UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Aula 04: Sistemas Operacionais

Comunicação entre Processos (IPC)

Prof. Rodrigo Campiolo
Prof. Rogério A. Gonçalves[1]

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Departamento de Computação (DACOM)
Campo Mourão, Paraná, Brasil

Ciência de Computação

BCC34G - Sistemas Operacionais

Sumário

Comunicação entre Processos

Processos podem ser

- **Independentes:** não compartilham dados com outros processos.
- **Cooperativos:** podem afetar ou ser afetados por outros processos em execução no sistema.

- Os processos possuem seu próprio espaço de endereçamento, mas há situações precisam enviar/receber/acessar informações de outros processos.
- Mecanismo de IPC (**I***nter***p***rocess* **C***ommunication*).

Comunicação entre Processos

Motivos

- Compartilhamento de informações
- Aumento de desempenho na computação
- Modularidade
- Conveniência

- Vários mecanismos de IPC originaram-se do tradicional IPC do UNIX.

Dois modelos

- **Troca de mensagem**
- **Memória compartilhada**

Comunicação entre Processos

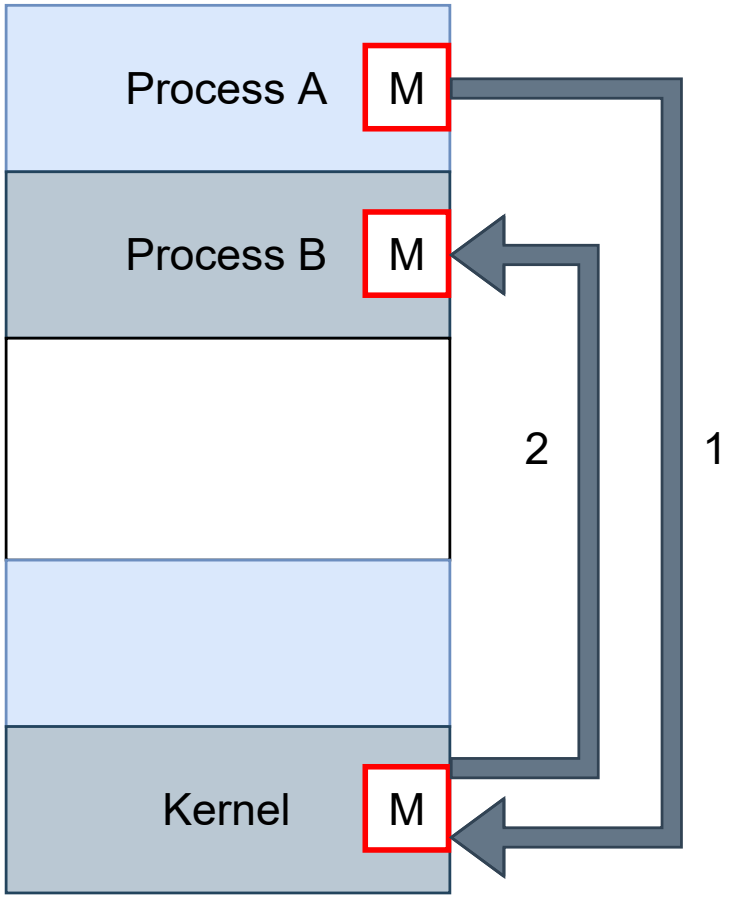
Troca de Mensagem

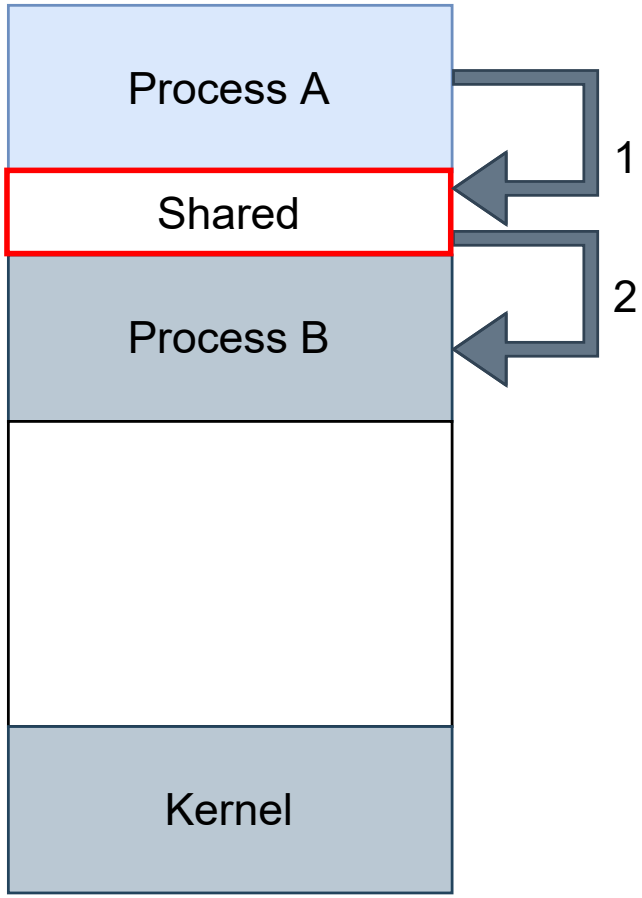
- Ocorre por meio de troca de mensagens entre os processos.
- Útil para trocar quantidades menores de dados.
- Mais fácil de implementar.
- Normalmente implementado com uso de chamadas de sistema.

Memória Compartilhada

- Uma região de memória é compartilhada entre os processos.
- A troca de informações ocorre por leitura/escrita de dados nessa região.
- Mais rápida, chamadas de sistemas só criam a região.

Comunicação entre Processos





Passagem de mensagem

Memória Compartilhada

Comunicação entre Processos

- Sinais

Mecanismos por troca de mensagens

- Pipe
- Fifo
- Filas de mensagens (mqueue)
- Sockets
- RPC

Mecanismos por memória compartilhada

- Memória compartilhada (shared memory)

Sinais

- Um dos primeiros mecanismos de comunicação interprocessos disponíveis em sistemas UNIX.
- São interrupções de software que notificam ao processo que um evento ocorreu, são utilizados pelo núcleo.
- Permitem somente o envio de uma palavra de dados (código do sinal (1 a 64)) para outros processos.
- Não permitem que processos especifiquem dados para trocar com outros processos.

Sinais

- Dependem do SO e das interrupções geradas por software suportadas pelo processador do sistema.
- Quando ocorre um sinal, o SO determina qual processo deve receber o sinal e como esse processo responderá ao sinal.

Sinais

Quando sinais são gerados?

