

Sistemas Operacionais

III - Comunicação entre Tarefas ^{*†}

Prof. Carlos Alberto Maziero
PPGIA CCET PUCPR
<http://www.ppgia.pucpr.br/~maziero>

30 de julho de 2008

Resumo

Muitas implementações de sistemas complexos são estruturadas como várias tarefas inter-dependentes, que cooperam entre si para atingir os objetivos da aplicação, como por exemplo em um navegador Web. Para que as várias tarefas que compõem uma aplicação possam cooperar, elas precisam comunicar informações umas às outras e coordenar suas atividades, para garantir que os resultados obtidos sejam coerentes. Este módulo apresenta os principais conceitos, problemas e soluções referentes à comunicação entre tarefas.

^{*}Copyright (c) 2006 Carlos Alberto Maziero. É garantida a permissão para copiar, distribuir e/ou modificar este documento sob os termos da Licença de Documentação Livre GNU (*GNU Free Documentation License*), Versão 1.2 ou qualquer versão posterior publicada pela *Free Software Foundation*. A licença está disponível em <http://www.gnu.org/licenses/gfdl.txt>.

[†]Este texto foi produzido usando exclusivamente software livre: Sistema Operacional *Linux* (distribuições *Fedora* e *Ubuntu*), compilador de texto \LaTeX , gerenciador de referências *BibTeX*, editor gráfico *Inkscape*, criador de gráficos *GNUPlot* e processador PS/PDF *GhostScript*, entre outros.

Sumário

1	Objetivos	3
2	Escopo da comunicação	3
3	Formas de comunicação	4
3.1	Comunicação direta ou indireta	5
3.2	Sincronismo	5
3.3	Formato de envio	6
3.4	Capacidade dos canais	8
3.5	Número de participantes	9
4	Exemplos de mecanismos de comunicação	10
4.1	UNIX Message queues	11
4.2	Pipes	13
4.3	Memória compartilhada	14

1 Objetivos

Nem sempre um programa seqüencial é a melhor solução para um determinado problema. Muitas vezes, as implementações são estruturadas na forma de várias tarefas inter-dependentes que cooperam entre si para atingir os objetivos da aplicação, como por exemplo em um navegador Web. Existem várias razões para justificar a construção de sistemas baseados em tarefas cooperantes, entre as quais podem ser citadas:

Atender vários usuários simultâneos : um servidor de banco de dados ou de e-mail completamente seqüencial atenderia um único cliente por vez, gerando atrasos intoleráveis para os demais clientes. Por isso, servidores de rede são implementados com vários processos ou threads, para atender simultaneamente todos os usuários conectados.

Uso de computadores multi-processador : um programa seqüencial executa um único fluxo de instruções por vez, não importando o número de processadores presentes no hardware. Para aumentar a velocidade de execução de uma aplicação, esta deve ser “quebrada” em várias tarefas cooperantes, que poderão ser escalonadas simultaneamente nos processadores disponíveis.

Modularidade : um sistema muito grande e complexo pode ser melhor organizado dividindo suas atribuições em módulos sob a responsabilidade de tarefas inter-dependentes. Cada módulo tem suas próprias responsabilidades e coopera com os demais módulos quando necessário. Sistemas de interface gráfica, como os projetos *Gnome* [Gnome, 2005] e *KDE* [KDE, 2005], são geralmente construídos dessa forma.

Construção de aplicações interativas : navegadores Web, editores de texto e jogos são exemplos de aplicações com alta interatividade; nelas, tarefas associadas à interface reagem a comandos do usuário, enquanto outras tarefas comunicam através da rede, fazem a revisão ortográfica do texto, renderizam imagens na janela, etc. Construir esse tipo de aplicação de forma totalmente seqüencial seria simplesmente inviável.

Para que as tarefas presentes em um sistema possam cooperar, elas precisam **comunicar**, compartilhando as informações necessárias à execução de cada tarefa, e **coordenar** suas atividades, para que os resultados obtidos sejam consistentes (sem erros). Este módulo visa estudar os principais conceitos, problemas e soluções empregados para permitir a comunicação entre tarefas executando em um sistema.

2 Escopo da comunicação

Tarefas cooperantes precisam trocar informações entre si. Por exemplo, a tarefa que gerencia os botões e menus de um navegador Web precisa informar rapidamente

as demais tarefas caso o usuário clique nos botões *stop* ou *reload*. Outra situação de comunicação freqüente ocorre quando o usuário seleciona um texto em uma página da Internet e o arrasta para um editor de textos. Em ambos os casos ocorre a transferência de informação entre duas tarefas distintas.

