UTFPR - Universidade Tecnológica Federal do Paraná

DACOM - Departamento de Computação

Bacharelado em Ciência da Computação

Sistemas Operacionais

Comunicação entre Processos (IPC)

Prof. Rogério A. Gonçalves

Prof. Rodrigo Campiolo

Tópico 4 - IPC

- Comunicação entre processos.
- Sinais.
- Pipe e Fifo.
- Fila de mensagens.
- Sockets.
- Memória compartilhada.

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 (***Inter****p****rocess*** ***C****ommunication*).

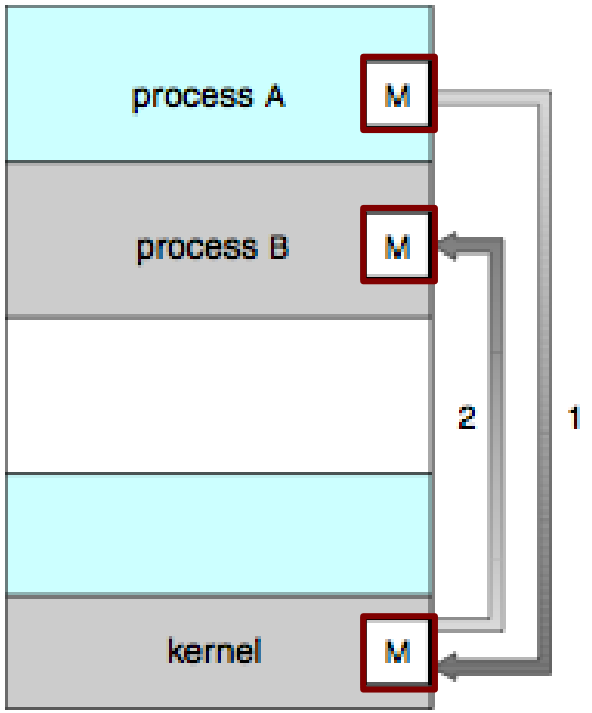
Comunicação entre Processos

- 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***

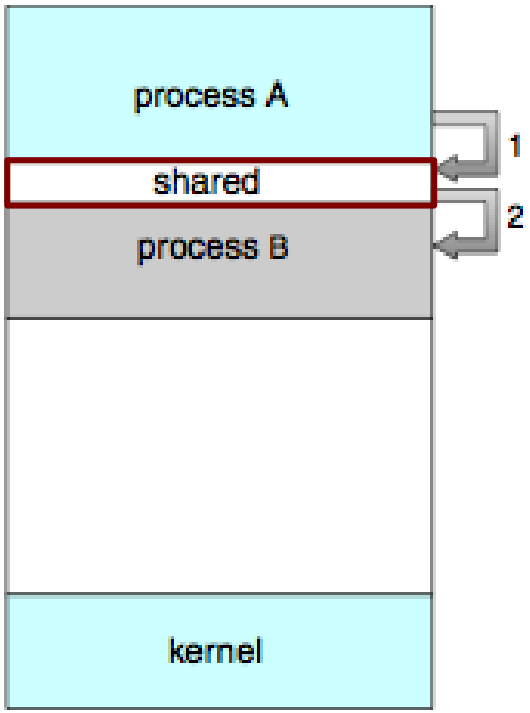
Comunicação entre Processos

- **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 de comunicação do terminal ou morte do processo controlador
2	SIGINT	Abortar	Interrupção de teclado
3	SIGQUIT	Descarregar	Sair do teclado
4	SIGILL	Descarregar	Instrução ilegal
5	SIGTRAP	Descarregar	Rastro/armadilha de ponto de ruptura
6	SIGABRT	Descarregar	Abortar sinal da função **abort**
7	SIGBUS	Descarregar	Erro de barramento
8	SIGFPE	Descarregar	Exceção de ponto flutuante
9	SIGKILL	Abortar	Sinal de matar
10	SIGUSR1	Abortar	Sinal 1 definido pelo usuário
11	SIGSEGV	Descarregar	Referência inválida à 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***
     - os 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.