- Criados pelo núcleo em resposta a interrupções e exceções, os sinais são enviados a um processo ou thread.
- Em consequência da execução de uma instrução (como falha de segmentação).
- Em um outro processo (como quando um processo encerra outro) ou em um evento assíncrono.

Sinais POSIX

Sinal	Tipo	Ação default	Descrição
1	SIGHUP	Abortar	Detectada interrupção ou morte do processo controlador
2	SIGINT	Abortar	Interrupção do teclado
3	SIGQUIT	Descarregar	Sair do teclado
4	SIGILL	Descarregar	Instrução ilegal
5	SIGTRAP	Descarregar	Rastreamento/ponto de ruptura
6	SIGABRT	Descarregar	Abortar sinal de função **abort**
7	SIGBUS	Descarregar	Erro de barramento
8	SIGGFPE	Descarregar	Exceção de ponto flutuante
9	SIGKILL	Abortar	Sinal de matar
10	SIGUSR1	Abortar	Sinal 1 definido pelo usuário
11	SIGSEGV	Descarregar	Referência inválida na memória
12	SIGUSR2	Abortar	Sinal 2 definido pelo usuário
13	SIGPIPE	Abortar	Pipe rompido: escrever para pipe com nenhum leitor
14	SIGALRM	Abortar	Sinal de temporizador da função **alarm**
15	SIGTERM	Abortar	Sinal de encerramento
16	SIGSTKFLT	Abortar	Falha de pilha no co-processador
17	SIGCHLD	Ignorar	Filho parado ou encerrado
18	SIGCONT	Continuar	Continuar se estiver parado
19	SIGSTOP	Parar	Parar processo
20	SIGTSTP	Parar	Parar digitado no dispositivo de terminal

Sinais POSIX

- Estão definidos 64 sinais.

```
rogerio@guarani:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

- Um kill -9 pid tem o mesmo efeito de um kill -SIGKILL pid

Sinais: Tratamento

Processos podem

- **Capturar:** especificando uma rotina que o SO chama quando entrega o sinal.
- **Ignorar:** Neste caso depende da ação padrão do SO para tratar o sinal.
- **Mascarar um sinal:** Quando um processo mascara um sinal de um tipo específico, o SO não transmite mais sinais daquele tipo para o processo até que ele desbloqueie a máscara do sinal.

Sinais: Tratamento

Um processo/thread pode tratar um sinal

1. **Capturando o sinal**
   - quando um processo capta um sinal, chama o tratador para responder ao sinal.
2. **Ignorando o sinal**
   - processos podem ignorar todos, exceto os sinais SIGSTOP e SIGKILL.
3. **Executando a ação default**
   - Ação definida pelo núcleo para esse sinal.

Sinais: Tratamento

Ações default

- **Abortar:** terminar imediatamente.
- **Descarga de memória:** copia o contexto de execução antes de sair para um arquivo do núcleo (memory dump).
- **Ignorar.**
- **Parar** (isto é, suspender).
- **Continuar** (isto é, retomar).

Sinais: Bloqueio

- Um processo ou thread pode bloquear um sinal.
- O sinal não é entregue até que o processo/thread pare de bloqueá-lo.
- Enquanto o tratador de sinal estiver em execução, os sinais desse tipo são bloqueados por default.
- Ainda é possível receber sinais de um tipo diferente (não bloqueados).
- Os sinais comuns não são enfileirados.
- Os sinais de tempo real podem ser enfileirados.

Sinais: Exemplo 1

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

/* função tratadora de sinais. */
void sig_handler(int signo){
    if (signo == SIGINT)
        printf("received SIGINT\n");
}

int main(void){
    /* Associa a função tratadora de sinais */
    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("\ncan't catch SIGINT\n");

    /* exibe o PID */
    printf("My PID is %d.\n", getpid());

    /* Simulando uma execução de nada */
    while(1)
        sleep(1);

    return 0;
}
```

```
$gcc ex01_simple_signal_handler.c -o ex01
$./ex01
My PID is 6450.
^Creceived SIGINT
^Creceived SIGINT
```

Sinais: Exemplo 2

```c
/*
 * Exemplo: http://www.gnu.org/software/libc/manual/
 *               html_node/Handler-Returns.html#Handler-Returns
 */

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

/* Flag que controla a terminação do loop. */
volatile sig_atomic_t keep_going = 1;

/* Tratador para o sinal SIGALRM. Reseta o flag e se reabilita. */
void catch_alarm(int sig) {
    puts("Alarme!");
    keep_going = 0;
    signal(sig, catch_alarm);
}

void do_stuff(void) {
    puts("Fazendo alguma coisa enquanto aguarda o alarme.");
}

int main(void) {
    /* Estabelece um tratador para sinais SIGALRM. */
    signal(SIGALRM, catch_alarm);

    /* Define um alarme para daqui a 10 segundos.
     * Interromperá o laço. */
    alarm(10);

    /* Fica em loop executando. */
    while (keep_going)
        do_stuff();

    puts("Terminou.");

    return EXIT_SUCCESS;
}
```

```
$gcc ex02_signal_alarm.c -oex02
$./ex02
Fazendo alguma coisa enquanto aguarda o alarme.
Fazendo alguma coisa enquanto aguarda o alarme.
Fazendo alguma coisa enquanto aguarda o alarme.
Fazendo alguma coisa enquanto aguarda o alarme.
Fazendo alguma coisa enquanto aguarda o alarme.
Fazendo alguma coisa enquanto aguarda o alarme.
Fazendo alguma coisa enquanto aguarda o alarme.
Fazendo alguma coisa enquanto aguarda o alarme.
Fazendo alguma coisa enquanto aguarda o alarme.
Fazendo alguma coisa enquanto aguarda o alarme.
Alarme!
Terminou.
$
```

