Universidade Federal do Rio Grande do Norte
Centro de Tecnologia
Departamento de Engenharia Elétrica
Laboratório de Engenharia de Computação e Automação

Curso de Graduação

"Programação em Tempo Real"

AUTOR: Celso Alberto Saibel Santos

Natal - Outubro de 2000

Prefácio

O material apresentado nesta apostila é uma coletânea dos seguintes trabalhos:

1. *Vous avez dit: IPC sous UNIX System V* de autoria de Olivier Bernard, Bernard Bolz, Thierry Elsensohn, Laboratoire LGI−IMAG, École Nationale Supérieure d Informatique et de Mathematiques Appliquées de Grenoble, 1992;

2. *An Introduction 4.4BSD Interprocess Communication Tutorial* de autoria de Stuart Sechrest, Computer Science Research Group, Dep. of Electrical Enngineering and Computer Science, University of California, Berkeley, 1990

3. *An Advanced 4.4BSD Interprocess Communication Tutorial* de autoria de S.J. Leffler et alli, Computer Science Research Group, Dep. of Electrical Enngineering and Computer Science, University of California, Berkeley, 1986

4. *UNIX Network Programming*, W.R.Stevens, Prentice Hall

Parte I

A parte I da apostila, envolvendo os capítulos 1, 2, 3 e 4 apresenta algumas noções de base do sistema operacional `UNIX`. Estas noções são fundamentais para o entendimento da maior parte dos exemplos apresentados nos capítulos seguintes da apostila.

O capítulo 1 trata de generalidades como as chamadas de sistema, os descritores de arquivos e as formas de identificação do usuário.

O capítulo 2 apresenta uma introdução à utilização de processos, assim como as primitivas principais para o gerencimento dos processos `UNIX`.

Capítulo 1

Generalidades

1.1 Chamadas de Sistema x Rotinas de biblioteca

Como primeiro passo deste curso, é fundamental fazer-se a distinção entre uma chamada de sistema (ou primitiva) e uma rotina de biblioteca. Quando uma chamada de sistema é feita, o usuário solicita ao sistema operacional (SO) que realize uma operação em seu nome. Por exemplo, `read()` é uma chamada de sistema que solicita ao SO para encher um buffer com dados gravados sobre um periférico. Uma rotina de biblioteca, por outro lado, não requer normalmente a utilização do sistema para realizar a operação desejada. É o caso da função `sin()` que é calculada através de uma soma dos termos de uma série finita, enquanto que a função `popen()`é um subprograma da biblioteca que permite a abertura de um *pipe* especificando o comando a ser executado, utilizando para isso as primitivas `pipe()`, `fork()`, `open()` e `close()`. As primitivas serão explicadas no manual `UNIX` 2, ao passo que as rotinas de biblioteca estão no manual número 3 (para buscar as informações no manual 2, por exemplo, dev-se digitar o comando shell `man 2 <comando>`

1.2 Gestão de erros

Todas as funções descritas anteriormente possuem um valor de retorno definido. A maior parte do tempo, em caso de erro durante a execução de uma primitiva, o valor de retorno é igual a -1. Neste caso, a variável global errno será atualizada e seu valor indicará um código de erro preciso. Este código de erro pode ser obtido através da função `perror()`. É necessário neste caso que o arquivo `<errno.h>` seja incluído do cabeçalho do programa para que `errno` e `perror()` possam ser utilizadas.

1.3 Funcionamento do comando `man`

O sistema UNIX conta com um manual *on-line* definindo todas as primitivas suportadas, o qual pode ser acessado através do comando shell: `man <número_do_manual> <nome_da_primitiva>` A ajuda solicitada aparecerá então na tela sobre a forma de um texto que pode ser navegado através das teclas `Pgup`, `Pgdn`, etc.

A lista de manuais disponíveis no sistema estão organizados em vários sub-diretórios a partir do diretório `/usr/man/`. Os arquivos são apresentados de duas formas: compactados (`nome.2.z`, por exemplo),

ou não compactado (`nome.2`, por exemplo). Para o primeiro caso, deve-se antes descompactar o arquivo num arquivo temporário através do comando:

```
pcat nome.2.z > /tmp/nome
```

Para o segundo caso, deve-se apenas redirecionar o arquivo para um arquivo temporário usando:

```
cat nome.2 > /tmp/nome
```

Estes arquivos podem ser então impressos utilizando-se os comandos tradicionais de impressão.

1.4 Informações relativas ao usuário

As informações relativas aos usuários do sistema estão localizadas nos arquivos `/etc/passwd`, `/etc/group` e `/var/run/utmp` e `/var/log/wtmp`) - a localização exata deste arquivo depende da versão `libc` do sistema.

1.4.1 `Login`

Cada usuário do sistema dispõe de um nome (uma cadeia de caracteres) de `login` único para permitir sua correta identificação. Este nome é utilizado apenas programas de nível usuário, como o correio eletrônico (o *kernel* do sistema não utiliza este nome). A chamada de sistema `getlogin()` permite a obtenção do `login` do usuário executando um programa que busca no arquivo `utmp` o terminal onde ele está sendo executado, e retornando o login associado a esse terminal.

Valor de retorno: ponteiro sobre uma cadeia de caracteres ou `NULL` em caso de erro.

1.4.2 Direitos de acesso em `UNIX`

Cada arquivo possui, em seu nó de indexação, um identificador (ou) ID do usuário, um identificador (ou ID) do grupo do proprietário. O nó de indexação contém ainda um número binário de 16 bits para o qual os 9 primeiros dígitos (os mais à direita) constituem o direito de acesso interpretados como autorização para a leitura, a escrita e a execução pelo proprietário, grupo, ou outros usuários. Se o bit está em 0, o direito de acesso é negado, se está em 1, o contrário.

Os nove primeiros bits são detalhados a seguir:

- Bit 8 - leitura pelo proprietário
- Bit 7 - escrita pelo proprietário
- Bit 6 - execução pelo proprietário
- Bit 5 - leitura pelos membros do grupo
- Bit 4 - escrita pelos membros do grupo
- Bit 3 - execução pelos membros do grupo
- Bit 2 - leitura por outros usuários do sistema

- Bit 1 - escrita por outros usuários do sistema
- Bit 0 - execução por outros usuários do sistema

Para modificar os direitos de acesso dos arquivos, deve-se utilizar o comando shell `chmod`. Existem uma série de opções para esse comando, as quais podem ser visualizadas via `man`. Algumas opções mais usadas são as seguintes:

- o : para modificar unicamente os 3 bits associados aos outros usuários;
- g : para modificar unicamente os 3 bits associados aos membros do grupo;
- u : para modificar unicamente os 3 bits associados ao usuário (proprietário do login);

Se nada é definido, as mudanças são aplicadas a todos os campos.

Exemplo:

Muitas vezes, durante a utilização de primitvas de I/O (entrada/saída), às vezes é necessário indicar os direitos de acesso associados ao arquivo, utilizando-se para isso um valor inteiro associado obtido da conversão do valor binário em octal. Por exemplo, para ter a autorização de leitura, escrita e execução pelo proprietário e pelo grupo, e a autorização de leitura e execução pelos outros, deve utilizar o seguinte código durante uma chamada da rotina de criação do arquivo :

1111111101 em binário que é equivalente a 775 em octal

Nos 7 bits restantes do número usado pelo nó de indexação, dois são associados a modificação do ID do usuário (bit 11 set-uid), e a modificação da identificador do grupo (bit 10 set-gid). Quando o bit 11 (respectivamente 10) é posicionado em 1, o indicador de permissão de execução para o proprietário (respectivamente, grupo) visualizado pelo comando `ls` está em *s* (veja o próximo exemplo). Seja o exemplo de um programa tendo o bit *s* (*s* para set-uid-bit) colocado em 1 (*setado* para o proprietário. Na execução deste programa, o usuário efetivo é mudado para o proprietário do arquivo contendo o programa. Uma vez que o usuário efetivo que determina os direitos de acesso (ver 1.4.3), isto vai permitir ao usuário de liberar temporariamente o direitos de acesso de qualquer um outro. Este observação se aplica também aos identificadores de grupo. É este mecanismo que permite a todo o usuário de modificar o arquivo `/etc/passwd`, mesmo sendo o *root* é o proprietário, através do comando shell `passwd()`. O bit *s* é colocado em 1 para o arquivo de referência `/bin/passwd` contendo o programa realizando este comando.

A primitiva `system()` permite a execução de um comando shell dentro de um programa e será mais detalhada nos próximos capítulos.

Exemplo:

```
                /* arquivo test_s.c */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{
  FILE * fp;
     system("echo Meu login e: $LOGNAME");
     printf("Vou tentar abrir o arquivo  /home/saibel/teste\n");
     if((fp=fopen("/home/saibel/teste","w+")) == NULL )
          perror("Error fopen()");
     else
          printf ("O arquivo teste esta aberto\n");
     exit(0);
}
```

Resultado da execução:

Seja `saibel` o proprietário do arquivo test_s conectado à máquina `euler`. Um arquivo `teste` é criado através do programa `test_s`), sendo que *par défaut* no sistema usado, somente para o proprietário do arquivo são assegurados os direitos de leitura e escrita deste arquivo. Considere agora que existe um usuário `usertest`, conectado à máquina `lyapunov`, do mesmo grupo de trabalho de `saibel`, que tenta abrir o arquivo `teste`, utilizando o programa `test_s`, o qual não lhe pertence (note que não há obriagatoriedade que ambos os usuários estejam conectados sob diferentes máquinas para que o exemplo seja executado; isso foi feito nesse exemplo devido a forma de visualização do `prompt` utilizada *par défaut* no sistema).

```
euler:~> chmod g+w test_s
euler:~> ls -la test_s
-rwxrwxr-x   1 saibel   prof        12333 Sep 25 10:08 test_s*
euler:~> test_s
Meu login e: saibel
Vou tentar abrir o arquivo  /home/saibel/teste
O arquivo teste esta aberto
euler:~> ls -la /home/saibel/teste
-rw-r--r--   1 saibel   prof            0 Sep 25 10:16 /home/saibel/teste

lyapunov:~> test_s
Meu login e: usertest
Error fopen(): Permission denied
```

Evidentemente a tentativa do usuário `usertest` irá fracassar, uma vez que ele não possui direito de acesso ao arquivo `/home/saibel/teste`.

Agora, `saibel` vai colocar em 1 o bit *s* do programa `test_s`, continuando entretanto, a deixar interditado o acesso ao arquivo `teste` aos outros usuários do grupo.

```
euler:~> chmod u+s test_s
euler:~> ls -al test_s
```

```
-rwsrwxr-x   1 saibel   prof        12337 Sep 25 10:28 test_s*

lyapunov:~> /home/saibel/test_s
Meu login e: usertest
Vou tentar abrir o arquivo  /home/saibel/teste
O arquivo teste esta aberto
```

O usuário `usertest` executando o programa `/home/users/saibel/test_s`, deverá conseguir desta vez abrir o arquivo `teste`. Na verdade, graças ao bit *s*, `usertest` foi assumiu a identidade do proprietário do arquivo `test_s` durante sua execução.

Observação: Em alguns sistemas (como no caso do Laboratório de Engenharia de Computação do LECA), a operação de mudança de identidade do usuário durante a execução de um programa (do qual ele não é proprietário) não é permitida por razões de segurança. Assim, a saída gerada em nosso sistema será:

```
lyapunov:~> /home/saibel/test_s
/home/saibel/test_s: Operation not permitted.
```

Resumindo, pode-se dizer que um processo tem o direito de acesso a um arquivo se:

1. O ID do usuário efetivo do processo é o identificador do super-usuário (*root*);
2. O ID do usuário efetivo do processo é idêntico ao ID do proprietário do arquivo, e o modo de acesso do arquivo permite o direito de acesso desejado no campo **proprietário**;
3. O ID do usuário efetivo do processo não é o ID do proprietário do arquivo, e o ID do grupo efetivo do processo é idêntico ao ID do grupo do arquivo, e o modo de acesso do arquivo permite o direito de acesso desejado no campo **grupo**;
4. O ID do usuário efetivo do processo não é o ID do proprietário do arquivo, e o ID do grupo efetivo do processo não é o ID do grupo do arquivo, e o modo de acesso do arquivo permite o direito de acesso desejado no campo **outros**.

1.4.3 Identificadores dos usuários

Cada processo tem dois valores inteiros associados: o identificador (ou ID) do usuário real e o identificador (ou ID) do usuário efetivo.

O ID do usuário real identifica em qualquer caso o usuário executando o processo.

O ID de usuário efetivo é utilizado para determinar as permissões do processo. Estes dois valores são em geral iguais. Através da mudança do ID de usuário efetivo, um processo poderá ganhar permissões associadas ao novo ID do usuário (e perder temporariamente àquelas associadas ao identificador do usuário real).

Exemplo:

```
                /* arquivo test_acces.c */
#include <errno.h>
#include <stdio.h>
#include <stdlib.h> /* primitiva system */

int main()
{
FILE *fp ;
     system("echo Meu login e: $LOGNAME");
     printf("Vou tentar abrir o arquivo  /home/saibel/teste\n");
     if((fp=fopen("/home/saibel/teste","w")) == NULL )
          perror("Error fopen()");
     else
          printf ("O arquivo teste esta aberto\n");
     printf("Vou tentar abrir o arquivo  /home/adelardo/teste\n");
     if((fp=fopen("/home/adelardo/teste","w")) == NULL )
          perror("Error fopen()");
     else
          printf ("O arquivo teste esta aberto\n");
     exit(0);
}
```

O programa `test_s` anterior é modificado agora pela adição de comandos para a abertura de um arquivo pertencente a `usertest`. O bit *s* é colocado em 1 para `test_acces`.

```
euler:~> ls -l test_acces
-rwsr-xr-x   1 saibel   prof        12521 Sep 25 10:59 test_acces*
euler:~> ls -l /home/saibel/teste
-rw-rw-r--   1 saibel   prof       0 Sep 25 10:59 /home/saibel/teste

lyapunov:~> ls -l /home/usertest/teste
-rw-rw-r--   1 usertest   prof     0 Sep 25 10:30 /home/usertest/teste
```

Agora, `saibel` lança o programa `test_acces`. Pode ser observado que ele tem o acesso ao arquivo teste que lhe pertence, mas não ao arquivo de `usertest`.

```
euler:~> test_acces
Meu login e: saibel
Vou tentar abrir o arquivo  /home/saibel/teste
O arquivo teste esta aberto
Vou tentar abrir o arquivo  /home/usertest/teste
Error fopen(): Permission denied
euler:~>
```

Do seu lado, `usertest` lança `test_acces` e obtém os direitos de acesso sobre o arquivo `teste` pertencendo a `saibel`, mas ele perde os direitos de acesso sobre seu próprio arquivo `teste`:

```
lyapunov:~> test_acces
Meu login e: usertest
Vou tentar abrir o arquivo  /home/saibel/teste
O arquivo teste esta aberto
Vou tentar abrir o arquivo  /home/usertest/teste
Error fopen(): Permission denied
lyapunov:~>
```

Obtenção e modificação dos identificadores do usuário

Os identificadores (ou IDs) do usuário real e efetivo são obtidos com a ajuda das chamadas de sistema `getuid()` e `geteuid()`.

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

uid_t getuid(void)  /* determina o ID do usuário real */
uid_t geteuid(void) /* determina o ID do usuário efetivo */
```

Valor de retorno: identificador real ou efetivo. Não existe erro possível neste caso.
Estes identificadores podem ser modificados pela chamada de sistema `setuid()`.

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

int setuid(uid_t uid)    /* muda o ID do usuário */
uid_t uid                /* novo valor do ID */
```

Valor de retorno: -1 em caso de erro.

Além disso, a utilização das primitivas implicam no salvamento (*backup*) do identificador precedente. A gestão de `setuid()` é definida pelas três regras:

1. Se essa chamada de sistema é empregada pelo super-usuário, ele posicionará o ID do usuário efetivo, assim que o ID do usuário real para o valor definido como argumento. Ele permitirá assim que o super-usuário de se tornar não importa qual usuário.

2. Se o ID do usuário real é igual ao valor passado como argumento, então o ID do usuário efetivo terá esse valor. Isto permite ao processo de encontrar as permissões do usuário que o executa, após a aplicação da terceira regra;

3. Se o usuário salvo é igual ao valor passado como argumento, o ID do usuário efetivo receberá este valor. Isto permite a um processo de se executar temporariamente com as permissões de um outro usuário. O programa pode encontrar as permissões originais através da aplicação da segunda regra.

O exemplo a seguir tenta esclarecer as dúvidas relativas ao assunto:
O código foi escrito e compilado sobre o login `saibel`, com os IDs real e ID efetivo valendo 1198.
O programa é lançado sucessivamente da conta `saibel`, e após da conta `usertest` (ID real e ID efetivo valendo).

```
                /* arquivo test_uid.c */
#include <errno.h>
#include <stdio.h>
#include <stdlib.h> /* primitiva system */
#include <unistd.h>
#include <sys/types.h>

int main()
{
        system("echo Meu login e: $LOGNAME");
        printf("Meu user id real e: %d\n",getuid()) ;
        printf("Meu user id efetivo e: %d\n",geteuid()) ;
        if (setuid(4010) == -1) {
               perror("Error setuid()") ;
               exit(-1) ;
        }
        printf("Meu user id real : %d\n",getuid()) ;
        printf("Meu user id efetivo: %d\n",geteuid()) ;

        if (setuid(1198) == -1) {
               perror("Error setuid()") ;
               exit(-1) ;
        }
        printf("Meu user id real : %d\n",getuid()) ;
        printf("Meu user id efetivo : %d\n",geteuid()) ;
        exit(0);
}
```

Resultado da execução:

```
euler:~/> ls -l test_uid
-rwxr-xr-x   1 saibel   profs       12633 Sep 26 20:47 test_uid*
euler:~/> test_uid
Meu login e: saibel
Meu user id real e: 1198
Meu user id efetivo e: 1198
Error setuid(): Operation not permitted
```

O primeiro `setuid()` é interrompido, dado que nenhuma das regras da aplicação foi respeitada. Se por outro lado o programa é lançado a partir do login usertest, não proprietário do arquivo a execução será:

```
lyapunov:~/> /home/saibel/test_uid
Meu login e: usertest
```

```
Meu user id real e: 1275
Meu user id efetivo e: 1275
Meu user id real : 1275
Meu user id efetivo: 1275
Error setuid():  Operation not permitted
```

O primeiro `setuid()` é executado com sucesso: o ID do usuário real 1275 é igual ao valor passado como argumento. Para poder acessar o segundo setuid(), o proprietário do arquivo, `saibel`, deve posicionar o bit *s* em 1 para que `usertest` seja durante o tempo de execução proprietário do arquivo.

```
euler:~/> chmod u+s test_uid
euler:~/> ls -l test_uid
-rwsrwxr-x   1 saibel   profs       12633 Sep 26 20:47 test_uid*

lyapunov:~/> /home/saibel/test_uid
Meu login e: usertest
Meu user id real e: 1275
Meu user id efetivo e: 1198
Meu user id real : 1275
Meu user id efetivo: 1275
Meu user id real : 1275
Meu user id efetivo: 1198
```

Antes do lançamento do programa, os IDes real e efetivo são iguais (1275). Em seguida, o ID efetivo torna-se 1198 quando o programa está sendo executado, em razão da presença do bit *s*. Na chamada de `setuid(1275)`, o ID real é igual ao argumento (2ª regra), ele então gravou o valor 1198 no ID que foi salvo. Na chamada `setuid(1198)` o ID salvo 1198 é igual ao argumento (3ª regra), o processo encontra seu ID inicial, e portanto suas permissões associadas.

Observação: As mesmas observações da seção 1.4.2 são válidas nesse caso.

Interesse do que foi apresentado:

1. gestão do correio eletrônico;

2. gestão das impressoras;

3. gestão do banco de dados - possibilidade de criar arquivos pertencentes ao usuário e não à base inteira);

4. `Login` - depois de pedir o nome do usuário e sua senha, `login` verifica-os consultando `/etc/passwd`, e se eles estão conformes, `setuid()` e `setgid()` são executados para posicionar os IDs do usuário e do grupo real e efetivo aos valores de entrada correspondente a `/etc/passwd`.

1.4.4 Identificadores de grupo

O princípio é idêntico a aquele dos IDs de usuário, sabendo-se que vários usuários podem pertencer ao mesmo grupo, permitindo a estes de ter acesso aos arquivos do grupo, e recusando acesso aos outros.

As chamadas de sistema para obter os IDs do grupo real e efetivo são `getgid()` e `getegid()`, enquanto que para modificá-los é `setgid()`.

Note que na maior parte dos sistemas, um usuário pertence apenas a um grupo por vez. Para mudá-lo de grupo, deve-se executar o comando `newgrp()`, que muda o ID do grupo e executa um novo shell.

Observação: se o novo grupo ao qual o usuário quer se juntar instalou uma senha do grupo (normalmente o caso), esta será demandada quando o comando for executado.

1.5 Entrada e Saída

1.5.1 Noção de tabela de nós de indexação

Esta tabela está localizada no início de cada região de disco contendo um sistema de arquivos `UNIX`. Cada nó de indexação (ou *inode* desta tabela corresponde a um arquivo e contém as informações necessárias essenciais sobre os arquivos gravados no disco:

1. O tipo do arquivo (detalhado a seguir);
2. O número de links (número de arquivos dando acesso ao mesmo arquivo);
3. O proprietário e seu grupo;
4. O conjunto de direitos de acesso associados ao arquivo para o proprietário do arquivo, o grupo ao qual ele pertence, e enfim todos os outros usuários do sistema;
5. O tamanho em número de bytes;
6. As datas do último acesso, da última modificação, e da última mudança de estado (quando o nó de indexação foi modificado);
7. Os ponteiros para os blocos do disco contendo o arquivo propriamente dito.

A estrutura `stat` correspondente dentro do arquivo `<sys/stat.h>`. Uma saída simples a partir de `stat` seria da seguinte forma:

```
File: "/"
Size: 1024         Allocated Blocks: 2            Filetype: Directory
Mode: (0755/drwxr-xr-x)         Uid: (    0/  root)  Gid: (    0/ system)
Device:  0,0   Inode: 2         Links: 20
Access: Wed Jan  8 12:40:16 1986(00000.00:00:01)
Modify: Wed Dec 18 09:32:09 1985(00021.03:08:08)
Change: Wed Dec 18 09:32:09 1985(00021.03:08:08)
```

Observação: Esta tabela não contém nem o nome do arquivo, nem os dados, apenas informações lógicas associadas aos arquivos.

1.5.2 Tipos de arquivos

Existem três tipos de arquivos em `UNIX`:

1. Os arquivos ordinários ou comuns : tabelas lineares de bytes sem nome, identificados pelo número de indexação;
2. Os diretórios: a utilidade deste arquivos é de facilitar a procura de um arquivo por um nome ao invés de por seu número de indexação na tabela de nós; o diretório é portanto constituído de uma tabela de duas colunas contendo o nome que o usuário deu ao arquivo, e de outro, o número de indexação que permite o acesso a esse arquivo. Este par é chamado de *link*.
3. Os arquivos especiais: trata-se de um tipo de periférico, ou de uma FIFO (*first in, first out*)

1.5.3 Descritores de arquivo e ponteiros para os arquivos

Foi visto que o nó de indexação de um arquivo é a estrutura de identificação do arquivo dentro de um sistema. Quando um processo quiser manipular um arquivo, ele vai simplesmente utilizar um inteiro chamado descritor de arquivo. A associação desse descritor ao nó de indexação deste arquivo se faz durante a chamada da primitiva `open()` (ver 1.5.4), com o descritor tornando-se então o nome local de acesso desse arquivo no processo. Cada processo `UNIX` dispõe de 20 descritores de arquivo, numerados de 0 a 19. Por convenção, os três primeiros são sempre abertos no início da vida de um processo:

- O descritor de arquivo 0 é a entrada padrão (geralmente o teclado);
- O descritor de arquivo 1 é associado a saída padrão (normalmente a tela);
- O descritor de arquivo 2 é a saída de erro padrão (normalmente a tela).

Os outros 17 descritores estão disponíveis para os arquivos. Esta noção de descritor de arquivo é usada para a interface de Entrada/Saída de baixo nível, especialmente com as primitivas `open()`, `write()`, etc. Por outro lado, quando as primitivas da biblioteca padrão de entrada/saída são usadas, os arquivos são encontrados através dos ponteiros para os objetos do tipo `FILE` (tipo definido dentro da `<stdio.h>`).

Existem três ponteiros definidos neste caso:

- `stdin` que aponta para o buffer da saída padrão (geralmente o teclado);
- `stdout` que aponta para o buffer da saída padrão (normalmente a tela);
- `stderr` que aponta para o buffer da saída de erro padrão (normalmente a tela).

1.5.4 Descrição de algumas primitivas e funções

Primitivas open() e creat()

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

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

Valor de retorno: descritor do arquivo ou -1 em caso de erro.

As chamadas `open()` e `creat()` possibilitam a criação de um arquivo no sistema.

A primitiva `creat()` realiza a criação de um arquivo, com o nome definido pelo parâmetro path. O inteiro perm corresponde ao número octal (direito de acesso) conforme descrito em 1.4.2.

Se o arquivo não existia, ele é aberto em modo escrita. Senão, a autorização de escrita não é necessária (apenas a autorização para execução já seria suficiente). Neste caso, usuário e grupo efetivos tornam-se proprietários do arquivo. Nem os proprietários dos arquivos, nem as autorizações são modificados, mas seu tamanho é fixado em 0, e o arquivo é aberto em modo escrita (o argumento perm será então ignorado).

Para criar um arquivo de nome teste_create com as autorizações de leitura e escrita para o proprietário e o grupo, deve-se escrever:

```
if ( (fd=creat("teste\_create", 0666) ) == -1)
       perror("Error creat()");
```

A primitiva `open()` permite a abertura (ou a criação) de um arquivo de nome igual à string apontada pelo ponteiro `pathname`. O parâmetro `mode` determina as permissões para o uso do arquivo que foi criado. Ele só é utilizado quando `open()` é usado na criação de um arquivo.O parâmetro `flags` define a forma de abertura do arquivo, podendo receber uma ou mais constantes simbólicas separadas por operadores — (ou), definidas no arquivo `<fcntl.h>`, como mostra a tabela 1.1.