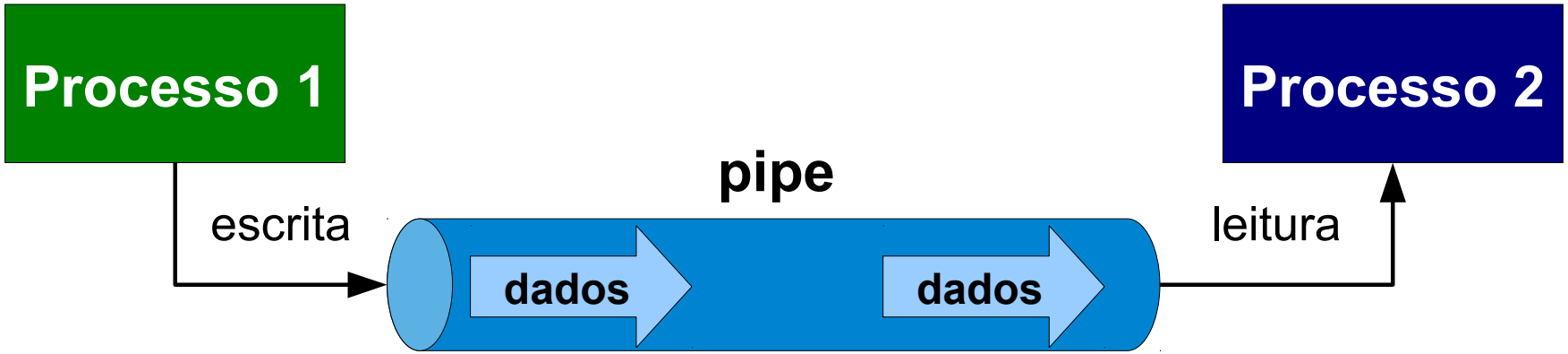
Pipes e Fifos

- ***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

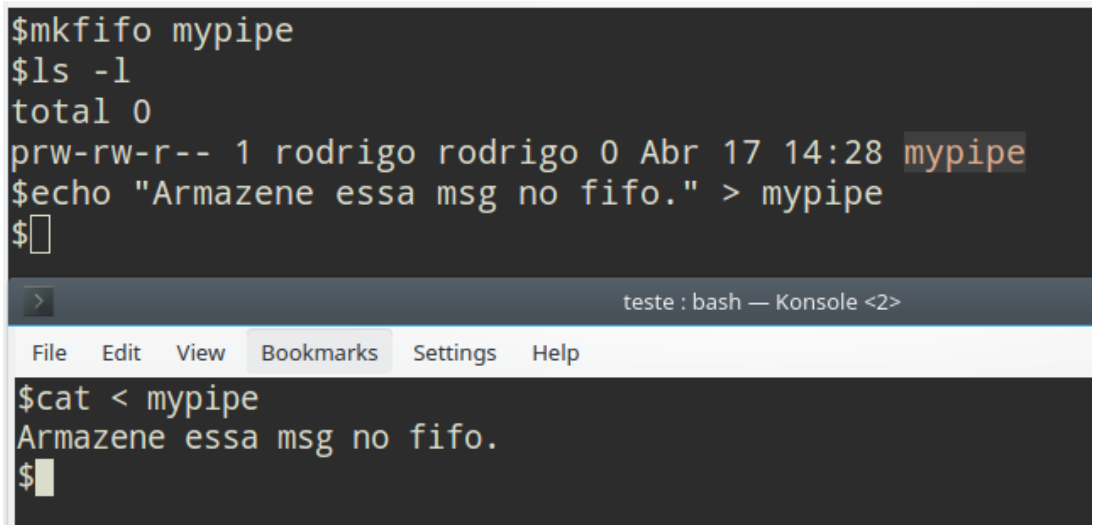
```c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

/* 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:

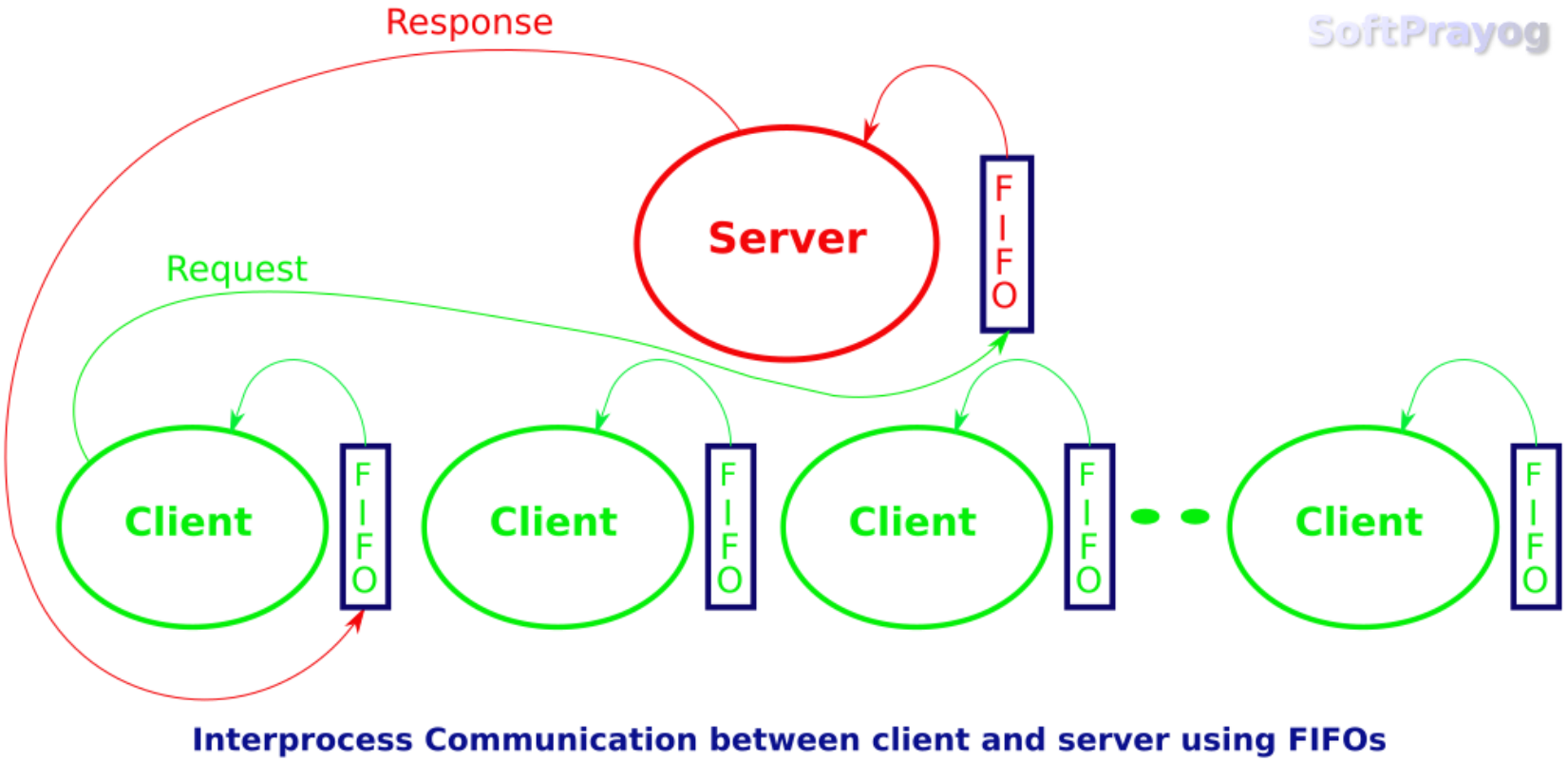
```
int mkfifo (const char *filename, mode_t mode)
```

FIFOS são canais nomeados (*named pipes*).

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

