Capítulo 9

Mecanismos de comunicação

Neste capítulo são apresentados alguns mecanismos de comunicação usados com frequência em sistemas operacionais, com ênfase em sistemas UNIX. Mais detalhes sobre estes e outros mecanismos podem ser obtidos em [Stevens, 1998; Robbins and Robbins, 2003]. Mecanismos de comunicação implementados nos sistemas Windows são apresentados em [Petzold, 1998; Hart, 2004].

9.1 Pipes

Um dos mecanismos de comunicação entre processos mais simples de usar no ambiente UNIX é o *pipe*, ou "cano". Um *pipe* é um canal de comunicação unidirecional entre dois processos. Na interface de linha de comandos UNIX, o *pipe* é frequentemente usado para conectar a saída padrão (*stdout*) de um processo à entrada padrão (*stdin*) de outro processo, permitindo assim a comunicação entre eles. A linha de comando a seguir traz um exemplo do uso de *pipes*:

```
1 $ who | grep marcos | sort
```

Esse comando lança simultaneamente os processos `who`, `grep` e `sort`, conectados por dois *pipes*. O comando `who` gera uma listagem de usuários conectados ao computador em sua saída padrão. O comando `grep marcos` é um filtro que lê as linhas de sua entrada padrão e envia para sua saída padrão somente as linhas contendo a *string* "`marcos`". O comando `sort` ordena a linhas recebidas em sua entrada padrão e as envia para sua saída padrão.

Ao associar esses comandos com *pipes*, é produzida uma lista ordenada das linhas de saída do comando `who` que contêm a *string* `marcos`, como mostra a figura 9.1. Deve-se observar que todos os processos envolvidos são lançados simultaneamente; suas ações são coordenadas pelo comportamento síncrono dos *pipes*.

O *pipe* pode ser classificado como um canal de comunicação local entre dois processos (1:1), unidirecional, síncrono, orientado a fluxo, confiável e com capacidade finita (os *pipes* do Linux armazenam 64 KBytes por default). O *pipe* é visto pelos processos como um arquivo, ou seja, o envio e a recepção de dados são feitos pelas chamadas de sistema `write` e `read`, como em arquivos normais[1].

[1] As funções `scanf`, `printf`, `fprintf` e congêneres normalmente usam as chamadas de sistema `read` e `write` em suas implementações.

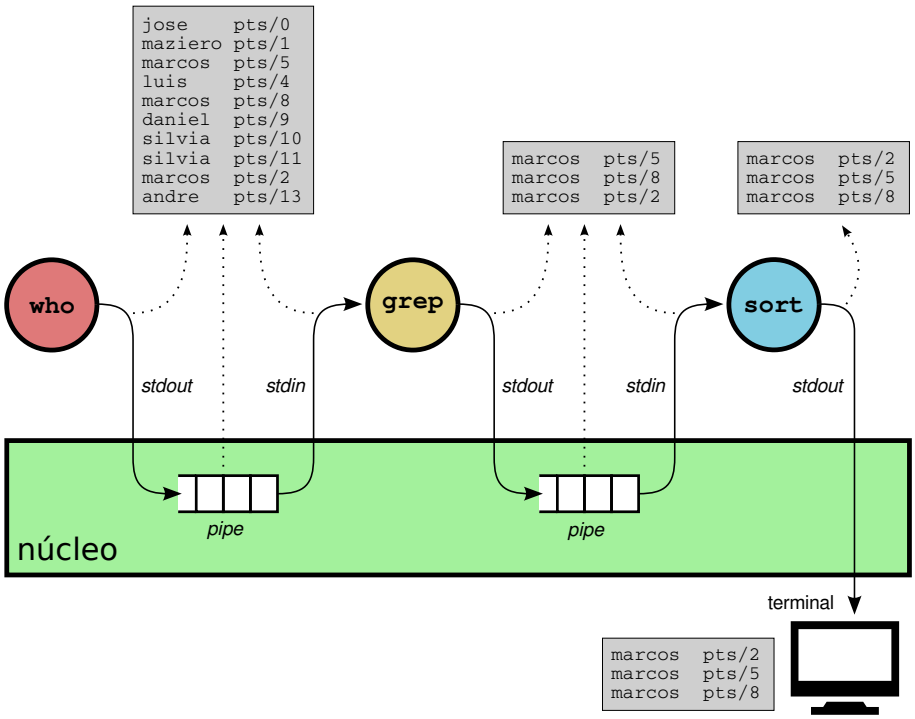

Figura 9.1: Comunicação através de *pipes*.