Implementar a comunicação entre tarefas pode ser simples ou complexo, dependendo da situação. Se as tarefas estão no mesmo processo, elas compartilham a mesma área de memória e a comunicação pode então ser implementada facilmente, usando variáveis globais comuns. Entretanto, caso as tarefas pertençam a processos distintos, não existem variáveis compartilhadas; neste caso, a comunicação tem de ser feita por intermédio do núcleo do sistema operacional, usando chamadas de sistema. Caso as tarefas estejam em computadores distintos, o núcleo deve implementar mecanismos de comunicação específicos, fazendo uso do suporte de rede disponível. A figura 1 ilustra essas três situações.

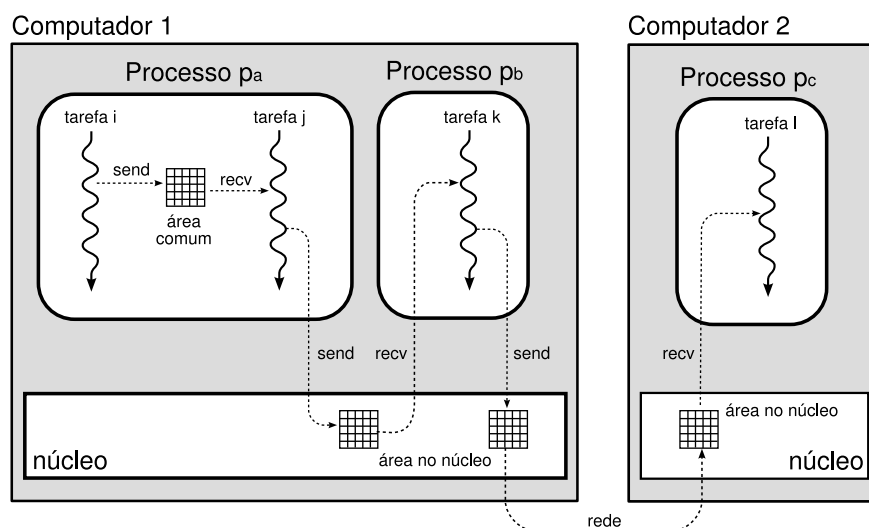


Figura 1: Comunicação intra-processo ($t_i \rightarrow t_j$), inter-processos ($t_j \rightarrow t_k$) e inter-sistemas ($t_k \rightarrow t_l$).

3 Formas de comunicação

A implementação da comunicação entre tarefas pode ocorrer de várias formas. Ao definir os mecanismos de comunicação oferecidos por um sistema operacional, seus projetistas devem considerar muitos aspectos, como o formato dos dados a transferir, o sincronismo exigido nas comunicações, a necessidade de *buffers* e o número de emissores/receptores envolvidos em cada ação de comunicação. Um termo muito empregado para designar genericamente os mecanismos de comunicação é **IPC - Inter-Process Communication**.

3.1 Comunicação direta ou indireta

De forma mais abstrata, a comunicação entre tarefas pode ser implementada por duas primitivas básicas: *enviar (dados, destino)*, que envia os dados relacionados ao destino indicado, e *receber (dados, origem)*, que recebe os dados previamente enviados pela origem indicada. Essa abordagem, na qual o emissor identifica claramente o receptor e vice-versa, é denominada **comunicação direta**.

Poucos sistemas empregam a comunicação direta; na prática são utilizados mecanismos de **comunicação indireta**, por serem mais flexíveis. Na comunicação indireta, emissor e receptor não precisam se conhecer, pois não interagem diretamente entre si. Eles se relacionam através de um **canal de comunicação**, que é criado pelo sistema operacional, geralmente a pedido de uma das partes. Assim, as primitivas de comunicação não designam diretamente tarefas, mas canais de comunicação aos quais as tarefas estão associadas: *enviar (dados, canal)* e *receber (dados, canal)*.

3.2 Sincronismo

Em relação aos aspectos de sincronismo do canal de comunicação, a comunicação entre tarefas pode ser:

Síncrona : quando as operações de envio e recepção de dados bloqueiam (suspendem) as tarefas envolvidas até a conclusão da comunicação: o emissor será bloqueado até que a informação seja recebida pelo receptor, e vice-versa. A figura 2 apresenta os diagramas de tempo de duas situações frequentes em sistemas com comunicação síncrona.

