Comunicação Entre Processos (IPC)

Engenharia de Computação
Pontifícia Universidade Católica de Campinas

Prof. Dr. Denis M. L. Martins

Objetivos de Aprendizagem

Ao final desta aula, você será capaz de:

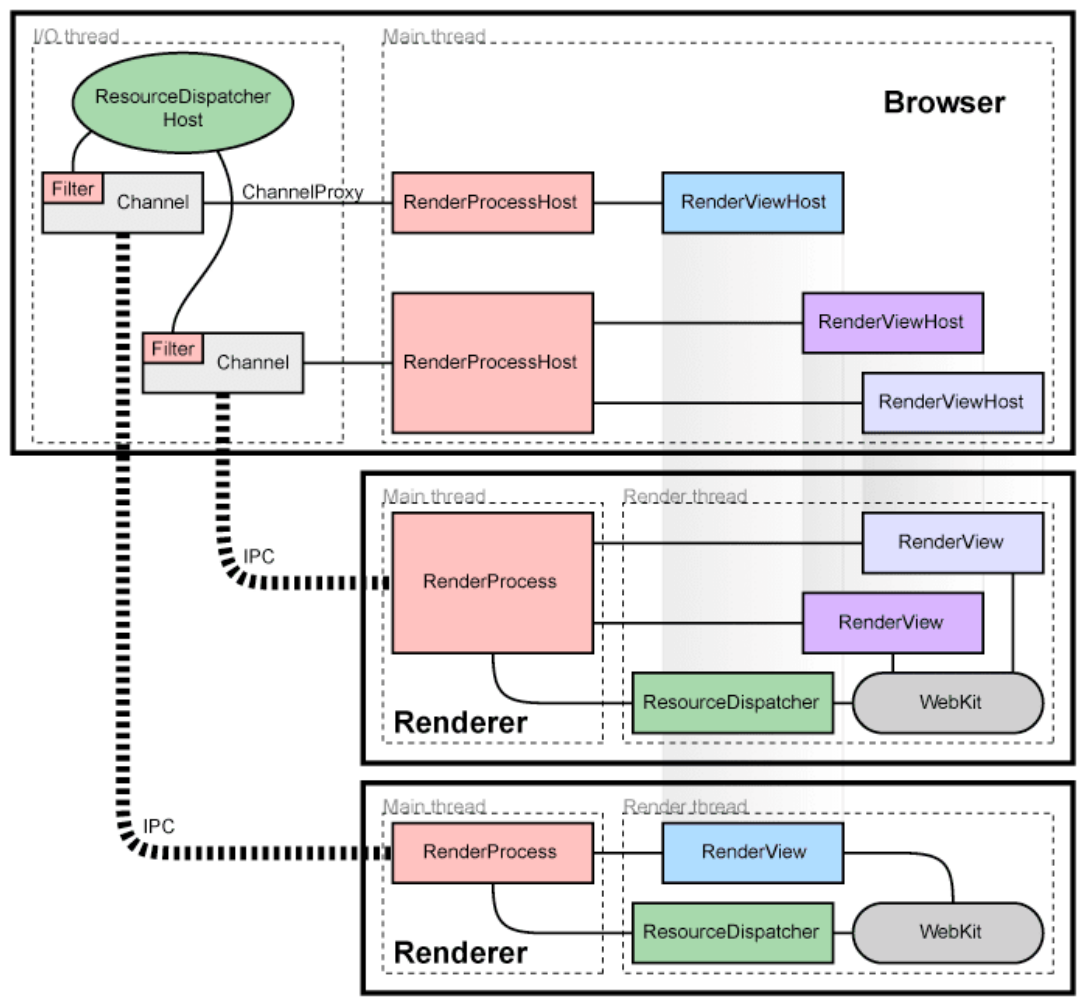
- Definir o conceito de Comunicação entre Processos (IPC) e sua importância no desenvolvimento de aplicações multi-processos.
- Explicar os diferentes mecanismos de IPC disponíveis:
  - Memória Compartilhada (Shared Memory)
  - Message Queues (Filas de Mensagens)
  - Sockets (Conexões Rede)
  - Pipes (Tubos)