Sinais: Exemplo 3

```c
/*
 * Fonte: http://www.gnu.org/software/libc/manual/html_node/
 *        Signaling-Yourself.html#Signaling-Yourself
 */
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

/* Define uma função tratadora de sinais. */
void sig_handler(int signo){
    if (signo == SIGTERM){
        printf("received SIGTERM\n");
        printf("Eu deveria ter finalizado...\n");
    }

    if (signo == SIGALRM){
        printf("received SIGALRM\n");
        //raise(SIGKILL);
        kill(getpid(), SIGKILL);
    }
}

int main(void){

    /* Associa a função tratadora de sinais */
    if (signal(SIGTERM, sig_handler) == SIG_ERR)
        printf("\ncan't catch SIGTERM\n");

    if (signal(SIGALRM, sig_handler) == SIG_ERR)
        printf("\ncan't catch SIGALRM\n");

    alarm(10);

    /* exibe o PID */
    printf("My PID is %d.\n", getpid());

    /* Entra em loop para pode dar tempo de receber sinais. */
    while(1)
        sleep(1);

    return 0;
}
```

```
$gcc ex03_signal_raise.c -oex03
$./ex03
My PID is 8283.
received SIGTERM
Eu deveria ter finalizado...
received SIGALRM
Killed
$
```

sinais : bash — Kon

File Edit View Bookmarks Settings Help

```
$kill -SIGTERM 8283
$
```

Sinais: Exemplo 4

```c
/**
 * Tutorial: https://github.com/angrave/SystemProgramming/wiki/
 * Signals%2C-Part-2%3A-Pending-Signals-and-Signal-Masks
 */

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

sigset_t set;   //conjutno de sinais a serem bloqueados/mascarados

/* Define uma função tratadora de sinais. */
void sig_handler(int signo){
    printf("received a %d\n", signo);
}

int main(void){
    /* Associa a função tratadora de sinais */
    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("\ncan't catch SIGINT\n");
    if (signal(SIGQUIT, sig_handler) == SIG_ERR)
        printf("\ncan't catch SIGQUIT\n");
    if (signal(SIGHUP, sig_handler) == SIG_ERR)
        printf("\ncan't catch SIGHUP\n");

    sigemptyset(&set);          //inicializa o conjunto com vazio
    //sigfillset(&set);         // adiciona todos os sinais
    sigaddset(&set, SIGQUIT);   // adiciona o sinal SIGQUIT
    sigaddset(&set, SIGINT);    // adiciona o sinal SIGINT
    sigprocmask(SIG_SETMASK, &set, NULL);   //aplica o mascaramento
    // SIGKILL e SIGSTOP não podem ser mascarados

    printf ("My PID is %d.\n", getpid());

    /* Entra em loop para pode dar tempo de receber sinais. */
    while(1)
       sleep(1);
    return 0;
}
```

```
$gcc ex04_signal_mask.c -oex04
$./ex04
My PID is 8364.
User defined signal 1
$./ex04
My PID is 8369.
received a 1
```

sinais : bash —

File Edit View Bookmarks Settings Help

```
$kill -SIGINT 8369
$kill -SIGQUIT 8369
$kill -SIGHUP 8369
$
```

Troca de Mensagens

- O SO fornece mecanismos para permitir que os processos cooperativos se comuniquem entre si por meio de troca de mensagens.
- Comunicação e sincronização de ações sem compartilhar o mesmo espaço de endereços, por exemplo, entre processos locais ou em um ambiente distribuído.

Troca de Mensagens

- Sistema de mensagem – processos se comunicam entre si trocando mensagens sem o uso de variáveis compartilhadas.

Há **duas operações** básicas

- **send** (mensagem) – tamanho da mensagem fixo ou variável
- **receive** (mensagem)

Se P e Q quiserem se comunicar, eles precisam

- estabelecer um enlace(link) de comunicação entre eles.
- trocar mensagens por meio de **send/receive**.

Questões de implementação

- Como os enlaces (links) são estabelecidos?
- Um enlace pode estar associado a mais de dois processos?
- Quantos enlaces pode haver entre cada par de processos em comunicação?
- Qual é a capacidade de um enlace?
- O tamanho de uma mensagem que o enlace suporta é fixo ou variável?
- O enlace é unidirecional ou bidirecional?

Sincronismo no envio/recebimento

- A passagem de mensagens pode ser com bloqueio ou sem bloqueio.

Bloqueio é considerado **síncrono (blocking)**

- **Envio com bloqueio:** emissor é bloqueado até que a mensagem é recebida.
- **Recepção com bloqueio:** receptor é bloqueado até que a uma mensagem esteja disponível.

Não bloqueio é considerado **assíncrono (nonblocking)**

- **Envio sem bloqueio:** emissor envia a mensagem e continua sua execução.
- **Recepção sem bloqueio:** receptor recebe uma mensagem válida ou nulo.

Pipes e Fifos

- **Pipes** e **Fifos** são canais para a comunicação entre processos, geralmente criados por chamadas de sistema. Os dados são tratados como se estivessem em uma fila.

Pipes

- Não possuem nome e são herdados de um processo.

Fifos (Named pipes)

- Continuam existindo mesmo depois que o processo terminar.

Pipes

O acesso ao pipe é controlado por descritores de arquivo.

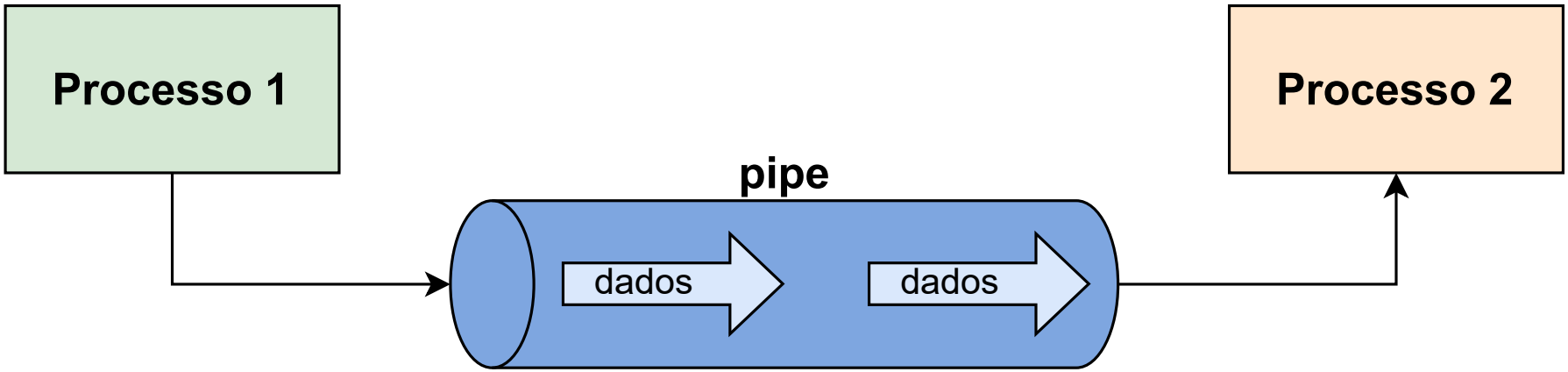
- Podem ser passados entre processos relacionados (por exemplo, pai e filho).

Pipes nomeados (FIFOs).

- Podem ser acessados por meio da árvore de diretório.

- Limitação: buffer de tamanho fixo.

Pipe



terminal

```
$ ps -aux | grep rogerio
...
```

Pipe

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

- `pipe()` cria um pipe, um canal de dados unidirecional que pode ser usado em IPC.