FIFO

- Exemplo



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

FIFO

- Exemplo
  - 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 (-l**rt**) 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."

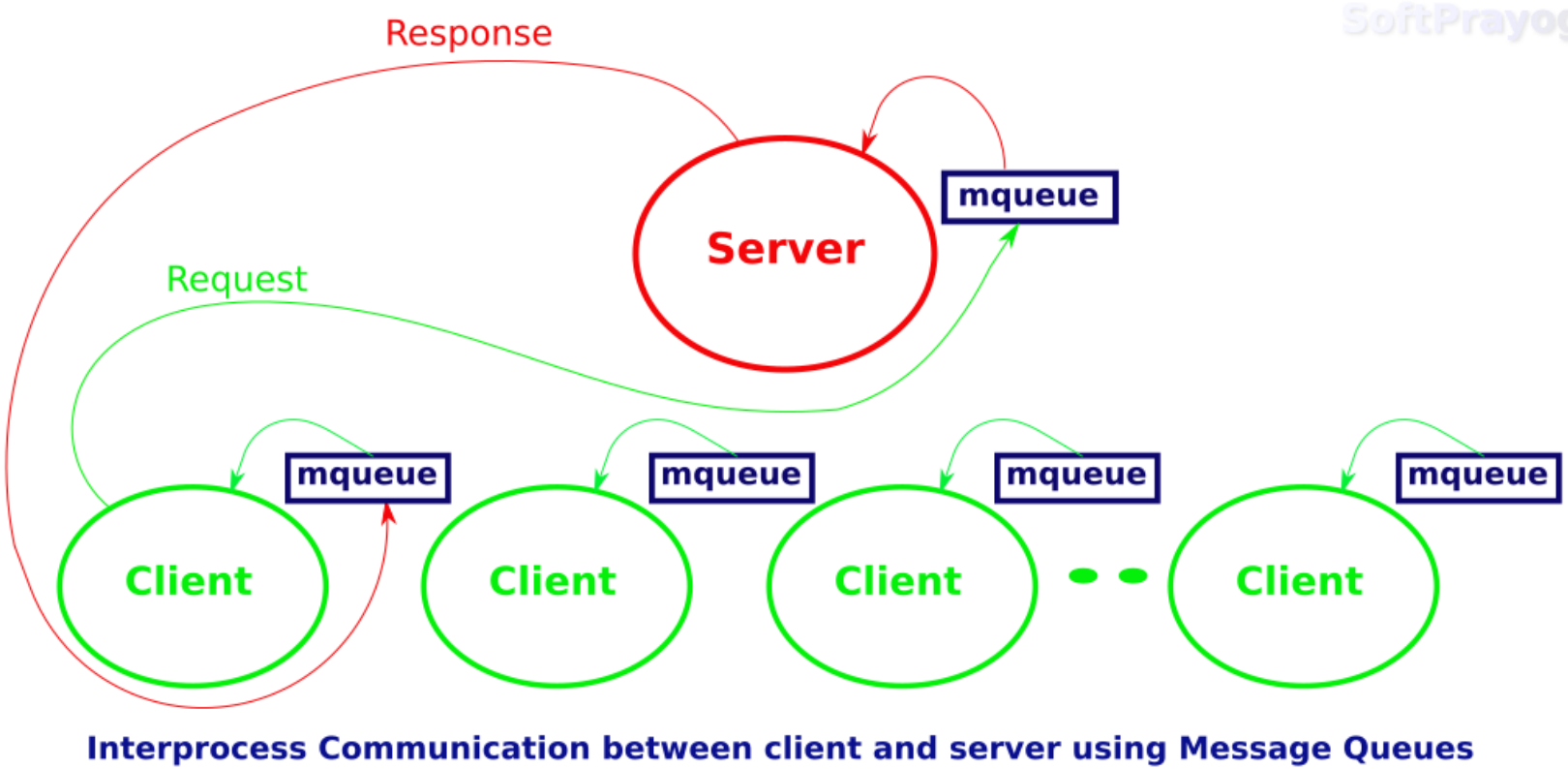
Fonte: https://users.pja.edu.pl/~jms/qnx/help/watcom/clibref/mq_overview.html

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



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