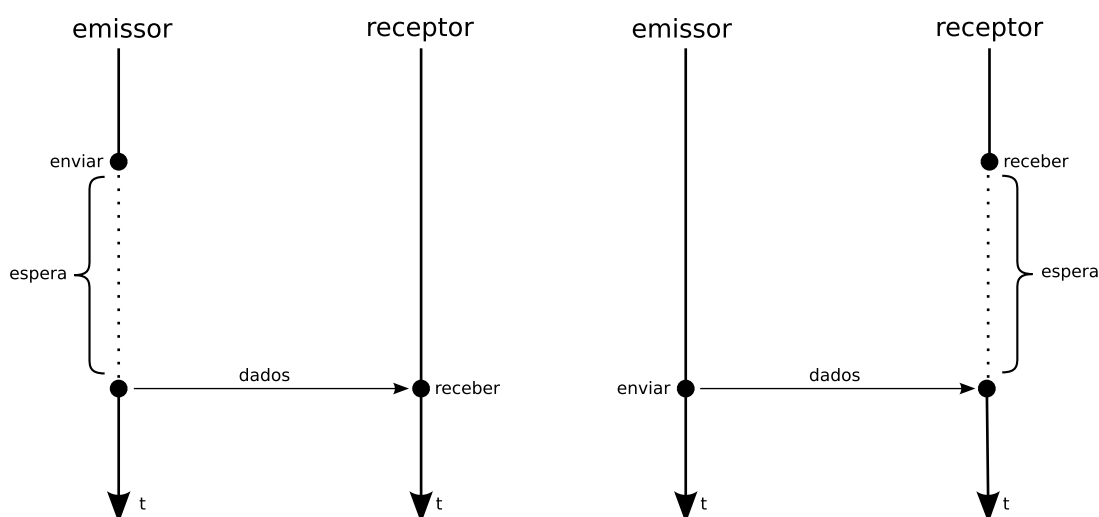


Figura 2: Comunicação síncrona.

Assíncrona : em um sistema com comunicação assíncrona, as primitivas de envio e recepção não são bloqueantes: caso a comunicação não seja possível no momento

em que cada operação é invocada, esta retorna imediatamente com uma indicação de erro. Deve-se observar que, caso o emissor e o receptor operem ambos de forma assíncrona, torna-se necessário criar um *buffer* para armazenar os dados da comunicação entre eles. Sem esse *buffer*, a comunicação se tornará inviável, pois raramente ambos estarão prontos para comunicar ao mesmo tempo. Esta forma de comunicação esta representada no diagrama de tempo da figura 3.

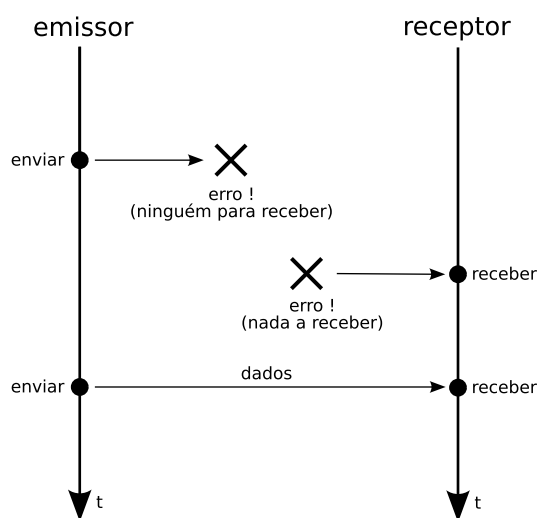


Figura 3: Comunicação assíncrona.

Semi-síncrona : primitivas de comunicação semi-síncronas têm um comportamento síncrono (bloqueante) durante um prazo pré-definido. Caso o prazo se esgote sem que a comunicação tenha ocorrido, a primitiva retorna com uma indicação de erro. Para refletir esse comportamento, as primitivas de comunicação recebem um parâmetro adicional : *enviar (dados, destino, prazo)* e *receber (dados, origem, prazo)*. A figura 4 ilustra duas situações em que ocorre esse comportamento.

3.3 Formato de envio

A informação enviada pelo emissor ao receptor pode ser vista basicamente de duas formas: como uma **seqüência de mensagens** independentes, cada uma com seu próprio conteúdo, ou como um **fluxo seqüencial** e contínuo de dados, imitando o comportamento de um arquivo com acesso seqüencial.

Na abordagem baseada em mensagens, cada mensagem consiste de um pacote de dados que pode ser tipado ou não. Esse pacote é recebido ou descartado pelo receptor em sua íntegra; não existe a possibilidade de receber “meia mensagem” (figura 5). Exemplos de sistema de comunicação orientados a mensagens incluem as *message queues* do UNIX e os protocolos de rede IP e UDP, apresentados na seção 4.