Valor	mneumônico	Função
00000	O_RDONLY	Abertura de arquivo somente em modo leitura
00001	O_WRONLY	Abertura de arquivo somente em modo escrita
00002	O_RDWR	Abertura de arquivo em modo leitura e escrita
00004	O_NDELAY	Impede o bloqueio de um processo numa chamada de leitura sobre um pipe, um pipe tipo FIFO (*First In First Out*) ou um arquivo especial, ainda não aberto em escrita; retorna um erro numa chamada de escrita sobre um pipe, um pipe tipo FIFO ou um arquivo especial, ainda não aberto em leitura, ainda sem bloquear o processo.
00010	O_APPEND	Abertura em escrita no fim do arquivo
00400	O_CREAT	Criação do arquivo, caso ele ainda não exista (único caso de utilização do argumento `flag`)
01000	O_TRUNC	Truncar o tamanho do arquivo em 0 caso o arquivo exista
02000	O_EXCL	Se O_CREAT está *setado*, provoca um erro se o arquivo já existe

Tabela 1.1: Flags utilizados com a primitiva `open()`

Exemplo:

Para criar e abrir o arquivo `teste_open` em modo escrita com as permissões **leitura e escrita para o proprietário e para o grupo**, deve-se escrever o seguinte código:

```
if ((fd=open(‘‘test_open’’, O_WRONLY| O_CREAT| O_TRUNC, 0666)) == -1)
    perror("Error open()");
```

Função `fdopen()`

```
#include <stdio.h>

FILE *fdopen (int fildes, const char *mode);
```

Valor de retorno: ponteiro sobre o arquivo associado ao descritor `fields`, ou a constante prédefinida (dentro de `<stdio.h>`) e `NULL` em caso de erro.

A função `fdopen()` associa uma *stream* com o descritor de arquivo existente `fildes`. O parâmetro `mode` da *stream* indica a forma de abertura do arquivo.

Esta função faz a ligação entre as manipulações de arquivos da biblioteca padrão `C`, que utiliza ponteiros para os objetos do tipo `FILE` (`fclose()`, `fflush()`, `fprintf()`, `fscanf()`, ...), e as primitivas de baixo nível (`open()`, `write()`, `read()`, ...) que utilizam descritores de arquivo do tipo `int`. O detalhamento da biblioteca padrão em `C` não faz parte do escopo deste manual e maiores detalhes devem ser buscados na bibliografia recomendada.

Observação:
O arquivo deve, anteriormente, ter sido aberto através da primitiva `open()`. Por outro lado, o parâmetro `mode` escolhido deve ser compatível com o modo utilizado durante a abertura do arquivo com o `open()`. Este parâmetro pode ter os seguintes valores:

- *r* : o arquivo é aberto em modo leitura
- *w* : o arquivo é criado em aberto em modo escrita. Se ele já existia, seu tamanho é colocado em 0
- *a* : abertura em escrita no fim do arquivo (com sua criação, se ele não existia anteriormente)

Exemplo:

```
/*  Abertura precedente por open(), por exemplo em leitura  */
if ( (fd=open("meu\_arquivo", O\_RDONLY, 0666) ) == -1)
        perror("Error open()");

/*  Associação de fp (do tipo FILE*) a fd (de tipo int)     */
if ( (fd=open(fd, "r") ) == -1)
        perror("Error fdopen()");
```

Primitiva close()

```
#include <unistd.h>

int close(int fd)
```

Valor de retorno: 0 para sucesso ou -1 em caso de erro.

Fecha o decritor de arquivo `fd`. Se `fd` é a última cópia de um descritor de arquivo particular, todos os recursos associados a ele serão liberados. Esta primitiva não vai esvaziar o buffer associado ao processo, ela simplesmente vai liberar o descritor do arquivo para um reutilização eventual.

Primitivas dup() - dup2()

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Valor de retorno: novo descritor de arquivo ou -1 em caso de erro.

Esta primitiva cria uma cópia de um descritor de arquivo existente (`oldfd`) e fornece um novo descritor (`newfd`) tendo exatamente as mesmas características que aquele passado como argumento na chamada. Ela garante que o valor de retorno seja o menor entre todos os valores de descritores possíveis.

`dup` vai usar o menor número de descritor disponível para criar o novo descritor, enquanto `dup2` determina que `newfd` será a cópia de `oldfd`, fechando antes `newfd` se ele já estiver aberto.

Exemplo:

```
                /* arquivo test_dup.c */

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

int main()
{
 int fd ;               /* descritor a ser duplicado  */
 int retour1=10 ;       /* valor de retorno de dup  */
 int retour2=10 ;       /* valor de retorno de dup2 */

     if ((fd=open("./fic",O_RDWR|O_CREAT|O_TRUNC,0666))==-1) {
          perror("Error open()") ;
          exit(1) ;
     }
```

```
     close(0) ;         /* fechamento da saida entrada stdin */

     if ((retour1 = dup(fd)) == -1) {  /* duplicacao */
          perror("Error dup()") ;
          exit(1) ;
     }

     if ((retour2 = dup2(fd,1)) == -1) {  /* duplicacao de stdout */
          perror("Error dup2()") ;
          exit(1) ;
     }

     printf ("valor de retorno de dup()  : %d \n",retour1) ;
     printf ("valor de retorno de dup2() : %d \n",retour2) ;
     exit(0);
}
```

Resultado da execução:

```
euler> test_dup
euler>
euler> cat fic
valor de retorno de dup()  : 0
valor de retorno de dup2() : 1
```

A chamada à primitiva `dup()` redirige a entrada padrão para o arquivo `fic`, de descritor `fd`, e a chamada à `dup2()` redirige a saída padrão para este mesmo arquivo. O resultado da execução não pode ser desta forma visualizado na tela; deve-se então editar o arquivo `fic`. Note que a chamada de `dup2()` não obriga o fechamento do descritor a ser duplicado.

Primitiva write()

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t nbytes)
```

Valor de retorno: número de bytes escritos ou -1 em caso de erro, e `errno` é setado apropriadamente. Se `nbytes` valer 0 e `fd` referenciar um arquivo regular, o valor 0 será retornado, sem causar qualquer outro efeito.

Esta primitiva escreve num arquivo aberto respresentado pelo descritor de arquivo `fd`, os `nbytes` apontados por `buf`. Note que a escrita não se faz diretamente no arquivo, passando antes por um buffer do *kernel* (método *kernel buffering*).

Primitiva read()

```
#include <unistd.h>

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

Valor de retorno: número de bytes lidos, 0 ou `EOF` para indicar o fim de linha, ou -1 em caso de erro. Não é um erro retornar um número de bytes menor do que foi especificado (isto pode acontecer se o `read()` for interrompido por um sinal ou quando poucos bytes estão disponíveis momentaneamente).

Esta primitiva lê os `nbytes` bytes no arquivo aberto representado por `fd`, e os coloca dentro do buffer apontado por `buf`.

Observação:
As operações de abertura de arquivos (semelhantes a `open()`), e de duplicação de descritores (semelhantes a `dup()`) estão reunidas dentro da primitiva `fcntl()`, que não será detalhada aqui (veja o arquivo localizado em`/usr/include/fcntl.h` para maiores informações).

Exemplo 1: Redirecionamento da saída padrão

Este programa executa o comando shell `ps`, depois redireciona o resultado para o arquivo `fic_saída`. Assim, a execução deste programa não deve imprimir nada na tela. A primitiva `system()` executa o comando passado como argumento.

```
                /* arquivo test_dup2.c */

#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

#define STDOUT 1

int main()
{
 int fd ;
     /* associa fic_saida ao descritor fd */
     if ((fd = open("fic_saida",O_CREAT|O_WRONLY|
                    O_TRUNC,0666)) == -1){
          perror("Error na abertura de fic_saida") ;
          exit(1) ;
     }
     dup2(fd,STDOUT) ;  /* duplica a saida padrao */
     system("ps") ;     /* executa o comando */
     exit(0);
}
```

Resultado da execução:

```
euler:~/> test_dup2
euler:~/> more fic_saida
  PID TTY STAT  TIME COMMAND
 9819  ?  S    0:01 -tcsh
11815  ?  S    0:00 test_dup2
11816  ?  R    0:00 ps
```

Note que outros redirecionamentos seguem o mesmo princípio, e que também é possível a realização de redirecionamentos de entrada e de saída.

Exemplo 2: Cópia de um arquivo

O programa `test_copy` a seguir copia um arquivo para outro. Ele é semelhante ao comando shell `cp`.

```
                /* arquivo test_copy.c */

#include <errno.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

#define TAILLEBUF 512

void copia_fic(src,dest)   /* copia um arquivo */
char *src, *dest ;
{
 int srcfd,destfd ;
 int nlect, necrit, n ;
 char buf[TAILLEBUF] ;

      if ((srcfd = open(src,O_RDONLY)) == -1){
           perror("Error: abertura do arquivo fonte") ;
           exit(1) ;
      }

      if ((destfd = creat(dest,0666)) == -1){
           perror("Error: Criacao do arquivo desino");
           exit(1) ;
      }
      write(1,"12332423",8);
      while ((nlect = read(srcfd,buf,sizeof(buf))) != 0){
```

```
            if (nlect == -1){
                 perror("Error read()") ;
                 exit(1) ;
            }
            necrit = 0 ;
            do {
                 if ((n = write(destfd,&buf[necrit],nlect-necrit)) == -1)
                 {
                      perror("Error write()") ;
                      exit(1) ;
                 }
                 necrit += n ;
            } while (necrit < nlect) ;

       }
       if (close(srcfd) == -1 || close(destfd) == -1){
            perror("Error close()") ;
            exit(1) ;
       }
}

int main(argc,argv)
int argc ;
char *argv[] ;
{
     if (argc != 3) {
          printf("Uso: copia_fic arquivo1 arquivo2\n") ;
          exit(1) ;
     }
     printf("Estou copiando ...\n") ;
     copia_fic(argv[1],argv[2]) ;
     printf("\nAcabei!\n") ;
     exit(0);
}
```

Resultado da execução:

```
euler:~/> test_copy fonte destino
Estou copiando ...
12332423
Acabei!
euler:~/> cmp fonte destino
euler:~/>
```

1.6 Chaves de acesso: Função `ftok`

Uma chave nada mais é do que um valor inteiro longo. Ela é utilizada para identificar uma estrutura de dados que vai ser referenciada por um programa. Existe uma função que permite a criação de chaves de maneira única no sistema, denominada `ftok()`.

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

key_t ftok(char *path, char proj)
```

Valor de retorno: valor de uma chave única para todo o sistema ou -1 em caso de erro.

A função `ftok()` usa o nome do arquivo apontado por `path`, que é único no sistema, como uma cadeia de caracteres, e o combina com um identificador `proj` para gerar uma chave do tipo key_t no sistema IPC.

Também é possível criar funções gerando chaves utilizando-se parâmetros associados ao usuário, como seu número de identificação (`uid`) e seu número de grupo (`gid`). Por exemplo, com a função:

```
#define KEY(n)  ((getuid() % 100) * 100 + getgid() + n )
```

Capítulo 2

Processos em UNIX

2.1 Introdução

A norma POSIX fornece dois mecanismos para a criação de atividades concorrentes. O primeiro, abordado neste capítulo, corresponde ao tradicional mecanismo `fork` do UNIX e a sua chamada de sistema associada, o `wait`. A chamada `fork` dentro de um processo provoca a criação de um clone perfeito deste processo para a execução.

POSIX permite ainda que cada um dos processos criados contenham diversas *threads* (tarefas) de execução. Essas *threads* têm acesso às mesmas posições de memória e rodam num espaço de endereçamento único. Uma introdução ao uso dessas primitivas será apresentada na seqüência deste texto.

2.1.1 Processo: Uma definição

Um processo é um ambiente de execução que consiste em um segmento de instruções, e dois segmentos de dados (*data* e *stack*). Deve-se, entretanto, notar a diferença entre um processo e um programa: um programa nada mais é que um arquivo contendo instruções e dados utilizados para inicializar segmentos de instruções e de dados do usuário de um processo.

2.1.2 Identificadores de um processo

Cada processo possui um identificador (ou ID) único denominado `pid`. Como no caso dos usuários, ele pode estar associado a um grupo, e neste caso será utilizado o identificador denominado `pgpr`. As diferentes primitivas permitindo o acesso aos diferentes identificadores de um processo são as seguintes:

```
#include <unistd.h>

pid_t getpid()                       /* retorna o ID do processo  */
pid_t getppid()                      /* retorna o ID do pai do processo  */
int setpgid(pid_t pid, pid_t pgid);  /* seta o valor do ID do grupo do */
                                     /* especificado por pid para pgid */
pid_t getpgid(pid_t pid);            /* retorna o ID do grupo de processos */
                                     /* especificado por pid */
```

```
int setpgrp(void);                  /* equivalente a setpgid(0,0) */
pid_t getpgrp(void);                /* equivalente a getpgid(0) */
```

Valor de retorno: 0 se `setpgid` e `setpgrp` são executados com sucesso e, -1 em caso de erro.

Exemplo:

```
                /* arquivo test_idf.c */
#include <stdio.h>
#include <unistd.h>

int main()
{
     printf("Eu sou o processo %d de pai %d e de grupo %d\n",getpid()
             ,getppid(),getpgrp()) ;
     exit(0);
}
```

Resultado da execução:

```
euler:~> test_idf
Eu sou o processo 28448 de pai 28300 e de grupo 28448
```

Observe que o pai do processo executando `test_idf` é o processo `tcsh`. Para confirmar a afirmação, faça um `ps` na janela de trabalho:

```
euler:~> ps
  PID TTY STAT  TIME COMMAND
28300  ?  S    0:00 -tcsh
28451  ?  R    0:00 ps
```

Observação:
Grupos de processo são usados para distribuição de sinais, e pelos terminais para controlar as suas requisições. As chamadas `setpgid` e `setpgrp` são usadas por programas como o `csh()` para criar grupo de processos na implementação de uma tarefa de controle e não serão utilizadas no decorrer do curso.

2.2 As primitivas envolvendo processos

2.2.1 A primitiva `fork()`

```
#include <unistd.h>

pid_t fork(void)       /* criação de um processo filho */
pid_t vfork(void);     /* funciona como um alias de fork */
```

Valor de retorno: 0 para o processo filho, e o identificador do processo filho para o processo pai; -1 em caso de erro (o sistema suporta a criação de um número limitado de processos).

Esta primitiva é a única chamada de sistema que possibilita a criação de um processo em UNIX. Os processos pai e filho partilham o mesmo código. O segmento de dados do usuário do novo processo (filho) é uma cópia exata do segmento correspondente ao processo antigo (pai). Por outro lado, a cópia do segmento de dados do filho do sistema pode diferir do segmento do pai em alguns atributos específicos (como por exemplo, o pid, o tempo de execução, etc.). Os filhos herdam uma duplicata de todos os descritores dos arquivos abertos do pai (se o filho fecha um deles, a cópia do pai não será modificada). Mais ainda, os ponteiros para os arquivos associados são divididos (se o filho movimenta o ponteiro dentro de um arquivo, a próxima manipulação do pai será feita a partir desta nova posição do ponteiro). Esta noção é muito importante para a implementação dos *pipes* (tubos) entre processos.

Exemplo:

```
                /* arquivo test_fork1.c */

/* Descritores herdados pelos processos filhos */

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

int main()
{
int pid ;
int fd ; /* descritor de arquivo associado ao arquivo agenda */
char *telephone ;
int r ; /* retorno de read */
int i ;
char c ;
     printf("Oi, eu sou o processo %d\n",getpid()) ;
     printf("Por favor, envie-me o seu numero de telefone\n") ;
     printf("E o 123456789 ? Ok, ja anotei na minha agenda\n") ;
     if((fd=open("agenda",O_CREAT|O_RDWR|O_TRUNC,S_IRWXU))==-1)
     {
          perror("impossivel de abrir a agenda") ;
          exit(-1) ;
     }
     telephone="123456789" ;
     if(write(fd,telephone,9)==-1)
     {
```

```
     perror("impossivel de escrever na agenda") ;
     exit(-1) ;
}
printf("Enfim, acabei de anotar e estou fechando a agenda\n") ;
close(fd) ;
printf("O que ? Eu me enganei ? O que foi que eu anotei ?\n") ;
printf("\tO pai reabre sua agenda\n") ;
             if((fd=open("agenda",O_RDONLY,S_IRWXU))==-1)      {
     perror("impossivel de abrir a agenda") ;
     exit(-1) ;
}
printf("\tNesse instante, o pai gera um filho\n") ;
pid=fork() ;
if(pid==-1) /* erro */
{
     perror("impossivel de criar um filho") ;
     exit(-1) ;
}
else if(pid==0) /* filho */
{
     sleep(1) ; /* o filho dorme para agrupar as mensagens */
     printf("\t\tOi, sou eu %d\n",getpid()) ;
     printf("\t\tVoces sabem, eu tambem sei ler\n") ;
     printf("\tO filho entao comeca a ler a agenda\n") ;
     for(i=1;i<=5;i++)
     {
          if(read(fd,&c,1)==-1)
          {
               perror("impossivel de ler a agenda") ;
               exit(-1) ;
          }
          printf("\t\tEu li um %c\n",c) ;
     }
     printf("\tMinha agenda ! Diz o pai\n") ;
     printf("\te supreso o filho fecha a agenda...\n") ;
     close(fd) ;
     sleep(3) ;
     printf("\tO filho entao se suicida de desgosto!\n") ;
     exit(1) ;
}
else /* pai */
{
     printf("De fato, eu apresento a voces meu filho %d\n",pid) ;
     sleep(2) ;
```

```
          printf("Oh Deus ! Eu nao tenho mais nada a fazer\n");
          printf("Ah-ha, mas eu ainda posso ler minha agenda\n") ;
          while((r=read(fd,&c,1))!=0)
          {

               if(r==-1)
               {
                    perror("impossivel de ler a agenda") ;
                    exit(-1) ;
               }
               printf("%c",c) ;
          }
          printf("\n") ;
          printf("ENFIM ! Mas onde estao todos ?\n") ;
          sleep(3) ;
          close(fd) ;
     }
     exit(0);
}
```

Resultado da Execução:

```
euler:~> test_fork1
Oi, eu sou o processo 28339
Por favor, envie-me o seu numero de telefone
E o 123456789 ? Ok, ja anotei na minha agenda
Enfim, acabei de anotar e estou fechando a agenda
O que ? Eu me enganei ? O que foi que eu anotei ?
        O pai reabre sua agenda
        Nesse instante, o pai gera um filho
                Oi, sou eu 28340
                Voces sabem, eu tambem sei ler
        O filho entao comeca a ler a agenda
                Eu li um 1
                Eu li um 2
                Eu li um 3
                Eu li um 4
                Eu li um 5
        Minha agenda ! Diz o pai
        e supreso o filho fecha a agenda...
        O filho entao se suicida de desgosto!
Oi, eu sou o processo 28339
Por favor, envie-me o seu numero de telefone
E o 123456789 ? Ok, ja anotei na minha agenda
Enfim, acabei de anotar e estou fechando a agenda
```

```
O que ? Eu me enganei ? O que foi que eu anotei ?
        O pai reabre sua agenda
        Nesse instante, o pai gera um filho
De fato, eu apresento a voces meu filho 28340
Oh Deus ! Eu nao tenho mais nada a fazer
Ah-ha, mas eu ainda posso ler minha agenda
6789
ENFIM ! Mas onde estao todos ?
```

Observação:
Três pontos principais devem ser observados no exemplo anterior:

1. O filho herda os descritores "abertos" do pai - uma vez que o filho pode ler a agenda sem que seja necessário abri-la.

2. O filho pode fechar um descritor aberto pelo pai, sendo que esse descritor continuará aberto para o pai.

3. Os dois processos compartilham do mesmo ponteiro sobre o arquivo duplicado na chamada da primitiva fork! Note que quando o pai vai ler o arquivo, ele vai se movimentar dentro desse arquivo da mesma forma que seu filho.

Comportamento da saída no console:
Note que se o pai e o filho vivem, uma interrupção de teclado (via `CTRL-c`) irá destruir os dois processos. Entretanto, se um filho vive enquanto seu pai está morto, uma interrupção pode não matá-lo. Veja o exemplo de programa a seguir.

Exemplo:

```
            /* arquivo test_fork2.c */

/* Testa as reacoes do console quando um pai
 * morre e o filho continua vivo */

#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
int pid ;
   printf("Eu sou o pai %d e eu vou criar um filho \n",getpid()) ;
   pid=fork() ; /* criacao do filho */
   if(pid==-1) /* erro */
```

```
   {
      perror("impossivel de criar um filho\n") ;
   }
   else if(pid==0) /* acoes do filho */
   {
      printf("\tOi, eu sou o processo %d, o filho\n",getpid()) ;
      printf("\tO dia esta otimo hoje, nao acha?\n") ;
      printf("\tBom, desse jeito vou acabar me instalando para sempre\n");
      printf("\tOu melhor, assim espero!\n") ;
      for(;;) ; /* o filho se bloqueia num loop infinito */
   }
   else /* acoes do pai */
   {
       sleep(1) ; /* para separar bem as saidas do pai e do filho */
       printf("As luzes comecaram a se apagar para mim, %d\n",getpid()) ;
       printf("Minha hora chegou : adeus, %d, meu filho\n",pid) ;
       /* e o pai morre de causas naturais */
    }
    exit(0);
}
```

Resultado da Execução:

```
euler:~> test_fork2
Eu sou o pai 28637 e eu vou criar um filho
        Oi, eu sou o processo 28638, o filho
        O dia esta otimo hoje, nao acha?
        Bom, desse jeito vou acabar me instalando para sempre
        Ou melhor, assim espero!
As luzes comecaram a se apagar para mim, 28637
Minha hora chegou : adeus, 28638, meu filho
```

Se o comando shell ps é executado no console, a seguinte saída é obtida:

```
euler:~> ps
  PID TTY STAT  TIME COMMAND
28300  ?  S    0:00 -tcsh
28638  ?  R    0:04 test_fork2
28639  ?  R    0:00 ps
```

Note que o filho permanece rodando! Tente interrompê-lo via teclado usando `CTRL-c` ou `CTRL-d`. Ele não quer morrer, não é verdade? Tente matá-lo diretamente com um sinal direto através do comando shell `kill`.

```
kill -9 <pid>
```

No caso do exemplo anterior, deveria ser feito:

```
kill -9 28638
```

Desta vez estamos certos que *Jason* morreu! ?

```
euler:~> kill -9 28638
euler:~> ps
  PID TTY STAT  TIME COMMAND
28300  ?  S    0:00 -tcsh
28666  ?  R    0:00 ps
```

Problema com os buffers de saída:

```
          /* arquivo test_fork3.c */

/* O filho herda uma copia do buffer de saida do pai */

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

int main()
{
int pid ;
     printf(" 1") ;
     pid=fork() ;
     if(pid==-1) /* error */
     {
          perror("impossivel de criar um filho") ;
          exit(-1) ;
     }
     else if(pid==0) /* filho */
     {
          printf(" 2") ;
          exit(0) ;
     }
     else /* pai */
     {
          wait(0) ; /* o pai aguarda a morte de seu filho */
          printf(" 3") ;
          exit(0);
     }
}
```

Resultado da execução:
Contrariamente ao que poderia ser intuitivamente imaginado, o resultado da execução não será

```
 1 2 3
```

mas

```
 1 2 1 3
```

Parece estranho... O que teria acontecido? A resposta é que o filho, no seu nascimento, herda o "1"que já estava colocado no buffer de saída do pai (note que nem o caracter de retorno de linha, nem um comando para esvaziar a saída padrão foram enviados antes da criação do filho). Mais tarde, na sua morte, o buffer de saída do filho é esvaziado, e a seguinte saída de impressão será obtida: 1 2. Finalmente, o pai terminará sua execução e imprimirá por sua vez: 1 3.

Uma possível solução para o problema é mostrada no programa a seguir, através da utilização da primitiva `fflush`, que será detalhada no fim do capítulo.

```
          /* arquivo test_fork4.c */

/* Solucao para o filho que herda uma copia do buffer nao vazio
 * de saida do pai */

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

int main()
{
  int pid ;
     printf(" 1") ;
     fflush(stdout); /* o buffer vai ser esvaziado pelo flush */
     pid=fork() ;
     if(pid==-1) /* error */
     {
          perror("impossivel de criar um filho") ;
          exit(-1) ;
     }
     else if(pid==0) /* filho */
     {
          printf(" 2") ;
          exit(0) ;
     }
```

```
    else /* pai */
    {
         wait(0) ; /* o pai aguarda a morte de seu filho */
         printf(" 3") ;
         exit(0);
    }
}
```

A primitiva `wait()`

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

pid_t wait(int *status)    /* espera a morte de um filho */
pid_t waitpid(pid_t pid, int *status, int options)

int *status                /* status descrevendo a morte do filho */
```

Valor de retorno: identificador do processo morto ou -1 em caso de erro.

A função `wait` suspende a execução do processo até a morte de seu filho. Se o filho já estiver morto no instante da chamada da primitiva (caso de um processo zumbi, abordado mais a frente), a função retorna imediatamente.

A função `waitpid` suspende a execução do processo até que o filho especificado pelo argumento `pid` tenha morrido. Se ele já estiver morto no momento da chamada, o comportamento é idêntico ao descrito anteriormente.

O valor do argumento `pid` pode ser:

- < -1 : significando que o pai espera a morte de qualquer filho cujo o ID do grupo é igual so valor de `pid`;
- -1 : significando que o pai espera a morte de qualquer filho;
- 0 : significando que o pai espera a morte de qualquer processo filho cujo ID do grupo é igual ao do processo chamado;
- > 0 : significando que o pai espera a morte de um processo filho com um valor de ID exatamente igual a `pid`.

Se `status` é não nulo (`NULL`), `wait` e `waitpid` armazena a informação relativa a razão da morte do processo filho, sendo apontada pelo ponteiro `status`. Este valor pode ser avaliado com diversas macros que são listadas com o comando shell `man 2 wait`.

