    default: puts("Erro!");
}

// Captura informacoes sobre o sistema

```



```
uname(&info);

printf("Informacoes do sistema:\n");
printf("\tNome do S.O.: %s\n", info.sysname);
printf("\tRelease do S.O.: %s\n", info.release);
printf("\tVersao do S.O.: %s\n", info.version);
printf("\tHardware: %s\n", info.machine);
printf("\tNome do host: %s\n", info.nodename);

puts("Digite algo e tecle <enter> para encerrar.");
scanf("%s", str);
return 1;
}
```

3.2 Exercício

Compile o programa anterior e execute-o.

Antes de compilar o programa, mude para o diretório onde se encontra o arquivo `infoProcesso.c`, com o seguinte comando:

```
cd Processos
```

Para compilar o programa utilize o comando a seguir:

```
gcc infoProcesso.c -o infoProcesso
```

Capítulo 4

Tratamento de Sinais

Sinais são usados para notificar um processo ou segmento de um evento particular. Pode-se comparar o tratamento de sinais com interrupções de hardware, que ocorrem quando um subsistema de hardware, por exemplo uma interface de E/S de disco, gera uma interrupção para o processador quando a E/S é concluída.

Este evento, por sua vez, faz com que o processador chame um tratador de interrupções. Assim, o processamento subsequente pode ser feito no sistema operacional com base na fonte e da causa da interrupção.

Observe como isso pode ser feito no programa `sinais.c` a seguir

```
/**
 * Programa para exemplificar o tratamento de sinais.
 * Desenvolvido por Prof. Andre Leon S. Gradvohl, Dr.
 * Atualizado em: 03/08/2011
 */
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

void trataSinal(int numSinal)
{
    switch(numSinal)
    {
        case SIGINT:
            fprintf(stderr, "Tentou usar o Ctrl-C\n");
            break;
        case SIGHUP:
            fprintf(stderr, "Recebi um sinal HUP\n");
            fprintf(stderr, "Agora ignorando o SIGHUP\n");
            /* SIG_IGN e usado para ignorar sinais. SIGKILL e SIGSTOP
             * nao podem ser ignorados.
             */
            signal(SIGHUP, SIG_IGN);
        case SIGQUIT:
            fprintf(stderr, "Recebi um sinal de termino!\n Adeus!\n");
            exit(0);
    }
}

int main()
{
    //Registrando os sinais.
    signal(SIGINT, trataSinal);
    signal(SIGHUP, trataSinal);
}
```

```
signal(SIGQUIT, trataSinal);

fprintf(stderr, "use o comando 'kill -HUP %d' ou \n", getpid());
fprintf(stderr, "'kill -QUIT %d' para encerrar o processo\n", getpid());

while(1)
{
    puts(".");
    sleep(2);
}
return 0;
}
```

4.1 Exercício

Antes de compilar o programa, mude para o diretório onde se encontra o arquivo `sinais.c`, com o seguinte comando:

```
cd ../Sinais
```

Agora para compilar o programa utilize o comando a seguir:

```
gcc sinais.c -o sinais.o
```

Depois de compilado, será necessário abrir uma segunda janela do terminal. Na primeira janela, você executará o programa `./sinais.o`.

Depois que o programa entrar em execução, tente pressionar as teclas `Ctrl` + `c` para ver se o programa termina.

Para encerrar de fato o programa, na segunda janela, utilize o comando `kill` para enviar um sinal de término para o programa. Para isso, utilize o comando a seguir:

```
kill -QUIT <pid>
```

onde `<pid>` é o identificador do processo na primeira janela.

Importante: para saber o identificador do processo `./sinais` que está em execução na primeira janela, use o comando a seguir:

```
ps -ef | grep sinais.o
```

Capítulo 5

Disparando vários processos

A primitiva `fork()` é utilizada para, a partir de um processo, criar outro processo com as mesmas características do primeiro. Na verdade, a primitiva `fork()` faz uma cópia do processo pai em um processo filho, fazendo com que ambos continuem a sua execução do ponto imediatamente posterior à primitiva `fork()`.

A primitiva `fork()` tem três saídas distintas:

- -1 se houve problemas (nesse caso o filho não é criado);
- 0, para o processo filho;
- identificador do filho, para o processo pai.

Observe o programa a seguir e tente entender o funcionamento da primitiva `fork()`.