Caso a comunicação seja definida como um fluxo contínuo de dados, o canal de comunicação é visto como o equivalente a um arquivo: o emissor “escreve” dados nesse

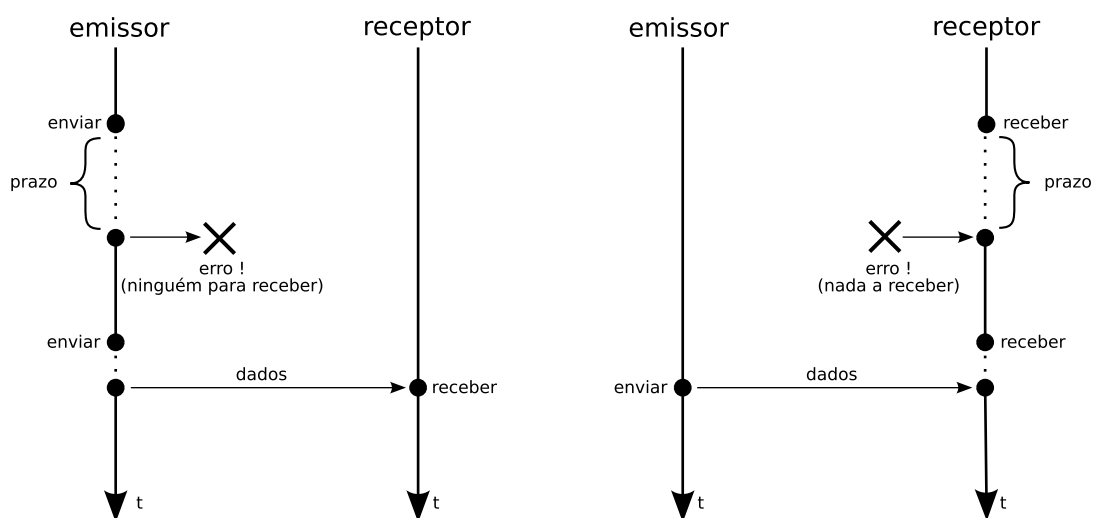


Figura 4: Comunicação semi-síncrona.

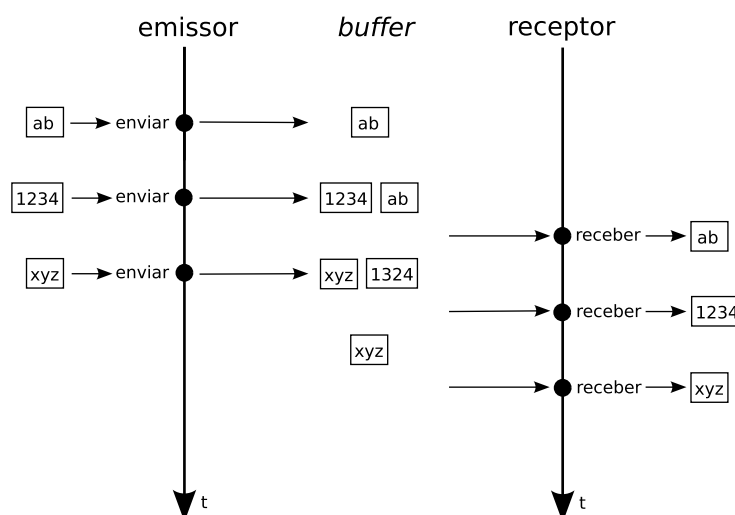


Figura 5: Comunicação baseada em mensagens.

canal, que serão “lidos” pelo receptor respeitando a ordem de envio dos dados. Não há separação lógica entre os dados enviados em operações separadas: eles podem ser lidos byte a byte ou em grandes blocos a cada operação de recepção, a critério do receptor. A figura 6 apresenta o comportamento dessa forma de comunicação.

Exemplos de sistemas de comunicação orientados a fluxo de dados incluem os *pipes* do UNIX e o protocolo de rede TCP/IP (este último é normalmente classificado como *orientado a conexão*, com o mesmo significado). Nestes dois exemplos a analogia com o conceito de arquivos é tão forte que os canais de comunicação são identificados por descritores de arquivos e as chamadas de sistema *read* e *write* (normalmente usadas com arquivos) são usadas para enviar e receber os dados. Esses exemplos são apresentados em detalhe na seção 4.