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

Sockets

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

Socket Unix

- Comunicação entre processos em uma mesma máquina.
- Socket UNIX é conhecido por um **pathname**.
- Um servidor mapeia o **pathname** para o socket.
- Proveem comunicação bidirecional ao usar *stream sockets*.
- Clientes usando sockets mantêm conexão individual com o servidor.

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

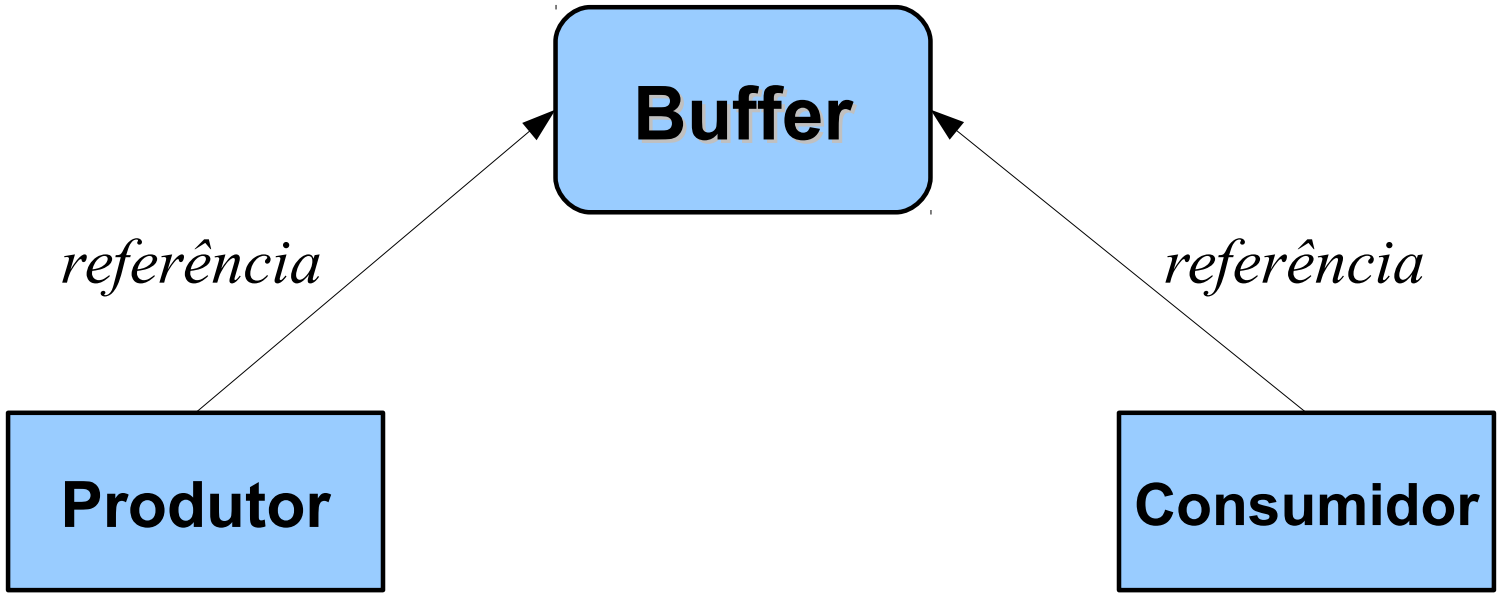
Memória Compartilhada

- 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

Problema Produtor-Consumidor



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.

Implementação de Memória Compartilhada

- 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.
  - As páginas do tmpfs podem ser trocadas.
  - É possível definir as permissões.
  - O sistema de arquivo não exige formatação.

Atividade Prática

- ***Acessar no moodle.***