O código de retorno via `status` indica a morte do processo que pode ser devido uma:

- uma chamada `exit()`, e neste caso, o byte à direita de status vale 0, e o byte à esquerda é o parâmetro passado a `exit` pelo filho;

- uma recepção de um sinal fatal, e neste caso, o byte à direita de status é não nulo. Os sete primeiros bits deste byte contém o número do sinal que matou o filho.

Exemplo:

```
                /* arquivo test_wait1.c */

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

int main()
{
   int pid ;
   printf("\nBom dia, eu me apresento. Sou o processo %d.\n",getpid()) ;
   printf("Estou sentindo uma coisa crescendo dentro de minha barriga...");
   printf("Sera um filho?!?!\n") ;

   if (fork() == 0) {
      printf("\tOi, eu sou %d, o filho de %d.\n",getpid(),getppid()) ;
      sleep(3) ;
      printf("\tEu sao tao jovem, e ja me sinto tao fraco!\n") ;
      printf("\tAh nao... Chegou minha hora!\n") ;
      exit(7) ;
   }
   else {
      int ret1, status1 ;
      printf("Vamos esperar que este mal-estar desapareca.\n") ;
      ret1 = wait(&status1) ;
      if ((status1&255) == 0) {
         printf("Valor de retorno do wait(): %d\n",ret1) ;
         printf("Parametro de exit(): %d\n",(status1>>8)) ;
         printf("Meu filho morreu por causa de um simples exit.\n") ;
      }
      else
         printf("Meu filho nao foi morto por um exit.\n") ;
         printf("\nSou eu ainda, o processo %d.", getpid());
         printf("\nOh nao, recomecou! Minha barriga esta crescendo
                 de novo!\n");
      if ((pid=fork()) == 0) {
         printf("\tAlo, eu sou o processo %d, o segundo filho de %d\n",
```

```
            getpid(),getppid()) ;
         sleep(3) ;
         printf("\tEu nao quero seguir o exemplo de meu irmao!\n") ;
         printf("\tNao vou morrer jovem e vou ficar num loop infinito!\n") ;
         for(;;) ;
      }
      else {
         int ret2, status2, s ;
         printf("Este aqui tambem vai ter que morrer.\n") ;
         ret2 = wait(&status2) ;
         if ((status2&255) == 0) {
            printf("O filho nao foi morto por um sinal\n") ;
         }
         else {
            printf("Valor de retorno do wait(): %d\n",ret2) ;
            s = status2&255 ;
            printf("O sinal assassino que matou meu filho foi: %d\n",s) ;
         }
      }
   }
   exit(0);
}
```

Resultado da execução:

```
euler:~/> test_wait &
[1] 29079

euler:~/> Bom dia, eu me apresento. Sou o processo 29079.
Estou sentindo uma coisa crescendo dentro de minha barriga...Sera um filho?!?!
Vamos esperar que este mal-estar desapareca.
        Oi, eu sou 29080, o filho de 29079.
        Eu sao tao jovem, e ja me sinto tao fraco!
        Ah nao... Chegou minha hora!
Valor de retorno do wait(): 29080
Parametro de exit(): 7
Meu filho morreu por causa de um simples exit.

Sou eu ainda, o processo 29079.
Oh nao, recomecou! Minha barriga esta crescendo de novo!
Este aqui tambem vai ter que morrer.
        Alo, eu sou o processo 29081, o segundo filho de 29079
        Eu nao quero seguir o exemplo de meu irmao!
        Nao vou morrer jovem e vou ficar num loop infinito!
```

```
euler:~/> ps
  PID TTY STAT  TIME COMMAND
28300  ?  S    0:01 -tcsh
29079  ?  S    0:00 test_wait
29081  ?  R    5:06 test_wait
29103  ?  R    0:00 ps
euler:~/> kill -8 29081
euler:~/> Valor de retorno do wait(): 29081
O sinal assassino que matou meu filho foi: 8.

[1]    Done                          test_wait
euler:~/>
```

O programa é lançado em *background* e, após o segundo filho estiver bloqueado num laço infinito, um sina será lançado para interromper sua execução através do comando shell

```
kill <numero-do-sinal> <pid-filho2>
```

Observações:

Após a criação dos filhos, o processo pai ficará bloqueado na espera de que estes morram. O primeiro filho morre pela chamada de um `exit()`, sendo que o parâmetro de `wait()` irá conter, no seu byte esquerdo, o parâmetro passado ao `exit()`; neste caso, este parâmetro tem valor 7.

O segundo filho morre com a recepção de um sinal, o parâmetro da primitiva `wait()` irá conter, nos seus 7 primeiros bits, o número do sinal (no exemplo anterior ele vale 8).

Observações relativas aos processos zumbis Um processo pode se terminar quando seu pai não está a sua espera. Neste caso, o processo filho vai se tornar um processo denominado zumbi (*zombie*). Ele é neste caso identificado pelo nome `<defunct>` ou `<zombie>` ao lado do nome do processo. Seus segmentos de intruções e dados do usuário e do sistema são automaticamente suprimidos com sua morte, mas ele vai continuar ocupando a tabela de processo do kernel. Quando seu fim é esperado, ele simplesmente desaparace ao fim de sua execução.

Exemplo:

```
                /* arquivo test_defunct.c */

#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
```

```
int pid ;
     printf("Eu sou %d e eu vou criar um filho\n",getpid()) ;
     printf("Vou em bloquear em seguida num loop infinito\n") ;
     pid = fork() ;
     if(pid == -1) /* erro */
     {
          perror("E impossivel criar um filho") ;
          exit(-1) ;
     }
     else if(pid == 0) /* filho */
     {
          printf("Eu sou %d o filho e estou vivo\n",getpid()) ;
          sleep(10) ;
          printf("Vou me suicidar para manter minha consciencia
                  tranquila\n") ;
          exit(0) ;
     }
     else /* pai */
     {
          for(;;) ; /* pai bloqueado em loop infinito */
     }
}
```

Resultado da execução:
Lançando a execução em *background*, tem-se o seguinte resultado:

```
euler:~/> test_defunct &
Vou em bloquear em seguida num loop infinito
Eu sou 29733 o filho e estou vivo
euler:~/>
```

Fazendo-se um `ps`, obtém-se:

```
 euler:~/> ps
PID TTY STAT  TIME COMMAND
28300  ?  S    0:02 -tcsh
29732  ?  R    0:01 test_defunct
29733  ?  S    0:00 test_defunct
29753  ?  R    0:00 ps
```

Após os 10 segundos, o processo filho anuncia sua morte na tela:

```
Vou me suicidar para manter minha consciencia tranquila
```

Refazendo-se um `ps`, obtém-se:

```
euler:~/> ps
  PID TTY STAT  TIME COMMAND
28300  ?  S    0:02 -tcsh
29732  ?  R    1:35 test_defunct
29733  ?  Z    0:00 (test_defunct <zombie>)
29735  ?  R    0:00 ps
```

Tente matar o processo filho usando a primitiva `kill`:

```
euler:~/> kill -INT 29733
euler:~/> ps
  PID TTY STAT  TIME COMMAND
28300  ?  S    0:02 -tcsh
29732  ?  R    2:17 test_defunct
29733  ?  Z    0:00 (test_defunct <zombie>)
29736  ?  R    0:00 ps
```

Não funciona, não é mesmo! Óbvio, ele é um zumbi!!! Note o resultado do `ps` para se certificar. Tente agora matar o pai.

```
euler:~/> kill -INT 29732
[1]    Interrupt                     test_defunct
euler:~/>ps
  PID TTY STAT  TIME COMMAND
28300  ?  S    0:02 -tcsh
29750  ?  R    0:00 ps
```

Finalmente o processo pai, junto com seu filho zumbi foram finalizados.

2.2.2 Primitiva exit()

```
#include <unistd.h>

void _exit(int status);  /* terminacao do processo */
int status  /* valor retornado ao processo pai como status
             * saida do processo filho */
```

Valor de retorno: única primitiva que não retorna.

Todos os descritores de arquivos abertos são automaticamente fechados. Quando um processo faz `exit`, todos os seus processos filho são herdados pelo processo *init* de ID igual a 1, e um sinal SIGCHLD é automaticamente enviado ao seu processo pai.

Por convenção, um código de retorno igual a 0 significa que o processo terminou normalmente (veja o valor de retorno dos procedimentos `main()` dos programas de exemplo. Um código de retorno não nulo (em geral -1 ou 1) indicará entretanto a ocorrência de um erro de execução.

2.2.3 As Primitivas `exec()`

```
#include <unistd.h>

extern char **environ;

int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int  execle( const char *path, const char *arg , ...,
             char* const envp[]);

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

As primitivas `exec()` constituem na verdade uma família de funções (`execl, execlp, execle, execv, execvp`) que permitem o lançamento da execução de um programa externo ao processo. Não existe a criação efetiva de um novo processo, mas simplesmente uma substituição do programa de execução.

Existem seis primitivas na família, as quais podem ser divididas em dois grupos: os `execl()`, para o qual o número de argumentos do programa lançado é conhecido; e os `execv()`, para o qual esse número é desconhecido. Em outras palavras, estes grupos de primitivas se diferenciam pelo número de parâmetros passados.

O parâmetro inicial destas funções é o caminho do arquivo a ser executado.

Os parâmetros `char arg, ...` para as funções `execl`, `execlp` e `execle` podem ser vistos como uma lista de argumentos do tipo `arg0, arg1, ..., argn` passadas para um programa em linha de comando. Esses parâmetros descrevem uma lista de um ou mais ponteiros para strings não-nulas que representam a lista de argumentos para o programa executado.

As funções `execv` e `execvp` fornecem um vetor de ponteiros para strings não-nulas que representam a lista de argumentos para o programa executado.

Para ambos os casos, assume-se, por convenção, que o primeiro argumento vai apontar para o arquivo associado ao nome do programa sendo executado. A lista de argumento deve ser terminada pelo ponteiro `NULL`.

A função `execle` também especifica o ambiente do processo executado após o ponteiro `NULL` da lista de parâmetros ou o ponteiro para o vetor `argv` com um parâmetro adicional. Este parâmetro adicional é um vetor de ponteiros para strings não-nulas que deve também ser finalizado por um ponteiro `NULL`. As outras funções consideram o ambiente para o novo processo como sendo igual ao do processo atualmente em execução.

Valor de retorno: Se alguma das funções retorna, um erro terá ocorrido. O valor de retorno é -1 neste caso, e a variel global `errno` será setada para indicar o erro.

Na chamada de uma função `exec()`, existe um recobrimento do segmento de instruções do processo que chama a função. Desta forma, não existe retorno de um `exec()` cuja execução seja correta (o endereço de retorno desaparece). Em outras palavras, o processo que chama a função `exec()` morre.

O código do processo que chama uma função `exec()` será sempre destruído, e desta forma, não existe muito sentido em utilizá-la sem que ela esteja associada a uma primitiva `fork()`.

Exemplo:

```
                /* arquivo test_exec.c */

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

int main()
{
      execl("/bin/ls","ls","test_exec.c",NULL) ;
      printf ("Eu ainda nao estou morto\n") ;
      exit(0);
}
```

Resultado da execução:

```
euler:~/> test_exec
test_exec.c
```

O comando `ls` é executado, mas o `printf` não. Isto mostra que o processo não retorna após a execução do `execl`.

O exemplo seguinte mostra a utilidade do `fork` neste caso.

Exemplo:

```
                /* arquivo test_exec_fork.c */

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

int main()
{
     if ( fork()==0 ) execl( "/bin/ls","ls","test_exec.c",NULL) ;
     else {
          sleep(2) ;  /* espera o fim de ls para executar o printf() */
          printf ("Eu sou o pai e finalmente posso continuar\n") ;
     }
     exit(0);
}
```

Resultado da execução:

```
euler:~/> test_exec_fork
test_exec.c
Eu sou o pai e finalmente posso continuar
```

Neste caso, o filho morre após a execução do `ls`, e o pai continuará a viver, executando então o `printf`.

Comportamento em relação aos descritores abertos

A princípio, os descritores de arquivos abertos antes da chamada `exec()` continuama abertos, exceto se for determinado o contrário (via primitiva `fcntl()`. Um dos efeitos do recobrimento dos segmentos dos processo numa chamada `exec` é a destruição do *buffer* associado ao arquivo na região usuário, e portanto a perda de informações contidas por ele. Para contornar o problema, deve-se forçar o esvaziamento completo do *buffer* antes da chamada `exec` de forma a não perder seus dados, utilizando-se para isso a função `flush()`.

Exemplo 1:

```
                /* arquivo test_buffer1.c */
#include <stdio.h>
#include <unistd.h>

int main()
{
     printf("Voce nao vai conseguir ler este texto") ;
     execl("/bin/ls","ls","test_buffer1.c",NULL) ;
     exit(0);
}
```

Resultado da execução:

```
euler:~/> test_buffer1
test_buffer1.c
```

A mensagem não é impressa, pois o *buffer* de saída não foi esvaziado antes de ser destruído pela chamada `exec`.

Exemplo 2:

```
                /* arquivo test_buffer2.c */
#include <stdio.h>
#include <unistd.h>

int main()
{
     printf("Voce nao vai conseguir ler este texto\n") ;
     execl("/bin/ls","ls","test_buffer1.c",NULL) ;
     exit(0);
}
```

Resultado da execução:

```
euler:~/> test_buffer2
Voce nao vai conseguir ler este texto
test_buffer1.c
```

A mensagem agora é impressa, pois o caracter `\n` esvazia o *buffer* de saída e retorna à linha.

Exemplo 3:

```
              /* arquivo test_buffer3.c */
#include <stdio.h>
#include <unistd.h>

int main()
{
     printf("Voce nao vai conseguir ler este texto\n") ;
     fflush(stdout) ;
     execl("/bin/ls","ls","test_buffer1.c",NULL) ;
     exit(0);
}
```

Resultado da execução:

```
euler:~/> test_buffer3
Voce nao vai conseguir ler este texto
test_buffer1.c
```

O resultado é semelhante ao anterior, só que desta vez, o *buffer* é esvaziado através da primitiva `fflush`.

Observações: `stdout` corresponde à saída padrão em UNIX, que é neste caso a tela. Note também que comandos internos do shell não podem ser executados através de `exec()`. Por exemplo, não teria nenhum efeito a utilização de um processo filho associado a `exec()` para a execução do comando shell `cd`. Isto porque o atributo mudado no processo filho (no caso o diretório corrente) não pode ser remontado ao pai uma vez que o filho morrerá imediatamente após a execução do `exec`. Verifique a afirmação através do exemplo a seguir:

Exemplo:

```
               /* arquivo test_cd.c */

/* a mudanca de diretorio so e valida */
/* durante o tempo de execucao do processo */

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

int main()
{
     if(chdir("..")==-1) /* retorno ao diretorio predecente */
       {  perror("impossivel de achar o diretorio especificado") ;
       exit(-1);
```

```
      }
    /* sera executado um pwd que vai matar o processo */
    /* e que vai retornar o diretorio corrente onde ele esta */
    if(execl("/bin/pwd","pwd",NULL)==-1)
    {
         perror("impossivel de executar o pwd") ;
         exit(-1) ;
    }
    exit(0);
}
```

Faça um `pwd` no diretorio corrente. Lance o programa `test_cd` e verifique então que o diretório foi momentaneamente alterado durante a execução deste. Espere o fim do programa e execute novamente um `pwd`. O diretório corrente não foi alterado como era de se esperar.

2.2.4 Primitiva system()

```
#include <stdlib.h>

int system (const char * string)
```

Esta primitiva executa um comando especificado por `string`, chamando o programa `/bin/sh/ -c string`, retornando após o comando ter sido executado. Durante a execução do comando, SIGCHLD será bloqueado e SIGINT e SIGQUIT serão ignorados (estes sinais serão detalhados no próximo capítulo).

Valor de retorno: O código de retorno do comando. Em caso de erro, retorna 127 se a chamada da primitiva `execve()` falhar, ou -1 se algum outro erro ocorrer.

Capítulo 3

Os sinais

3.1 Os sinais gerados pelo sistema

3.1.1 Introdução

Um sinal é uma interrupção por software que é enviada aos processos pelo sistema para informá-los da ocorrência de eventos "anormais" dentro do ambiente de execução (por exemplo, violação de memória, erro de entrada e saída, etc). Deve-se notar que este mecanismo possibilita ainda a comunicação entre diferentes processos.

Um sinal (à exceção de `SIGKILL`) é tratado de três maneiras diferentes em `UNIX`:

- ele pode ser simplesmente ignorado. Por exemplo, o programa pode ignorar as interrupções de teclado geradas pelo usuário (é exatamente o que se passa quando um processo é lançado em *background*).
- ele pode ser interceptado. Neste caso, na recepção do sinal, a execução de um processo é desviado para o procedimento específico especificado pelo usuário, para depois retomar a execução no ponto onde ele foi interrompido.
- Seu comportamento *par défaut* pode ser aplicado a um processo após a recepção de um sinal.

3.1.2 Tipos de sinal

Os sinais são identificados pelo sistema por um número inteiro. O arquivo `/usr/include/signal.h` contém a lista de sinais acessíveis ao usuário. Cada sinal é caracterizado por um mneumônico. Os sinais mais usados nas aplicações em `UNIX` são listados a seguir:

- `SIGHUP` (1) Corte: sinal emitido aos processos associados a um terminal quando este se "desconecta". Ele é também emitido a cada processo de um grupo quando o chefe termina sua execução.
- `SIGINT` (2) Interrupção: sinal emitido aos processos do terminal quando as teclas de interrupção (`INTR` ou `CTRLc`) do teclado são acionadas.
- `SIGQUIT` (3)* Abandono: idem com a tecla de abandono (`QUIT` ou `CTRLD`).

- `SIGILL` (4)* Instrução ilegal: emitido quando uma instrução ilegal é detectada.
- `SIGTRAP` (5)* Problemas com trace: emitido após cada intrução em caso de geração de traces dos processos (utilização da primitiva ptrace())
- `SIGIOT` (6)* Problemas de intrução de E/S: emitido em caso de problemas materiais
- `SIGEMT` (7) Problemas de intrução emulador: emitido em caso de erro material dependente da implementação
- `SIGFPE` (8)* Emitido em caso de erro de cálculo em ponto flutuante, assim como no caso de um número em ponto flutuante em formato ilegal. Indica sempre um erro de programação.
- `SIGKILL` (9) Destruição: arma absoluta para matar os processos. Não pode ser ignorada, nem interceptada (veja `SIGTERM` para uma morte mais suave para processos)
- `SIGBUS` (10)* Emitido em caso de erro sobre o barramento
- `SIGSEGV` (11)* Emitido em caso de violação da segmentação: tentativa de acesso a um dado fora do domínio de endereçamento do processo.
- `SIGSYS` (12)* Argumento incorreto de uma chamada de sistema
- `SIGPIPE` (13) Escrita sobre um pipe não aberto em leitura
- `SIGALRM` (14) Relógio: emitido quando o relógio de um processo para. O relógio é colocado em funcionamento através da primitiva `alarm()`
- `SIGTERM` (15) Terminação por software: emitido quando o processo termina de maneira normal. Pode ainda ser utilizado quando o sistema quer por fim à execução de todos os processos ativos.
- `SIGUSR1` (16) Primeiro sinal disponível ao usuário: utilizado para a comunicação interprocessos.
- `SIGUSR2` (17)Primeiro sinal disponível ao usuário: utilizado para a comunicação interprocessos.
- `SIGCLD` (18) Morte de um filho: enviado ao pai pela terminção de um processo filho
- `SIGPWR` (19) Reativação sobre pane elétrica

Observação: Os sinais marcados por * geram um arquivo `core` no disco quando eles não são corretamente tratados.

Para maior portabilidade dos programas que utilizam sinais, pode-se pensar em aplicar as seguintes regras: evitar os sinais `SISIOT`, `SIGEMT`, `SIGBUS` e `SIGSEGV` que são dependentes da implementação. O mais correto seria interceptá-los para imprimir uma mensagem relativa a eles, mas não se deve nunca tentar atribuir uma significação qualquer que seja para a ocorrência destes sinais.

3.1.3 Tratamento dos processos zumbis

O sinal `SIGCLD` se comporta diferentemente dos outros. Se ele é ignorado, a terminação de um processo filho, sendo que o processo pai não está em espera, não irá acarretar a criação de um processo zumbi (veja seção 2.2.1)

Exemplo: O programa a seguir gera um processo zumbi quando o pai é informado da morte do filho através de um sinal `SIGCLD`.

```
                /* arquivo test_sigcld.c */

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

int main() {
     if (fork() != 0) while(1) ;
     exit(0);
}
```

Resultado da execução:

```
euler:~/> test_sigcld &
euler:~/> ps
  PID TTY          TIME CMD
  675 pts/0    00:00:00 tcsh
 1038 pts/0    00:00:01 test_sigcld
 1039 pts/0    00:00:00 test_sigcld <defunct>
 1040 pts/0    00:00:00 ps
```

No próximo programa, o pai ignora o sinal `SIGCLD`, e seu filho não vai mais se tornar um processo zumbi.

```
               /* arquivo test_sigcld2.c */

#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int main() {
     signal(SIGCLD,SIG_IGN) ;/* ignora o sinal SIGCLD */
     if (fork() != 0)
         while(1) ;
     exit(0);
}
```

Resultado da execução:

```
euler:~/> test_sigcld2 &
euler:~/> s
  PID TTY          TIME CMD
  675 pts/0    00:00:00 tcsh
 1055 pts/0    00:00:01 test_sigcld2
 1057 pts/0    00:00:00 ps
```

Observação: A primitiva `signal()` será mais detalhada na seção 3.2.2 do texto.

3.1.4 Sinal SIGHUP: tratamento de aplicações duráveis

O sinal `SIGHUP` pode ser incômodo quando o usuário deseja que um processo continue a ser executado após o fim de sua seção de trabalho (aplicação durável). De fato, se o processo não trata esse sinal, ele será interrompido pelo sistema no instante de "deslogagem". Existem diferentes soluções para solucionar este problema:

1. Utilizar o comando shell *at* ou `@` que permite de lançar uma aplicação numa certa data, via um processo do sistema, denominado deamon. Neste caso, o sinal `SIGHUP` não terá nenhuma influencia sobre o processo, uma vez que ele não está ligado a nenhum terminal.

2. Incluir no código da aplicação a recepção do sinal `SIGHUP`

3. Lançar o programa *em background* (na verdade, um processo lançado em *background* trata automaticamente o sinal `SIGHUP`

4. Lançar a aplicação associada ao comando `nohup`, que provocará uma chamada à primitiva `trap`, e que redigire a saída padrão sobre `nohup.out`.

3.2 Tratamento dos sinais

3.2.1 Emissão de um sinal

Primitiva `kill()`

```
#include <signal.h>
int kill(pid,sig) /* emissao de um sinal */
int pid ;         /* id do processo ou do grupo de destino */
int sig ;         /* numero do sinal */
```

Valor de retorno: 0 se o sinal foi enviado, -1 se não foi.

A primitiva `kill()` emite ao processo com número pid o sinal de número sig. Além disso, se o valor inteiro sig é nulo, nenhum sinal é enviado, e o valor de retorno vai informar se o número de pid é um número de um processo ou não.

Utilização do parâmetro `pid`:

- Se `pid` > 0: pid designará o processo com ID igual a `pid`.
- Se `pid` = 0: o sinal é enviado a todos os processos do mesmo grupo que o emissor. Esta possibilidade é geralmente utilizada com o comando shell `kill`. O comando `kill -9 0` irá matar todos os processos rodando em *background* sem ter que indicar os IDs de todos os processos envolvidos.
- Se `pid` = 1:
  - Se o processo pertence ao super-usuário, o sinal é enviado a todos os processos, exceto aos processos do sistema e ao processo que envia o sinal.
  - Senão, o sinal é enviado a todos os processos com ID do usuário real igual ao ID do usuário efetivo do processo que envia o sinal (é uma forma de matar todos os processos que se é proprietário, independente do grupo de processos ao qual se pertence)
- Se `pid` < 1: o sinal é enviado a todos os processos para os quais o ID do grupo de processos (`pgid`) é igual ao valor absoluto de `pid`.

Note finalmente que a primitiva `kill()` é na maioria das vezes executada via o comando shell `kill`.

Primitiva `alarm()`

```
#include <unistd.h>
unsigned int alarm(unsigned int secs)  /* envia um sinal SIGALRM */
```

Valor de retorno: tempo restante no relógio se já existir um alarme armado anteriormente ou 0 se não exitir. Se o `secs` for igual a 0, ele retorna o valor do tempo restante no relógio, sem portanto re-armar o alarme.

A primitiva `alarm()` envia um sinal `SIGALRM` ao processo chamando após um intervalo de tempo `secs` (em segundos) passado como argumento, depois reinicia o relógio de alarme. Na chamada da primitiva, o relógio é reiniciado a `secs` segundos e é decrementado até 0. Esta primitiva pode ser utilizada, por exemplo, para forçar a leitura do teclado dentro de um dado intervalo de tempo. O tratamento do sinal deve estar previsto no programa, senão o processo será finalizado ao recebê-lo.

Exemplo 1:

```
            /* arquivo test_alarm.c */

/* testa os valores de retorno de alarm() */
/* assim que seu funcionamento */

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

```
void it_horloge(int sig) /* rotina executada na recepção de SIGALRM */
{ printf("recepção do sinal %d :SIGALRM\n",sig) ; }

main() {
  unsigned sec ;
  signal(SIGALRM,it_horloge) ; /* interceptação do sinal */
  printf("Fazendo alarm(5)\n") ;
  sec = alarm(5) ;
  printf("Valor retornado por alarm: %d\n",sec) ;
  printf("Principal em loop infinito (CTRLc para acabar)\n") ;
  for(;;) ;
}
```

Resultado da execução:

```
euler:~/> test_alarm
Fazendo alarm(5)
Valor retornado por alarm: 0
Principal em loop infinito (CTRLc para acabar)
recepção do sinal 14 :SIGALRM
```

Exemplo 2:

```
                /* arquivo test_alarm2.c */

/*
 * teste dos valores de retorno de alarm() quando 2
 * chamadas a alarm() sao feitas sucessivamente
 */

#include <unistd.h>
#include <stdio.h>
#include <signal.h>

void it_horloge(int sig) /* tratamento do desvio sobre SIGALRM */
{
     printf("recepcao do sinal %d : SIGALRM\n",sig) ;
     printf("atencao, o principal reassume o comando\n") ;
}

void it_quit(int sig) /* tratamento do desvio sobre SIGALRM */
{
     printf("recepcao do sinal %d : SIGINT\n",sig) ;
     printf("Por que eu ?\n") ;
     exit(1) ;
```

```
}

int main()
{
unsigned sec ;
     signal(SIGINT,it_quit); /* interceptacao do ctrl-c */
     signal(SIGALRM,it_horloge); /* interceptacao do sinal de alarme */
     printf("Armando o alarme para 10 segundos\n");
     sec=alarm(10);
     printf("valor retornado por alarm: %d\n",sec) ;
     printf("Paciencia... Vamos esperar 3 segundos com sleep\n");
     sleep(3) ;
     printf("Rearmando alarm(5) antes de chegar o sinal precedente\n");
     sec=alarm(5);
     printf("novo valor retornado por alarm: %d\n",sec);
     printf("Principal em loop infinito (ctrl-c para parar)\n");
     for(;;);
}
```

Observação: A interceptação do sinal só tem a finalidade de fornecer uma maneira elegante de sair do programa, ou em outras palavras, de permitir um redirecionamento da saída padrão para um arquivo de resultados.