O uso de pipes na linha de comando UNIX é trivial, mas seu uso na construção de programas é um pouco mais complexo. Vários exemplos do uso de pipes UNIX na construção de programas são apresentados em [Robbins and Robbins, 2003].

Os *pipes* padrão têm vida curta: eles só existem durante a execução da linha de comando ou do processo que os criou, sendo destruídos logo em seguida. Por outro lado, os *pipes nomeados* (*named pipes*, ou *FIFOs*) permanecem desde sua criação até serem explicitamente destruídos ou o sistema ser encerrado. Um *pipe* nomeado é basicamente um *pipe* independente de processos e que tem um nome próprio, para que os processos interessados possam encontrá-lo. Esse nome é baseado na árvore de diretórios do sistema de arquivos, como se fosse um arquivo (mas ele não usa o disco).

Pipes nomeados podem ser criados na linha de comandos em Linux. No Windows, eles podem ser criados dentro de programas. A listagem a seguir apresenta um exemplo de criação, uso e remoção de um *pipe* nomeado usando comandos em Linux:

```
# cria um pipe nomeado, cujo nome é/tmp/pipe
$ mkfifo /tmp/pipe

# mostra o nome do pipe no diretório
$ ls -l /tmp/pipe
prw-rw-r-- 1 maziero maziero 0 sept. 6 18:14 pipe|

# envia dados (saída do comando date) para o pipe nomeado
$ date > /tmp/pipe

# EM OUTRO TERMINAL, recebe dados do pipe nomeado
$ cat < /tmp/pipe
Thu Sep 6 2018, 18:01:50 (UTC+0200)

# remove o pipe nomeado
$ rm /tmp/pipe
```

9.2 Filas de mensagens

As filas de mensagens são um bom exemplo de implementação do conceito de *mailbox* (vide Seção 8.3.6), permitindo o envio e recepção ordenada de mensagens tipadas entre processos em um sistema operacional. As filas de mensagens foram definidas inicialmente na implementação UNIX *System V*, sendo ainda suportadas pela maioria dos sistemas. O padrão *POSIX* também define uma interface para manipulação de filas de mensagens, sendo mais recente e de uso recomendado. Nos sistemas Windows, filas de mensagens podem ser criadas usando o mecanismo de *MailSlots* [Russinovich et al., 2008].

As filas de mensagens são mecanismos de comunicação entre vários processos (N:M ou N:1, dependendo da implementação), confiáveis, orientadas a mensagens e com capacidade finita. As operações de envio e recepção podem ser síncronas ou assíncronas, dependendo da implementação e a critério do programador.

As principais chamadas para usar filas de mensagens POSIX na linguagem C são:

- `mq_open`: abre uma fila já existente ou cria uma nova fila;
- `mq_setattr` e `mq_getattr`: permitem ajustar ou obter atributos (parâmetros) da fila, que definem seu comportamento, como o tamanho máximo da fila, o tamanho de cada mensagem, etc.;
- `mq_send`: envia uma mensagem para a fila; caso a fila esteja cheia, o emissor fica bloqueado até que alguma mensagem seja retirada da fila, abrindo espaço para o envio; a variante `mq_timedsend` permite definir um prazo máximo de espera: caso o envio não ocorra nesse prazo, a chamada retorna com erro;
- `mq_receive`: recebe uma mensagem da fila; caso a fila esteja vazia, o receptor é bloqueado até que surja uma mensagem para ser recebida; a variante `mq_timedreceive` permite definir um prazo máximo de espera;
- `mq_close`: fecha o descritor da fila criado por `mq_open`;

- `mq_unlink`: remove a fila do sistema, destruindo seu conteúdo.