```
/**
 * Programa para ilustrar a criacao de um processo filho
 * a partir do processo pai.
 * Desenvolvido por:
 *   Prof. Andre Leon S. Gradvohl, Dr.
 *
 * Ultima atualizacao:
 *   26/11/2003
 *
 * Para compilar:
 *   gcc PaiFilho.c -o PaiFilho
 */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    int pid;
    int paiPid;
    int ret;
    int i=0;

    pid = getpid();
    printf("Pronto para o fork. Meu id e:%d\n",pid);

    sleep(1);

    ret = fork();

    if (ret < 0) // Problemas no fork
    {
        perror("Impossivel fazer o fork!\n");
        return 1;
    }
}
```

```

if (ret == 0) // Se verdade, sou o processo filho
{
    pid = getpid();
    paiPid = getppid();
    printf("Sou o processo filho!\n");
    printf("\tMeu id e: %d.\n",pid);
    printf("\tO id do meu Pai e: %d\n", paiPid);
    return 0;
}
else // Senao sou o processo pai
{
    pid = getpid();
    printf("Sou o processo Pai!\n");
    printf("\tMeu id e: %d. \n",pid);
    printf("\tO id do meu filho e: %d\n", ret);
    return 0;
}
}

```

5.1 Exercício

Compile o programa anterior e execute-o.

Antes de compilar o programa, mude para o diretório onde se encontra o arquivo `PaiFilho.c`, com o seguinte comando:

```
cd ../Processos
```

Para compilar o programa utilize o comando a seguir:

```
gcc PaiFilho.c -o PaiFilho
```

5.2 Criando processos zumbis

Um processo zumbi é o processo que já terminou sua execução, mas que ainda está na tabela de processos por algum motivo. Um desses motivos é que, por algum *bug* no sistema operacional, a tabela de processos ainda não foi atualizada, eliminando o identificador do processo.

A princípio, um processo zumbi não é um problema sério para o sistema operacional. No entanto, a presença de zumbis pode indicar *bugs* no sistema ou problemas de segurança do tipo *Denial of service*.

No exemplo a seguir, vamos forçar a criação de processos zumbis.

```

/**
 * Programa exemplo para ilustrar a existencia de processos zumbis.
 * Desenvolvido por:
 *   Prof. Andre Leon S. Gradvohl, Dr.
 *
 * Ultima atualizacao:
 *   02/04/2019
 *
 * Para compilar:
 *   gcc zumbi.c -o zumbi.o
 */
#include <stdio.h>
#include <stdlib.h>

```

```

#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>

int main ()
{
    pid_t pidFilho;

    // Executa um fork() para criar um processo filho.
    pidFilho = fork ();

    if (pidFilho > 0) {
        // Processo Pai vai dormir por 30 segundos e
        // sair, sem uma chamada para o wait.
        fprintf(stderr, "Processo Pai. PID: %d dormindo 30 segundos\n", getpid());
        fprintf(stderr, "Em outra janela, execute o comando a seguir:\n\t");
        fprintf(stderr, "top -p %d -p %d\n", getpid(), pidFilho);
        sleep(30);
        exit(0);
    }
    else if (pidFilho == 0) {
        // Processo Filho vai sair imediatamente
        fprintf(stderr, "Processo Filho. PID: %d\n", getpid());
        exit(0);
    }
    else if (pidFilho == -1)
    { // Erro no fork()
        perror("Falha no fork().");
        exit(1);
    }
    else // Isso nao deve acontecer.
    {
        fprintf(stderr, "Valor de retorno %d da chamada ao fork() desconhecido.",
            ↪ pidFilho);
        exit(2);
    }
    return 0;
}

```

5.3 Exercício

Antes de compilar o programa `zumbi.c`, abra uma outra janela do terminal. Você precisará executar o comando `top` na segunda janela, enquanto o programa é executado na primeira.

Compile o programa `zumbi.c` com o seguinte comando:

```
gcc zumbi.c -o zumbi.o
```

Agora, execute o programa `./zumbi.o` em uma janela e na outra execute o comando a seguir:

```
top -p <id_pai> -p <id_filho>
```

Os valores para `<id_pai>` e `<id_filho>` serão fornecidos pelo programa `zumbi.o`.

5.4 Processos pai e filho diferentes

A princípio, a primitiva `fork()` cria um processo filho exatamente igual ao seu processo pai. Entretanto, cada um deles fica em um espaço de memória diferente.

Contudo, há situações em que é necessário que cada processo – pai e filho – execute códigos diferentes. No exemplo a seguir, ilustra-se a primitiva `execvp()` para executar programas diferentes a partir de um determinado processo.

A primitiva `execvp()` faz parte de uma família de primitivas que substitui a imagem do processo atual por uma nova. A imagem de um processo são os códigos (programa) que aquele processo executa e os respectivos dados.

A sintaxe da primitiva `execvp()` é a seguinte:

```
int execvp(const char *file, char *const argv[]);
```

onde:

- O valor de retorno é sempre `-1`. Mas, se isso acontecer, significa que houve um erro na execução da primitiva;
- `file` é nome do programa;
- `argv[]` é um vetor de *strings* com os argumentos do programa. Importante: a primeira posição do vetor `argv` deve ter o caminho completo para o programa e última posição do vetor deve ter valor `NULL`.

O exemplo a seguir ilustra o programa que representa os processos pai e filho.

pai.c

```
/**
 * Programa exemplo para ilustrar a criacao de processos filhos
 * a partir de processos pais. Alem da criacao, os processos filhos
 * assumirao processos diferentes dos processos pais.
 *
 * Desenvolvido por:
 * Prof. Andre Leon S. Gradvohl, Dr.
 *
 * Ultima atualizacao:
 * 02/04/2019
 *
 * Para compilar:
 * gcc pai.c -o pai.o
 *
 * Observacao: precisa que o programa filho.c esteja compilado.
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>

int main(void)
{
    pid_t filho; // Identificador do procesos filho.
    int statusFilho; // Status de saida do filho.
    pid_t c; // Identificador do filho que sera retornado pelo wait.
    char *args[3]; // Lista de argumentos para o processo filho.

    // Define os argumentos para o programa filho.
    args[0] = "./filho.o"; // Nome do programa filho.
    args[1] = "2"; // Argumentos para o programa filho.
    args[2] = NULL; // Indica o fim dos argumentos para o programa.
```

```

filho = fork(); // Cria o processo filho atraves da primitiva fork().
if (filho == 0) // Se o processo filho foi criado, este if sera verdadeiro
{
    printf("PID do filho = %ld\n", (long) getpid());

    /**
     * Substitui a imagem do filho pela imagem do programa
     * "filho.o", com os respectivos argumentos.
     */
    execvp(args[0], args);

    /**
     * Se o processo filho alcanca este ponto,
     * entao a primitiva execvp falhou.
     */
    fprintf(stderr, "O processo filho nao pode executar a primitiva execvp.\n
    ↪ ");
    exit(1);
}
else // O processo pai entrara neste else.
{
    if (filho == (pid_t)(-1))
    {
        fprintf(stderr, "Fork falhou.\n");
        exit(1);
    }
    else
    {
        printf("Esperando o filho terminar!\n");
        c = wait(&statusFilho); //Esperando o filho terminar.
        printf("Pai: filho (%ld) terminou com status = %d\n", (long) c,
        ↪ statusFilho);
    }
}
return 0;
}

```

filho.c