Resultado da execução:

```
euler:~/> test_alarm2
Armando o alarme para 10 segundos
valor retornado por alarm: 0
Paciencia... Vamos esperar 3 segundos com sleep
Rearmando alarm(5) antes de chegar o sinal precedente
novo valor retornado por alarm: 7
Principal em loop infinito (ctrl-c para parar)
recepcao do sinal 14 : SIGALRM
atencao, o principal reassume o comando
recepcao do sinal 2 : SIGINT
Por que eu ?
euler:~/>
```

Pode-se notar que o relógio é reinicializado para o valor de 5 segundos durante a segunda chamada de `alarm()`, e que mais ainda, o valor retornado é o estado atual do relógio. Finalmente, pode-se observar que o relógio é decrementado ao longo do tempo. As duas últimas linhas da execução são geradas após um sinal `CTRL-c` do teclado.

Observação: A função `sleep()` chama a primitiva `alarm()`. Deve-se então utilizá-la com maior prudência se o programa já manipula o sinal `SIGALRM`.

Exemplo usando `sleep()`:

Implementação de uma versão da função `sleep()` que utiliza as primitivas `pause()` e `alarm()`. O princípio de funcionamente é simples: um processo arma um alarme (via `alarm()`) e se posiciona em pausa (via `pause()`). Na chegada do sinal `SIGALRM`, `pause()` será interrompida e o processo termina sua execução.

```
                /* arquivo test_sleep.c */

 /* utilizacao de pause() e de alarm() para
  * implementar uma primitiva sleep2 */

#include <errno.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

void nullfcn()   /* define-se aqui uma funcao executada quando */
{ }              /* o sinal SIGALRM é interceptado por signal() */
                 /* esta funcao nao faz absolutamente nada */

void sleep2(int secs) /* dorme por secs segundos */
{
  if( signal(SIGALRM,nullfcn) )
     {
          perror("error: reception signal") ;
          exit(-1) ;
     }
     alarm(secs) ; /* inicializa o relogio a secs segundos */
     pause() ;     /* processo em espera por um sinal */
}

int main() /* so para testar sleep2() */
{
     if(fork()==0)
     {
          sleep(3) ;
          printf("hello, sleep\n") ;
     }
     else /* pai */
     {
          sleep2(3) ;
          printf("hello, sleep2\n") ;
     }
     exit(0);
```

```
}
```

Resultado da execução:
Após 3 segundos, deve-se obter indiferentemente:

```
hello, sleep2
hello, sleep
```

Ou então:

```
hello, sleep2
hello, sleep
```

Observação: O interesse da função `nullfunc()` é de se assegurar que o sinal que desperta o processo não provoque o comportamento *par défaut* e que não seja ignorado, de forma a garantir que a pausa (via `pause()`) possa ser interrompida.

3.2.2 Recepção de sinais:

Primitive `signal()`

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Valor de retorno: o valor anterior do manipulador do sinal, ou SIG_ERR (normalmente -1) quando houver erro.

A chamada de sistema `signal()` define um novo manipulador (*handler*) para o sinal especificado pelo número `signum`. Em outras palavras, ela intercepta o sinal `signum`. O manipulador do sinal é "setado" para `handler`, que é um ponteiro para uma função que pode assumir um dos três seguintes valores:

- `SIG_DFL`: indica a escolha da ação *défaut* para o sinal. A recepção de um sinal por um processo provoca a terminação deste processo, menos para `SIGCLD` e `SIGPWR`, que são ignorados *par défaut*. No caso de alguns sinais, existe a criação de um arquivo de imagem core no disco.
- `SIG_IGN`: indica que o sinal deve ser ignorado: o processo é imunisado contra este sinal. Lembrando sempre que o sinal `SIGKILL` nunca pode ser ignorado.
- Um ponteiro para uma função (nome da função): implica na captura do sinal. A função é chamada quando o sinal chega, e após sua execução, o tratamento do processo recomeça onde ele foi interrompido. Não se pode proceder um desvio na recepção de um sinal `SIGKILL` pois esse sinal não pode ser interceptado, nem para `SIGSTOP`.

Pode-se notar então que é possível de modificar o comportamento de um processo na chegada de um dado sinal. É exatamente isso que se passa para um certo número de processos *standards* do sistema: o shell, por exemplo, ao receber um sinal `SIGINT` irá escrever na tela o *prompt* (e não será interrompido).

Primitive `pause()`

```
#include <unistd.h>
int pause(void)    /* espera de um sinal qualquer */
```

Valor de retorno: sempre retorna -1.

A primitiva `pause()` corresponde a uma espera simples. Ela não faz nada, nem espera nada de particular. Entretanto, uma vez que a chegada de um sinal interrompe toda primitiva bloqueada, pode-se dizer que `pause()` espera simplesmente a chegada de um sinal.

Observe o comportamento de retorno clássico de um primitiva bloqueada, isto é o posicionamento de errno em `EINTR`. Note que, geralmente, o sinal esperado por `pause()` é o relógio de `alarm()`.

Exemplo:

```
                /* arquivo test_pause.c */

/* teste sobre o valor retornado por pause() */

#include <errno.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>

void it_main(sig) /* tratamento sobre o 1o SIGINT */
int sig ;
{
     printf("recepcao do sinal numero : %d\n",sig) ;
     printf("vamos retomar o curso ?\n") ;
     printf("é o que o os profs insistem em dizer geralmente!\n") ;
}

void it_fin(sig) /* tratamento sobre o 2o SIGINT */
int sig ;
{
     printf("recepcao do sinal numero : %d\n",sig) ;
     printf("ok, tudo bem, tudo bem ...\n") ;
     exit(1) ;
}

int main()
{
     signal(SIGINT,it_main) ; /* interceptacao do 1o SIGINT */
     printf("vamos fazer uma pequena pausa (cafe!)\n") ;
     printf("digite CTRL-c para imitar o prof\n") ;
     printf("retorno de pause (com a recepcao do sinal): %d\n",pause()) ;
     printf("errno = %d\n",errno) ;
```

```
     signal(SIGINT,it_fin) ; /* rearma a interceptacao: 2o SIGINT */
     for(;;) ;
     exit(0) ;
}
```

Resultado da execução:

```
euler:~/> test_pause
vamos fazer uma pequena pausa (cafe!)
digite CTRL-c para imitar o prof
recepcao do sinal numero : 2
vamos retomar o curso ?
é o que o os profs insistem em dizer geralmente!
retorno de pause (com a recepcao do sinal): -1
errno = 4
recepcao do sinal numero : 2
ok, tudo bem, tudo bem ...
euler:~/>
```

A primitiva `pause()` é interrompida pelo sinal `SIGINT`, retorna -1 e errno é posicionado em 4: `interrupted system call`.

3.3 Processos manipulando sinais: Exemplos

3.3.1 Herança de sinais com `fork()`

Os processos filhos recebem a imagem da memória do pai, herdando seu comportamento em relação aos sinais. O próximo exemplo descreve este fenômeno:

```
                /* arquivo test_sign_fork.c */

/* heranca pelo filho do comportamento do pai
 * em relacao aos sinais */

#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void fin()
{
     printf("SIGINT para os processos %d\n",getpid()) ;
     exit(0) ;
}
```

```
int main()
{
     signal(SIGQUIT,SIG_IGN) ;
     signal(SIGINT,fin) ;
     if (fork()>0){
          printf("processo pai : %d\n",getpid()) ;
          while(1) ;
     }
     else {
          printf("processo filho : %d\n",getpid()) ;
          while(1) ;
     }
     exit(0);
}
```

Resultado da execução:

```
euler:~/> test_sign_fork
processo pai : 1736
processo filho : 1737
SIGINT para os processos 1737
SIGINT para os processos 1736
euler:~/>
```

3.3.2 Comunicação entre processos

O programa a seguir é um exemplo simples da utlização das primitivas de emissão e recepção de sinais com o objetivo de permitir a comunicação entre dois processos. A execução deste programa permite ainda de assegurar que o processo executando a rotina de desvio é mesmo aquele que recebeu o sinal.

```
                /* arquivo test_kill_signal.c */

/* comunicacao simples entre dois processos
 *  atraves de sinais */

#include <errno.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

void it_fils()
{
   printf("- Sim, sim. E darei um jeito nisso eu mesmo... ARGHH...\n") ;
   kill (getpid(),SIGINT) ;
}
```

```
void fils()
{
   signal(SIGUSR1,it_fils) ;
   printf("- Papai conte mais uma vez, como foi que voce me fez?\n") ;
   while(1) ;
}

int main()
{
 int pid ;

  if ((pid=fork())==0) fils() ;
  else {
    sleep(2) ;
    printf("- Filhinho, quer ir passear no reino dos mortos?\n") ;
    kill (pid,SIGUSR1) ;
    sleep(1);
  }
  exit(0);
}
```

Resultado da execução:

```
euler:~/> test_kill_signal
- Papai conte mais uma vez, como foi que voce me fez?
- Filhinho, quer ir passear no reino dos mortos?
- Sim, sim. E darei um jeito nisso eu mesmo... ARGHH...
```

Um processo criou um filho que parece não estar vivendo muito feliz. Este processo vai enviar então ao filho um sinal `SIGUSR1` ao filho. Com a recepção desse sinal, o filho desesperado decide de enviar a si mesmo um sinal `SIGINT` para se suicidar.

3.3.3 Controle da progressão de uma aplicação

Todos aqueles que já lançaram programas de simulação ou de cálculo numérico muito longos devem ter pensado numa forma de saber como está progredindo a aplicação durante a sua execução. Isto é perfeitamente possível através do envio do comando shell `kill` aos processos associados ao sinal. Os processos podem então, após a recepção deste sinal, apresentar os dados desejados. O exemplo a seguir mostra um programa que ajuda a resolver este problema:

```
                /* arquivo verificacao.c */

#include <errno.h>
#include <stdio.h>
```

```
#include <signal.h>
#include <time.h>
#include <unistd.h>

/* as variaveis a serem editadas devem ser globais */
long somme = 0 ;

void it_verificacao()
{
 long t_date ;
     signal(SIGUSR1, it_verificacao) ;/* reativo SIGUSR1 */
     time(&t_date) ;
     printf("\n Data do teste : %s ", ctime(&t_date)) ;
     printf("valor da soma : %d \n", (int) somme) ;
}

int main()
{
     signal(SIGUSR1,it_verificacao) ;
     printf ("Enviar o sinal USR1 para o processo %d \n",getpid()) ;
     while(1) {
       sleep(1);
       somme++ ;
     }
     exit(0);
}
```

Resultado da execução:
Se o programa é lançado em *background*, se o usuário digitar o comando shell `kill -USR1 pid`, ele irá obter as variáveis de controle desejadas. A primitiva `ctime()` usada no programa retorna um ponteiro para uma cadeia de caracteres contendo a data sob a forma:

```
"Wed Jun 30 21:49:08 1993\n"
```

3.4 Conclusão

A exceção de `SIGCLD`, os sinais que são recebidos por um processo não são memorizados: ou eles são ignorados, ou eles põem fim na execução dos processos, ou ainda eles são interceptados e tratados por algum procedimento. Por esta razão, os sinais não são apropriados para a comunicação interprocessos... Uma mensagem sob a forma de sinal pode ser perdida se o sinal é recebido num momento onde o tratamento para esse tipo de sinal é simplesmente ignorá-lo. Após a captura de um sinal por um processo, esse processo vai readotar seu comportamento *par défaut* em relação a esse sinal. Assim, no caso de se desejar captar um mesmo sinal várias vezes, é conveniente fazer a redefinição do comportamento do

processo pela primitiva `signal()`. Geralmente, a interceptação do sinal deve ser rearmada o mais cedo possível (deve ser a primeira instrução efetuada no procedimento de desvio para tratamento do sinal).

Um outro problema é que os sinais têm um comportamento um excessivamente abrupto em relação à execução do programa: na sua chegada, eles vão interromper o trabalho em curso. Por exemplo, a recepção de um sinal enquanto o processo espera um evento (algo que pode acontecer durante a utilização das primitivas `open(), read(), write(), pause(), wait(),...`), laça a execução imediata da rotina de desvio; em seu retorno, a primitiva interrompida reenvia uma mensagem de erro, mesmo sem ser totalmente completada (errno é posicionado em `EINTR`). Por exemplo, quando um processo pai que intercepta os sinais de interrupção e de abandono está em espera da terminação de um filho, é possível que um sinal de interrupção ou de abandono tire o pai da espera no wait() antes que o filho tenha terminado sua execução. Neste caso, um processo `<defuntc>` será criado. Uma forma de contornar esse problema é ignorar certos sinais antes da chamadas de tais primitivas (levando irremediavelment a outros problemas, uma vez que esses sinais não serão tratados de forma alguma).

3.5 Lista de Sinais em `LINUX`

A lista completa de sinais disponíveis em `LINUX` aplicações é mostrada nas tabelas 3.1 e 3.2. Alguns dos sinais relacionados nas tabelas são dependentes da arquitetura, sendo a primeira lista conforme à norma `POSIX.1`):

Sinal	Valor	Ação	Descrição
SIGHUP	1	A	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	A	Interrupt from keyboard
SIGQUIT	3	A	Quit from keyboard
SIGILL	4	A	Illegal Instruction
SIGABRT	6	C	Abort signal from abort(3)
SIGFPE	8	C	Floating point exception
SIGKILL	9	AEF	Kill signal
SIGSEGV	11	C	Invalid memory reference
SIGPIPE	13	A	Broken pipe: write to pipe with no readers
SIGALRM	14	A	Timer signal from alarm(2)
SIGTERM	15	A	Termination signal
SIGUSR1	30,10,16	A	User-defined signal 1
SIGUSR2	31,12,17	A	User-defined signal 2
SIGCHLD	20,17,18	B	Child stopped or terminated
SIGCONT	19,18,25		Continue if stopped
SIGSTOP	17,19,23	DEF	Stop process
SIGTSTP	18,20,24	D	Stop typed at tty
SIGTTIN	21,21,26	D	tty input for background process
SIGTTOU	22,22,27	D	tty output for background process

Tabela 3.1: Sinais mais usados em UNIX (POSIX.1)

Alguns outros sinais importantes em `UNIX` são apresentados na tabela 3.2.

Sinal	Valor	Ação	Descrição
SIGTRAP	5	CG	Trace/breakpoint trap
SIGIOT	6	CG	IOT trap. A synonym for SIGABRT
SIGEMT	7,-,7	G	
SIGBUS	10,7,10	AG	Bus error
SIGSYS	12,-,12	G	Bad argument to routine (SVID)
SIGSTKFLT	-,16,-	AG	Stack fault on coprocessor
SIGURG	16,23,21	BG	Urgent condition on socket (4.2 BSD)
SIGIO	23,29,22	AG	I/O now possible (4.2 BSD)
SIGPOLL		AG	A synonym for SIGIO (System V)
SIGCLD	-,-,18	G	A synonym for SIGCHLD
SIGXCPU	24,24,30	AG	CPU time limit exceeded (4.2 BSD)
SIGXFSZ	25,25,31	AG	File size limit exceeded (4.2 BSD)
SIGVTALRM	26,26,28	AG	Virtual alarm clock (4.2 BSD)
SIGPROF	27,27,29	AG	Profile alarm clock
SIGPWR	29,30,19	AG	Power failure (System V)
SIGINFO	29,-,-	G	A synonym for SIGPWR
SIGLOST	-,-,-	AG	File lock lost
SIGWINCH	28,28,20	BG	Window resize signal (4.3 BSD, Sun)
SIGUNUSED	-,31,-	AG	Unused signal

Tabela 3.2: Outros sinais usados em UNIX (POSIX.1)

Na tabela, - significa que o sinal é ausente; existem 3 valores que são dados: o primeiro é normalmente válido para arquiteturas `SPARC` e `ALPHA`, o do meio para `i386` e `ppc`, e o último para `MIPS`. O sinal 29 é `SIGINFO/SIGPWR` para `ALPHA` e `SIGLOST` para `SPARC`).

As letras no campo "Ação" têm o seguinte significado:

- A - Ação Default é terminar o processo
- B - Ação Default é ignorar o sinal
- C - Ação Default é to dump core.
- D - Ação Default é parar o processo (stop)
- E - Sinal não pode ser mascarado ou tratado
- F - Sinal não pode ser ignorado
- G - Não é um sinal conformante à norma `POSIX.1`

Parte II

A parte II da apostila abrange os capítulos 3 e 4, tratando dos mecanismos de base para as interações entre processos `UNIX` : sinais e tubos.

O capítulo 3 apresenta os sinais gerados pelo sistema e a forma como estes podem ser utilizados para a comunicação entre processos.

O capítulo 4 introduz os tubos (ou *pipes*) e suas características principais, mostrando como eles podem ser utilizados de maneira efetiva para a comunicação entre processos. `UNIX`

Capítulo 4

Os tubos ou *pipes* de comunicação

4.1 Introdução

Os tubos (ou *pipes*) constituem um mecanismo fundamental de comunicação unidirecional entre processos. Eles são um mecanismo de I/O com duas extremidades, ou *socketc*, correspondendo na verdade a filas de caractereres do tipo FIFO (First In First Out): as informações são introduzidas numa das extremidades (num *socket*) e retiradas em outra (outro *socket*). Por exemplo, quando o usuário executa o comando shell `prog1|prog2`, está se fazendo o chamado "piping" entre a saída de um programa `prog1` para a entrada de outro `prog2`.

Os tubos são implementados como arquivos (eles possuem um *i-node* na tabela de indexação), mesmo se eles não têm um nome definido no sistema. Assim, o programa especifica sua entrada e sua saída somente como um descritor na tabela de índices, o qual aparece como uma variável ou constante, podendo a fonte (entrada) e o destino (saída) serem alteradas sem que para isso o texto do programa tenha que ser alterado. No caso do exemplo, na execução de `prog1` a saída padrão (*stdout*) é substituída pela entrada do tubo. No `prog2`, de maneira similar, a entrada padrão (*stdin*) é substituída pela saída de `prog1`.

A técnica dos tubos é freqüentemente utilizada nos *shells* para redirecionar a saída padrão de um comando para a entrada de um outro.

4.2 Particularidades dos tubos

- Uma vez que eles não têm nomes, os tubos de comunicação são temporários, existindo apenas em tempo de execução do processo que os criou;
- A criação dos tubos é feita através de uma primitiva especial: `pipe()`;
- Vários processos podem fazer leitura e escrita sobre um mesmo tubo, mas nenhum mecanismo permite de diferenciar as informações na saída do tubo;
- A capacidade é limitada (em geral a 4096 bytes). Se a escrita sobre um tubo continua mesmo depois do tudo estar completamente cheio, ocorre uma situaçào de bloqueio (*dead-lock*);
- Os processos comunicando-se através dos tubos devem ter uma ligação de parentesco, e os tubos religando processos devem ter sidos abertos antes da criação dos filhos (veja a passagem de descritores

de arquivos abertos durante a execução do `fork()` na seção 2.2.1);

- É impossível fazer qualquer movimentação no interior de um tubo.
- Com a finalidade de estabelecer um diálogo entre dois processos usando tubos, é necessário a abertura de um tubo em cada direção.

4.3 Criação de um tubo

4.3.1 A Primitiva `pipe()`

```
#include <unistd.h>

int pipe(int desc[2]);
```

Valor de retorno: 0 se a criação tiver sucesso, e -1 em caso de falha.

A primitiva `pipe()` cria um par de descritores, apontando para um *i-node*, e coloca-os num vetor apontado por `desc`:

- desc[0] contém o número do descritor pelo qual pode-se ler no tubo
- desc[1] contém o número do descritor pelo qual pode-se escrever no tubo

Assim, a escrita sobre `desc[1]` introduz dados no tubo, e a leitura em `desc[0]` extrai dados do tubo.

4.4 Segurança do sistema

No caso em que todos os descritores associados aos processos susceptíveis de ler num tubo estiverem fechados, um processo que tenta escrever neste tubo deve receber um sinal `SIGPIPE`, sendo então interrompido se ele não possuir uma rotina de tratamento deste sinal.

Se um tubo está vazio, ou se todos os descritores susceptíveis de escrever sobre ele estiverem fechados, a primitiva `read()` retornará o valor 0 (fim de arquivo lido).

Exemplo 1: emissão de um sinal `SIGPIPE`

```
                /* arquivo test_pipe_sig.c */

/* teste de escrita num tubo fechado a leitura */

#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

void it_sigpipe()
{
     printf("Sinal SIGPIPE recebido \n") ;
```

```
}
int main()
{
     int p_desc[2] ;
     signal(SIGPIPE,it_sigpipe) ;
     pipe(p_desc) ;
     close(p_desc[0]) ;  /* fechamento do tubo em leitura */
     if (write(p_desc[1],"0",1) == -1)
          perror("Error write") ;
     exit(0);
}
```

Resultado da execução:

```
euler:~/> test_pipe_sig
Sinal SIGPIPE recebido
Error write: Broken pipe
```

Neste exemplo, tenta-se escrever num tubo sendo que ele acaba de ser fechado em leitura; o sinal `SIGPIPE` é emitido e o programa é desviado para a rotina de tratamento deste sinal. No retorno, a primitiva `write()` retorna -1 e `perror` imprime na tela a mensagem de erro.

Exemplo 2: Leitura num tubo fechado em escrita.

```
                /* arquivo test_pipe_read.c */

/* teste de leitura num tubo fechado em escrita */

#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
  int i, p_desc[2] ;
  char c ;
        pipe(p_desc) ; /* criacao do tubo */
        write(p_desc[1],"AB",2) ; /* escrita de duas letras no tubo */
        close(p_desc[1]) ;        /* fechamento do tubo em escrita */

        /* tentativa de leitura no tuno */
        for (i=1; i<=3; i++) {
          if ( (read(p_desc[0],&c,1) == 0) )
                        printf("Tubo vazio\n") ;
                else
```

```
                        printf("Valor lido: %c\n",c) ;
        }
        exit(0);
}
```

Resultado da execução:

```
euler:~/> test_pipe_read
Valor lido: A
Valor lido: B
Tubo vazio
euler:~/>
```

Este exemplo mostra que a leitura num tubo é possível, mesmo se este tiver sido fechado para a escrita. Obviamento, quando o tubo estiver vazio, `read()` vai retornar o valor 0.

4.5 Aplicações das primitivas de entrada e saída

É possível utilizar as funções da biblioteca padrão sobre um tubo já aberto, associando a esse tubo - por meio da função `fopen()` - um ponteiro apontando sobre uma strutura do tipo `FILE`:

- `write()` : os dados são escritos no tubo na ordem em que eles chegam. Quando o tubo está cheio, `write()` se bloqueia esperando que uma posição seja liberada. Pode-se evitar este bloqueio utilizando-se o *flag* `O_NDELAY`.
- `read()` : os dados são lidos no tubo na ordem de suas chegadas. Uma vez retirados do tubo, os dados não poderão mais serem relidos ou restituídos ao tubo.
- `close()` : esta função é mais importante no caso de um tubo que no caso de um arquivo. Não somente ela libera o descritor de arquivo, mas quando o descritor de arquivo de escritura está fechado, ela funciona como um fim de arquivo para a leitura.
- dup() : esta primitiva combinada com `pipe()` permite a implementaçãão dos comandos religados por tubos, redirecionando a saída padrão de um comando para a entrada padrão de um outro.