A listagem a seguir implementa um "consumidor de mensagens", ou seja, um programa que cria uma fila para receber mensagens. O código apresentado segue o padrão *POSIX* (exemplos de uso de filas de mensagens no padrão *System V* estão disponíveis em [Robbins and Robbins, 2003]). Para compilá-lo em Linux é necessário efetuar a ligação com a biblioteca de tempo real *POSIX* (usando a opção `-lrt`).

```
1  // Arquivo mqrecv.c: recebe mensagens de uma fila de mensagens POSIX.
2  // Em Linux, compile usando: cc -Wall mqrecv.c -o mqrecv -lrt
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <mqueue.h>
7  #include <sys/stat.h>
8
9  #define QUEUE "/my_queue"
10
11 int main (int argc, char *argv[])
12 {
13   mqd_t queue ;               // descritor da fila de mensagens
14   struct mq_attr attr ;       // atributos da fila de mensagens
15   int  msg ;                  // as mensagens são números inteiros
16
17   // define os atributos da fila de mensagens
18   attr.mq_maxmsg = 10 ;       // capacidade para 10 mensagens
19   attr.mq_msgsize = sizeof(msg) ; // tamanho de cada mensagem
20   attr.mq_flags = 0 ;
21
22   // abre ou cria a fila com permissoes 0666
23   if ((queue = mq_open (QUEUE, O_RDWR|O_CREAT, 0666, &attr)) < 0)
24   {
25     perror ("mq_open") ;
26     exit (1) ;
27   }
28
29   // recebe cada mensagem e imprime seu conteudo
30   for (;;)
31   {
32     if ((mq_receive (queue, (void*) &msg, sizeof(msg), 0)) < 0)
33     {
34       perror("mq_receive:") ;
35       exit (1) ;
36     }
37     printf ("Received msg value %d\n", msg) ;
38   }
39 }
```

A listagem a seguir implementa o programa produtor das mensagens consumidas pelo programa anterior. Vários produtores e consumidores de mensagens podem operar sobre uma mesma fila, mas os produtores de mensagens devem ser lançados após um consumidor, pois é este último quem cria a fila (neste código de exemplo).

```c
// Arquivo mqsend.c: envia mensagens para uma fila de mensagens POSIX.
// Em Linux, compile usando: cc -Wall mqsend.c -o mqsend -lrt

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

#define QUEUE "/my_queue"

int main (int argc, char *argv[])
{
  mqd_t queue ;      // descritor da fila
  int   msg ;        // mensagem a enviar

  // abre a fila de mensagens, se existir
  if((queue = mq_open (QUEUE, O_RDWR)) < 0)
  {
    perror ("mq_open") ;
    exit (1) ;
  }

  for (;;)
  {
    msg = random() % 100 ; // valor entre 0 e 99

    // envia a mensagem
    if (mq_send (queue, (void*) &msg, sizeof(msg), 0) < 0)
    {
      perror ("mq_send") ;
      exit (1) ;
    }
    printf ("Sent message with value %d\n", msg) ;
    sleep (1) ;
  }
}
```

Deve-se observar que o arquivo /my_queue referenciado em ambas as listagens serve unicamente como identificador comum para a fila de mensagens; nenhum arquivo de dados com esse nome será criado pelo sistema. As mensagens não transitam por arquivos, apenas pela memória do núcleo. Referências de recursos através de nomes de arquivos são frequentemente usadas para identificar vários mecanismos de comunicação e coordenação em UNIX, como filas de mensagens, semáforos e áreas de memória compartilhadas (vide Seção 9.3).

9.3 Memória compartilhada

A comunicação entre tarefas situadas em processos distintos deve ser feita através do núcleo, usando chamadas de sistema. Não existe a possibilidade de acesso a variáveis comuns a ambos, pois suas áreas de memória são distintas e isoladas. A comunicação através do núcleo pode ser ineficiente caso seja frequente e o volume de dados a transferir seja elevado, por causa das trocas de contexto envolvidas nas chamadas de sistema. Para essas situações, seria conveniente ter uma área de memória

comum que possa ser acessada direta e rapidamente pelos processos interessados, sem o custo da intermediação do núcleo.