Conceitos Fundamentais

Definição e Importância

- **Definição**: *IPC (Inter-Process Communication)* refere-se aos mecanismos que permitem a troca de dados entre processos distintos.
- **Importância**: Essencial em sistemas multitarefa, permitindo modularidade, concorrência e eficiência.
    - Permite que processos cooperem, sincronizem ações e compartilhem informações. Exemplo: Integração entre componentes existentes (reuso de software).
    - Lembre-se: *Sem comunicação, os processos são originialmente isolados*.
        - Uma aplicação faz tudo, sem modularização.
        - Potenciais falhas de segurança (*reliability issues*)

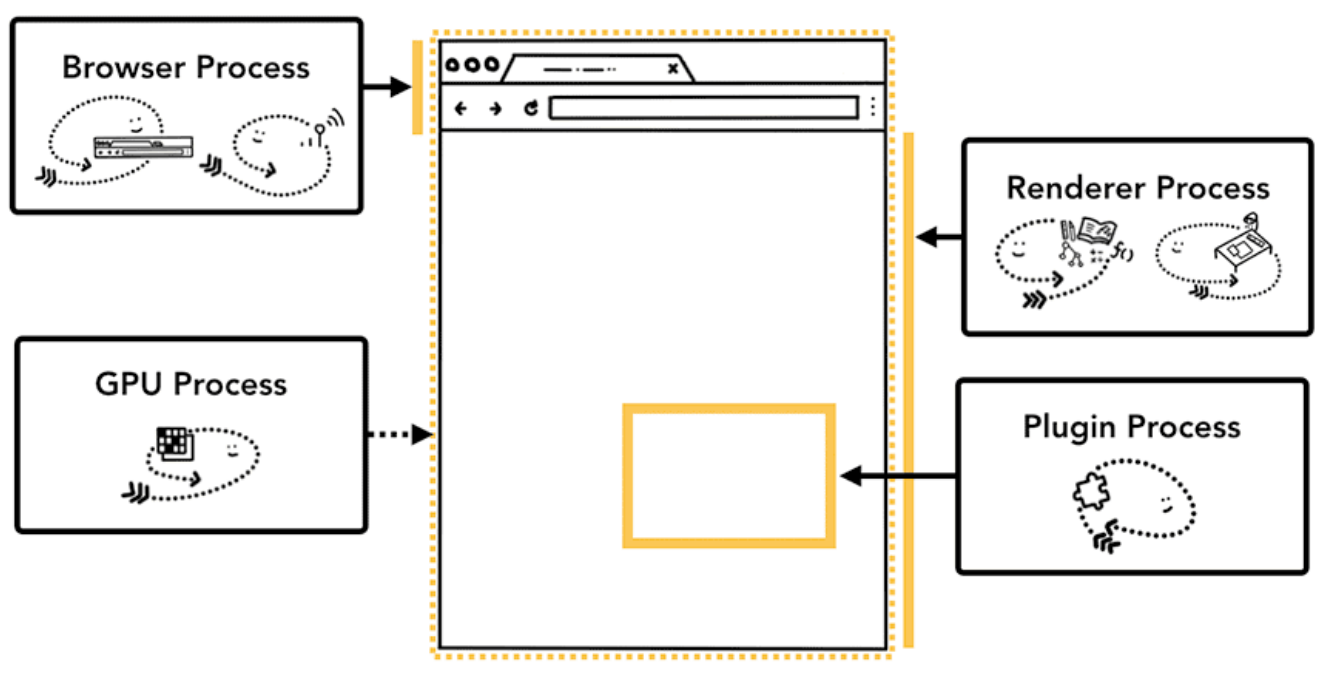
Exemplo de IPC: Chromium



- Navegadores baseados em Chromium são multiprocesso, utilizando três tipos diferentes de processos:
  - **Processo do Navegador** (*Browser Process*): Gerencia a interface do usuário, operações de entrada e saída em disco e rede.
  - **Processo Renderizador** (*Renderer Process*): Renderiza as páginas web, lidando com HTML, JavaScript. Um novo processo renderizador é criado para cada site aberto.
  - **Processo de Plugin** (*Plug-in Process*): Executa os processos para cada tipo de plugin.
- Cada aba encapsulada em um novo processo.
- Fonte da imagem: Chromium Project