4.5.1 Exemplos globais

4.5.2 Implementação de um comando com tubos

Este exemplo permite observar como as primitivas `pipe()` e `dup()` podem ser combinadas com o objetivo de produzir comandos *shell* do tipo `ls|wc|wc`. Note que é necessário fechar os descritores não utilizados pelos processos que executam a rotina.

```
                /* arquivo test_pipe.c */

/* este programa é equivalente ao comando shell ls|wc|wc */
```

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

int p_desc1[2] ;
int p_desc2[2] ;

void faire_ls()
{
     /* saida padrao redirecionada para o 1o. tubo */
     close (1) ;
     dup(p_desc1[1]) ;
     close(p_desc1[1]) ;

     /* fechamento dos descritores nao-utilizados */
     close(p_desc1[0]) ;
     close(p_desc2[1]) ;
     close(p_desc2[0]) ;

     /* executa o comando */
     execlp("ls","ls",0) ;
     perror("impossivel executar ls ") ;
}

void faire_wc1()
{
     /* redirecionamento da entrada padrao para o 1o. tubo*/
     close(0) ;
     dup(p_desc1[0]) ;
     close(p_desc1[0]) ;
     close(p_desc1[1]) ;

     /* redirecionamento da saida padrao para o 2o. tubo*/
     close(1) ;
     dup(p_desc2[1]) ;
     close(p_desc2[1]) ;
     close(p_desc2[0]) ;

     /* executa o comando */
     execlp("wc","wc",0) ;
     perror("impossivel executar o 1o. wc") ;

}
```

```
void faire_wc2()
{
     /* redirecionamento da entrada padrao para o 2o. tubo*/
     close (0) ;
     dup(p_desc2[0]) ;
     close(p_desc2[0]) ;

     /* fechamento dos descritores nao-utilizados */
     close(p_desc2[1]) ;
     close(p_desc1[1]) ;
     close(p_desc1[0]) ;

     /* executa o comando */
     execlp("wc","wc",0) ;
     perror("impossivel executar o 2o. wc") ;
}

int main()
{
     /* criacao do primeiro tubo*/
     if (pipe(p_desc1) == -1)
          perror("Impossivel criar o 1o. tubo") ;

     /* criacao do segundo tubo */
     if (pipe(p_desc2) == -1)
          perror("impossivel criar o 1o. tubo") ;

     /* lancamento dos filhos */
     if (fork() == 0) faire_ls() ;
     if (fork() == 0) faire_wc1() ;
     if (fork() == 0) faire_wc2() ;
     exit(0);
}
```

Resultado da execução:

```
euler:~/> ls|wc|wc
euler:~/>       1       3      24
euler:~/> test_pipe
euler:~/>       1       3      24
```

4.5.3 Comunicação entre pais e filhos usando um tubo

Exemplo 1: Envio de uma mensagem ao usuário

Este programa permite que processos troquem mensagens com ajuda do sistema de correio eletrônico.

```
          /* arquivo test_pipe_mail.c */

/* teste do envio de um mail usando tubos */

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

int main()
{
    FILE *fp;
    int pid, pipefds[2];
    char *username, *getlogin();
    /* da o nome do usuario */
    if ((username = getlogin()) == NULL) {
        fprintf(stderr, "quem e voce?\n");
        exit(1);    }
     /* Cria um tubo. Isto deve ser feito antes do fork para que
      * o filho possa herdar esse tubo
      */
    if (pipe(pipefds) < 0) { perror("Error pipe");
        exit(1);    }
    if ((pid = fork()) < 0) {    perror("Error fork");
        exit(1); }
     /* Codigo do filho:
      * executa o comando mail e entao envia ao username
      * a mensagem contida no tubo */
    if (pid == 0) {
  /* redirige a stdout para o tubo; o comando executado em seguida tera
   como entrada (uma mensagem) a leitura do tubo */
        close(0);
        dup(pipefds[0]);
        close(pipefds[0]);
  /* fecha o lado de escrita do tubo, para poder ver a saida na tela */
        close(pipefds[1]);
        /* executa o comando mail */
        execl("/bin/mail", "mail", username, 0);
        perror("Error execl");
        exit(1);
```

```
    }
    /* Codigo do pai:
     * escreve uma mensagem no tubo */
    close(pipefds[0]);
    fp = fdopen(pipefds[1], "w");
    fprintf(fp, "Hello from your program.\n");
    fclose(fp);
    /* Espera da morte do processo filho */
    while (wait((int *) 0) != pid) ;
    exit(0);
}
```

Resultado da execução: O usuário que executa o programa vai enviar a si mesmo um mensagem por correio eletrônico. A mensagem deve ser exatamente igual à mostrada a seguir:

```
Date: Fri, 13 Oct 2000 10:28:34 -0200
From: Celso Alberto Saibel Santos <saibel@leca.ufrn.br>
To: saibel@leca.ufrn.br

Hello from your program.
```

Exemplo 2: Enfoca mais uma vez a herança dos descritores através de um `fork()`. O programa cria um tubo para a comunicação entre um processo pai e seu filho.

```
                /* arquivo test_pipe_fork.c */

/* Testa heranca dos descritores na chamada do fork().
 * O programa cria um tubo e depois faz um fork. O filho
 * vai se comunicar com o pai atraves desse tubo
*/

#include <errno.h>
#include <stdio.h>
#include <unistd.h>

#define DATA "Testando envio de mensagem usando pipes"

int main()
{
   int sockets[2], child;
   char buf[1024];
   /* criacao de um tubo */
   if ( pipe(sockets) == -1 ) {
          perror("Error opening stream socket pair") ;
          exit(10);
```

```
   }
   /* criacao de um filho */
   if ( (child = fork()) == -1)
     perror ("Error fork") ;
   else if (child) {
     /* Esta ainda e a execução do pai. Ele lê a mensagem do filho */
     if ( close(sockets[1]) == -1) /* fecha o descritor nao utilizado */
          perror("Error close") ;
     if (read(sockets[0], buf, 1024) < 0 )
       perror("Error: reading message");
     printf("-->%s\n", buf);
     close(sockets[0]);
   } else {
     /* Esse e o filho. Ele escreve a mensagem para seu pai */
      if ( close(sockets[0]) == -1) /* fecha o descritor nao utilizado */
        perror("Error close") ;
      if (write(sockets[1], DATA, sizeof(DATA)) < 0 )
        perror("Error: writing message");
      close(sockets[1]);
   }
   sleep(1);
   exit(0);
}
```

Resultado da execução:

```
euler:~/> test_pipe_fork
-->Testando envio de mensagem usando pipes
euler:~/>
```

Um tubo é criado pelo processo pai, o qual logo após faz um `fork`. Quando um processo faz um `fork`, a tabela de descritores do pai é automaticamente copiada para o processo filho.

No programa, o pai faz um chamada de sistema `pipe()` para criar um tubo. Esta rotina cria um tubo e inclui na tabela de descritores do processos os descritores para os *sockets* associados às duas extremidades do tubo. Note que as extremidades do tubo não são equivalentes: o *sockets* com índice 0 está sendo aberto para leitura, enquanto que o de índice 1 está sendo aberto somente para a escrita. Isto corresponde ao fato de que a entrada padrão na tabela de descritores é associada ao primeiro descritor, enquanto a saída padrão é associada ao segundo.

Após ser criado, o tubo será compartilhado entre pai e filho após a chamada `fork`.

A tabela de descritores do processo pai aponta para ambas as extremidades do tubo. Após o `fork`, ambas as tabelas do pai e do filho estarão apontando para o mesmo tubo (herança de descritores). O filho então usa o tubo para enviar a mensagem para o pai.

4.5.4 Utilização dos tubos

É possível que um processo utilize um tubo tanto para a escrita quanto para a leitura de dados. Este tubo

não tem mais a função específica de fazer a comunicação entre processos, tornando-se muito mais uma implementação da estrutura de um arquivo. Isto permite, em certas máquinas, de ultrapassar o limite de tamanho da zona de dados. O mecanismo de comunicação por tubos apresenta um certo número de incovenientes como o não armazenamento da informação no sistema e a limitação da classe de processos podendo trocar informações via tubos.

4.6 As FIFOS ou tubos com nome

Uma FIFO combina as propriedades dos arquivos e dos tubos:

- como um arquivo, ela tem um nome e todo processo que tiver as autorizações apropriadas pode abri-lo em leitura ou escrita (mesmo se ele não tiver ligação de parentesco com o criador do tubo). Assim, um tubo com nome, se ele não estiver destruído, persistirá no sistema, mesmo após a terminação do processo que o criou.

- Uma vez aberto, uma FIFO se comporta muito mais como um tubo do que como um arquivo: os dados escritos são lidos na ordem "First In First Out", seu tamanho é limitado, e além disso, é impossível de se movimentar no interior do tubo.

4.6.1 Criação de um tubo com nome

Primitiva `mknod()`

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

int mknod(const char *pathname, mode_t mode, dev_t dev)
```

Valor de retorno: 0 se a criação do tubo tem sucesso, e -1 caso contrário.

A primitiva `mknod()` permite a criação de um nó (arquivo normal ou especial, ou ainda um tubo) cujo nome é apontado por `pathname`, especificado por `mode` e `dev`. O argumento `mode` especifica os direitos de acesso e o tipo de nó a ser criado. O argumento `dev` não é usado na criação de tubos com nome, devendo ter seu valor igual a 0 neste caso.

A criação de um tubo com nome é o único caso onde o usuário normal tem o direito de utilizar esta primitiva, reservada habitualmente ao super-usuário. Afim de que a chamada `mknod()` tenha sucesso, é indispensável que o *flag* `S_IFIFO` esteja "setado" e nos parâmetros de `mode` os direitos de acesso ao arquivo estejam indicados: isto vai indicar ao sistema que uma FIFO vai ser criada e ainda que o usuário pode utilizar `mknod()` mesmo sem ser *root*.

Dentro de um programa, um tubo com nome pode ser eliminado através da primitiva `unlink(const char *pathname)`, onde `pathname` indica o nome do tubo a ser destruído.

Exemplo:

```
              /* arquivo test_fifo.c */

/* este programa mostra a criacao e destruicao de tubos
 * com nome
 */

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

int main()
{
     printf("Vou criar um tubo de nome 'fifo1'\n") ;
     printf("Vou criar um tubo de nome 'fifo2'\n") ;
     if (mknod("fifo1",S_IFIFO | O_RDWR, 0) == -1) {
          perror("Criacao de fifo1 impossivel") ;
          exit(1) ;
     }
     if (mknod("fifo2",S_IFIFO | O_RDWR, 0) == -1) {
          perror("Criacao de fifo2 impossivel") ;
          exit(1) ;
     }
     sleep(10) ;
     printf("Vou apagar o tubo de nome 'fifo1'\n") ;
     unlink("fifo1") ;
     exit(0);
}
```

Resultado da execução:

O programa é lançado em *background* e pode-se verificar (através do comando shell `ls -l fifo*`) que os tubos denominados `fifo1` e `fifo2` foram criados e depois, que o tubo `fifo1` foi destruído.

```
euler~/> test_fifo &
[2] 812
Vou criar um tubo de nome 'fifo1'
Vou criar um tubo de nome 'fifo2'
euler:~/> ls -l fifo*
p---------   1 saibel   prof            0 Sep 27 10:07 fifo1|
p---------   1 saibel   prof            0 Sep 27 10:07 fifo2|
euler:~/> Vou apagar o tubo de nome 'fifo1'
```

```
[2]    Done                          test_fifo
euler:~/> ls -l fifo*
p---------   1 saibel   prof            0 Sep 27 10:07 fifo2|
```

Observações:

- Note que a presença do bit *p* indica que `fifo1` e `fifo2` são tubos (*pipes*) com nome;
- Pode-se notar ainda que o tubo denominado `fifo2` permanece no sistema, mesmo após a morte do processo que o criou.
- A eliminação de um tubo com nome pode ser feita a partir do shell, como no caso de um arquivo comum, usando-se o comando `rm`

4.6.2 Manipulação das FIFOs

As instruções `read()` e `write()` são bloqueantes:

- Na tentativa de leitura de uma FIFO vazia, o processo ficará em espera até que haja um preenchimento suficiente de dados dentro da FIFO;
- Na tentativa de escrita de uma FIFO cheia, o processo irá esperar que a FIFO seja sucessivamente esvaziada para começar a preenchê-la com seus dados.

Neste caso ainda, a utilização do *flag* `O_NDELAY` permite de manipular o problema de bloqueio, uma vez que nesse caso as funções `read()` e `write()` vão retornar um valor nulo.

Parte III

A parte III da apostila é constituída pelos capítulos 5 a 9, apresentando as extensões em `UNIX` desenvolvidas para a comunicação efetiva entre processos: os semáforos (capítulo 5), os segmentos de memória compartilhada (capítulo 6) e a troca demensagens (capítulo 7). Tais extensões são comumente conhecidas pela sigla IPC (*InterProcess Communication*).

A adição de facilidades de comunicação interprocessos foi um dos grandes avanços no sistema `UNIX` visando o desenvolvimento de aplicações concorrentes. A idéia básica é que esta interface IPC seja similar a um I/O em um arquivo qualquer do sistema.

O sistema `UNIX` possui uma série de descritores para manipular a leitura e escrita em arquivos, os quais aumentam consideravelmente a flexibilidade do usuário no tratamento de I/O. Por exemplo, um programa pode criar um descritor, enquanto outro diferente vai usá-lo (pense no caso de um comando shell `ps` que pode imprimir seu resultado num arquivo ao invés da tela, simplesmente através da associação adequada dos descritores). Os tubos são uma outra forma de descritor permitindo a transmissão de dados (em um único sentido) entre processos que tenham algum grau de parentesco.

Os descritores não são, entretanto, a única interface de comunicação em `UNIX`. O mecanismo de sinal também permite o envio de informações entre processos, entretando alguns problemas reduzem a sua flexibilidade na comunicação interprocessos: o processo que recebe apenas o tipo do sinal sem conhecer efetivamente o emissor desse sinal; mais ainda, o número de sinais a serem utilizados é bastante reduzido.

A utilização dos mecanismos IPC providos pelo `UNIX` tornou-se rapidamente um padrão para o desenvolvimento de aplicações multiprocessadas.Basicamente, os semáforos, os segmentos de memória compartilhada, e as filas de mensagens representam os três tipos de mecanismos avançados de comunicação interprocessos, reagrupados sob a denominação *System V IPC*.

Como será visto nos próximos capítulos, existem uma série de semelhanças tanto no que diz respeito às primitivas que implementam os mecanismos de comunicação, quanto nas informações mantidas pelo kernel do sistema para controlá-los.

O capítulo 8 apresentará, finalmente, uma série de exemplos clássicos implementados com a utilização dos conceitos apresentados anteriormente.

Capítulo 5

Os Semáforos

5.1 Introdução

Os semáforos são objetos IPC utilizados para a sincronização entre processos. Eles constituem também uma solução para resolver o problema de exclusão mútua, permitindo a solução de conflitos de acesso concorrentes de processos distintos a um mesmo recurso.

5.1.1 Comandos de *status* usando IPC

A maior parte dos sistemas `UNIX` fornecem ao usuário um conjunto de comandos que permitem o acesso às informações relacionadas aos três mecanismos implementados em IPC (semáforos, memória compartilhada e filas de mensagens). Os comandos `ipcs` e `ipcrm` são bastante úteis ao programador durante o desenvolvimento de aplicações.

O comando `ipcs -<recurso>` fornece informações atualizadas de cada um dos recursos IPC implementados no sistema. O tipo de recurso pode ser especificado da seguinte forma:

- `ipcs -m` informações relativas aos segmentos de memória compartilhada
- `ipcs -s` informações sobre os semáforos
- `ipcs -q` informações sobre as filas de mensagens
- `ipcs -a` todos os recursos (opção *par défaut* se nenhum parâmetro for especificado).

O formato de saída das informações pode ainda ser especificado.

O comando `ipcrm` permite que recursos IPC que tenham acidentalmente restado no sistema após a execução da aplicação possam ser destruídos via linha de comando. Esse comando exige um parâmetro especificando o tipo de recurso a ser destruído, assim como o identificador associado a esse recurso. Sua sintaxe é a seguinte: `ipcrm [sem|shm|msg] <id>`.

Suponha que o sistema produza a seguinte saída para o comando `ipcs`. A execução do comando `ipcrm msg 1152` irá destruir a fila com `id`=1152.

```
euler:~/> ipcs
```

```
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems      status

------ Message Queues --------
key        msqid      owner      perms      used-bytes  messages
0x7b045862 1152       saibel     600        0           0

euler:~/> ipcrm msg 1152
resource deleted
euler:~/> ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes  messages
```

5.2 Princípio

Inicialmente, o usuário deve associar um valor de uma chave ao semáforo criado. O sistema irá então retornar um identificador único de semáforo ao qual estão agrupados `n` semáforos (i.e., um grupo de semáforos) numerados de `0` a `(n-1)`. Para especificar um semáforo, o usuário deverá então indicar um identificador do grupo de semáforos e o número de semáforos a serem criados.

A cada semáforo é associado um valor, sempre positivo, que poderá ser incrementado ou decrementado pelo usuário segundo suas necessidades. Considere, por exemplo, `N` como o valor inicial, e `n` o valor do incremento determinado pelo usuário:

- Se `n>0` : o processo do usuário aumenta o valor do semáforo de `n` e continua sua execução;
- Se `n<0` :
  - se `N+n` é maior ou igual a 0 o processo do usuário diminui o valor do semáforo de `|n|` e continua sua execução;
  - se `N+n` é menor que 0 o processo do usuário se bloqueia, esperando até que `N+n` seja maior ou igual a 0;
- Se `n=0` :
  - se `N=0` o processo continua sua execução;
  - se `N` é negativo o processo do usuário se bloqueia, esperando até que `N`=0;

Como será visto, o bloqueio dos processos é "parametrizável", isto é, pode-se especificar que o processo não será bloqueado pelo sistema, sendo enviado apenas um código de erro ao processo que segue sua execução normalmente após o recebimento deste código.

Vale salientar que a cada identificador de semáforo é associada uma lista de permissões de acesso a este (de maneira similar ao caso de arquivos e tubos). Estas permissões são necessárias para controlar as operações sobre os valores dos semáforos. Elas são inoperantes para as duas manipulações a seguir:

- a destruição de um identificador de semáforo
- a modificação dos direitos (permissões) de acesso ao semáforo

Essas modificações só são permitidas para o super-usuário do sistema, para o próprio criador do semáforo, ou por seu proprietário. Note ainda que para o "bom funcionamento" dos mecanismos, as operações sobre os semáforos são consideradas indivisíveis (não podem ser interrompidas).

5.2.1 Estruturas associadas aos semáforos

Cada conjunto de semáforos do sistema é associado a diversas estruturas de dados, tais como `semid_ds`, `sembuf` e `sem`. Todas estas estrturas estão definidas em `<sys/sem.h>`) e `<bits/sem.h>`). A compreensão destas estrturas facilitará o entendimento de como o sistema trabalha ao receber as chamadas das primitivas `semget()`, `semctl()` e `semop()`.

Por exemplo, a estrutura `sembuf` é usada como argumento para as operações envolvendo a primitiva `semop()`, tem a seguinte forma:

```
struct sembuf
{
  short int sem_num;            /* número do semáforo */
  short int sem_op;             /* operação no semáforo */
  short int sem_flg;            /* flag da operação */
};
```

A estrutura de dados do conjunto de semáforos é a seguinte:

```
struct semid_ds
{
  struct ipc_perm sem_perm;             /* operation permission struct */
  __time_t sem_otime;                   /* last semop() time */
  __time_t sem_ctime;                   /* last time changed by semctl() */
  struct sem *__sembase;                /* ptr to first semaphore in array */
  struct sem_queue *__sem_pending;      /* pending operations */
  struct sem_queue *__sem_pending_last; /* last pending operation */
  struct sem_undo *__undo;              /* ondo requests on this array */
  unsigned short int sem_nsems;         /* number of semaphores in set */
};
```

A estrutura de dados usada para passar as informações sobre as permissões de acesso para as operações `IPC` é mostrada a seguir:

```
struct ipc_perm
  {
    __key_t __key;                      /* Key.  */
    unsigned short int uid;             /* Owner's user ID.  */
    unsigned short int gid;             /* Owner's group ID.  */
    unsigned short int cuid;            /* Creator's user ID.  */
    unsigned short int cgid;            /* Creator's group ID.  */
    unsigned short int mode;            /* Read/write permission.  */
    unsigned short int __seq;           /* Sequence number.  */
  };
```

5.3 A Função semget()

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/sem.h>

int semget ( key_t key, int nsems, int semflg )
```

Valor de retorno: o identificador (ou ID) do conjunto de semáforos `semid`, e -1 em caso de erro.

A função `semget()` é utilizada para criar um novo conjunto de semáforos, ou para obter o ID de um conjunto de semáforos já existentes. O primeiro argumento, `key`, é uma chave indicando o nome (valor numérico) de um conjunto de semáforos. O segundo argumento, `nsems`, indica o número de semáforos do conjunto. O último argumento, `semflg`, é um *flag* especificando os direitos de acesso ao semáforo.

Um novo conjunto de semáforos será criado se `key` tiver valor `IPC_PRIVATE` (=0), ou se ele não for `IPC_PRIVATE`, anenhum conjunto de semáforos existente será associado a `key`, e `IPC_CREAT` é colocado em `semflg`.

A presença dos campos `IPC_CREAT` e `IPC_EXCL` em `semflg` têm a mesma função em se tratando da existência ou não do conjunto de semáforos, que no caso das presenças de `IPC_CREAT` e `IPC_EXCL` associadas a primitiva `open()` apresentada no capítulo 1. Em outras palavras, a função `semget` falha se `semflg` contiver os flags `IPC_CREAT` e `IPC_EXCL` e o conjunto de semáforos já existir para uma outra chave.

O argumento `semflg` é definido como a combinação de diferentes constantes pré-definidas, permitindo de especificar os direitos de acesso, e os comandos de controle (usando a combinação clássica por intermédio dos operadores OU). Note que existe uma grande semelhança entre os direitos de acesso para a utilização dos semáforos e aqueles associados aos arquivos em `UNIX`: as mesmas noções de autorização de acesso para outros ou para o grupo são definidas da mesma forma (ver capítulo 1, seção XXXX).

As constantes utilizadas com `semflg` pré-definidas em `<sys/sem.h>` e em `<sys/ipc.h>` são as seguintes:

```
/* Mode bits for `msgget', `semget', and `shmget'.  */
#define IPC_CREAT       01000           /* Create key if key does not exist. */
#define IPC_EXCL        02000           /* Fail if key exists.  */
#define IPC_NOWAIT      04000           /* Return error on wait.  */
```

```
/* Control commands for ‘msgctl’, ‘semctl’, and ‘shmctl’.  */
#define IPC_RMID        0               /* Remove identifier.  */
#define IPC_SET         1               /* Set ‘ipc_perm’ options.  */
#define IPC_STAT        2               /* Get ‘ipc_perm’ options.  */
#define IPC_INFO        3               /* See ipcs.  */
```

5.3.1 Como criar um conjunto de semáforos

Para criar um conjunto de semáforos, deve-se obedecer os seguintes passos:

1. `key` deve contar um valor identificando um conjunto de semáforos (diferente de `IPC_PRIVATE`=0);
2. `semflg` deve conter os direitos de acesso desejados para o semáforo, e a constante `IPC_CREATE`;
3. Se deseja-se testar a existência de um conjunto correspondente a uma chave `key` determinada, deve-se adicionar (usando o operador OU) ao argumento `semflg` a constante `IPC_EXCL`. A chamada à função `semget()` irá falhar no caso da existência de um conjunto associado à chave.

Note ainda que durante a criação de um conjunto de semáforos, um certo número de campos da estrutura `semid_ds` são iniciados (proprietário, modo de acesso, etc). Faça `man semget` para obter a lista completa dos parâmetros modificados com a chamada `semget`. No caso do exemplo, o valor `0600` garante ao usuário as permissões de escrita e leitura do array de semáforos.

Exemplo

O programa `test_semget.c` cria um conjunto de 4 semáforos associados à chave 123.

```
            /* arquivo test_semget.c */

/* exemplo de uso de semget() */

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>

#define KEY 123

int main()
{
  int semid ;  /* identificador dos semaforos */
  char *path = "nome_de_arquivo_existente" ;

    /* alocacao de 4 semaforos */
```

```
    if (( semid = semget(ftok(path,(key_t)KEY), 4,
                      IPC_CREAT|IPC_EXCL|0600)) == -1) {
         perror("Erro de semget") ;
         exit(1) ;
    }
  printf(" O semid do conjunto do semaforo e : %d\n",semid) ;
  printf(" Este conjunto e identificado pela chave unica : %d\n"
                           ,(int) ftok(path,(key_t)KEY)) ;
  exit(0);
}
```

Resultado da execução:

```
euler:~/> test_semget
 O semid do conjunto do semaforo e : 1920
 Este conjunto e identificado pela chave unica : 2063902908

euler:~/> test_semget
Erro de semget: File exists

euler:~/> ipcs -s

------ Semaphore Arrays --------
key       semid     owner     perms     nsems     status
0x7b04a8bc 1920      saibel    600       4
```

5.4 A Função `semctl()`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#if defined(__GNU_LIBRARY__) && !defined(_SEM_SEMUN_UNDEFINED)
/* union semun is defined by including <sys/sem.h> */
#else
/* according to X/OPEN we have to define it ourselves */
union semun {
      int val;                    /* value for SETVAL */
      struct semid_ds *buf;       /* buffer for IPC_STAT, IPC_SET */
      unsigned short int *array;  /* array for GETALL, SETALL */
      struct seminfo *__buf;      /* buffer for IPC_INFO */
};
#endif
```

```
int  semctl  (int  semid, int semnum, int cmd, union semun arg)

/* Commands for 'semctl'.  */
#define GETPID               11             /* get sempid */
#define GETVAL               12             /* get semval */
#define GETALL               13             /* get all semval's */
#define GETNCNT              14             /* get semncnt */
#define GETZCNT              15             /* get semzcnt */
#define SETVAL               16             /* set semval */
#define SETALL               17             /* set all semval's */
```

Valor de retorno: Depende do valor do argumento `cmd`:

- Se `cmd = GETPID` : valor de `sempid`
- Se `cmd = GETVAL` : valor de `semval`
- Se `cmd = GETALL` : valor de `all`
- Se `cmd = GETNCNT` : valor de `semncnt`
- Se `cmd = GETZCNT` : valor de `semzcnt`

Para todos os outros valores de `cmd`, o valor de retorno é 0 em caso de sucesso, e -1 em caso de erro.

A função `semctl` é utilizada para examinar e mudar (controlar) os valores de cada um dos componentes de um conjunto de semáforos. Ela executa as ações de controle definidas em `cmd` no conjunto de semáforos (ou no `semnun`-ésimo semáforo do conjunto) identificado por `semid`. O primeiro semáforo do grupo é identificado pelo valor `0` e o último por `semnum-1`. O último argumento `arg` é uma variável do tipo `union semun`.

Observação: Em alguns casos o usuário deve definir a `union` dentro do seu arquivo fonte se este não for definido dentro de `<sys/sem.h>`). Pode se testar a macro `_SEM_SEMUN_UNDEFINED` para se verificar se a `union` está ou não definida.