A maioria dos sistemas operacionais atuais oferece mecanismos para o compartilhamento de áreas de memória entre processos (*shared memory areas*). As áreas de memória compartilhadas e os processos que as acessam são gerenciados pelo núcleo, mas o acesso ao conteúdo de cada área é feito diretamente pelos processos, sem intermediação do núcleo.

A criação e uso de uma área de memória compartilhada entre dois processos p_a e p_b em um sistema UNIX pode ser resumida na seguinte sequência de passos, ilustrada na Figura 9.2:

1. O processo p_a solicita ao núcleo a criação de uma área de memória compartilhada;
2. o núcleo aloca uma nova área de memória e a registra em uma lista de áreas compartilháveis;
3. o núcleo devolve ao processo p_a o identificador (*id*) da área alocada;
4. o processo p_a solicita ao núcleo que a área identificada por *id* seja anexada ao seu espaço de endereçamento;
5. o núcleo modifica a configuração de memória do processo p_a para incluir a área indicada por *id* em seu espaço de endereçamento;
6. o núcleo devolve a p_a um ponteiro para a área alocada;
7. O processo p_b executa os passos 4-6 e também recebe um ponteiro para a área alocada;
8. Os processos p_a e p_b comunicam através de escritas e leituras de valores na área de memória compartilhada.

Deve-se observar que, ao solicitar a criação da área de memória compartilhada, p_a define as permissões de acesso à mesma; por isso, o pedido de anexação da área de memória feito por p_b pode ser recusado pelo núcleo, se violar as permissões definidas por p_a.

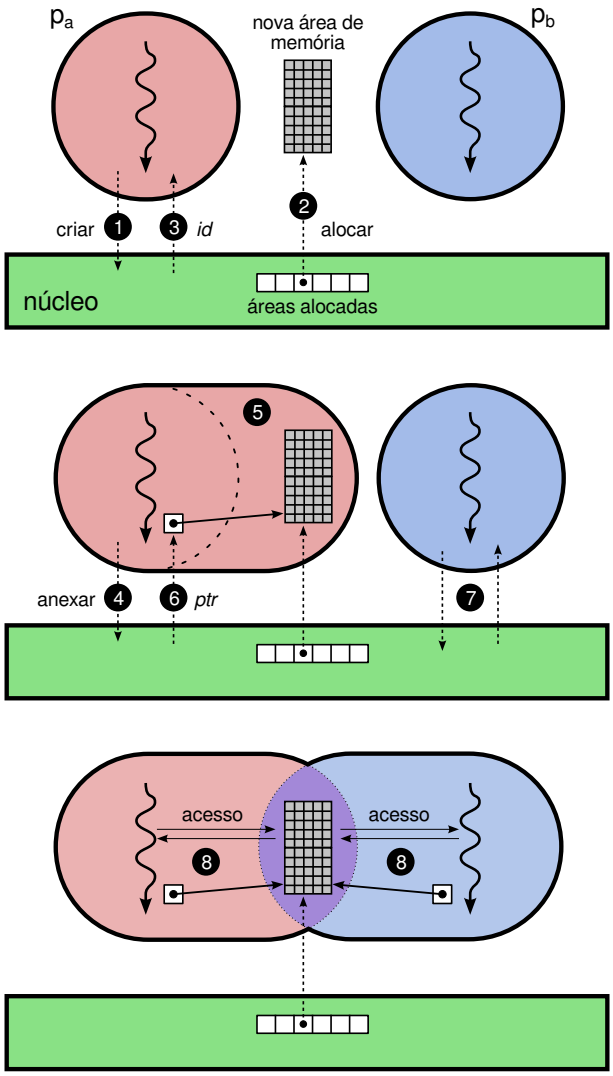

Figura 9.2: Criação e uso de uma área de memória compartilhada.

A Listagem 9.1 exemplifica a criação e uso de uma área de memória compartilhada, usando o padrão POSIX (exemplos de implementação no padrão *System V* podem ser encontrados em [Robbins and Robbins, 2003]). Para compilá-lo em Linux é necessário efetuar a ligação com a biblioteca de tempo real *POSIX*, usando a opção `-lrt`. Para melhor observar seu funcionamento, devem ser lançados dois ou mais processos executando esse código simultaneamente.