Exemplo de IPC: Chrome

Nas versões mais modernas do navegador Chrome, há também o tipo **Processo de GPU** (*GPU Process*), que lida com tarefas de GPU e processam solicitações de múltiplas aplicações e as renderiza na mesma "superfície". Na imagem: Diferentes processos apontando para diferentes partes da interface do usuário (UI) do navegado. Fonte da imagem: Google Developers



Necessidade de IPC

- **Paralelismo**: Executar tarefas simultaneamente para melhorar o desempenho.
- **Modularidade**: Dividir um programa complexo em partes menores, cada uma executada por um processo separado (exemplo: cliente/servidor).
- **Robustez**: Se um processo falhar, os outros podem continuar funcionando (dependendo da implementação).
- **Recursos Compartilhados**: Permitir que processos acessem e manipulem recursos comuns de forma controlada.

Programa em Isolamento

Observe o programa abaixo:

```c
#include <stdio.h>

int main(void){
    printf("Hello, world\n");
    return 0;
}
```

Programa em Isolamento (cont.)

Observe o programa abaixo:

```c
#include <stdio.h>

int main(void){
    printf("Hello, world\n");
    return 0;
}
```

- Pode interagir com outros programas usando: `./hello_world | grep Hello`
  - Modelo de comunicação pipeline: $P_0 \rightarrow P_1 \rightarrow P_2 \ldots \rightarrow P_N$
  - A interação é estática, mas não é voluntária por parte do programa.
  - O programa foi projetado como um aplicativo independente.
- "Filosofia Unix": faça uma coisa bem feita (modularidade).

Criando interação

Observe agora os programas abaixo:

```c
// writer.c
int main(void) {
        FILE *fp = fopen("myf.txt", "w");
        fprintf(fp, "Hello");
        fclose(fp);
        return 0;
}
```

```c
// reader.c
int main(void) {
        char a[20];
        FILE *fp = fopen("myf.txt", "r");
        fscanf(fp, "%s", a);
        printf("%s\n", a);
        fclose(fp);
        return 0;
}
```

Criando interação (cont.)

- Os dois programas realmente interagem.
- Não há protocolo de interação.
- Se uma das aplicações não existisse, a outra ainda poderia ser um programa válido e significativo.

Mas o que acontece se iniciarmos o programa `reader` antes do `writer`?