Os possíveis comandos para um semáforo

Os diferentes comandos possíveis para `semctl()` podem ser encontrados fazendo-se o comando shell `man semctl()`. Neste caso, as seguintes informações são disponibilizadas:

- IPC_RMID : O conjunto de semáforos identificado por `semid` é destruído. Somente o super-usuário ou um processo tendo o número do usuário `sem_permid` pode destruir um conjunto. Todos os processos em espera sobre os semáforos destruídos são automaticamente desbloqueados, recebendo ao mesmo tempo um código de erro.
- IPC_SET : dá ao identificador do grupo, ao identificador do usuário e aos direitos de acesso ao semáforo os valores contidos no campo `sem_perm` da estrutura apontada por `arg.buf`. A hora da modificação também é atualizada no membro `sem_ctime` da estrutura.

- IPC_STAT : a estrutura associada a `semid` é copiada para o endereço apontado por `arg.buf`
- GETNCNT : a chamada retorna o valor de `semncnt` para o `semun`-ésimo semáforo do conjunto (i.e. o número de processos em espera de um incremento do valor do `semun`-ésimo semáforo).
- GETPID : a chamada retorna o valor de `sempid` para o `semun`-ésimo semáforo do conjunto (i.e. o `pid` do processo que executou a última operação `semop` sobre o `semun`-ésimo semáforo).
- GETVAL : a chamada retorna o valor de `semval` para o `semun`-ésimo semáforo do conjunto. O processo que chama a primitiva deve ter permissão de leitura para a operação.
- GETALL : a chamada retorna `semval` para o todos os semáforos do conjunto em `arg.array`.
- GETZCNT : a chamada retorna o valor `semzcnt` para o `semun`-ésimo semáforo do conjunto (i.e. o número de processos esperando que o valor `semval` do `semun`-ésimo semáforo torne-se 0).O processo que chama a primitiva deve ter ter permissão de leitura para a operação. operação.
- SETVAL : seta `semval` do `semun`-ésimo semáforo do conjunto para o valor de `arg.val`, atualizando também o membro `sem_ctime` da estrutura. Esta ação corresponde, na verdade, à inicialização do valor do semáforo.
- SETALL : seta `semval` de todos os semáforos identificados em `arg.array`, atualizando também o membro `sem_ctime` da estrutura. Esta ação corresponde, na verdade, à inicialização do valor do semáforo.

Exemplo:

```
                /* arquivo test_semctl.c */
/* exemplo de uso de semctl() */

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

#define KEY 123

union semun {
              int val ;
              struct semid_ds buf[2] ;
              unsigned short int array[4] ;
              struct seminfo *__buf;
};

int main()
```

```
{
  struct sembuf sempar;
  int semid, semval , sempid;
  union semun arg;
  char *path = "nome_de_arquivo_existente" ;
    /*
     * recuperacao do identificador do
     * do conjunto de semaforos do projeto 123
     */
     if (( semid = semget(ftok(path,(key_t)KEY),0,0)) == -1 ) {
          perror ("Error semget()") ;
          exit(1) ;
      }
      printf("O conjunto de semaforos tem semid : %d\n",semid) ;
      printf("A chave de acesso unica e : %d\n",ftok(path,(key_t)KEY)) ;

    /*
     * leitura do 3o. semaforo
     */
      if ( (semval = semctl(semid,2,GETVAL,arg)) == -1){
          perror("Error semctl() GETVAL") ;
          exit(1) ;
     }
     else {
       printf("O valor do terceiro semaforo e : %d\n",semval) ;
     }
    /*
     * atualizacao do 3o. semaforo
     */
     sempar.sem_num = 2 ;
     sempar.sem_op = 1 ;
     sempar.sem_flg = SEM_UNDO ;
     if (semop(semid, &sempar, 1) == -1) {
          perror("Error semop()") ;
          exit(-1);
     }
    /*
     * leitura do 3o. semaforo
     */
      if ( (semval = semctl(semid,2,GETVAL,arg)) == -1){
          perror("Error semctl() GETVAL") ;
          exit(1) ;
     }
     else printf("O valor do terceiro semaforo e : %d\n",semval) ;
```

```
    /*
     * leitura do pid do processo que executou a ultima operacao
     */
      if (( sempid = semctl(semid,2,GETPID,arg) )== -1){
          perror("Error semctl()") ;
          exit(1) ;
     }
     else
       {
         printf("O valor do pid do processo que\n");
         printf("\t realizou o ultimo semop no semaforo e : %d\n",sempid);
         printf("\t Meu pid e : %d\n",getpid()) ;
       }

    /*
     * destruicao do semaforo
     */
     if (semctl(semid,0,IPC_RMID,0)==-1){
             perror("Impossivel de destruir o semaforo") ;
             exit(1) ;
     }
     else printf("O semaforo com semid %d foi destruido\n",semid) ;

     exit(0);
}
```

Resultado da execução:

Após o programa `test_semget` ser executado, uma chave com ID igual a 1024 foi criada no sistema. O programa `test_semctl` recupera essa chave dos semáforos e realiza uma operação sobre o terceiro semáforo.

```
euler:~/> test_semget
 O semid do conjunto do semaforo e : test_semget
 O semid do conjunto do semaforo e : 1024
 Este conjunto e identificado pela chave unica : 2063837372
euler:~/> ipcs -s

------ Semaphore Arrays --------
key       semid     owner     perms     nsems     status
0x7b03a8bc 1024      saibel    600       4

euler:~/> test_semctl
```

```
O conjunto de semaforos tem semid : 1024
A chave de acesso unica e : 2063837372
O valor do terceiro semaforo e : 0
O valor do terceiro semaforo e : 1
O valor do pid do processo que
         realizou a ultima operacao no semaforo e : 1067
         Meu pid e : 1067
O semaforo com semid 1024 foi destruido
euler:~/> ipcs -s

------ Semaphore Arrays --------
key       semid     owner     perms     nsems     status
```

5.5 A Função semop()

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/sem.h>

int semop ( int semid, struct sembuf *sops, unsigned nsops )
```

Valor de retorno: o valor de `semval` do último semáforo manipulado ou -1 em caso de erro.

A função `semop()` permite de efetuar operações sobre os semáforos identificado por `semid`. Cada um dos (nsops) elementos no array de estruturas do tipo `struct sembuf` apontado por sops especifica uma operação a ser realizada no semáforo.

A estrutura `sembuf` (ver seção 5.2.1) especifica o número do semáforo que será tratado (`sem_num`), a operação que será realizada sobre este semáforo(`sem_op`), e os flags de controle da operação (`sem_flag`).

O tipo de operação realizada vai depender do valor armazenado no membro `sem_op`:

- Se `sem_op < 0` (pedido de recurso) então:
  - Se `semval` é maior ou igual a —`sem_op`—, então `semval` = `semval` - —`sem_op`—;
  - Se `semval` é menor que `sem_op`, então o processo se bloqueia até que `semval` seja maior ou igual a —`sem_op`—;
- Se `sem_op = 0`, o processo deve ler as permissões de acesso do conjunto de semáforos.
  - Se `semval` é menor ou igual a 0, a chamada retorna;
  - Se `semval` é maior que 0, o processo se bloqueia até que `semval`=0
- Se `sem_op > 0` (restituição de recurso): então `semval` = `semval` + —`sem_op`—

Quando as operações `semop()` no semáforo utilizarem apenas os valores de `sem_op` iguais a 1 ou -1, tem-se o funcionamento dos semáforos conhecidos como *semáforos de Dijsktra* (veja seção 5.6).

Existem entretanto uma gama de operações possíveis utilizando `sem_op`. A implementação torna-se nesse caso muito mais complexa e a demonstração das garantias de exclusão mútua entre as operações é extremamente delicada.

Os flags reconhecidos em `sem_flg` são `IPC_NOWAIT` e `SEM_UNDO`. Se a operação coloca `SEM_UNDO` como flag, ele será desfeita quando o processo termina (`exit()`). Em resumo:

- `IPC_NOWAIT` : evita o bloqueio do processo, retornando um código de erro.
- `SEM_UNDO` : todas as modificações feitas sobre os semáforos são desfeitasquando o processo morre. Durante toda a vida do processo, as operações efetuadas com o flag `SEM_UNDO` sobre todos os semáforos de todos os identificadores são acumulados, e na morte do processo, o sistema "refaz"todas essa operações ao inverso. Isto possibilita não bloquear indefinidamente os processos sobre os semáforos, após a morte acidental de um processo. Note que este tipo de procedimento é custoso, tanto do ponto de vista de carga para a CPU, quanto do ponto de vista da memória gasta.

Exemplo:
Um processo (executando o programa `processo1`) cria um conjunto de semáforos, fixa o valor de um dos semáforos do conjunto em 1, depois demanda um recurso. Ele se coloca em espera por 10 segundos. Um segundo processo (executando o programa `processo2`) recupera o identifcador `semid` do conjunto de semáforos, depois também demanda um recurso. Ele fica bloqueado até que o primeiro processo acabe sua espera e libere o recurso.

`Resultado da execução`
Os dois processos são lançados em *background* e o resultado deve ser o seguinte:

```
euler:~/> processo1 &
[2] 967
euler:~/>
processo1: acabo de criar um conjunto de semaforos : 768
processo1: vou demandar um recurso
processo1: Esperando 10 sec
euler:~/> processo2 &
[3] 968
euler:~/>
processo2: trata os semaforos : semid 768
 demanda do processo2 : semaforo nao disponivel
 demanda do processo2 : semaforo nao disponivel
 demanda do processo2 : semaforo nao disponivel
 demanda do processo2 : semaforo nao disponivel
 demanda do processo2 : semaforo nao disponivel
 demanda do processo2 : semaforo nao disponivel
 demanda do processo2 : semaforo nao disponivel
 demanda do processo2 : semaforo nao disponivel
processo1: Acabei minha espera: liberando o recurso
```

```
morte de processo1

[2]    Done                          processo1
semaforo alocado ao processo2

[3]    Done                          processo2
euler:~/>
```

5.6 Semáforos de Dijsktra

Os semáforos de Dijkstra são uma solução simples para o problema da exclusão mútua. Duas operações básicas podem ser feitas sobre estes semáforos: `P` (aquisição) e `V` (liberação).

Quando a operação `P` é realizada sobre um semáforo, seu valor é decrementado de 1 se ele é diferente de 0; senão for o caso, o processo tentando executar`P` será bloqueado e colocado numa fila de espera associada ao semáforo.

Quando a operação Q é realizada sobre um semáforo, seu valor é incrementado se seu valor é incrementado de 1 se não existe processo algum na fila de espera; senão for o caso, o valor do semáforo não é modificado, e o primeiro processo da fila é liberado.

5.6.1 Implementação dos semáforos de Dijkstra

O programa `dijsktra.h` realiza a implementação dos semáforos de Dijkstra a partir dos mecanismos de semáforos `IPC` disponíveis no sistema. A função `sem_create()` possibilita criação de um semáforo. As operações P e V são realizadas pelas funções `P()` e `V()`. A função `sem_delete()` possibilita a destruição do semáforo.

```
                /* arquivo test_sem_dijkstra.c */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <unistd.h>
#include "dijkstra.h"

#define KEY 456

int main()
{
  int sem ;
  sem = sem_create(KEY,1) ;
  printf("\nCriacao do semaforo do identificador %d\n",sem) ;
  // P(sem);
 if (fork() == 0) {
```

```
      printf("\tEu sou o FILHO e fazer P sobre o semaforo\n");
      P(sem);
      printf("\tEu sou o FILHO e vou dormir 10 segundos...\n") ;
      sleep(10) ;
      printf("\tEu sou o FILHO e vou fazer V sobre o semaforo\n") ;
      V(sem) ;
       sleep(1);
   }
   else {
      printf("Eu sou o PAI e vou dormir 2 segundos...\n") ;
      sleep(2);
      printf("Eu sou o PAI e vou me bloquear fazendo P sobre o semaforo\n");
      P(sem) ;
      printf("Eu sou o PAI e acabei de me desbloquear\n") ;
      sem_delete(sem) ;
      printf("Eu sou o PAI e vou acabar o processamento\n\n");
   }
   exit(0);
}
```

Exemplo de utilização dos semáforos de Dijkstra

Inicialmente, o semáforo (modelando um recurso compartilhado) é criado com valor 1. O processo filho garante a posse do recurso fazendo a operação `P` sobre o semáforo, levando seu valor para 0. O processo pai é bloqueado sobre um semáforo (fazendo `P` sobre um semáforo com valor nulo). Ele se desbloqueará quando ser filho liberar o recurso demandado por ele, fazendo a operação `V` que colocará novamente o valor do semáforo em 1.

Resultado da execução:

```
euler:~> test_sem_dijkstra

Criacao do semaforo do identificador 128
Eu sou o PAI e vou dormir 2 segundos...
        Eu sou o FILHO e fazer P sobre o semaforo
        Eu sou o FILHO e vou dormir 10 segundos...
Eu sou o PAI e vou me bloquear fazendo P sobre o semaforo
        Eu sou o FILHO e vou fazer V sobre o semaforo
Eu sou o PAI e acabei de me desbloquear
Eu sou o PAI e vou acabar o processamento
```

5.7 Conclusão

O mecanismo dos semáforos é um pouco complexo de ser implementado como mostraram os exemplos. Por outro lado, a aplicação deste mecanismo é fundamental em situações onde o acesso a recursos compartilhados deve ser feito de maneira exclusiva entre processo. Neste caso, deve ser possível demonstrar que este acesso exclusivo aos recursos é garantido, que não existe inter-bloqueio entre processos concorrendo pelos recursos e que esta concorrência é justa (ou seja, deve ser garantido que todos os processos têm a mesma chance de adquirir o recurso compratilhado). Se esta análise parece complicada usando os semáforos de Dijkstra, ele será muito mais delicada em aplicações concorrentes gerais usando todo o espectro primitivas IPC. A utilização de `abordagens formais` para a concepção e análise deste tipo de aplicação torna-se desta forma, fundamental.

Capítulo 6

Memória Compartilhada

6.1 Introdução

O compartilhamento de uma região de memória entre dois ou mais processos (executando programas) corresponde a maneira mais rápida deles efetuarem uma troca de dados. A zona de memória compartilhada (denominada segmento de memória compartilhada) é utilizada por cada um dos processos como se ela fosse um espaço de endereçamento que pertencesse a cada um dos programas. Em outras palavras, o compartilhamento de memória permite aos processos de trabalhar sob um espaço de endereçamento comum em memória virtual. Em conseqüência, este mecanismo é dependente da forma de gerenciamento da memória; isto significa que as funcionalidades deste tipo de comunicação interprocessos são fortemente ligadas ao tipo de arquitetura (máquina) sobre a qual a implementação é realizada.

6.2 Princípio da memória compartilhada

Um processo pode criar um segmento de memória compartilhada e suas estruturas de controle através da função `shmget()`. Durante essa criação, os processos devem definir: as permissões de acesso ao segmento de memória compartilhada; o tamanho de bytes do segmento e; a possibilidade de especificar que a forma de acesso de um processo ao segmento será apenas em modo leitura. Para poder ler e escrever nessa zona de memória, é necessário estar de posse do identificador (ID) de memória comum, chamado `shmid`. Este identificador é fornecido pelo sistema (durante a chamada da função `shmget()`) para todo processo que fornece a chave associada ao segmento. Após a criação de um segmento de memória compartilhada, duas operações poderão ser executadas por um processo:

- acoplamento (*attachment*) ao segmento de memória compartilhada, através da função `shmat()`;
- desacoplamento da memória compartilhada, utilizando a função `shmdt()`.

O acoplamento à memória compartilhada permite ao processo de se associar ao segmento de memória: ele recupera, executando `shmat()`, um ponteiro apontando para o início da zona de memória que ele pode utilizar, assim como todos os outros ponteiros para leitura e escrita no segmento.

O desacoplamento da memória compartilhada permite ao processo de se desassociar de um segmento quando ele não desejar mais utilizá-lo. Após esta operação, o processo perde a possibilidade de ler ou escrever neste segmento de memória compartilhada.

6.3 A Função `shmget()`

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflg);
```

Valor de retorno: o identificador do segmento de memória compartilhada `shmid`, ou -1 em caso de erro.

Esta função é encarregada de buscar o elemento especificado (pela chave de acesso `key`) na estrutura `shmid_ds` e, caso esse elemento não exista, de criar um novo segmento de memória compartilhada, com tamanho em bytes igual a `size`. Além da chave de acesso `key` e do tamanho do segmento (`size`), um terceiro argumento (`shmflg`) é empregado para definir os direitos de acesso ao segmento criado.

O argumento `key` pode conter os seguintes valores:

- IPC_PRIVATE (=0): indicando que a zona de memória não tem chave de acesso, e que somente o processo proprietário tem acesso ao segmento de memória compartilhada.
- o valor desejado para a chave de acesso do segmento de memória. Observe que para a geração de uma chave única no sistema deve-se utilizar a função `ftok` apresentada na seção 1.6

O argumento `shmflg` é bastante semelhante ao `semflg` utilizado para semáforos (ver seção 5.3. Este flag corresponde à combinação de diferentes constantes pré-definidas através do operador lógico OU). O argumento `shmflg` permite assim a especificação dos direitos de acesso ao segmento de memória compartilhada criado. As possíveis constantes a serem combinadas são: `IPC_CREAT`, `IPC_EXCL` similares àquelas dos semáforos e, `SHM_R` (=0400) e `SHM_W`(=200) que dão o direito de leitura e escrita ao segmento. Note que a combinação destas útimas constantes pode ser igualmente representada pelo octal 0600.

Existe muita semelhança entre os direitos de acesso aos segmentos criados e aos arquivos no sistema `UNIX` através da noção de direitos de leitura e escrita para o usuário, para o grupo e para outros. O número octal definido de maneira similar àquela mostrada em 1.4.2 pode ser utilizado.

6.3.1 Estrutura associada a uma memória comum: `shmid_ds`

Quando um novo segmento de memória é criado, as permissões de acesso definidas pelo parâmetro `shmflg` são copiadas no membro `shm_perm` da estrutura `shmid_ds` que define efetivamente o segmento. A estrutura `shmid_ds` é mostrada a seguir:

```
struct shmid_ds
     {
       struct    ipc_perm shm_perm;  /* operation permissions */
       int  shm_segsz;               /* size of segment (bytes) */
       time_t    shm_atime;          /* last attach time */
       time_t    shm_dtime;          /* last detach time */
       time_t    shm_ctime;          /* last change time */
       unsigned short shm_cpid;      /* pid of creator */
```

```
    unsigned short shm_lpid;      /* pid of last operator */
    short     shm_nattch;         /* no. of current attaches */
  };
```

Os campos no membro `shm_perm` são os seguintes:

```
struct ipc_perm
  {
    key_t  key;
    ushort uid;   /* owner euid and egid */
    ushort gid;
    ushort cuid;  /* creator euid and egid */
    ushort cgid;
    ushort mode;  /* lower 9 bits of shmflg */
    ushort seq;   /* sequence number */
  };
```

6.3.2 Como criar um segmento de memória compartilhada

O procedimento é exatamente o mesmo que aquele empregaado para gerar um conjunto de semáforos (seção 5.3). As seguintes regras gerais devem ser entretanto observadas:

- `key` deve contar um valor identificando o segmento (diferente de `IPC_PRIVATE`= 0);
- `shmflg` deve conter os direitos de acesso desejadas para o segmento, e ainda a constante `IPC_CREAT`.
- Se deseja-se testar a existência ou não de um segmento associado a uma chave especificada, deve-se adicionar (OU lógico) a constante `IPC_EXCL` ao argumento `shmflg`. A chamada `shmget` irá falhar se esse segmento existir.

Note finalmente que durante a criação do segmento de memória compartilhada, um certo número de membros da estrutura `shmid_ds` serão também inicializados (por exemplo, o proprietário, os modos de acesso, a data de criação, etc). Faça `man shmget` para maiores detalhes.

Exemplo de utilização de `shmget`

Este programa cria um segmento de memória compartilhada associado à chave 123.

```
/* fichier test_shmget.c */
/* exemplo de utilizacap de shmget() */

#include <errno.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
```

```
#define KEY 123

int main()
{
  int shmid ;  /* identificador da memoria comum */
  int size = 1024 ;
  char *path="nome_de_arquivo_existente" ;

     if (( shmid = shmget(ftok(path,(key_t)KEY), size,
                   IPC_CREAT|IPC_EXCL|SHM_R|SHM_W)) == -1) {
          perror("Erro no shmget") ;
          exit(1) ;
     }
     printf("Identificador do segmento: %d \n",shmid) ;
     printf("Este segmento e associado a chave unica: %d\n",
                  ftok(path,(key_t)KEY)) ;
     exit(0);
}
```

Resultado da execução:

Lançando duas vezes a execução do programa, tem-se o seguinte resultado:

```
euler:~> test_shmget
Identificador do segmento: 36096
Este segmento e associado a chave unica: 2063804629
euler:~> ipcs -m

------ Shared Memory Segments --------
key       shmid     owner     perms     bytes     nattch    status
0x7b0328d5 36096     saibel    600       1024      0

euler:~> test_shmget
Erro no shmget: File exists
```

6.4 A Função `shmctl()`

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Valor de retorno: 0 em caso de sucesso, senão -1.

A função `semctl()` é utilizada para examinar e modificar as informações relativas ao segmento de memória compartilhada. De maneira intuitiva, ela permite ao usuário de receber informações relativas

ao segmento, de setar o proprietário ou grupo, de especificar as permissões de acesso, e finalmente, de destruir o segmento.

A função utiliza três argumentos: um identificador do segmento de memória compartilhada `shmid`, um parâmetro de comando `cmd` e um ponteiro para uma estrutura do tipo `shmid_ds` associada pelo sistema ao segmento de memória onde a operação será executada.

O argumento `cmd` pode conter os seguintes valores:

- `IPC_RMID (0)`: O segmento de memória será destruído. O usuário deve ser o proprietário, o criador, ou o super-usuário para realizar esta operação; todas as outras operações em curso sobre esse segmento irão falhar;
- `IPC_SET (1)`: dá ao identificador do grupo, ao identificador do usuário, e aos direitos de acesso, os valores contidos no campo `shm_perm` da estrutura apontada por `buf`; a hora da modificação é também atualizada (membro `shm_ctime`);
- `IPC_STAT (2)`: é usada para copiar a informação sobre a memória compartilhada no buffer `buf`;

O super usuário pode ainda evitar ou permitir o *swap* do segmento de memória compartilhada usando os valores `SHM_LOCK (3)` (evitar o *swap* e `SHM_UNLOCK (4)` (permitir o *swap*).

Exemplo:

Neste exemplo, supõe-se que o segmento de memória compartilhada tem a chave de acesso 123 utilizada no exemplo anterior:

```
                /* arquivo test_shmctl.c */

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

#define KEY 123

struct shmid_ds buf ;

int main()
{
  char *path = "nome_de_arquivo_existente" ;
  int shmid ;
  int size = 1024 ;

  /* recuperacao do identificador do segmento associado a chave 123 */
     if (( shmid = shmget(ftok(path,(key_t)KEY),size,0)) == -1 ) {
          perror ("Erro shmget()") ;
```

```
          exit(1) ;
     }
      /* recuperacao das informacoes realtivas ao segmento */
     if ( shmctl(shmid,IPC_STAT,&buf) == -1){
          perror("Erro shmctl()") ;
          exit(1) ;
     }
     printf("ESTADO DO SEGMENTO DE MEMORIA COMPARTILHADA %d\n",shmid) ;
     printf("ID do usuario proprietario: %d\n",buf.shm_perm.uid) ;
     printf("ID do grupo do proprietario: %d\n",buf.shm_perm.gid) ;
     printf("ID do usuario criador: %d\n",buf.shm_perm.cuid) ;
     printf("ID do grupo criador: %d\n",buf.shm_perm.cgid) ;
     printf("Modo de acesso: %d\n",buf.shm_perm.mode) ;
     printf("Tamanho da zona de memoria: %d\n",buf.shm_segsz) ;
     printf("pid do criador: %d\n",buf.shm_cpid) ;
     printf("pid  (ultima operacao): %d\n",buf.shm_lpid) ;

     /* destruicao do segmento */
     if ((shmctl(shmid, IPC_RMID, NULL)) == -1){
          perror("Erro shmctl()") ;
          exit(1) ;
     }
     exit(0);
}
```

Resultado da execução

```
euler:~/> test_shmctl
ESTADO DO SEGMENTO DE MEMORIA COMPARTILHADA 35968
ID do usuario proprietario: 1145
ID do grupo do proprietario: 1000
ID do usuario criador: 1145
ID do grupo criador: 1000
Modo de acesso: 384
Tamanho da zona de memoria: 1024
pid do criador: 930
pid  (ultima operacao): 0
euler:~> ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
```

6.5 Função `shmat()`

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

void *shmat ( int shmid, const void *shmaddr, int shmflg )
```

Valor de retorno: endereço do segmento de memória compartilhada, ou -1 em caso de erro.

Antes que o processo possa utilizar um segmento de memória criado por outro processo, ele deve inicialmente se acoplar a esse segmento. É exatamente a função `shmat()` que faz esse papel. Ela "acopla" *(*attaches) o segmento de memória compartilhada identificado por `shmid` ao segmento de dados do processo que a chamou. A função exige três argumentos: o identificador do segmento `shmid`, um ponteiro `shmaddr` especificando o endereço de acoplamento e um conjunto de flags, `shmflg`.

O endereço de acoplamento é especificado através dos dois últimos parâmetros `shmaddr` e `shmflg`:

- Se `shmaddr` é 0, o segmento é acoplado ao primeiro endereço possível determinado pelo sistema (caso mais comum na prática);
- Se `shmaddr` não é 0, observa-se então `shmflg`:
  - Se o flag `SHM_RND` não está posicionado em `shmflg`, o acoplamento ocorre no endereço especificado por `shmaddr`;
  - Se `SHM_RND` está posicionado em `shmflg`, o acoplamento ocorre no endereço especificado pelo menor inteiro resultante da divisão de `shmaddr` por `SHMLBA`, uma constante pré-definida do sistema em `<sys/shm.h>` associada ao tamanho da página na memória física.

Observações:

Quando a função `shmat` é chamada, o sistema verifica se existe espaço suficiente no espaço de endereçamento da memória virtual do processo ao qual deve ser acoplado o segmento de memória compartilhada. Se esta não for o caso, um código de erro será retornado. Note ainda que não existe efetivamente uma cópia da zona de memória, mais simplesmente um redirecionamento do endereçamento para o segmento de memória que está sendo compartilhado.

Exemplo:

Suponha que um segmento de memória compartilhada tenha sido criado anteriormente através do programa `test_shmget`. O programa `test_shmat` vai reacoplar um processo ao segmento e escrever na memória comum, uma cadeia de caracteres. O programa `test_shmat2` irá então se acoplar à mesma zona de memória e ler então seu conteúdo. O programa `test_shmctl` irá então obter informações sobre o segmento de memória antes de destruí-lo.