Deve-se observar que não existe nenhuma forma de coordenação ou sincronização implícita no acesso à área de memória compartilhada. Assim, dois processos podem escrever sobre os mesmos dados simultaneamente, levando a possíveis inconsistências. Por essa razão, mecanismos de coordenação adicionais (como os apresentados no Capítulo 10) podem ser necessários para garantir a consistência dos dados armazenados em áreas compartilhadas.

Listing 9.1: Memória Compartilhada

```
1  // Arquivo shm.c: cria e usa uma área de memória compartilhada POSIX.
2  // Em Linux, compile usando: cc -Wall shm.c -o shm -lrt
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <fcntl.h>
7  #include <unistd.h>
8  #include <sys/types.h>
9  #include <sys/stat.h>
10 #include <sys/mman.h>
11
12 int main (int argc, char *argv[])
13 {
14   int fd, value, *ptr ;
15
16   // Passos 1 a 3: abre/cria uma area de memoria compartilhada
17   fd = shm_open ("/sharedmem", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR) ;
18   if (fd == -1) {
19     perror ("shm_open") ;
20     exit (1) ;
21   }
22
23   // ajusta o tamanho da area compartilhada para sizeof (value)
24   if (ftruncate (fd, sizeof (value)) == -1) {
25     perror ("ftruncate") ;
26     exit (1) ;
27   }
28
29   // Passos 4 a 6: mapeia a area no espaco de enderecamento deste processo
30   ptr = mmap (NULL, sizeof(value), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0) ;
31   if (ptr == MAP_FAILED) {
32     perror ("mmap") ;
33     exit (1) ;
34   }
35
36   for (;;) {
37     // Passo 8: escreve um valor aleatorio na area compartilhada
38     value = random () % 1000 ;
39     (*ptr) = value ; // escreve na area
40     printf ("Wrote value %i\n", value) ;
41     sleep (1) ;
42
43     // Passo 8: le e imprime o conteudo da area compartilhada
44     value = (*ptr) ; // le da area
45     printf ("Read value %i\n", value) ;
46     sleep (1) ;
47   }
48 }
```

Exercícios

1. Classifique as filas de mensagens UNIX de acordo com os tipos de comunicação discutidos no Capítulo 8.

2. Classifique os *pipes* UNIX de acordo com os tipos de comunicação discutidos no Capítulo 8.

3. Classifique as áreas de memória compartilhadas de acordo com os tipos de comunicação discutidos no Capítulo 8.

4. Sobre as afirmações a seguir, relativas aos mecanismos de comunicação, indique quais são incorretas, justificando sua resposta:

   (a) As filas de mensagens POSIX são um exemplo de canal de comunicação com capacidade nula.

   (b) A memória compartilhada provê mecanismos de sincronização para facilitar a comunicação entre os processos.

   (c) A troca de dados através de memória compartilhada é mais adequada para a comunicação em rede.

   (d) Processos que se comunicam por memória compartilhada podem acessar a mesma área da RAM.

   (e) Os pipes Unix são um bom exemplo de comunicação M:N.

   (f) A comunicação através de memória compartilhada é particularmente indicada para compartilhar grandes volumes de dados entre dois ou mais processos.

   (g) As filas de mensagens POSIX são um bom exemplo de canal de eventos.

   (h) Nas filas de mensagens POSIX, as mensagens transitam através de arquivos em disco criados especialmente para essa finalidade.

   (i) Em UNIX, um *pipe* é um canal de comunicação unidirecional que liga a saída padrão de um processo à entrada padrão de outro.

Referências

J. Hart. *Windows System Programming, 3rd edition*. Addison-Wesley Professional, 2004.

C. Petzold. *Programming Windows, 5th edition*. Microsoft Press, 1998.

K. Robbins and S. Robbins. *UNIX Systems Programming*. Prentice-Hall, 2003.

M. Russinovich, D. Solomon, and A. Ionescu. *Microsoft Windows Internals, Fifth Edition*. Microsoft Press, 2008.

R. Stevens. *UNIX Network Programming*. Prentice-Hall, 1998.