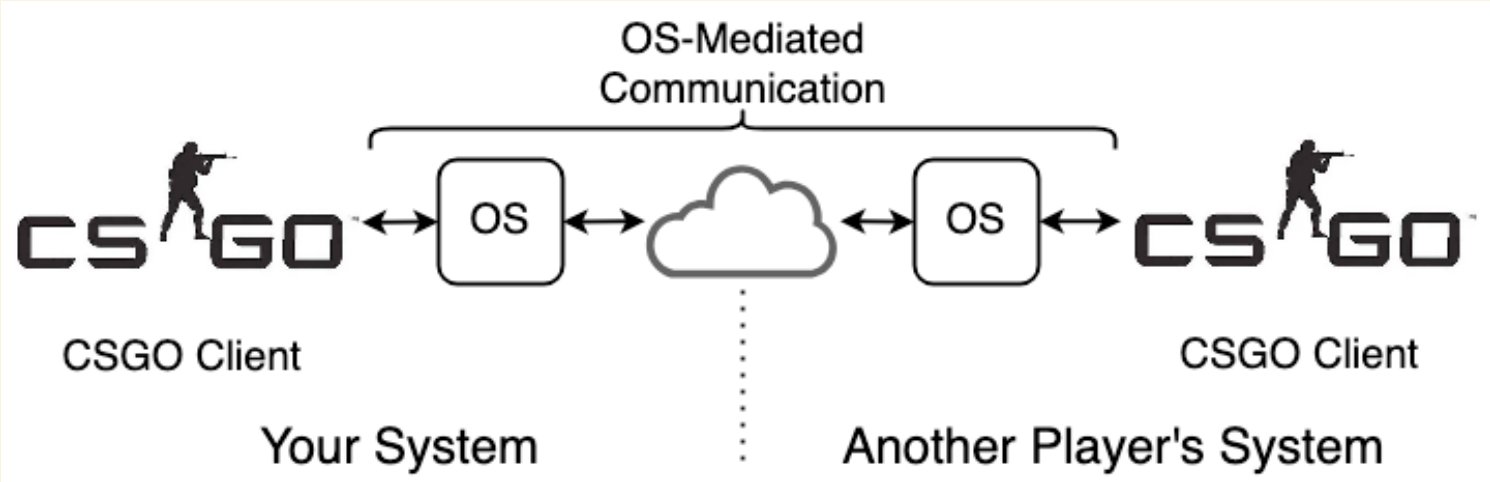
- Note a necessidade de sincronização (ex: mutex, semáforos...)

Papel do Sistema Operacional na IPC

- Faz a mediação da interação entre aplicativos.
- Fornece primitivas/mecanismos para a interação entre aplicativos.
- Funções:
  - Registro (Registry): Identificação dos pontos finais dos aplicativos.
  - Correio (Post Office): Passagem confiável de mensagens entre aplicações.
  - Política (Policy): Garantia do controle de acesso e segurança.
  - Campainha (Doorbell): Notificação do aplicativo sobre mensagens recebidas.

Papel do Sistema Operacional na IPC (cont)

- SO é como um legislador que determina como a interação ocorre.
  - Permite ou nega o envio de dados/notificações.
  - Garante a entrega correta de dados/notificação.



- Na imagem: Apps (CS GO) interagindo por meio do SO. Fonte da Imagem: OER Operating Systems