```
                /* arquivo test_shmat.c */

/*
```

```
 * exemplo de utilizacao de shmat()
 * escrita num segmento de memoria compartilhada
 */

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

#define KEY 123
#define KEY 123
#define MSG "Mensagem escrita na memoria comum"

int main()
{
  int shmid ;  /* identificador da memoria comum */
  int size = 1024 ;
  char *path="nome_de_arquivo_existente" ;
  char *mem ;
  int flag = 0;

     /*
     * recuperacao do shmid
     */
     if (( shmid = shmget(ftok(path,(key_t)KEY), size,0)) == -1) {
          perror("Erro no shmget") ;
          exit(1) ;
     }
     printf("Sou o processo com pid: %d \n",getpid()) ;
     printf("Identificador do segmento recuperado: %d \n",shmid) ;
     printf("Este segmento e associado a chave unica: %d\n",
                   ftok(path,(key_t)KEY)) ;
    /*
     * acoplamento do processo a zona de memoria
     * recuperacao do pornteiro sobre a area de memoria comum
     */
     if ((mem = shmat (shmid, 0, flag)) == (char*)-1){
          perror("acoplamento impossivel") ;
          exit (1) ;
     }

    /*
     * escrita na zona de memoria compartilhada
```

```
     */
     strcpy(mem,MSG);
     exit(0);
}

                /* fichier test_shmat2.c */

/*
 * programa para ler o conteudo de um segmento de memoria
 * compartilhada que foi preenchido anteriormente por outro processo
 */

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

#define KEY 123

int main()
{
  int shmid ;  /* identificateur de la memoire commune */
  int size = 1000 ;
  char *path="nome_de_arquivo_existente" ;
  char *mem ;
  int flag = 0 ;

      /*
     * recuperacao do shmid
     */
     if (( shmid = shmget(ftok(path,(key_t)KEY), size,0)) == -1) {
          perror("Erro no shmget") ;
          exit(1) ;
     }
     printf("Sou o processo com pid: %d \n",getpid()) ;
     printf("Identificador do segmento recuperado: %d \n",shmid) ;
     printf("Este segmento e associado a chave unica: %d\n",
                  ftok(path,(key_t)KEY)) ;
    /*
     * acoplamento do processo a zona de memoria
     * recuperacao do pornteiro sobre a area de memoria comum
     */
     if ((mem = shmat (shmid, 0, flag)) == (char*)-1){
```

```
          perror("acoplamento impossivel") ;
          exit (1) ;
     }
    /*
     * tratamento do conteudo do segmento
     */
     printf("leitura do segmento de memoria compartilhada:\n");
     printf("\t==>%s\n",mem) ;
     exit(0);
}
```

Resultado da execução:

```
euler:~/> test_shmget
Identificador do segmento: 41600
Este segmento e associado a chave unica: 2063804629
euler:~/> test_shmat
Sou o processo com pid: 1250
Identificador do segmento recuperado: 41600
Este segmento e associado a chave unica: 2063804629
euler:~/> test_shmat2
Sou o processo com pid: 1251
Identificador do segmento recuperado: 41600
Este segmento e associado a chave unica: 2063804629
leitura do segmento de memoria compartilhada:
        ==>Mensagem escrita na memoria comum
euler:~/> test_shmctl
ESTADO DO SEGMENTO DE MEMORIA COMPARTILHADA 41600
ID do usuario proprietario: 1145
ID do grupo do proprietario: 1000
ID do usuario criador: 1145
ID do grupo criador: 1000
Modo de acesso: 384
Tamanho da zona de memoria: 1024
pid do criador: 1249
pid  (ultima operacao): 1251
```

Note que após o lançamento em seqüência dos programas, o processo com `pid` = 1249, correspondente à execução de `test_shmget` cria o segmento de memória. Depois, esse segmento será acessado por dois processos, sendo que o último é aquele com `pid` = 1251, correspondente à execução de `test_shmat2`.

6.6 Função `shmdt()`

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

```
# include <sys/shm.h>

int shmdt ( const void *shmaddr)
```

Valor de retorno: 0 em caso de sucesso e -1 em caso de erro.

A função `shmdt()` desacopla (*detaches*)o segmento de memória compartilhada especificado pelo endereço `shmaddr` do espaço de endereçamento processo que a chama. Obviamente, o segmento desacoplado deve ser um dentre os segmentos previamente acoplados pelo processo usando `shmat`. Este segmento não poderá mais utilizado pelo processo após a chamada da função.

Exemplo:

```
                /* arquivo test_shmdt.c */
/*
 * este programa permite a leitura do conteudo de um segmento de
 * memoria compartilhada que foi preenchido por algum processo
 * anteriormente. O processo vai se desacoplar do segmento apos
 * a leitura
 */

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

#define KEY 123
#define MSG "Mensagem escrita na memoria comum"

int main()
{
  int shmid ;  /* identificador da memoria comum */
  int size = 1024 ;
  char *path="nome_de_arquivo_existente" ;
  char *mem ;
  int flag = 0;

     /*
     * recuperacao do shmid
     */
     if (( shmid = shmget(ftok(path,(key_t)KEY), size,0)) == -1) {
          perror("Erro no shmget") ;
          exit(1) ;
     }
```

```
     printf("Sou o processo com pid: %d \n",getpid()) ;
     printf("Identificador do segmento recuperado: %d \n",shmid) ;
     printf("Este segmento e associado a chave unica: %d\n",
                    ftok(path,(key_t)KEY)) ;
    /*
     * acoplamento do processo a zona de memoria
     * recuperacao do ponteiro sobre a zona de memoria comum
     */
     if ((mem = shmat (shmid, 0, flag)) == (char*)-1){
          perror("acoplamento impossivel") ;
          exit (1) ;
     }
    /*
     * tratamento do conteudo do segmento
     */
     printf("leitura do segmento de memoria compartilhada:\n");
     printf("\t==>%s\n",mem) ;
    /*
     * desacoplamento do segmento
     */
     if (shmdt(mem)== -1){
          perror("acoplamento impossivel") ;
          exit(1) ;
     }
    /*
     * destruicao do segmento
     */
     if ( shmctl(shmid, IPC_RMID,0) == -1){
          perror("destruicao impossivel") ;
          exit(1) ;
     }
     exit(0);
}
```

Resultado da execução:
Após o lançamento em seqüência dos programas `test_shmget` (para criar um segmento de memória compartilhada), `test_shmat` (para acoplar um processo ao segmento e escrever uma mensagem na zona de memória compartilhada) e `test_shmdt` (para ler o conteúdo, desacoplar e destruir o segmento de memória compartilhada, tem-se a seguinte saída na janela de execução:

```
euler:~/> test_shmget
Identificador do segmento: 43136
Este segmento e associado a chave unica: 2063804629
euler:~/> test_shmat
```

```
Sou o processo com pid: 788
Identificador do segmento recuperado: 43136
Este segmento e associado a chave unica: 2063804629
euler:~/> test_shmdt
Sou o processo com pid: 789
Identificador do segmento recuperado: 43136
Este segmento e associado a chave unica: 2063804629
leitura do segmento de memoria compartilhada:
        ==>Mensagem escrita na memoria comum
```

Capítulo 7

As Filas de Mensagens

7.1 Introdução

Este capítulo apresenta a terceira e última facilidade IPC: as filas de mensagesns. A comunicação interprocessos por mensagens é realizada pela troca de dados, armazenados no sistema, sobre a forma de arquivos. Cada processo pode emitir ou receber mensagens durante uma comunicação. As filas de mensagens são semelhantes aos tubos, mas sem a complexidade associada à abertura e fechamento desses últimos. Entretanto, os problemas relativos à sincronização e bloqueio (leitura em um tubo cheio) no caso de tubos com nome podem ser evitados com as filas. Entretanto, existe um limite imposto para o tamanho dos blocos a serem inseridos na fila, bem como o tamanho máximo total de todos os blocos em todas as filas no sistema. Em `Linux`, por exemplo, dois valores são definidos para esses limites: `MSGMAX` (4096) e `MSGMNB` (16834), os quais definem, respectivamente, o tamanho máximo em bytes de uma mensagem individual e o tamanho máximo da fila.

7.2 Princípio

Da mesma maneira que para os semáforos e para os segmentos de memória compartilhada, uma fila de mensagem é associada a uma chave de acesso única (uma representação numérica no sistema). Esta chave é utilizada para definir e obter um identificador da fila de mensagens, denominada `msqis`, um valor fornecido pelo sistema ao processo oferecendo a chave. Um processo que deseja enviar uma mensagem deve inicialmente obter o identificador da fila `msqid`, utilizando para isso a função `msgget`. Ele utiliza então a função `msgsnd()` para armazenar sua mensagem (a qual está associada a um tipo de dados), dentro de um arquivo. De maneira similar, se um processo deseja ler uma mensagem, ele deve primeiramente buscar o identificador da fila (através da função `msgget()`, para depois ler a mensagem através da função `msgrcv()`.

7.3 Estrutura associada às mensagens: `msqid_ds`

Cada fila de mensagens é associada a um identificador denominado `msqid`. Esta fila tem também uma estrutura de dados associada (definida no arquivo `<sys/msg.h>`) que tem o seguinte formato:

```
/* Structure of record for one message inside the kernel.
   The type 'struct msg' is opaque.  */
struct msqid_ds
{
  struct ipc_perm msg_perm;     /* structure describing operation permission */
  struct msg *__msg_first;      /* pointer to first message on queue */
  struct msg *__msg_last;       /* pointer to last message on queue */
  __time_t msg_stime;           /* time of last msgsnd command */
  __time_t msg_rtime;           /* time of last msgrcv command */
  __time_t msg_ctime;           /* time of last change */
  struct wait_queue *__wwait;   /* ??? */
  struct wait_queue *__rwait;   /* ??? */
  unsigned short int __msg_cbytes;/* current number of bytes on queue */
  unsigned short int msg_qnum;  /* number of messages currently on queue */
  unsigned short int msg_qbytes;/* max number of bytes allowed on queue */
  __ipc_pid_t msg_lspid;        /* pid of last msgsnd() */
  __ipc_pid_t msg_lrpid;        /* pid of last msgrcv() */
};
```

7.4 Função `msgget()`

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/msg.h>

int msgget ( key_t key, int msgflg )
```

Valor de retorno: o identificador `msqid` da fila, ou -1 em caso de erro.

A função `msgget` é utilizada para criar uma nova fila de mensagens, ou para obter o identificador da fila `msqid` de uma fila de mensagens existente no sistema. Esta função recebe dois parâmetros: `key` é a chave indicando uma constante numérica representando a fila de mensagens; `msgflg` é um conjunto de flags especificando as permissões de acesso sobre a fila.

O parâmetro `key` pode conter os seguintes valores:

- IPC PRIVATE (=0): a fila de mensagens não tem chave de acesso e somente o proprietário ou o criador da fila poderão ter acesso à fila;
- um valor desejado para a chave de acesso da fila de mensagens.

O parâmetro `msgflg` é semelhante a `semflg` e a `shmflg`, consistindo de 9 flags de permissão de acesso. Estes flags são a combinação (de maneira clássica através do operador lógico OU) de diferentes constantes prédefinidas, permitindo de estabelecer direitos de acesso e os comandos de controle. As constantes pré-definidas estão normalmente no arquivo `<sys/msg.h>` e têm os seguintes valores:

```
#define IPC_CREAT 0001000 /* criacao de uma fila de mensagens */
#define IPC_EXCL 0002000  /* associado ao IPC_CREAT provoca um
                           * erro se a fila ja existe */
```

7.4.1 Como criar uma fila de mensagens

A criação de uma fila de mensagens é similar à criação de um conjunto de semáforos, ou de um segmento de memória compartilhada. As seguintes regras básicas devem ser respeitadas nesse caso:

- `key` deve conter um valor identificando a fila;
- `msgflg` deve conter os direitos de acesso desejados à fila e a constante `IPC_CREAT`;
- Se deseja-se testar a existência de uma fila, deve-se adicionar ao conjunto de flags a constante `IPC_EXCL`; assim, `msgget` vai falhar caso exista uma fila associada ao valor de chave `key`

Note ainda que durante a criação de um fila de mensagens, alguns campos da estrutura `msqid_ds` são iniciados (proprietário, modos de acesso, data de criação, etc.).

Exemplo:

Exemplo de utilização do `msgget()`: este programa cria uma fila de mensagens associada à chave 123, e verifica o conteúdo das estruturas do sistema próprias a essa fila.

```
                /* arquivo test_msgget.c */
/*
 *  exemplo de utilizacao de msgget()
 */

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

#define KEY 123

int main()
{
  int msqid ;  /* ID da fila de mensagens */
  char *path = "nome_de_arquivo_existente" ;
    /*
     * criacao de uma fila de mensagens para leitura se
     * ela ainda nao existe
     */
     if (( msqid = msgget(ftok(path,(key_t)KEY),
```

```
                        IPC_CREAT|IPC_EXCL|0600)) == -1) {
          perror("Erro de msgget") ;
          exit(1) ;
     }
   printf("identificador da fila: %d\n",msqid) ;
   printf("esta fila esta associada a chave unica : %#x\n"
                            ,ftok(path,(key_t)KEY)) ;
   exit(0);
}
```

Resultado da execução:

```
euler:~/> test_msgget
identificador da fila: 1152
esta fila esta associada a chave unica : 0x7b045862
euler:~/> ipcs -q

------ Message Queues --------
key       msqid     owner     perms     used-bytes  messages
0x7b045862 1152      saibel    600       0           0
```

7.5 Função `msgctl()`

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/msg.h>

int msgctl ( int msqid, int  cmd, struct msqid_ds *buf )
```

Valor de retorno: 0 em caso de sucesso e -1 em caso de erro.

A função `msgctl()` é utilizada para examinar e modificar os atributos de uma fila de mensagens existente. Ela recebe três parâmetros: um identificador da fila de mensagens (`msqid`); um comando a ser efetuado sobre a fila (`command`) e; um ponteiro para uma estrutura do tipo `msqid_ds` (`buf`).

O parâmetro `command` pode conter os seguintes valores e ações associadas:

- `IPC_RMID` (0): Destruir a fila de mensagens. Somente o processo tendo um número de usuário idêntico a `msg_perm.uid` ou o super-usuário podem destruir a fila. Todas as outras operações em curso sobre essa fila irão falhar e os processos bloqueados em espera de leitura ou escrita serão liberados.

- `IPC_SET` (1): Se o processo tem permissão de fazê-lo, este comando vai dar ao identificador do grupo, aos direitos de acesso da fila de mensagem, e ao número total de carateres dos textos, os valores contidos no campo `msg_perm` da estrutura apontada por `buf`. A hora da modificação é também atualizada.

- `IPC_STAT` (2): A estrutura `msqid_ds` apontada por `buf` refletirá os valores associados à fila de mensagens.

Exemplo:

Neste exemplo, supõe-se que o segmento de memória compartilhada de chave 123 foi criado anteriormente por um processo.

```
                /* arquivo test_msgctl.c */
/*
 * o programa recupera o ID de uma fila existente (criada com o
 * programa test_msgget.c e mostra a estrutura msqid_ds
 * associada a essa fila
 */

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <time.h>

#define KEY 123

int main()
{
  struct msqid_ds buf ;
  char *path = "nome_de_arquivo_existente" ;
  int msqid ;
     /* recuperacao do ID da fila de mensagens associada a chave 123 */
     if (( msqid = msgget(ftok(path,(key_t)KEY),0)) == -1 ) {
          perror ("Erreur msgget()") ;
          exit(1) ;
     }
      printf("A chave %#x esta associada a fila %d\n",
              ftok(path,(key_t)KEY), msqid);
     /* recuperacao na estrutura buf dos parametros da fila */
     if (msgctl(msqid,IPC_STAT,&buf) == -1){
          perror("Erreur msgctl()") ;
          exit(1) ;
     }
     else
     {
        printf("id da fila de mensagens     : %d\n",msqid) ;
```

```
        printf("id do proprietario          : %d\n",buf.msg_perm.uid) ;
        printf("id do grupo do proprietario : %d\n",buf.msg_perm.gid) ;
        printf("id do criador               : %d\n",buf.msg_perm.cuid) ;
        printf("id do grupo do criador      : %d\n",buf.msg_perm.cgid) ;
        printf("direitos de acesso          : %d\n",buf.msg_perm.mode) ;
        printf("nb atual de bytes na fila   : %d\n",buf.msg_cbytes) ;
        printf("nb de mensagens na fila     : %d\n",buf.msg_qnum) ;
        printf("nb maximal de bytes na fila : %d\n",buf.msg_qbytes) ;
        printf("pid do ultimo escritor      : %d\n",buf.msg_lspid) ;
        printf("pid do ultimo leitor        : %d\n",buf.msg_lrpid) ;
        printf("data da ultima escrita      : %s\n",ctime(&buf.msg_stime)) ;
        printf("data da ultima leitura      : %s\n",ctime(&buf.msg_rtime)) ;
        printf("data da ultima modificacao  : %s\n",ctime(&buf.msg_ctime)) ;
     }
   exit(0);
}
```

Resultado da execução:

```
euler:~/> test_msgctl
A chave 0x7b045862 esta associada a fila 1152
id da fila de mensagens      : 1152
id do proprietario           : 1145
id do grupo do proprietario  : 1000
id do criador                : 1145
id do grupo do criador       : 1000
direitos de acesso           : 384
nb atual de bytes na fila    : 0
nb de mensagens na fila      : 0
nb maximal de bytes na fila  : 16384
pid do ultimo escritor       : 0
pid do ultimo leitor         : 0
data da ultima escrita       : Wed Dec 31 21:00:00 1969

data da ultima leitura       : Wed Dec 31 21:00:00 1969

data da ultima modificacao   : Tue Oct 17 06:43:09 2000
```

7.6 Função `msgsnd()`

```
  # include <sys/types.h>
  # include <sys/ipc.h>
  # include <sys/msg.h>
```

```
int  msgsnd ( int msqid, struct msgbuf *msgp, int msgsz,
              int msgflg )
```

Valor de retorno: 0 se a mensagem é colocada na fila e -1 em caso de erro.

A função `msgsnd` permite a inserção de uma mensagem na fila. A estrutura da mensagem é limitada de duas maneiras: primeiramente, ela deve ser menor que o limite estabelecido pelo sistema; depois, ela deve respeitar o tipo de dado estabelecido pela função que receberá a mensagem. Esta função recebe três parâmetros em sua chamada: o identificador da fila `msqid`; um ponteiro `msgp` para a estrutura de tipo `msgbuf` que contem a mensagem a ser enviada; um inteiro `msg_sz` indicando o tamanho em bytes da mensagem apontada por `msgbuf` e; un flag `msgflg` que controla o modo de envio da mensagem.

Em relação ao valor do flag `msgflg`, ele poderá ser utilizado da seguinte forma:

- Se este parâmetro vale 0, ele provocará o bloqueio do processo chamando `msgsnd` quando a fila de mensagens estiver cheia;
- Se ele tem o flag `IPC_NOWAIT`, a função retorna imediatamente sem enviar a mensagem e com um valor de erro igual a -1 indicando que a fila esta cheia. Este indicador age da mesma maneira que `O_NDELAY` nos tubos com nome (ver seção 4.6.

A função `msgsnd` atualiza também a estrutura `msqid_ds`:

- incremento do número de mensagens da fila (`msg_qnum`);
- modificação do número (`pid`) do último processo que escreveu na fila (`msg_lspid`);
- modificação da data da última escrita (`msg_stime`).

A estrutura `msgbuf`

A estrutura `msgbuf` descreve a estrutura da mensagem propriamente dita. Ela é definida em `<sys/msg.h>` da seguinte maneira:

```
/* Template for struct to be used as argument for
 * 'msgsnd' and 'msgrcv'.  */
struct msgbuf
  {
    long int mtype;     /* type of received/sent message */
    char mtext[1];      /* text of the message */
  };
```

`mtype` é um inteiro longo positivo, o qual é usado para definir o tipo de mensagem na função de recepção. Uma boa regra de programação quando filas de mensagens são utilizadas, é definir `mtype` logo no início da estrutura que define a mensagem. Uma vez que `mtype` é usado para recepção, ela deve não só ser imperativamente declarada na sua estrutura, como também, ela deve conter um valor conhecido.

`mtext` é a mensagem a ser efetivamente enviada (array de bytes).

Estranhamente, a estrutura definida em IPC é definida para mensagens de tamanho igual a apenas 1 byte. Para contornar esse problema, costuma-se definir uma estrutura para envio de mensagens para uma fila da seguinte forma:

```
#define MSG_SIZE_TEXT 256
struct msgtext
 {
    long mtype ;                  /* type da mensagem */
    char mtexte[MSG_SIZE_TEXT] ; /* texto du mensagem */;
 }
```

Segundo as necessidades do programador, o tamanho máximo das mensagens podem ser reguladas através do parâmetro `MSG_SIZE_TEXT`. Observe que o campo `mtype` está localizado no início da estrutura.

Exemplo:

Exemplo de utilização da função `msgsnd`:

```
                    /* arquivo test_msgsnd.c */
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

#define KEY 123
#define MSG_SIZE_TEXT 256

int main()
{
int i = 1 ;
int msqid ;
char *path = "nome_de_arquivo_existente" ;

/* estrutura msg associada as mensagens */
struct msgtext {
     long mtype ;
     char mtext[MSG_SIZE_TEXT] ;
} msg ;

   /* recuperacao do identificador da fila de mensagens */
   if (( msqid = msgget(ftok(path,(key_t)KEY),0)) == -1 ) {
          perror ("Erro msgget()") ;
          exit(1) ;
   }
```

```
   printf("A chave %#x esta associada a fila %d\n",
           ftok(path,(key_t)KEY), msqid);
   msg.mtype = 1 ;           /* tipo das mensagens */
   while(i<=30)
   {
       /* escreve o texto da mensagem */
       sprintf(msg.mtext,"mensagem no %d de tipo %ld",i,msg.mtype) ;
       /* envia a mensagem a fila */
       if(msgsnd(msqid,&msg,strlen(msg.mtext),IPC_NOWAIT) == -1)
       {
             perror("Envio de mensagem impossivel") ;
             exit(-1) ;
       }
       printf("mensagem no %d de tipo %ld enviada a fila %d\n",
               i,msg.mtype,msqid) ;
       printf("-->texto da mensagem: %s\n",msg.mtext) ;
       i++ ;
     }
     exit(0);
}
```

Resultado da execução:

```
euler:~/> test_msgsnd
A chave 0x7b045862 esta associada a fila 1152
mensagem no 1 de tipo 1 enviada a fila 1152
-->texto da mensagem: mensagem no 1 de tipo 1
mensagem no 2 de tipo 1 enviada a fila 1152
-->texto da mensagem: mensagem no 2 de tipo 1

...

mensagem no 29 de tipo 1 enviada a fila 1152
-->texto da mensagem: mensagem no 29 de tipo 1
mensagem no 30 de tipo 1 enviada a fila 1152
-->texto da mensagem: mensagem no 30 de tipo 1
```

O estado atual da fila pode ser consultado através da execução do programa `test_msgctl`:

```
euler:~/> test_msgctl

A chave 0x7b045862 esta associada a fila 1152
id da fila de mensagens      : 1152
id do proprietario           : 1145
```

```
id do grupo do proprietario  : 1000
id do criador                : 1145
id do grupo do criador       : 1000
direitos de acesso           : 384
nb atual de bytes na fila    : 711
nb de mensagens na fila      : 30
nb maximal de bytes na fila  : 16384
pid do ultimo escritor       : 2192
pid do ultimo leitor         : 2190
data da ultima escrita       : Tue Oct 17 06:53:54 2000

data da ultima leitura       : Tue Oct 17 06:50:21 2000

data da ultima modificacao   : Tue Oct 17 06:43:09 2000
```

7.7 Função `msgrcv()`

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/msg.h>

int msgrcv ( int msqid, struct msgbuf *msgp, int msgsz,
             long msgtyp, int msgflg )
```

Valor de retorno: número de bytes da mensagem extraída da fila, ou -1 em caso de erro.

Esta função retira uma mensagem da fila. Ela recebe cinco parâmetros: o identificador da fila `msqid`; um ponteiro `msgp` para a estrutura de tipo `msgbuf` que contém a mensagem; um inteiro `msg_sz` indicando o tamanho máximo da mensgem a ser recebida; um inteiro longo `msgtyp` indicando o qual a mensagem a ser recebida e; um flag `msgflg` controlando o modo de execução da recepção da mensagem.

A função vai armazenar a mensagem lida numa estrutura apontada por `msgp`, a qual contém os seguintes elementos (idênticos aos usados para o envio da mensagem):

```
struct msgbuf
  {
    long int mtype;     /* type of received message */
    char mtext[1];      /* text of the message */
  };
```

O tamanho do campo `mtext` é fixado segundo as necessidades do programa (ver `msgsnd()` para maiores detalhes).