```

/**
 * Programa exemplo para ilustrar a criacao de processos filhos
 * a partir de processos pais. Alem da criacao, os processos filhos
 * assumirao processos diferentes dos processos pais.
 *
 * Desenvolvido por:
 * Prof. Andre Leon S. Gradvohl, Dr.
 *
 * Ultima atualizacao:
 * 02/04/2019
 *
 * Para compilar:
 * gcc filho.c -o filho.o
 *
 * Observacao: precisa que o programa pai.c esteja compilado.
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
    unsigned int tempo=0;
    printf("Sou o novo processo filho.\n");
    printf("\tMeu id: %d. Id do processo pai: %d\n",
        getpid(), getppid());
}

```



```
if (argc == 2) // Se a quantidade de argumentos na linha de comando for 2.
{
    tempo = atoi(argv[1]);
    printf("Colocando este processo (filho) para dormir %d segundos\n",
        ↳ tempo);
    sleep(tempo);
}
return 0;
}
```

5.5 Exercício

Compile os programas `pai.c` e `filho.c` separadamente com os seguintes comandos:

```
gcc pai.c -o pai.o
```

```
gcc filho.c -o filho.o
```

Agora execute o programa `./pai.o` e veja o resultado.

Capítulo 6

Compartilhamento de memória

Conforme discutido em sala de aula, é possível fazer com que dois ou mais processos compartilhem memória. Essa é uma forma para fazer com que dois processos possam se comunicar.

6.1 Primitivas para compartilhamento de memória

As primitivas usadas para fazer o compartilhamento e acesso são:

- `shmget`: retorna o identificador do segmento de memória compartilhado;
- `shmat`: anexa o segmento de memória compartilhado ao espaço de endereçamento do processo;
- `shmdt`: desanexa o segmento de memória compartilhado ao espaço de endereçamento do processo.

Observe o que os programas a seguir fazem. O primeiro é o programa `shm_serv.c` que disponibiliza um segmento de memória. O segundo é o programa `shm_cli.c` que acessa o segmento compartilhado.

`shm_serv.c`

```
/**
 * Programa desenvolvido para ilustrar o compartilhamento de
 * memoria principal entre processos.
 *
 * Baseado no programa shm_server.c disponivel em
 * http://www.cs.cf.ac.uk/Dave/C de David Marshall.
 *
 * Atualizado em: 15/12/2003
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <unistd.h>

// Definicao do Tamanho do segmento compartilhado
#define TamSegCompart 27

// Definicao do identificador do segmento compartilhado.
```

```

#define IDSegCompart 5678

int main(void)
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    // Criando/Atribuindo o ID do Segmento Compartilhado
    key = IDSegCompart;

    /* Criando o segmento.
     * 0666 -> Permissao para leitura e escrita,
     * para usuario, grupo e outros
     */
    shmid = shmget(key, TamSegCompart, IPC_CREAT | 0666);

    if (shmid < 0)
    {
        perror("Erro no shmget");
        return 1;
    }

    /* Vinculando o segmento ao espaco de enderecamento
     * Note que o segundo parametro e NULL. Isso significa
     * que a primitiva shmat vai encontrar um endereco nao
     * usado para vincular o segmento. Essa e a melhor forma.
     */
    shm = shmat(shmid, NULL, 0);

    if (shm == (char *) -1)
    {
        perror("Erro no shmat");
        return 1;
    }

    s = shm;

    // Inserindo alguns dados no segmento compartilhado.
    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = 0; // NULL

    /* Aguarda ate que o outro processo coloque um "*"
     * primeira posicao do segmento de memoria
     * compartilhado
     */
    while (*shm != '*')
        sleep(1);