O *array* `pipefd` é usado para retornar dois descritores de arquivo

- `pipefd[0]` referencia o lado de leitura.
- `pipefd[1]` referencia o lado de escrita.

- Os dados escritos são colocados em um *buffer* pelo núcleo até ser lido pelo lado de leitura.

Mais detalhes: **man pipe**

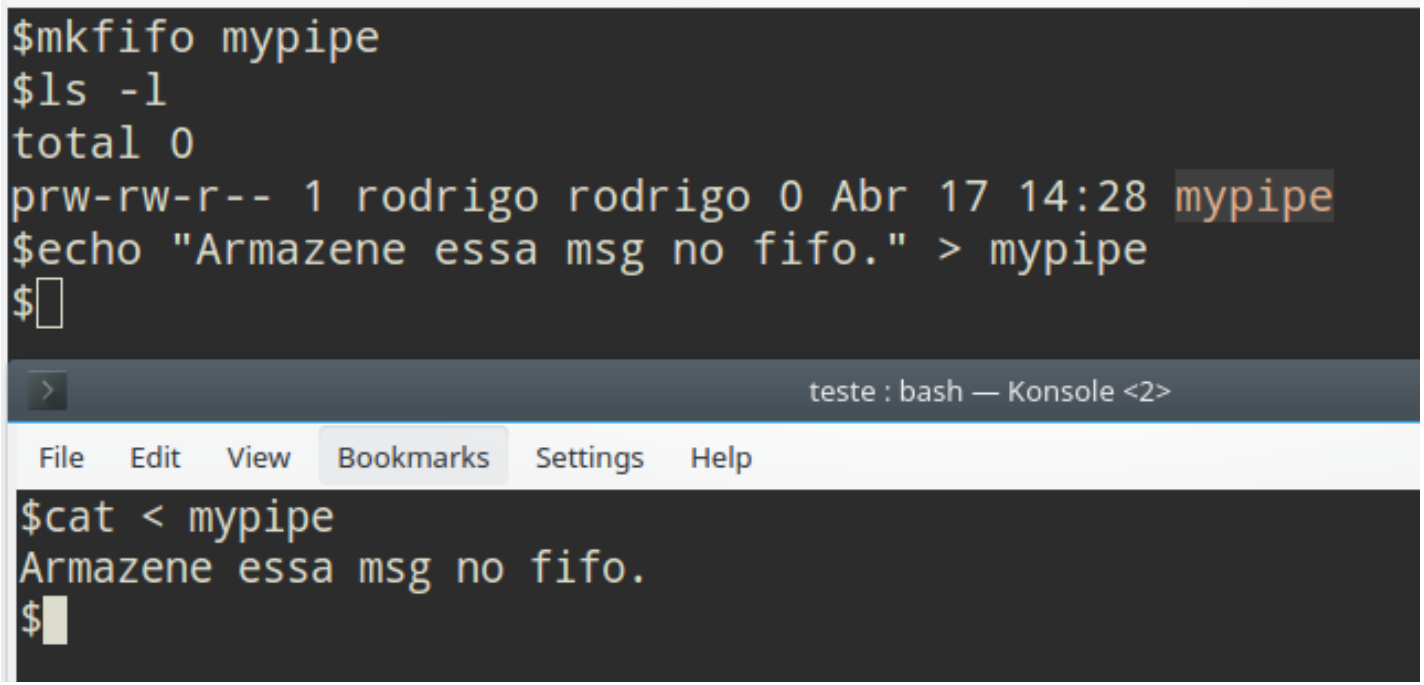
Exemplo Pipe

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

/* Programa principal */
int main(void) {
    pid_t pid;
    int mypipe[2];
    char buffer[40];

    /* Criar o pipe. */
    if (pipe(mypipe)) {
        fprintf(stderr, "Falha ao criar o Pipe.\n");
        return EXIT_FAILURE;
    }

    /* Criar o processo filho. */
    pid = fork();
    if (pid < (pid_t) 0) {
        /* pid negativo, falha no fork. */
        fprintf(stderr, "Falha ao executar o Fork.\n");
        return EXIT_FAILURE;
    } else if (pid == (pid_t) 0) {
        /* No processo filho. */
        close(mypipe[1]);
        read(mypipe[0], buffer, sizeof(buffer));
        printf("FILHO: ...%s\n", buffer);
        fflush(stdout);
        return EXIT_SUCCESS;
    } else {
        /* Processo pai. */
        close(mypipe[0]);
        printf("PAI: Digite algo para enviar: ");
        scanf("%40[^\n]", buffer);
        write(mypipe[1], buffer, sizeof(buffer));

        wait(NULL);
        return EXIT_SUCCESS;
    }
}
```

```
$gcc simple-pipe.c -osimple-pipe
$./simple-pipe
PAI: Digite algo para enviar: Olá filho.
FILHO: ...Olá filho.
$
```

Named Pipes (FIFO)

- FIFOS são canais nomeados (named pipes).
- É possível criar pipes nomeados usando o comando mkfifo.



FIFO

A mesma chamada de sistema está disponível em linguagem de programação

```
int mkfifo (const char *filename, mode_t mode)
```

- FIFOS são canais nomeados (named pipes).

Mais detalhes: **man fifo e man mkfifo**

FIFO - Exemplo 1

```c
/**
 * Lê mensagens de um FIFO e exibe na tela
 */
#include <stdio.h>      // standard io
#include <stdlib.h>     // standard lib
#include <errno.h>      // number of last error
#include <sys/stat.h>   // data returned stat() function (mkfifo)
#include <unistd.h>     // unix standard
#include <fcntl.h>      // file control options
#include <string.h>     // string operations

#define SERVER_FIFO "/tmp/serverfifo"

int main (int argc, char **argv)
{
    int fd_server, num_bytes_read;
    char buf [512];

    // cria um FIFO se inexistente
    if ((mkfifo (SERVER_FIFO, 0664) == -1) && (errno != EEXIST)) {
        perror ("mkfifo");
        exit (1);
    }

    // abre um FIFO
    if ((fd_server = open (SERVER_FIFO, O_RDONLY)) == -1)
        perror ("open");

    // lê e trata mensagens do FIFO
     while (1) {
         memset (buf, '\0', sizeof (buf));
         num_bytes_read = read (fd_server, buf, sizeof (buf));
         switch (num_bytes_read) {
             case -1:
                perror ("-- read error"); break;
             case  0:
                printf("-- None data...close and reopen fifo --\n");
                close(fd_server);
                fd_server = open (SERVER_FIFO, O_RDONLY);
                break;
             default:
                printf("Received %d bytes: %s\n", num_bytes_read, buf);
         }
     }

     if (close (fd_server) == -1)
        perror ("close");
}
```

```c
/**
 * Escreve N mensagens para o FIFO
 */
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