O parâmetro `msgtyp` indica qual a mensagem deve ser recebida:

- Se `msgtyp = 0`, a primeira mensagem da fila será lida, isto é, a mensagem na cabeça da lista será lida;

- Se `msgtyp > 0`, a primeira mensagem que tiver um valor igual a `msgtyp` deverá ser retirada da fila, desde que o flag `MSG_EXCEPT` não esteja setado no argumento `msgflg`; do contrário, a primeira mensagem com valor diferente de `msgtyp` será lida;
- Se `msgtyp < 0`, a primeira mensagem da fila com o mais baixo valor de tipo que é menor ou igual ao valor absoluto de `msgtyp` será lida;

Por exemplo, considerando três mensagens tendo por tipos 100, 200 e 300, a tabela a seguir indica o tipo de mensagem a ser retornada pelos diferentes valores de `msgtyp`:

`msgtyp`	tipo de mensagem retornada
0	100
100	100
200	200
300	300
100	100
200	100
300	100

Em relação ao parâmetro `msgflg`:

- Se ele contiver o flag `IPC_NOWAIT`, a chamada à `msgrcv` retorna imediatamente com um código de erro quando a fila não contiver uma mensagem do tipo desejado.
- Se ele não contiver o flag `IPC_NOWAIT`, o processo que chama a função `msgrcv` ficará bloqueado até que haja uma mensagem do tipo requerido (`msgtyp` na fila;
- Se ele contiver o flag `MSG_NOERROR`, a mensagem é truncada com um tamanho máximo `msgsz` bytes, sendo a parte truncada perdida;
- Se ele não contiver o flag `MSG_NOERROR`, a mensagem não é lida, e `msgrcv` retorna um código de erro.
- O flag `MSG_EXCEPT` é utlizado em conjunto com `msgtyp > 0` para a leitura da primeira mensagem da fila cujo tipo de mensagem difere de `msgtyp`.

Exemplo:

Exemplo de utilização da função `msgrcv()`:

```
                /* arquivo test_msgrcv.c */
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
#include <stdio.h>

#define KEY 123
#define MSG_SIZE_TEXT 256

/* estrutura msg associada as mensagens */
struct msgtext {
     long mtype ;
     char mtext[MSG_SIZE_TEXT] ;
} msg ;

int main()
{
int lg ;            /* tamanho da mensagem recebida */
long type = 1 ;     /* tipo de mensagem buscado */
int size_msg = 22 ; /* tamanho maximo do texto a ser recuperado */
int msqid ;         /* identificador da fila */
char *path = "nome_de_arquivo_existente" ;

   /* recuperacao do identificador da fila de mensagens */
   if (( msqid = msgget(ftok(path,(key_t)KEY),0)) == -1 ) {
          perror ("Erro msgget()") ;
          exit(1) ;
   }
   printf("A chave %#x esta associada a fila %d\n",
              ftok(path,(key_t)KEY), msqid);
      /* o programa vai ler na fila enquanto existirem mensagens
    * se as mensagens sao maiores que o tamanho maximo size_msg,
    * elas serao truncadas
    * quando a fila estiver vazia, o processo nao se bloqueia */
   while((lg=msgrcv(msqid,&msg,size_msg,type,IPC_NOWAIT|MSG_NOERROR)) != -1)
   {
       printf("texto da mensagem (tamanho %d) recebido: %s\n",
               lg,msg.mtext) ;
   }
   perror("nao ha mais mensagens") ;
   exit(-1) ;
}
```

Resultado da execução:

```
euler:~/> test_msgrcv
A chave 0x7b045862 esta associada a fila 1152
texto da mensagem (tamanho 22) recebido: mensagem no 1 de tipo
```

```
texto da mensagem (tamanho 22) recebido: mensagem no 2 de tipo

...

texto da mensagem (tamanho 22) recebido: mensagem no 29 de tipo
texto da mensagem (tamanho 22) recebido: mensagem no 30 de tipo
nao ha mais mensagens: No message of desired type
```

O programa lê na fila de mensagens enquanto mensagens do tipo desejado existirem na fila. Quando a fila estiver vazia, não haverá bloqueio de processos em leitura devido ao flag `IPC_NOWAIT`. Por outro lado, deve-se observar que as mensagens são truncadas (flag `MSG_NOERROR` ao tamanho especificado para a recepção da mensagem. Além disso, `;g` será igual ao tamanho da mensagem `size_msg` neste caso específico, uma vez que a mensagem é truncada.

Mais uma vez, pode-se constatar que a leitura (isto é a retirada) das mensagens da fila foi realmente realizada através da execução de `test_msgctl.c`:

```
euler:~/> test_msgctl
A chave 0x7b045862 esta associada a fila 1152
id da fila de mensagens      : 1152
id do proprietario           : 1145
id do grupo do proprietario  : 1000
id do criador                : 1145
id do grupo do criador       : 1000
direitos de acesso           : 384
nb atual de bytes na fila    : 0
nb de mensagens na fila      : 0
nb maximal de bytes na fila  : 16384
pid do ultimo escritor       : 2192
pid do ultimo leitor         : 2194
data da ultima escrita       : Tue Oct 17 06:53:54 2000

data da ultima leitura       : Tue Oct 17 06:57:17 2000

data da ultima modificacao   : Tue Oct 17 06:43:09 2000
```

Note que o número de mensagens na fila agora vale 0 e que a data da última escrita na fila é idêntica àquela obtida na execução precedente de `test_msgctl`.

Capítulo 8

Exemplos usando IPC

8.1 Exemplo1: *Rendezvous* de três processos

O exemplo dessa seção apresenta a implementação de um mecanimo de*rendez-vous* (ponto de encontro ou sincronização) entre três processos. O primeiro programa (`rdv1`) cria um conjunto de dois semáforos (um para a exclusão mútua e outro para o ponto de *rendez-vous*), e um segmento de memória compartilhada permitindo que cada um dos processos saibam na sua chegada, o número de processos que já estão no ponto de encontro. O segundo programa (`rdv2`) simula os processos que devem chegar a esse ponto. O terceiro programa (`rdv3`) serve para destruir os recursos criados por `rdv1`. Entretanto, se ele for lançado antes da chegada de no mínimo três ao ponto de *rendez-vous*, os recursos permanecerão intactos.

Para a execução, deve-se lançar inicialmente `rdv1` (para que a inicialização dos mecanismos IPC seja realizada) e depois `rdv2` no mínimo três vezes em *background*. Após a execução dos programas `rdv2`, o programa `rdv3` pode ser então lançado para destruir os mecanismos IPC implementados pela aplicação.

```
                /* arquivo rdv1.c */

/*-------------------------------------------------------
                     Rendez-vous de n processos
                     --------------------------

                mutex = 1 ; srdv = 0;
                n = 0     ;
                N numero de processos a esperar

                P(mutex, 1) ;
                n := n + 1 ;

                if (n < N) then

                           begin
                            V(mutex, 1);
                            P(srdv,, 1);
```

```
                            end

                           else

                            begin
                             V(mutex, 1);
                             V(srdv, N);
                            end;
-------------------------------------------------------------*/

/*          INICIALIZACAO
 * codigo do processo criador da memoria compartilhada e
 * dos semaforos
 */

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

#define     SHMKEY 75 /* chave para a memoria compartilhada */
#define     SEMKEY 76 /* chave para os semaforos */
#define     NBSEM 2   /* no. de semaforos (mutex e srdv) */

#define     RW 0600  /* permissao de leitura e escrita */

#define K 1024 /* tamanho do segmento de memoria */

int *pmem;
int shmid, semid;

int main()
{
 short initarray[NBSEM-1], outarray[NBSEM-1];

    /* criacao da memoria compartilhada */
     if ((shmid = shmget(SHMKEY,K,RW|IPC_CREAT))==-1){
          perror("Erro na criacao do segmento de memoria") ;
          exit(1) ;
     }
     pmem = (int *) shmat(shmid, 0, 0);
     pmem[0] = 0; /* no. de processos que chegaram (0 no inicio) */
```

```
     pmem[1] = 3; /* no. de processos esperados no ponto de encontro */

    /* criacao dos semaforos */
     if ((semid = semget(SEMKEY, NBSEM,RW|IPC_CREAT)) == -1){
          perror("Erro na criacao de semaforos") ;
          exit(1) ;
     }

    /* inicializacao dos semaforos */
     initarray[0] = 1;         /* mutex = 1 */
     initarray[1] = 0;         /* srdv  = 0 */
     if (semctl(semid, NBSEM-1, SETALL, initarray) == -1){
          perror ("inicializacao dos semaforos incorreta") ;
          exit(1) ;
     }

    /* leitura dos parametros dos semaforos */
     if (semctl(semid, NBSEM-1, GETALL, outarray) == -1){
          perror("leitura dos semaforos incorreta") ;
          exit(1);
     };

     printf("\n================INICIALIZACAO================\n") ;
     printf(" numero de processos esperados       : %d\n",pmem[1]) ;
     printf(" numero de processos que chegaram    : %d\n",pmem[0]) ;
     printf(" Para criar um novo processo, digite : rdv2 &\n");
     printf("=============================================\n") ;
     exit(0);
}

                /* arquivo rdv2.c */

/* codigo executaod pelos processos antes de chegar ao
 * ponto de encontro (rendez-vous)
 */

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

#define SHMKEY 75 /* chave para a memoria compartilhada */
```

```
#define SEMKEY 76 /* chave para os semaforos */

#define K 1024 /* tamanho do segmento de memoria */

int *pmem;
int shmid, semid;
struct sembuf pmutex[1], vmutex[1], psrdv[1], vsrdv[1] ;

int main()
{
    /*     recuperacao do shmid        */
     if ((shmid = shmget(SHMKEY, K, 0)) == -1){
          perror("Erro no shmget") ;
          exit(1) ;
     }
    /*      recuperacao do semid        */
     if ((semid = semget(SEMKEY, 2, 0)) == -1){
          perror ("Erro no semget") ;
          exit(1) ;
     }
    /*       acoplamento ao segmento     */
     if ((pmem = shmat(shmid, 0, 0)) == (int *)-1){
          perror("attachement non effectue") ;
          exit(1) ;
     }
    /*       demanda de recurso (P(mutex))      */
     pmutex[0].sem_op = -1;
     pmutex[0].sem_flg = SEM_UNDO;
     pmutex[0].sem_num = 0;
     if (semop(semid, pmutex, 1) == -1){
          perror("P(mutex) nao efetuado") ;
          exit(1) ;
     }
    /*     numero de processos que chegaram      */
     pmem[0] += 1;  /* n = n + 1 */
     printf("\nNo. de processos que chegaram  = %d de %d\n",
               pmem[0], pmem[1]);

     if ( pmem[0] < pmem[1] ){
         /*  liberacao do recurso (V(mutex))  */
          vmutex[0].sem_op = 1;
          vmutex[0].sem_flg = SEM_UNDO;
          vmutex[0].sem_num = 0;
          if (semop(semid, vmutex, 1) == -1){
```

```
                perror("V(mutex) nao efetuado") ;
                exit(1) ;
           }
           printf("Processo %d: vou me bloquear em espera \n",getpid());
          /*    demanda de recurso P(srdv)     */
           psrdv[0].sem_op = -1;
           psrdv[0].sem_flg = SEM_UNDO;
           psrdv[0].sem_num = 1;
           if (semop(semid, psrdv, 1) == -1){
                perror("P(srdv) nao efetuado") ;
                exit(1) ;
           }
      }
      else {
           printf("Processo %d: Vou liberar todos os processos bloqueados\n",
                   getpid());
           /*     liberacao de recurso (V(mutex))      */
           vmutex[0].sem_op = 1;
           vmutex[0].sem_flg = SEM_UNDO;
           vmutex[0].sem_num = 0;
           if (semop(semid, vmutex, 1) == -1){
                perror("liberacao final de mutex nao efetuada") ;
                exit(1) ;
           }
          /*     liberacao de 3 recursos     */
           vsrdv[0].sem_op = pmem[1];
           vsrdv[0].sem_flg = SEM_UNDO;
           vsrdv[0].sem_num = 1;
           if (semop(semid, vsrdv, 1) == -1){
                perror("liberacao final de srdv nao efetuada") ;
                exit(1) ;
           }
      }
      printf("Processo %d: acabo de passar o ponto de rendez-vous\n",
              getpid());
      exit(0);
}

                  /* arquivo rdv3.c */

/* remocao dos recursos criados por rdv1*/

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

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>

#define SHMKEY 75 /* chave para a memoria compartilhada */
#define SEMKEY 76 /* chave para os semaforos */
#define NBSEM 2   /* no. de semaforos (mutex e srdv) */

#define RW 0600  /* permissao de leitura e escrita */

#define K 1024 /* tamanho do segmento de memoria */

int *pmem;
int shmid, semid;
struct sembuf pmutex[1], vmutex[1], psrdv[1], vsrdv[1] ;

int main()
{
    /*     recuperacao do shmid        */
     if ((shmid = shmget(SHMKEY, K, 0)) == -1){
          perror("Erro no shmget") ;
          exit(1) ;
     }
    /*      recuperacao do semid        */
     if ((semid = semget(SEMKEY, 2, 0)) == -1){
          perror ("Erro no semget") ;
          exit(1) ;
     }
    /*       acoplamento ao segmento     */
     if ((pmem = shmat(shmid, 0, 0)) == (int *)-1){
          perror("attachement non effectue") ;
          exit(1) ;
     }
    /*       demanda de recurso (P(mutex))      */
     pmutex[0].sem_op = -1;
     pmutex[0].sem_flg = SEM_UNDO;
     pmutex[0].sem_num = 0;
     if (semop(semid, pmutex, 1) == -1){
          perror("P(mutex) nao efetuado") ;
          exit(1) ;
     }
     if ( pmem[0] > pmem[1] ){
     printf("\n===============FINALIZACAO==============\n") ;
```

```
      printf(" numero de processos esperados       : %d\n",pmem[1]) ;
      printf(" numero de processos que chegaram    : %d\n",pmem[0]) ;
      printf(" Os recursos serao destruidos nesse instante\n");
      printf("=============================================\n") ;
        /*      destruicao do semaforo    */
        if (semctl(semid,0,IPC_RMID,0)==-1){
              perror("Impossivel de destruir o semaforo") ;
              exit(1) ;
        }
        /*      destruicao do segmento de memoria compartilhada    */
        if (shmctl(shmid,IPC_RMID,0)==-1){
              perror("Impossivel de destruir o segmento de memoria") ;
              exit(1) ;
        }
      }
      else {
          /*  liberacao do recurso (V(mutex))  */
           vmutex[0].sem_op = 1;
           vmutex[0].sem_flg = SEM_UNDO;
           vmutex[0].sem_num = 0;
           if (semop(semid, vmutex, 1) == -1){
                perror("V(mutex) nao efetuado") ;
                exit(1) ;
           }
           printf("Processo %d: ainda nao posso destruir os recursos\n",
                  getpid());
           printf("Processo %d: faltam ainda %d processos de %d\n",
                  getpid(),(pmem[1]-pmem[0]), pmem[1]);
      }
      exit(0);
}
```

Resultado da execução:

```
margarida:~/> rdv1

===============INICIALIZACAO=============
 numero de processos esperados       : 3
 numero de processos que chegaram    : 0
 Para criar um novo processo, digite : rdv2 &
=========================================
margarida:~/> rdv2 &
[6] 949
margarida:~/>
```

```
No. de processos que chegaram  = 1 de 3
Processo 949: vou me bloquear em espera

margarida:~/> rdv3
Processo 951: ainda nao posso destruir os recursos
Processo 951: faltam ainda 2 processos de 3
margarida:~/graduacao/STR/Apostila/Chap8> rdv2 &
[7] 952
margarida:~/>
No. de processos que chegaram  = 2 de 3
Processo 952: vou me bloquear em espera

margarida:~> rdv2 &
[8] 953

No. de processos que chegaram  = 3 de 3
Processo 953: Vou liberar todos os processos bloqueados
Processo 949: acabo de passar o ponto de rendez-vous
Processo 952: acabo de passar o ponto de rendez-vous
Processo 953: acabo de passar o ponto de rendez-vous
[8]    Done                          rdv2

[7]    Done                          rdv2

[6]    Done                          rdv2
margarida:~/> rdv3

================FINALIZACAO===============
 numero de processos esperados       : 3
 numero de processos que chegaram    : 3
 Os recursos serao destruidos nesse instante
==========================================
```

8.2 Exemplo2: Cliente-servidor

Este exemplo implementa um problema clássico cliente-servidor utilizando mensagens. Dois tipos de estruturas podem ser utilizadas na comunicação:

1. As mensagens podem ser trocadas através de duas filas diferentes;
2. Uma fila única pode ser utlizada, com as mensagens sendo diferenciadas através da implementação de tipos distintos.

Nos dois casos, deve-se lançar inicialmente o programa `server<num>` em *backgroung*, e em seguida o programa `client<num>`. Uma mensagem *par défaut* será enviada se nenhum argumento for passado em

linha de comando para o programa `client<num>`. Se o usuário desejar passar outro texto de mensagem, basta digitar `client<num> <texto_da_mensagem>`.

8.2.1 Exemplo cliente-servidor com 1 fila

Código do servidor

```
               /* arquivo server1.c */

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

#define MSGKEY 13          /* chave associada a fila */

#define MSG_SIZE_TEXT 256  /* tamanho do texto das mensagens  */

#define MSG_SIZEMAX 260    /* MSG_SIZE_TEXT + sizeof(int) */
                           /* tamanho total das mensagens */

#define TYPE_CLIENT 1     /* tipo de mensagens enderacadas
                           * ao cliente */
#define TYPE_SERVEUR 2    /* tipo de mensagens enderecadas
                           * ao servidor  */
#define  RW 0600          /* flag para permissao leitura e escrita */

struct msgform {
     long     mtype;
     int      pid;
     char     mtext[MSG_SIZE_TEXT];
} ;

int main() {
struct msgform msg;         /* mensagem */
int msgid;                 /* identificador da fila de mensagens */

     /*     criacao da fila de mensagens      */
     if ((msgid = msgget(MSGKEY, RW|IPC_CREAT)) == -1) {
          perror("Erro na criacao da fila") ;
          exit(1) ;
```

```
    }

    for (;;) {
       /*       recepcao de mensagens       */
       if (msgrcv(msgid, &msg,MSG_SIZEMAX,TYPE_SERVEUR, 0) == -1) {
              perror("Erro na recepcao da mensagem") ;
              exit(1) ;
       }
       printf("O servidor %d recebeu uma mensagem do cliente %d\n",
                getpid(),msg.pid);
       printf("Texto da mensagem: %s\n", msg.mtext) ;

       /*        envio de mensagem         */
       msg.mtype  = TYPE_CLIENT ;
       msg.pid = getpid();
       if (msgsnd(msgid, &msg, MSG_SIZEMAX , 0) == -1) {
              perror("Erro no envio da mensagem") ;
              exit(1) ;
       }
   }
   exit(0);
}
```

Código do cliente

```
                /* arquivo client1.c */

#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>
#include <string.h>

#define MSGKEY     13     /* chave associada a fila */

#define MSG_SIZE_TEXT 256 /* tamanho do texto das mensagens  */

#define MSG_SIZEMAX 260   /* MSG_SIZE_TEXT + sizeof(int)     */
                          /* tamanho total das mensagens */

#define TYPE_CLIENT 1     /* tipo de mensagens enderecadas
```

```
                          * ao cliente */
#define TYPE_SERVEUR 2    /* tipo de mensagens enderecadas
                          * ao servidor  */

struct msgform {
     long     mtype;
     int      pid;
     char     mtext[MSG_SIZE_TEXT];
} ;

int main(int argc, char *argv[]) {

struct msgform msg;     /* mensagem */
int msgid;              /* identificador da fila do servidor */
char message1[32];

 /* definicao do texto da mensagem : client1 <argumento>
  * se <argumento> e vazio, um valor default e enviado */

 if (argc > 1) strcpy(message1,argv[1]);
 else  strcpy(message1,"Texto da mensagem");

  /*  recuperacao do id da fila de mensagens do servidor       */
  if ((msgid = msgget(MSGKEY,0)) == -1) {
          perror("Erro na criacao da fila do servidor") ;
          exit(1) ;
  }
  /*      envio de mensagens       */
     strcpy(msg.mtext,message1) ;
     printf("Cliente: envio de mensagem: %s\n",msg.mtext) ;
     msg.pid= getpid();
     msg.mtype = TYPE_SERVEUR ;
     if (msgsnd(msgid,&msg,MSG_SIZEMAX,0) == -1) {
          perror("Erro no envio de mensagem") ;
          exit(1) ;
     }
     /*      recepcao de mensagem (ack de recepcao)      */
     printf("Cliente: espera de ack do servidor\n") ;
     if (msgrcv(msgid, &msg, MSG_SIZEMAX, TYPE_CLIENT, 0) == -1) {
          perror("Erro na recepcao de mensagem") ;
          exit(1) ;
     }
     printf("Cliente %d recebeu um ack do servidor %d\n",
             getpid(), msg.pid);
```

```
     exit(0);
}
```

Resultado da execução:

```
lyapunov:~> server1 &
[2] 3575
lyapunov:~> client1
Cliente: envio de mensagem: Texto da mensagem
O servidor 3575 recebeu uma mensagem do cliente 3576
Texto da mensagem: Texto da mensagem
Cliente: espera de ack do servidor
Cliente 3576 recebeu um ack do servidor 3575
lyapunov:~> client1 "NOVA MENSAGEM DO USUARIO"
Cliente: envio de mensagem: NOVA MENSAGEM DO USUARIO
O servidor 3575 recebeu uma mensagem do cliente 3577
Texto da mensagem: NOVA MENSAGEM DO USUARIO
Cliente: espera de ack do servidor
Cliente 3577 recebeu um ack do servidor 3575
lyapunov:~>
```

8.2.2 Exemplo cliente-servidor com 2 filas

Código do servidor

```
                /* arquivo server2.c */

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

#define MSGKEY    13      /* chave associada a fila */

#define MSG_SIZE_TEXT 256 /* tamanho do texto das mensagens  */

#define MSG_SIZEMAX 260   /*  MSG_SIZE_TEXT + sizeof(int)   */
                          /* tamanho total das mensagens */

#define  RW 0600          /* flag para permissao leitura e escrita */

struct msgform {
```

```
     long     mtype;
     int      pid;
     char     mtext[MSG_SIZE_TEXT];
} ;

int main() {
struct msgform msg;       /* mensagem */
int msgid_serv;          /* identificador da fila do servidor */
int msgid_client;        /* identificador da fila do cliente  */

   /*      criacao da fila de mensagens do servidor       */
   if ((msgid_serv = msgget(MSGKEY, RW|IPC_CREAT)) == -1) {
         perror("Erro na criacao da fila do servidor") ;
         exit(1) ;
   }
   /*      criacao da fila de mensagens do cliente       */
   if ((msgid_client = msgget(MSGKEY+1, RW|IPC_CREAT)) == -1) {
          perror("Erro na criacao da fila do cliente") ;
          exit(1) ;
   }
   for (;;) {
     /*       recepcao de mensagens       */
     if (msgrcv(msgid_serv,&msg,MSG_SIZEMAX,1,0) == -1) {
            perror("Erro na recepcao da mensagem") ;
            exit(1) ;
     }
     printf("O servidor %d recebeu uma mensagem do cliente %d\n",
                 getpid(),msg.pid);
     printf("Texto da mensagem: %s\n", msg.mtext) ;
     /*        envio de mensagem         */
     msg.mtype = 1 ;
     msg.pid = getpid();
     if (msgsnd(msgid_client, &msg, MSG_SIZEMAX , 0) == -1) {
            perror("Erro no envio da mensagem") ;
            exit(1) ;
     }
   }
   exit(0);
}
```

Código do cliente

```
              /* arquivo client2.c */
```

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

#define MSGKEY     13     /* chave associada a fila */

#define MSG_SIZE_TEXT 256 /* tamanho do texto das mensagens */

#define MSG_SIZEMAX 260   /* MSG_SIZE_TEXT + sizeof(int)    */
                          /* tamanho total das mensagens */

struct msgform {
     long     mtype;
     int      pid;
     char     mtext[MSG_SIZE_TEXT];
} ;


int main(int argc, char *argv[]) {
struct msgform msg;       /* mensagem */
int msgid_serv;          /* id da fila do servidor */
int msgid_client;        /* id da fila do cliente  */
char message1[32];

 /* definicao do texto da mensagem : client1 <argumento>
  * se <argumento> e vazio, um valor default e enviado */

 if (argc > 1) strcpy(message1,argv[1]);
 else  strcpy(message1,"Texto da mensagem");

  /*   recuperacao do id da fila de mensagens do servidor   */
  if ((msgid_serv = msgget(MSGKEY,0)) == -1) {
        perror("Erro na recuperacao do ID da fila") ;
        exit(1) ;
  }
  /*   recuperacao do id da fila de mensagens do servidor   */
  if ((msgid_client = msgget(MSGKEY+1,0)) == -1) {
        perror("Erro na recuperacao do ID da fila") ;
        exit(1) ;
  }
```

```
  /*      envio de mensagem       */
  strcpy(msg.mtext,message1) ;
  msg.pid= getpid();
  msg.mtype = 1;
  printf("Cliente: envio da mensagem: %s\n",msg.mtext) ;
  if (msgsnd(msgid_serv,&msg,MSG_SIZEMAX,0) == -1) {
        perror("Erro no envio da mensagem") ;
        exit(1) ;
  }
  /*      recepcao de uma mensagem (ack de recepcao)       */
  printf("Cliente: espera de ack do servidor\n") ;
  if (msgrcv(msgid_client, &msg, MSG_SIZEMAX, 1, 0) == -1) {
        perror("Erro na recepcao da mensagem") ;
        exit(1) ;
  }
  printf("Cliente %d recebeu um ack do servidor %d\n",
          getpid(), msg.pid);
  exit(0);
}
```

Resultado da execução:

```
lyapunov:~> server2 &
[2] 3650
lyapunov:~> client2
Cliente: envio da mensagem: Texto da mensagem
O servidor 3650 recebeu uma mensagem do cliente 3651
Texto da mensagem: Texto da mensagem
Cliente: espera de ack do servidor
Cliente 3651 recebeu um ack do servidor 3650
lyapunov:~> client2 "NOVA MENSAGEM DE TEXTO!"
Cliente: envio da mensagem: NOVA MENSAGEM DE TEXTO!
O servidor 3650 recebeu uma mensagem do cliente 3654
Texto da mensagem: NOVA MENSAGEM DE TEXTO!
Cliente: espera de ack do servidor
Cliente 3654 recebeu um ack do servidor 3650
lyapunov:~>
```