    /* Desvincula o segmento compartilhado */
    if (shmdt(shm))
    {
        perror("Erro na shmdt");
        return 1;
    }

    return 0;
}

```

shm_cli.c

```
/**
 * Programa desenvolvido para ilustrar o compartilhamento de
 * memoria principal entre processos.
 *
 * Baseado no programa shm_client.c disponivel em
 * http://www.cs.cf.ac.uk/Dave/C de David Marshall.
 *
 * Atualizado em: 15/12/2003
 */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <unistd.h>

// Definicao do Tamanho do segmento compartilhado
#define TamSegCompart 27

// Definicao do identificador do segmento compartilhado.
#define IDSegCompart 5678

int main(void)
{
    int shmid;
    key_t key;
    char *shm, *s;

    // Criando/Atribuindo o ID do Segmento Compartilhado
    key = IDSegCompart;

    // Localizando o segmento.
    shmid = shmget(key, TamSegCompart, 0666);
    if (shmid < 0)
    {
        perror("Erro no shmget");
        return 1;
    }

    /* Vinculando o segmento ao espaco de enderecamento
     * Note que o segundo parametro e NULL. Isso significa
     * que a primitiva shmat vai encontrar um endereco nao
     * usado para vincular o segmento. Essa e a melhor forma.
     */
    shm = shmat(shmid, NULL, 0);

    if (shm == (char *) -1)
    {
        perror("Erro no shmat");
        return 1;
    }

    // Lendo o que o outro processo deixou na memoria
    for (s = shm; *s != 0 /*NULL*/; s++)
        putchar(*s);
    putchar('\n');

    /* Escrevendo '*' na primeira posicao de memoria
     * para notificar que ja leu o segmento.
     */
    *shm = '*';

    /* Desvincula o segmento compartilhado */
    if (shmdt(shm))
    {
        perror("Erro na shmdt");
        return 1;
    }
    return 0;
}
```

6.2 Exercício

Compile ambos os programas e, em seguida, execute em uma janela o programa `shm_serv.o` e em outra janela o programa `shm_cli.o`.

Antes de compilar o programa, mude para o diretório onde se encontram os arquivos `shm_serv.c` e `shm_cli.c`, com o seguinte comando:

```
cd ../CompartmentMem
```

Para compilar o programa utilize as linhas de comando a seguir:

```
gcc shm_serv.c -o shm_serv
```

```
gcc shm_cli.c -o shm_cli
```

Agora, em uma das janelas execute primeiro o programa `./shm_serv.o` e, depois, na segunda janela execute o programa `./shm_cli.o`. Veja o resultado.

Capítulo 7

Programação *Multithread*

Uma outra forma de fazer duas ou mais tarefas ao mesmo tempo é utilizado *multithreads*. Conforme já discutido em sala de aula, *multithreads* são diferentes linhas de execução em um processo.

Existem algumas bibliotecas para trabalhar com *multithreads*. Por exemplo, a POSIX *Threads* (PThreads) – que utilizaremos neste tutorial – e a OpenMP, cujo paradigma é diferente da PThreads.

Neste tutorial, serão utilizadas as seguintes primitivas da biblioteca PThreads:

- `pthread_create()`: responsável pela criação de uma *thread*.
- `pthread_exit()`: responsável por retornar um valor de uma *thread*.
- `pthread_join()`: adiciona uma barreira para aguardar por uma segunda *thread*;
- `pthread_self()`: obtém o identificador da *thread*.

Para ilustrar observe o programa `thrd.c` a seguir. Ele dispara duas *threads* que “dormem” um tempo aleatório.

`thrd.c`

```
/**
 * Este e um programa simples para exemplificar a utilizacao
 * de Threads.
 *
 * Desenvolvido por: Prof. M.Sc. Andre Leon S. Gradvohl
 * E-mail: andre_gradvohl@yahoo.com
 *
 * Outros arquivos necessarios para a execucao desse
 * programa sao:
 * - funcoes.h
 * - funcoes.c
 *
 * Ultima atualizacao: 14/Fev/2003
 */