#define SERVER_FIFO "/tmp/serverfifo"
#define N 5

char buf [512];

int main (int argc, char **argv)
{
    int fd_server;  // descritor para o FIFO

    if ((fd_server = open (SERVER_FIFO, O_WRONLY)) == -1) {
        perror ("open error: server fifo");
        return 1;
    }

    int value = 0;
    while (value < N) {
        // cria mensagens incrementalmente
        sprintf (buf, "Message %ld - Number %d", (long) getpid (), value++);
        printf("Sending: %s\n", buf);

        // envia mensagem para o fifo
        write (fd_server, buf, strlen (buf));

        sleep(1);  // somente para visualizacao
    }

    if (close (fd_server) == -1) {
        perror ("close error:server fifo");
        return 2;
    }

    return 0;
}
```

FIFO - Exemplo 2

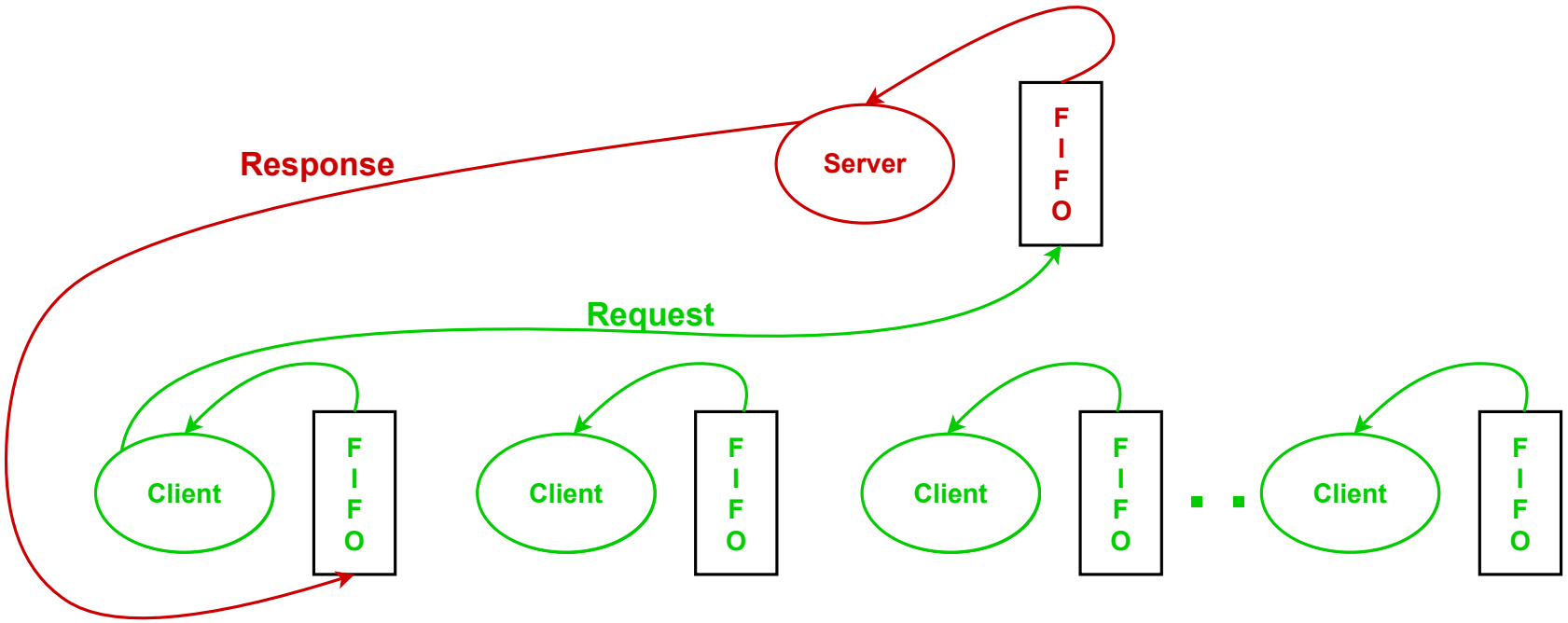


Figura 1: IPC usando FIFOs

Fonte: `https://www.softprayog.in/programming/interprocess-communication-using-fifos-in-linux`

FIFO - Exemplo 2

- Para compilar
  - `$ gcc server.c -o server`
  - `$ gcc client.c -o client`
- Para exibir os nomes dos fifos
  - `$ ls -l DIR`

Fila de Mensagens

- Permitem que os processos transmitam informações que são compostas por um tipo de mensagem e por uma área de dados de tamanho variável.
- Armazenadas em filas de mensagens, permanecem até que um processo esteja preparado para recebê-las.
- Processos relacionados podem procurar um identificador de fila de mensagens em um arranjo global de descritores de fila de mensagens.

Fila de Mensagens

O descritor de fila de mensagens contém

- Fila de mensagens pendentes;
- Fila de processos em espera de mensagens;
- Fila de processos em espera para enviar mensagens;
- Dados que descrevem o tamanho e o conteúdo da fila de mensagens.

POSIX Message Queue

- Disponível no Linux desde a versão 2.2.6.
- São identificadas por nomes definidos por uma cadeia de caracteres (string).
- No Linux, os nomes iniciam-se com /
- Qualquer processo que conheça o nome e tenha permissões, pode enviar e receber mensagens.
- No Linux, usa-se a biblioteca de tempo real para compilar (**-lrt**) e os nomes da funções iniciam-se com **mq_**

"POSIX message queues allow for an efficient, priority-driven IPC mechanism with multiple readers and writers."

Fonte: https://users.pja.edu.pl/~jms/qnx/help/watcom/clibref/mq_overview.html

POSIX Message Queue

Funções básicas (*mqueue.h*)

- `mq_open`: abre uma fila POSIX.
- `mq_close`: fecha o descritor da fila.
- `mq_send`: envia uma mensagem para uma fila.
- `mq_receive`: recebe uma mensagem de uma fila.
- `mq_unlink`: remove uma fila.
- `mq_setattr`: configura atributos de uma fila.

POSIX Message Queue - Exemplo

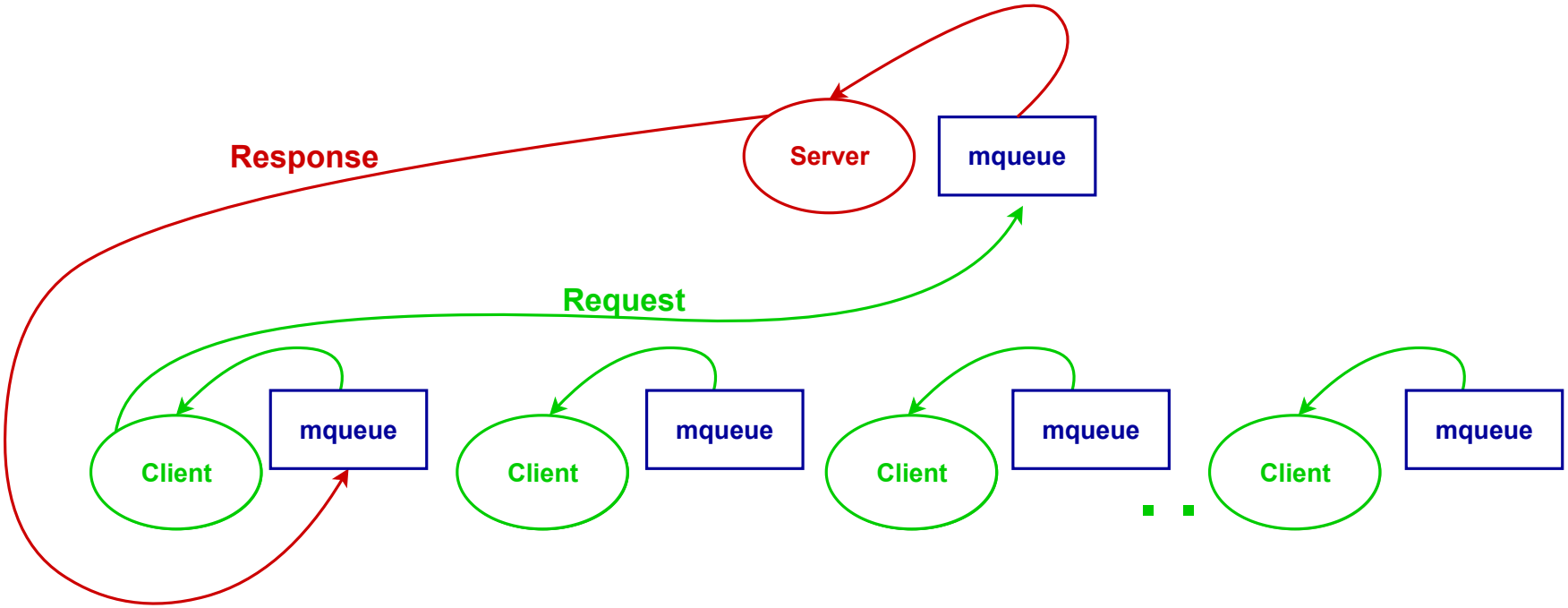


Figura 2: IPC usando Message Queues

Fonte: `https://www.softprayog.in/programming/interprocess-communication-using-posix-message-queues-in-linux`

POSIX Message Queue - Exemplo

- Para compilar