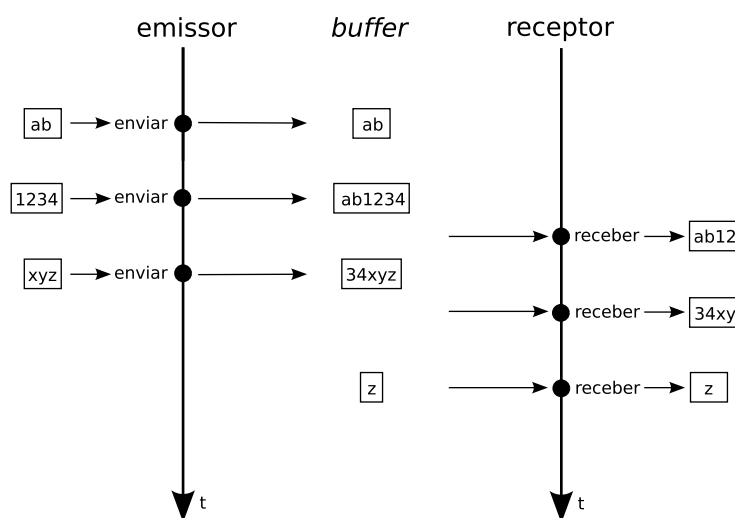


Figura 6: Comunicação baseada em fluxo de dados.

3.4 Capacidade dos canais

O comportamento síncrono ou assíncrono de um canal de comunicação pode ser afetado pela presença de *buffers* que permitam armazenar temporariamente os dados em trânsito, ou seja, as informações enviadas pelo emissor e que ainda não foram recebidas pelo receptor. Em relação à capacidade de *buffering* do canal de comunicação, três situações devem ser analisadas:

Capacidade nula ($n = 0$) : neste caso, o canal não pode armazenar dados; a comunicação é feita por transferência direta dos dados do emissor para o receptor, sem cópias intermediárias. Caso a comunicação seja síncrona, o emissor permanece bloqueado até que o destinatário receba os dados, e vice-versa. Essa situação específica (comunicação síncrona com canais de capacidade nula) implica em uma forte sincronização entre as partes, sendo por isso denominada *Rendez-Vous* (termo francês para “encontro”). A figura 2 ilustra dois casos de *Rendez-Vous*. Por outro lado, a comunicação assíncrona torna-se inviável usando canais de capacidade nula (conforme discutido na seção 3.2).

Capacidade infinita ($n = \infty$) : o emissor sempre pode enviar dados, que serão armazenados no *buffer* do canal enquanto o receptor não os consumir. Obviamente essa situação não existe na prática, pois todos os sistemas de computação têm capacidade de memória e de armazenamento finitas. No entanto, essa simplificação é útil no estudo dos algoritmos de comunicação e sincronização, pois torna menos complexas a modelagem e análise dos mesmos.

Capacidade finita ($0 < n < \infty$) : neste caso, uma quantidade finita (n) de dados pode ser enviada pelo emissor sem que o receptor os consuma. Todavia, ao tentar enviar dados em um canal já saturado, o emissor poderá ficar bloqueado até surgir

espaço no *buffer* do canal e conseguir enviar (comportamento síncrono) ou receber um retorno indicando o erro (comportamento assíncrono). A maioria dos sistemas reais opera com canais de capacidade finita.

Para exemplificar esse conceito, a figura 7 apresenta o comportamento de duas tarefas trocando dados através de um canal de comunicação com capacidade para duas mensagens e comportamento síncrono (bloqueante).

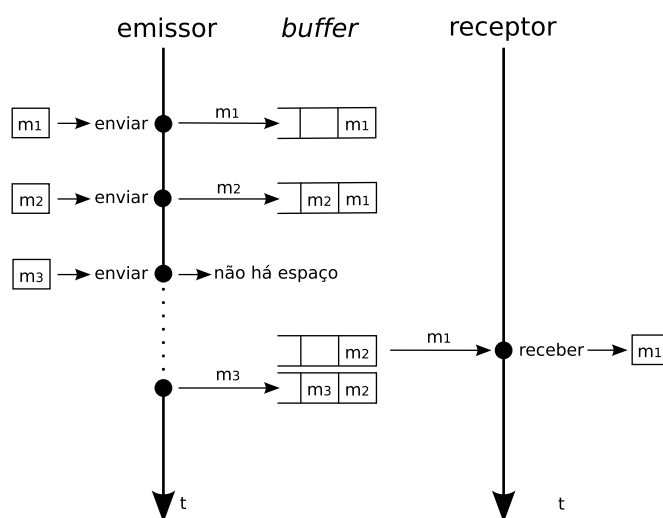


Figura 7: Comunicação síncrona usando um canal com capacidade 2.