/* Inclusao de cabecalhos*/
#include <stdio.h>
#include <unistd.h> // Cabecalho para a funcao sleep
#include <pthread.h> // Cabecalho especifico para threads POSIX
#include "funcoes.h" // Cabecalho para a funcoes que serao os threads.

pthread_t meutid;          /* Variavel que armazena o id do thread principal */
pthread_t outrosTIDs[2];  /* Vetor que armazena o id dos outros threads */
```

```

int main( void )
{
    meutid = pthread_self(); // Funcao que captura o id do thread.
    printf ("Meu Thread ID = %ld\n",meutid);

    printf("Disparando Thread Func1");
    /**
     * A funcao a seguir cria um thread (linha de execucao) para a funcao "func1
     * ↪ ".
     * O prototipo da funcao e:
     *   int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
     *                       void *(*start_routine, void*), void *arg);
     * Onde:
     *   "thread" e o identificador do thread que se quer criar.
     *   "attr" sao os atributos do Thread (Geralmente NULL).
     *   "start_routine" e a funcao onde estao os threads.
     *   "arg" sao os parametros da "start_routine".
     * A funcao retorna
     */
    pthread_create(&outrosTIDs[0], NULL, sub_a, NULL);
    printf("(id = %ld)\n", outrosTIDs[0]);

    printf("Disparando Thread Func2");
    /**
     * A funcao a seguir cria um thread (linha de execucao) para a funcao "func2
     * ↪ ".
     */
    pthread_create(&outrosTIDs[1], NULL, sub_b, NULL);
    printf("(id = %ld)\n", outrosTIDs[1]);

    printf("Aguardando finalizacao dos Threads id=%ld e id=%ld\n",
           outrosTIDs[0], outrosTIDs[1]);

    /**
     * A funcao a seguir bloqueia o processo ate que o thread indicado termine.
     * O prototipo da funcao e:
     *   int pthread_join(pthread_t thread, void **value_ptr);
     * Onde:
     *   "thread" e o identificador do thread que se espera terminar.
     *   "value_ptr" e o valor de retorno da funcao
     * A funcao retorna 0 se funcionou corretamente e um valor
     * diferente de 0 para indicar erro.
     */
    pthread_join(outrosTIDs[1], NULL);
    pthread_join(outrosTIDs[0], NULL);
    printf("Threads id=%ld e id=%ld finalizados\n", outrosTIDs[0], outrosTIDs
           ↪ [1]);

    return 1;
}

```

A definição das funções chamadas pelo programa principal estão no arquivo a seguir.

funcoes.c

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include "funcoes.h" // Cabecalho que contem o prototipo dessas funcoes

void *sub_a(void *arg)
{
    register int i=0;
    register int tempoEspera;

    for (i=0;i<30;i+=2)

```

```

    {
        tempoEspera =(rand() % 3)+1; //tempo aleatorio 1, 2 ou 3 seg.
        printf("\ni = %d. Tempo de Espera: %d\n",i, tempoEspera);
        sleep(tempoEspera); //Dorme um tempo.
    }
    pthread_exit(NULL);
}

void *sub_b(void *arg)
{
    register int j=1;
    register int tempoEspera;

    for (j=1;j<30;j+=2)
    {
        tempoEspera =(rand() % 3)+1; //tempo aleatorio 1, 2 ou 3 seg.
        printf("\njota = %d. Tempo de Espera: %d\n",j, tempoEspera);
        sleep(tempoEspera); //Dorme um tempo.
    }
    pthread_exit(NULL);
}

```

7.1 Exercício

Compile e execute o programa anterior. Antes de compilar o programa, mude para o diretório onde se encontram os arquivos `funcoes.c` e `thrd.c`, com o seguinte comando:

```
cd ../Thread
```

Para compilar, utilize a seguinte linha de comando:

```
gcc -lpthread funcoes.c thrd.c -o thrd
```

Observação: a chave `-lpthread` indica que será usada a biblioteca `pthread` para Linux.