```
$ gcc server.c -o server -lrt
$ gcc client.c -o client -lrt
```

- Para exibir os nomes das filas

```
$ ls /dev/mqueue
$ ipcs -q
```

Sockets

- Possibilita a comunicação entre processos locais e remotos.
- Comunicação bidirecional.
- Deve-se especificar:
  - domínio de comunicação (ex: AF_UNIX, AF_INET).
  - tipo de comunicação (ex: SOCK_STREAM, SOCK_DGRAM).
- No Linux, ao criar um socket, é devolvido um descritor de arquivo.

Sockets

Funções básicas

- `socket:` cria e devolve o descritor do socket.
- `connect:` estabelece conexão com o servidor.
- `bind:` associa um endereço de protocolo local (ex: porta) ao socket.
- `listen:` define o socket como passivo (socket do servidor) e limita o tamanho da fila interna de conexões.
- `accept:` aguarda conexões e devolve um descritor de socket para uma conexão.
- `send:` envia dados por *stream sockets*.
- `recv:` recebe dados por *stream sockets*.
- `close:` fecha a comunicação entre cliente e servidor.

Socket Unix

- Comunicação entre processos em uma mesma máquina.
- Socket UNIX é conhecido por um **pathname**.
- Um servidor mapeia o **pathname** para o socket.
- Há três tipos de endereçamento: *pathname*, *abstract* e *unnamed*.
- Proveem comunicação bidirecional ao usar *stream sockets*.
- Clientes usando sockets mantêm conexão individual com o servidor.

Exemplo Socket - UNIX/LOCAL

Servidor: aguarda conexão e recebe mensagens do cliente.

Fonte: `https://troydhanson.github.io/network/Unix_domain_sockets.html`

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>

//char *socket_path = "./mysocket";
char *socket_path = "\0myabstractsocket";

int main(int argc, char *argv[]) {
    int server_socket,       // descritor do socket
        client_socket,       // socket da conexao do cliente
        received_bytes;      // bytes recebidos
    struct sockaddr_un addr; // endereço socket
    char buf[100];           // buffer de comunicação

    /* cria um socket AF_UNIX do tipo SOCK_STREAM */
    if ( (server_socket = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        perror("socket error");
        exit(-1);
    }

    /* configura endereço do socket */
    memset(&addr, 0, sizeof(addr));
    addr.sun_family = AF_UNIX;

    if (*socket_path == '\0') {
        *addr.sun_path = '\0';
        strncpy(addr.sun_path+1, socket_path+1, sizeof(addr.sun_path)-2);
    } else {
        strncpy(addr.sun_path, socket_path, sizeof(addr.sun_path)-1);
        unlink(socket_path);  // desvincular path se existe
    }

    /* mapeia o socket para o socket_path */
    if (bind(server_socket, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
        perror("bind error");
        exit(-1);
    }

    /* configura para aguardar conexões */
    if (listen(server_socket, 5) == -1) {
        perror("listen error");
        exit(-1);
    }

    while (1) {
      /* aguarda conexões dos clientes */
      if ( (client_socket = accept(server_socket, NULL, NULL)) == -1) {
          perror("accept error");
          continue;
      }

      /* lê dados envidos pelos clientes */
      while ( (received_bytes = read(client_socket, buf, sizeof(buf))) > 0 ) {
          printf("read %u bytes: %.*s\n", received_bytes, received_bytes, buf);
      }

      /* trata erros */
      if (received_bytes == -1) {
          perror("read");
          exit(-1);
      } else if (received_bytes == 0) {
          printf("EOF\n");
          close(client_socket);
      }
    }

    return 0;
}
```

Exemplo Socket - UNIX/LOCAL

Cliente: conecta e envia mensagens para o servidor.
Fonte: `https://troydhanson.github.io/network/Unix_domain_sockets.html`