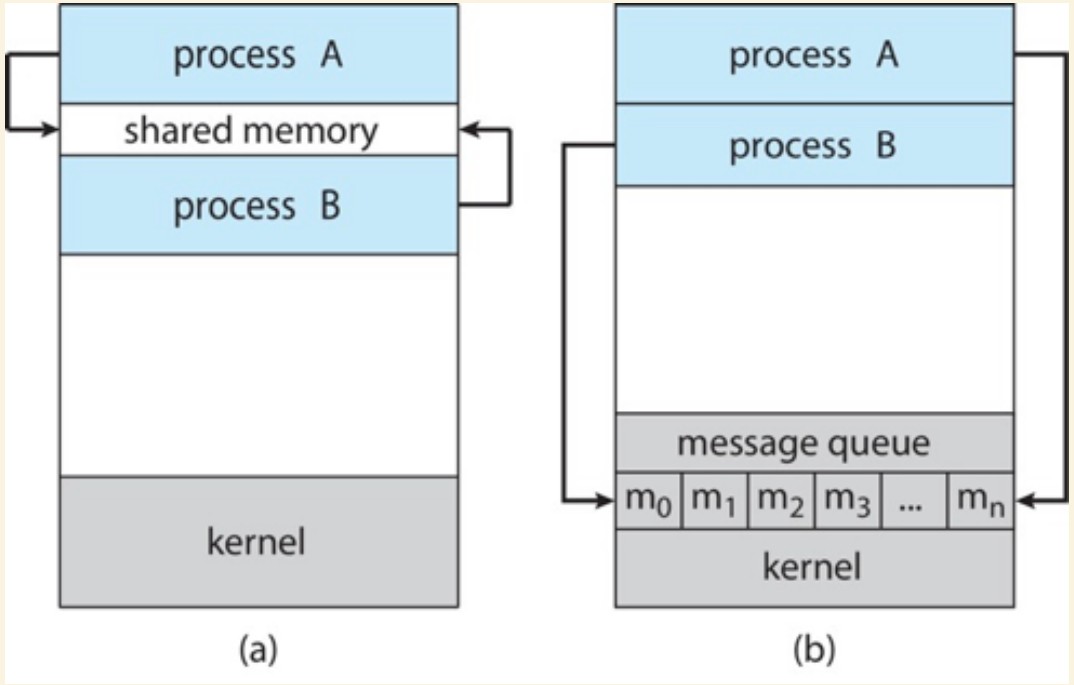
Mecanismos de IPC

Mecanismos de Comunicação

Critério	Classificação
Estrutura	Compartilhamento de memória ou troca de mensagens
Direção	Unidirecional ou bidirecional
Localidade	Local (mesma máquina) ou remota (via rede)
Sincronização	Bloqueante ou não bloqueante

Foco Nesta Aula

- Vamos focar em IPC por (a) Memória compartilhada e (b) Fila de Mensagem.

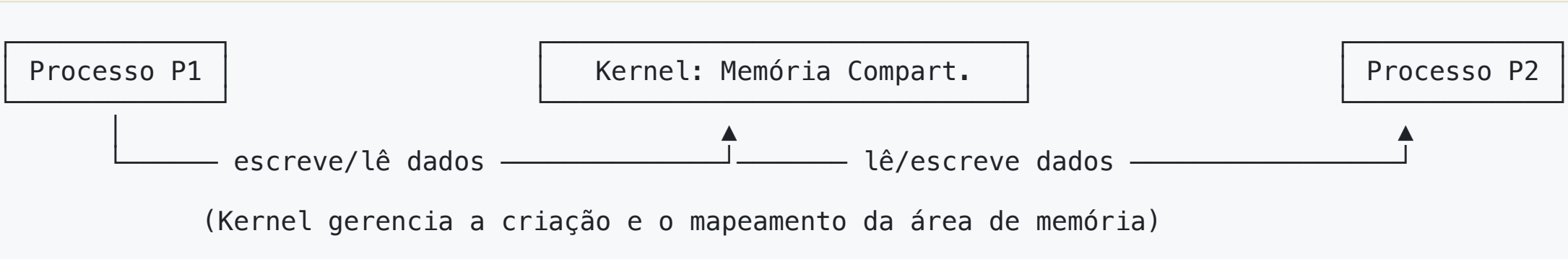


- Fonte da Imagem: A. Silberschatz *et. al*, **Operating Systems Concepts**, capítulo 3.

Mecanismos de IPC: Memória Compartilhada

- **Descrição**: Processos compartilham uma região de memória diretamente.
- **Vantagens**: Rápido, pois não há cópia de dados.
- **Desvantagens**: Complexidade no controle de concorrência. Requer sincronização (uso de mutex, semáforos) cuidadosa para evitar conflitos (race conditions).

Mecanismos de IPC: Memória Compartilhada (cont.)



- **Demo em C**: process_to_process_shared_memory.c.
- **Demo em Python**: process_to_process-with-sharedmemory.py por James Spurin.

Mecanismos de IPC: Memória Compartilhada (cont.)

Exemplo em Python:

```python
from multiprocessing import Process, Value

def processo_filho(shared_value):
    shared_value.value = 1024

if __name__ == "__main__":
    memoria_compartilhada = Value("i", 0)
    print("Valor inicial:", memoria_compartilhada.value)  # Saída: 0
    process = Process(target=processo_filho, args=(memoria_compartilhada,))
    process.start()
    process.join()
    print("Valor atualizado:", memoria_compartilhada.value)  # Saída: 1024
```

Problema Produtor-Consumidor

- Paradigma onde um processo produtor gera informações (e.g., adiciona itens) que são consumidas (e.g., remove itens) por um processo consumidor
  - Exemplo: web server e web browser.
- Problema de sincronização onde a ordem das operações precisa ser cuidadosamente controlada para evitar condições de corrida e garantir a integridade dos dados.
- Imagine o cenário abaixo:
  - O Produtor enche o buffer com dados rapidamente.
  - O Consumidor tenta ler o buffer, mas encontra-o cheio. Ele fica bloqueado esperando que o Produtor libere espaço.