3.5 Número de participantes

Nas situações de comunicação apresentadas até agora, cada canal de comunicação envolve apenas um emissor e um receptor. No entanto, existem situações em que uma tarefa necessita comunicar com várias outras, como por exemplo em sistemas de *chat* ou mensagens instantâneas (IM – *Instant Messaging*). Dessa forma, os mecanismos de comunicação também podem ser classificados de acordo com o número de tarefas participantes:

1:1 : quando exatamente um emissor e um receptor interagem através do canal de comunicação; é a situação mais frequente, implementada por exemplo nos *pipes* e no protocolo TCP.

M:N : quando um ou mais emissores enviam mensagens para um ou mais receptores. Duas situações distintas podem se apresentar neste caso:

- Cada mensagem é recebida por **apenas um receptor** (em geral aquele que pedir primeiro); neste caso a comunicação continua sendo ponto-a-ponto,

através de um canal compartilhado. Essa abordagem é conhecida como *mailbox* (figura 8), sendo implementada nas *message queues* UNIX e nos *sockets* do protocolo UDP. Na prática, o *mailbox* funciona como um *buffer* de dados, no qual os emissores depositam mensagens e os receptores as consomem.

- Cada mensagem é recebida por **todos os receptores** (cada receptor recebe uma cópia da mensagem). Essa abordagem é conhecida pelos nomes de *difusão* (*multicast*) ou *canal de eventos* (figura 9), sendo implementada por exemplo no protocolo UDP.

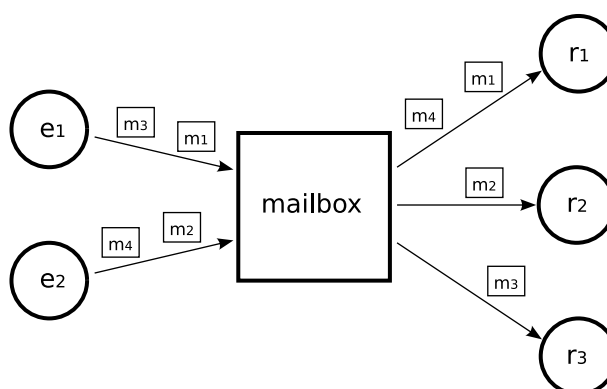


Figura 8: Comunicação M:N através de um *mailbox*.

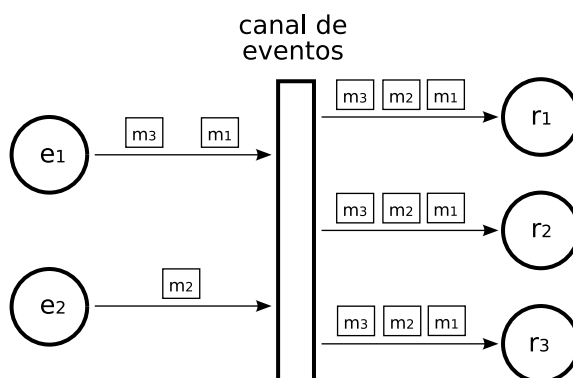


Figura 9: Difusão através de um canal de eventos.

4 Exemplos de mecanismos de comunicação

Nesta seção serão apresentados alguns mecanismos de comunicação usados com frequência 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].

4.1 UNIX Message queues

As filas de mensagens foram definidas inicialmente na implementação UNIX *System V*, sendo atualmente suportadas pela maioria dos sistemas. Além do padrão *System V*, o padrão *Posix* também define uma interface para manipulação de filas de mensagens. Esse mecanismo é um bom exemplo de implementação do conceito de *mailbox*, permitindo o envio e recepção ordenada de mensagens tipadas entre processos locais. As operações de envio e recepção podem ser síncronas ou assíncronas, a critério do programador.

As principais chamadas para uso de filas de mensagens POSIX são:

- `mq_open`: abre uma fila já existente ou cria uma nova fila;
- `mq_setattr` e `mq_getattr`: permitem ajustar ou obter atributos 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 o biblioteca de tempo-real *POSIX*, usando a opção `-lrt`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mqueue.h>
4 #include <sys/stat.h>
5
6 #define QUEUE "/my_queue"
7
8 int main (int argc, char *argv[])
9 {
10     mqd_t queue;           // descritor da fila de mensagens
11     struct mq_attr attr;   // atributos da fila de mensagens
12     int msg ;              // mensagens contendo um inteiro
13 }
```