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

/* especificar o path para mapear o socket */
//char *socket_path = "./mysocket";
char *socket_path = "\0myabstractsocket";

int main(int argc, char *argv[]) {
    int client_socket,          // descritor para o socket (file descriptor)
        sent_bytes;             // número de bytes enviados
    struct sockaddr_un addr;    // estrutura de endereço socket unix
    char buf[100];              // buffer para troca de mensagens

    /* cria socket UNIX do tipo SOCK_STREAM */
    if ( (client_socket = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        perror("socket error");
        exit(-1);
    }

    /* preenche estrutura de endereço */
    memset(&addr, 0, sizeof(addr));
    addr.sun_family = AF_UNIX;
    if (*socket_path == '\0') {   // socket não mapeado no sistema de arquivos (
        *addr.sun_path = '\0';
        strncpy(addr.sun_path+1, socket_path+1, sizeof(addr.sun_path)-2);
    } else {
        strncpy(addr.sun_path, socket_path, sizeof(addr.sun_path)-1);
    }

    /* conecta com o processo servidor */
    if (connect(client_socket, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
        perror("connect error");
        exit(-1);
    }

    /* lê dados do terminal e envia via sockets */
    while ( (sent_bytes = read(STDIN_FILENO, buf, sizeof(buf)) ) > 0) {
        if (write(client_socket, buf, sent_bytes) != sent_bytes) {
            if (sent_bytes > 0) fprintf(stderr,"partial write");
            else {
                perror("write error");
                exit(-1);
            }
        }
    }

    return 0;
}
```

Exemplo Socket - UNIX/LOCAL

Troca de mensagem entre pai e filho com *socketpair*.

```c
/** Demonstracao de como usar unnamed sockets para comunicação entre
    processos pai e filho.
    int socketpair(int domain, int type, int protocol, int sv[2]);
**/

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

#define HELLOREQUEST  "Hello, how are you?"
#define HELLORESPONSE "I am fine, thanks."

int main()
{
   int sockets[2], pid;
   char buf[1024];

   /* cria um par de sockets de domínio UNIX e tipo STREAM */
   if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) < 0) {
      perror("opening stream socket pair");
      exit(1);
   }

   pid = fork();

   if (pid) {     /* Processo PAI */
      /* PAI usa o sockets[1] para enviar/receber */
      close(sockets[0]);

      if (read(sockets[1], buf, 1024) < 0)
         perror("error reading message");

      printf("  PAI recv: %s\n", buf);

      if (write(sockets[1], HELLORESPONSE, sizeof(HELLORESPONSE)) < 0)
         perror("error writing message");

      close(sockets[1]);

   } else {       /* Processo FILHO. */
      /* FILHO usa o sockets[0] para enviar/receber */
      close(sockets[1]);

      if (write(sockets[0], HELLOREQUEST, sizeof(HELLORESPONSE)) < 0)
         perror("error writing message");

      if (read(sockets[0], buf, 1024) < 0)
         perror("error reading message");

      printf("FILHO recv: %s\n", buf);

      close(sockets[0]);
   }

   return 0;
}
```

Socket TCP/IP

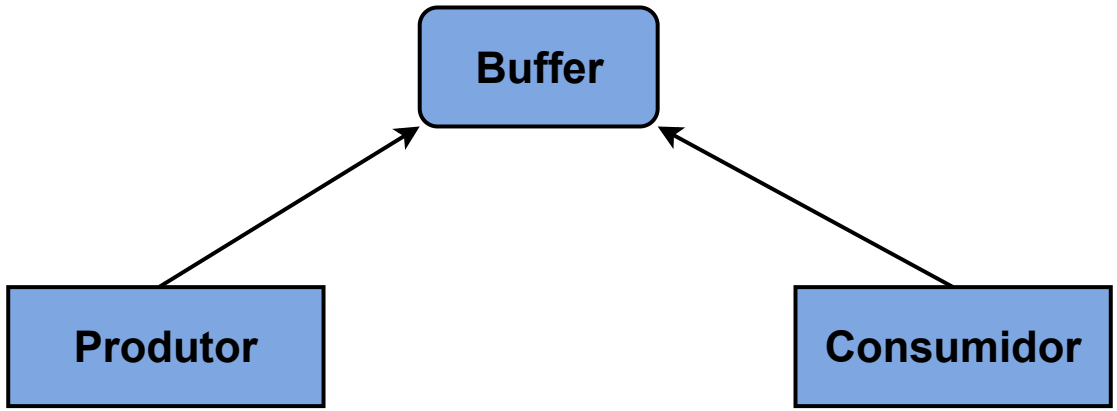
- Comunicação entre processos remotos ou local usando endereço de loopback (127.0.0.0/8).
- Socket é associado a endereço IP e porta.
- Exemplo: chat.

Memória Compartilhada

- Dois ou mais processos utilizam a região de memória compartilhada, conectando-a no seu espaço de endereçamento.
- Deve-se ter a garantia de que os dois processos não estejam gravando dados no mesmo local simultaneamente.
- **Exemplo:** Problema Produtor-Consumidor

Problema Produtor-Consumidor

- Paradigma para processos em cooperação
- **Processo produtor** produz informações que são consumidas por um **processo consumidor**
  - **Buffer ilimitado** não impõe limite prático sobre o tamanho do buffer
  - **Buffer limitado** assume que existe um tamanho de buffer fixo



Memória Compartilhada

Vantagens

- Melhora o desempenho de processos que acessam frequentemente dados compartilhados.
- Os processos podem compartilhar a mesma quantidade de dados que podem endereçar.

Interface padronizada

- Memória compartilhada System V.
- Memória compartilhada POSIX.
  - Não permite que os processos mudem privilégios de um segmento de memória compartilhada.

Memória Compartilhada

Funções para uso de memória compartilhada **POSIX**

- **shm_open:** cria ou abre um objeto de memória compartilhada.
- **shm_unlink:** remove um objeto de memória compartilhada.
- **ftruncate:** especifica o tamanho do segmento de memória compartilhada.
- **mmap:** mapeia o objeto de memória compartilhada dentro do espaço de endereçamento do processo.
- **munmap:** desassocia o objeto de memória compartilhado do espaço de endereçamento do processo.
- **close:** fecha o descritor alocado por *shm_open*.

Fonte: https://www.softprayog.in/programming/interprocess-communication-using-posix-shared-memory-in-linux

Memória Compartilhada

Funções para uso de memória compartilhada **SYSTEM V**

- **shmget:** aloca um segmento de memória compartilhada.
- **shmat:** anexa um segmento de memória compartilhada a um processo.
- **shmctl:** altera os atributos de um segmento de memória compartilhada.
- **shmdt:** desacopla um segmento de memória compartilhada.

Exemplo - POSIX Shared Memory

Produtor/Consumidor - versão simplificada [4].

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>

int main()
{
    const int SIZE = 4096;
    const char *name = "OS";
    const char *message1= "Operating Systems ";
    const char *message2= "Is Fun!";

    int shm_fd;   // descritor segmento de memoria compartilhada
    void *ptr;    // ponteiro segmento de memoria compartilhada

    /* cria segmento de memoria compartilhada */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configura o tamanho do segmento */
    ftruncate(shm_fd, SIZE);

    /* mapeia o segmento para o espaco de enderecamento do processo */
    ptr = mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED) {
        printf("Map failed\n");
        return -1;
    }

    /* escreve para a segmento de memoria compartilhada
     * obs: incrementa-se ponteiro a cada escrita */
    sprintf(ptr,"%s",message1);
    ptr += strlen(message1);
    sprintf(ptr,"%s",message2);
    ptr += strlen(message2);

    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>

int main()
{
    const char *name = "OS";
    const int SIZE = 4096;

    int shm_fd;   // descritor segmento de memoria compartilhada
    void *ptr;    // ponteiro segmento de memoria compartilhada

    /* abre segmento de memoria compartilhada */
    shm_fd = shm_open(name, O_RDONLY, 0666);
    if (shm_fd == -1) {
        printf("shared memory failed\n");
        exit(-1);
    }

    /* mapeia segmento no espaco de enderecamento do processo */
    ptr = mmap(NULL, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED) {
        printf("Map failed\n");
        exit(-1);
    }

    /* le segmento como uma string */
    printf("%s", (char *)ptr);

    /* remove segmento de memoria compartilhada */
    if (shm_unlink(name) == -1) {
        printf("Error removing %s\n",name);
        exit(-1);
    }

    return 0;
}
```