Capítulo 8

Problema do Produtor-Consumidor

Um dos problemas discutidos em sala de aula é o do produtor-consumidor. Em linhas gerais, existem dois processos, um produtor e um consumidor, que competem pelo uso de um recurso (no caso um *buffer*).

O produtor gera dados e os armazena no *buffer*. O consumidor, por sua vez, lê dados do *buffer* e os utiliza. A região crítica é o *buffer*, pois apenas um dos processos deve estar utilizando o *buffer* a cada instante. O sistema operacional deve prover meios de garantir essa exclusão mútua.

8.1 Problema do Produtor-Consumidor com *multithreads* e semáforos

Para resolver o problema do Produtor-Consumidor com multithread serão criados três semáforos *mutex*, *vazio* e *cheio*, conforme a solução vista em sala de aula.

Observe as primitivas para inicializar semáforos (*sem_init*), executar o *up* (*sem_wait*) e o *down* (*sem_post*).

Com base nessa explicação, observe o programa a seguir:

```
/**
 * Programa desenvolvido para ilustrar a solucao do problema do
 * produtor/consumidor com o uso de threads e semaforos.
 *
 * Comentado por Prof. Andre Leon S. Gradvohl, Dr.
 *
 * Atualizado em: 03/08/2011
 */
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define N 20
#define VEZES 60

struct timespec t;

sem_t mutex, vazio, cheio;
//vazio: semaforo utilizado pra controlar os sinais relativos as posicoes
//      ↳ vazias no buffer
//cheio: semaforo utilizado pra controlar os sinais relativos as posicoes
//      ↳ preenchidas no buffer
```

```

//mutex: semaforo binario utilizado p/ garantir exclusao mutua na regioa
    ↪ critica

int buffer[N], proxPosCheia, proxPosVazia, cont;
//buffer: utilizado para armazenar os dados produzidos pelo produtor e
    ↪ consumidos pelo consumidor
//proxPosCheia: proxima posicao cheia
//proxPosVazia: proxima posicao vazia
//cont: usado para controlar a quantidade de dados presentes no buffer

void *produtor(void *);
void *consumidor(void *);

int main(void)
{
    //Seta a semente da funcao geradora de numeros aleatorios
    srand(time(NULL));

    cont = 0;
    proxPosCheia = 0;
    proxPosVazia = 0;

    /*Inicializa os semaforos
    1o parametro: variavel semaforo
    2o parametro: indica se um semaforo sera compartilhado entre as
        ↪ threads de um processo ou entre processos
        o valor 0 indica q/ o semaforo sera compartilhado entre
            ↪ as threads de um processo (digit o comando
                "man sem_init" no shell do linux p/ ver os detalhes)
    3o parametro: valor inicial do semaforo
    */
    sem_init(&mutex, 0, 1);
    sem_init(&vazio, 0, N);
    sem_init(&cheio, 0, 0);

    pthread_t thd0, thd1;

    /*
    Inicializa as threads
    1o parametro: variavel thread
    2o parametro: indica se uma thread e "joinable", ou seja, se a thread
        ↪ nao sera finalizada ate chegar a uma chamada de funcao
            pthread_join().
    3o parametro: indica o nome do metodo que ira compor o trecho de
        ↪ codigo q/ sera executado pela thread
    4o parametro: utilizado qdo se necessita passar algum paramentro a
        ↪ thread. Pode se passar quaisquer tipos de dados,
            inclusive uma estrutura de dados qdo houver a
                ↪ necessidade de passar mais de um parametro.
            (dentro do metodo chamado realiza-se um cast p/
                ↪ recuperar os dados)
    */
    pthread_create(&thd0, 0, (void *) produtor, NULL);
    pthread_create(&thd1, 0, (void *) consumidor, NULL);

    //Esses dois metodos indicam q/ a tread nao sera finalizada ate
        ↪ ocorrer a chamada dos mesmos (como mencionado anteriormente)
    pthread_join(thd0,0);
    pthread_join(thd1,0);

    printf("\n");
    exit(0);
}

// Metodo que produz os itens q/ serao inseridos no buffer (numeros
    ↪ aleatorios)
int produz_item()
{
    int val;
    val = rand() % 100;
    printf("\nProduzindo item: %d", val);
    return val;
}