```
14 // define os atributos da fila de mensagens
15 attr.mq_maxmsg = 10 ; // capacidade para 10 mensagens
16 attr.mq_msgsize = sizeof(msg) ; // tamanho de cada mensagem
17 attr.mq_flags = 0 ;
18
19 umask (0) ; // mascara de permissoes (umask)
20
21 // caso a fila exista, remove-a para destruir seu conteudo antigo
22 mq_unlink (QUEUE) ;
23
24 // abre ou cria a fila com permissoes 0666
25 if ( (queue = mq_open (QUEUE, O_RDWR|O_CREAT, 0666, &attr)) == -1)
26 {
27     perror ("mq_open");
28     exit (1);
29 }
30
31 // recebe cada mensagem e imprime seu conteudo
32 while (1)
33 {
34     if ( (mq_receive (queue, (void*) &msg, sizeof(msg), 0)) == -1)
35     {
36         perror("mq_receive:") ;
37         exit (1) ;
38     }
39     printf ("Received msg value %d\n", msg);
40 }
41 }
```

A listagem a seguir implementa o programa produtor das mensagens consumidas pelo programa anterior:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mqueue.h>
4 #include <unistd.h>
5
6 #define QUEUE "/my_queue"
7
8 int main (int argc, char *argv[])
9 {
10     mqd_t queue; // descritor da fila
11     int msg; // mensagem a enviar
12
13     // abre a fila de mensagens, se existir
14     if( (queue = mq_open (QUEUE, O_RDWR)) == -1)
15     {
16         perror ("mq_open");
17         exit (1);
18     }
19
20     while (1)
```

```
21 {
22     msg = random() % 100 ; // valor entre 0 e 99
23
24     // envia a mensagem
25     if ( mq_send (queue, (void*) &msg, sizeof(msg), 0) == -1)
26     {
27         perror ("mq_send");
28         exit (1);
29     }
30     printf ("Sent message with value %d\n", msg);
31     sleep (1) ;
32 }
33 }
```

Deve-se observar que o arquivo /fila referenciado em ambas as listagens serve unicamente como identificador único para a fila de mensagens; nenhum arquivo com esse nome será criado pelo sistema. As mensagens não transitam por arquivos, apenas pela memória do núcleo. Referências de recursos na forma de nomes de arquivos são usadas para identificar vários mecanismos de comunicação e coordenação em UNIX, como filas de mensagens, semáforos e áreas de memória compartilhadas (vide seção 4.3).

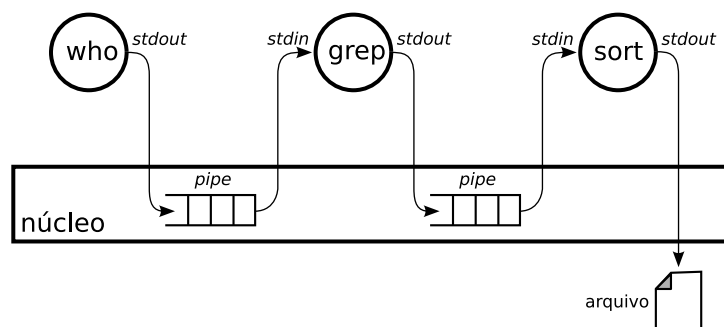
4.2 Pipes

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

```
# who | grep marcos | sort > login-marcos.txt
```

A saída do comando *who* é uma listagem de usuários conectados ao computador. Essa saída é encaminhada através de um *pipe* (indicado pelo caractere “|”) ao comando *grep*, que a filtra e gera como saída somente as linhas contendo a string “marcos”. Essa saída é encaminhada através de outro *pipe* ao comando *sort*, que ordena a listagem recebida e a deposita no arquivo *login-marcos.txt*. Deve-se observar que todos os processos envolvidos são lançados simultaneamente; suas ações são coordenadas pelo comportamento síncrono dos *pipes*. A figura 10 detalha essa sequência de ações.

O *pipe* é um canal de comunicação unidirecional entre dois processos (1:1), com capacidade finita (os *pipes* do Linux armazenam 4 KBytes por default), visto pelos processos como um arquivo, ou seja, a comunicação que ele oferece é baseada em fluxo. O envio e recepção de dados são feitos pelas chamadas de sistema *write* e *read*, que podem operar em modo síncrono (bloqueante, por default) ou assíncrono.

Figura 10: Comunicação através de *pipes*.

O uso de pipes na linha de comando é simples, mas seu uso na construção de programas pode ser complexo. Vários exemplos do uso de pipes UNIX na construção de programas são apresentados em [Robbins and Robbins, 2003].