Exemplo - System V Shared Memory

Produtor/Consumidor - versão simplificada.

Fonte: `http://www.cs.cf.ac.uk/Dave/C/node27.html`

```c
/* Tamanho do segmento compartilhado. */
#define SHMSZ     128

int main() {
    int shmid;        // id segmento memoria compartilhada
    key_t key;        // chave de acesso memoria compartilhada
    char *shm;        // ponteiro para memória compartilhada

    /* ID segmento de memória compartilhada */
    key = 5678;

    /* Cria o segmento compartilhado. */
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("Erro ao criar o segmento de shm (shmget).");
        exit(1);
    }
    printf ("ID regiao: %d. \nUse ipcs -m \n", shmid);

    /* Acoplamento do segmento criado ao espaco do processo */
    if ((shm = shmat(shmid, NULL, 0)) == (char*)-1) {
        perror("Erro ao acoplar o segmento ao processo (shmat).");
        exit(1);
    }

    /* Produzindo dados */
    strncpy(shm, "Olá Mundo!", 10);
    shm[10] = '\0';

    /* Aguardando a leitura do outro processo (espera ocupada).
      Consumidor rá mudar o primeiro '*' indicando leitura */
    while (*shm != '*')
        sleep(1);

    /* Desacoplamento da região de memória compartilhada. */
    if (shmdt(shm) == -1) {
        perror("Erro ao desacoplar o segmento (shmdt).");
        exit(1);
    }

    /* Destruição do segmento */
    if (shmctl(shmid, IPC_RMID, 0) == -1){
        perror("Erro ao destruir o segmento (shmctl).") ;
        exit(1) ;
    }
```

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define SHMSZ     128

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

    /* chave do segmento criado pelo produtor. */
    key = 5678;

    /* Localizando o segmento */
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("Erro ao acessar o segmento de shm (shmget).");
        exit(1);
    }

    /* Acoplamento do segmento ao processo. */
    if ((shm = shmat(shmid, NULL, 0)) == (char*)-1) {
        perror("Erro ao acoplar o segmento ao processo (shmat).");
        exit(1);
    }

    /* leitura do segmento de memoria compartilhado */
    for (s = shm; *s != '\0'; s++)
        putchar(*s);
    putchar('\n');

    /* Modificando o primeiro caracter do segmento para '*',
       indicando que os dados já foram lidos. */
    *shm = '*';

    /* Desacopla a região de memória compartilhada. */
    if (shmdt(shm) == -1) {
        perror("Erro ao desacoplar o segmento(shmdt).");
        exit(1);
    }
```

Implementação de Memória Compartilhada - POSIX

- Trata a região da memória compartilhada como um arquivo.
- As molduras de página de memória compartilhada são liberadas quando o arquivo é apagado.
- o **tmpfs** (sistema de arquivo temporário) armazena esses arquivos.
  - Geralmente montado no Linux em `/dev/shm`.
  - As páginas do `tmpfs` podem ser trocadas.
  - É possível definir as permissões.
  - O sistema de arquivo não exige formatação.

Atividades

- Fazer a lista de exercícios *L04 - IPC* disponível na plataforma Moodle.
- Fazer os exercícios práticos descritos no *Laboratório 05 - IPC com pipe, fifo, sockets, shm*.

Leitura Recomendada

1. Signal Handling. Disponível em: `https://www.gnu.org/software/libc/manual/html_node/Signal-Handling.html`.
2. Pipes and FIFOs. OpenCSF - Computer Systems Fundamentals. Disponível em: `https://opencsf.org/Books/csf/html/Pipes.html`, Michael S. Kirkpatrick [2].
3. Unix Socket - Quick Guide. Disponível em: `https://www.tutorialspoint.com/unix_sockets/socket_quick_guide.htm`
4. POSIX Shared Memory. Disponível em: `https://man7.org/training/download/lusp_pshm_slides.pdf`, Michael Kerrisk [1].
5. Capítulos 8 e 9. Sistemas Operacionais: Conceitos e Mecanismos, Maziero [3].

Referências I

[1] Kerrisk, M. (2020). *Linux/UNIX System Programming Fundamentals*. online. Disponível em `https://man7.org/training/download/Linux_System_Programming-man7.org-mkerrisk-NDC-TechTown-2020.pdf`.

[2] Kirkpatrick, M. S. (2021). *Computer Systems Fundamentals*. online. Disponível em `https://opencsf.org/`.

[3] Maziero, C. A. (2019). *Sistemas operacionais: conceitos e mecanismos*. online. Disponível em http://wiki.inf.ufpr.br/maziero/lib/exe/fetch.php?media=so:so-livro.pdf.

[4] Silberschatz, A., Galvin, P. B., and Gagne, G. (2015). *Fundamentos de sistemas operacionais*. LTC, 9 edition.