```

```

}

// Metodo utilizado p/ mostra o valor q foi consumido (meramente implementado
  ↳ p/ fins didaticos)
void consome_item(int item)
{
    printf("\nConsumindo item: %d", item);
}

//Metodo que a realiza a insercao do dado no buffer
void insere_item(int val)
{
    if(cont < N)
    {
        buffer[proxPosVazia] = val;
        // A utilizacao da divisao em modulo implementa um
        ↳ comportamento circular da utilizacao do buffer
        // ou seja, qdo o contador chegar no valor de N (N % N = 0) o
        ↳ valor da variavel voltara ao inicio do buffer
        proxPosVazia = (proxPosVazia + 1) % N;
        cont = cont + 1;
        if(cont == N)
            printf("\n##### Buffer completo
                ↳ #####");
    }
}

// Metodo que realiza a retirada do dado do buffer
int remove_item()
{
    int val;
    if(cont > 0)
    {
        val = buffer[proxPosCheia];
        proxPosCheia = (proxPosCheia + 1) % N;
        cont = cont - 1;
        return val;
    }
}

void *produtor(void *p_arg)
{
    int item;
    register int i=0;
    while(i++<VEZES)
    {
        item = produz_item();

        // sem_wait (realiza o down no semaforo (ver pag. 81 do livro
        ↳ Sistemas Operacionais - 2a edicao - Tanenbaum ))
        // sem_post (realiza o up no semaforo)
        sem_wait(&vazio);
        sem_wait(&mutex);

        insere_item(item);

        sem_post(&mutex);
        sem_post(&cheio);

        sleep(item%2);
    }
}

void *consumidor(void *p_arg)
{
    int item;
    register int i=0;
    while(i++<VEZES)
    {
        sem_wait(&cheio);
        sem_wait(&mutex);
    }
}

```

```
        item = remove_item();  
        sem_post(&mutex);  
        sem_post(&vazio);  
        consome_item(item);  
        sleep(item%3);  
    }  
}
```

8.2 Exercício

Antes de compilar o programa, mude para o diretório onde se encontra o arquivo `principal.c`, com o seguinte comando:

```
cd ../ProdCons
```

Compile o programa anterior com a seguinte linha de comando:

```
gcc -lpthread prod_cons.c -o prod_cons.o
```

Agora execute o programa `./prod_cons.o` e veja o resultado.

Licença de uso

This work is licensed under a Creative Commons “Attribution 4.0 International” license.



Essa licença permite que o usuário copie e redistribua o material em qualquer meio ou formato. Permite ainda que o usuário remixe, transforme, e use o material para complementar outros materiais para qualquer propósito, mesmo os comerciais.

Detalhes sobre a licença estão disponíveis no *site* a seguir:

<https://creativecommons.org/licenses/by/4.0>

Todos os códigos fontes na linguagem C utilizados neste texto, bem como o *script* para a instalação dos códigos fontes, os arquivos compactados e o código fonte em \LaTeX deste texto estão disponíveis no site do GitHub e indexados no site do Zenodo conforme os endereços a seguir.

GitHub: <https://github.com/gradvohl>

DOI: <https://doi.org/10.5281/zenodo.1307495>

Para citar este texto, use as informações a seguir:

GRADVOHL, A. L. S. Laboratório de Sistemas Operacionais (Versão 1.0). Zenodo. Disponível em <http://doi.org/10.5281/zenodo.1455756>, 2019.