4.3 Memória compartilhada

A comunicação entre tarefas situadas em processos distintos deve ser feita através do núcleo, usando chamadas de sistema, porque não existe a possibilidade de acesso a variáveis comuns a ambos. No entanto, essa abordagem pode ser ineficiente caso a comunicação seja muito volumosa e freqüente, ou envolva muitos processos. 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 pelo 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 utilizam são gerenciados pelo núcleo, mas o acesso ao conteúdo de cada área é feito diretamente pelos processos, sem intermediação ou coordenação do núcleo. Por essa razão, mecanismos de coordenação (apresentados na seção ??) podem ser necessários para garantir a consistência dos dados armazenados nessas áreas.

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 figura 11:

1. O processo p_a solicita ao núcleo a criação de uma área de memória compartilhada, informando o tamanho e as permissões de acesso; o retorno dessa operação é um identificador (*id*) da área criada.
2. O processo p_a solicita ao núcleo que a área recém-criada seja anexada ao seu espaço de endereçamento; esta operação retorna um ponteiro para a nova área de memória, que pode então ser acessada pelo processo.

3. O processo p_b obtém o identificador id da área de memória criada por p_a .
4. O processo p_b solicita ao núcleo que a área de memória seja anexada ao seu espaço de endereçamento e recebe um ponteiro para o acesso à mesma.
5. Os processos p_a e p_b acessam a área de memória compartilhada através dos ponteiros informados pelo núcleo.

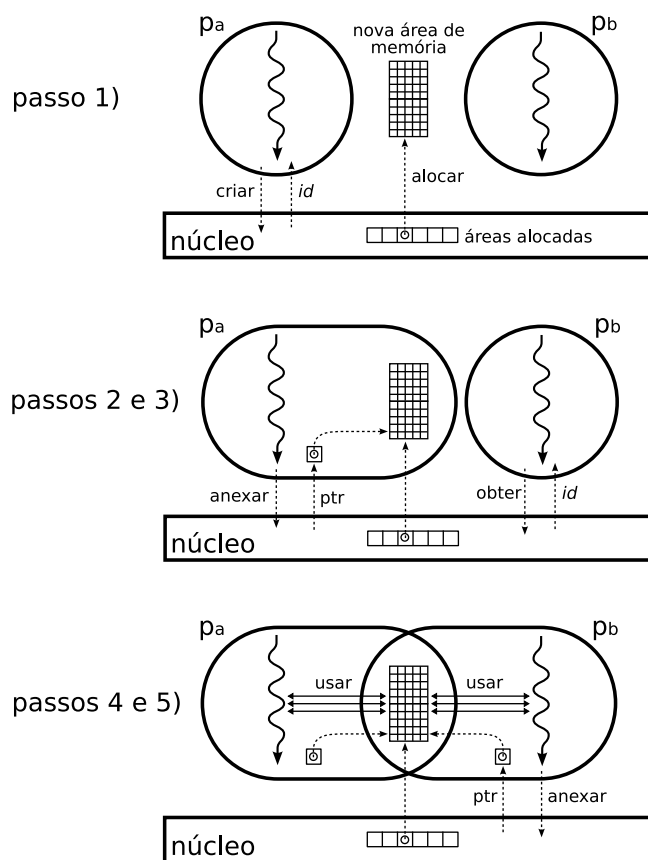


Figura 11: Criação e uso de uma área de memória compartilhada.

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 . A listagem a seguir 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.

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


Questões

1. Quais são as vantagens e desvantagens das abordagens a seguir, sob as óticas do sistema operacional e do programador de aplicativos?
 - (a) comunicação bloqueante ou não-bloqueante
 - (b) canais com *buffering* ou sem *buffering*
 - (c) comunicação por mensagens ou por fluxo
 - (d) mensagens de tamanho fixo ou variável
 - (e) comunicação 1:1 ou M:N
2. Explique como processos que comunicam por troca de mensagens se comportam em relação à capacidade do canal de comunicação, considerando as semânticas de chamada síncrona e assíncrona.

Exercícios

1. ...

Projetos

1. ...

Referências

- [Gnome, 2005] Gnome (2005). Gnome: the free software desktop project. <http://www.gnome.org>.
- [Hart, 2004] Hart, J. (2004). *Windows System Programming, 3rd edition*. Addison-Wesley Professional.
- [KDE, 2005] KDE (2005). KDE desktop project. <http://www.kde.org>.
- [Petzold, 1998] Petzold, C. (1998). *Programming Windows, 5th edition*. Microsoft Press.
- [Robbins and Robbins, 2003] Robbins, K. and Robbins, S. (2003). *UNIX Systems Programming*. Prentice-Hall.
- [Stevens, 1998] Stevens, R. (1998). *UNIX Network Programming*. Prentice-Hall.