Problema Produtor-Consumidor: Variantes

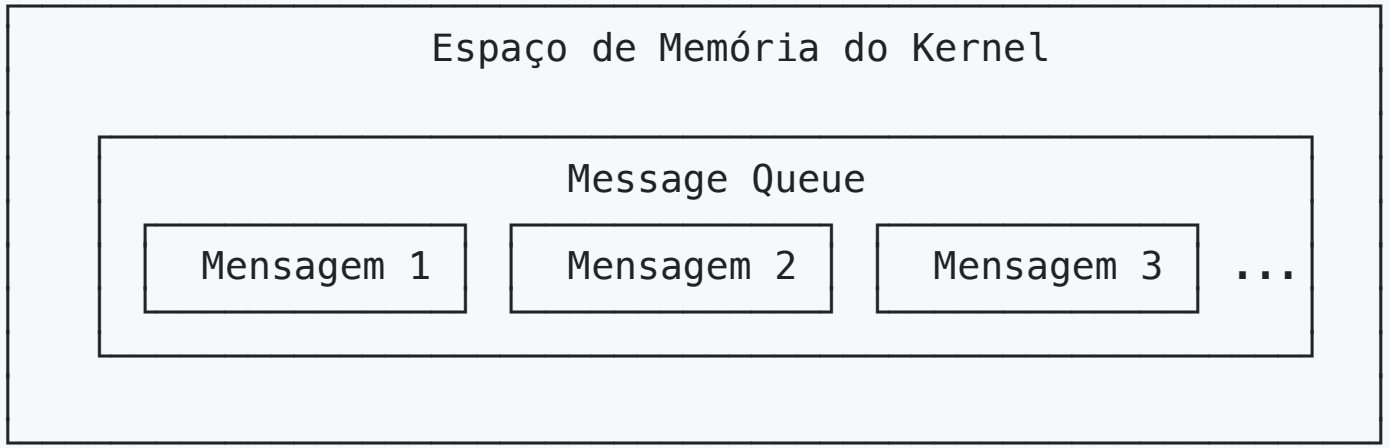
- **Buffer Ilimitado** (*Unbounded-Buffer*): Não impõe limites práticos ao tamanho do buffer.
  - *Produtor* nunca espera.
  - *Consumidor* espera se não houver buffer disponível para consumo.
- **Buffer com Tamanho Fixo** (*Bounded-Buffer*): Assume que existe um tamanho de buffer fixo.
  - *Produtor* deve esperar se todos os buffers estiverem cheios.
  - *Consumidor* espera se não houver buffer disponível para consumo.

Mecanismos de IPC: Fila de Mensagens

- **Descrição**: Processos enviam e recebem mensagens através de uma fila. A fila atua como um buffer.
  - Sem o uso de variáveis compartilhadas.
  - Envolve o kernel: Estrutura de dados FIFO controlada pelo sistema.
    - Overhead de sistema.
    - Chamadas `send(message)` e `receive(message)`
    - **POSIX:** `msgget`, `msgsnd`, `msgrcv`.
- **Vantagens**: Desacoplamento entre processos, fácil de escalar.
- **Desvantagens**: Pode introduzir latência devido à necessidade de enfileirar e desenfileirar mensagens.

Mecanismos de IPC: Fila de Mensagens (cont.)

(Kernel gerencia a criação, armazenamento e entrega das mensagens)



Mecanismos de IPC: Fila de Mensagens (cont.)

Demo/Exemplo em Python:

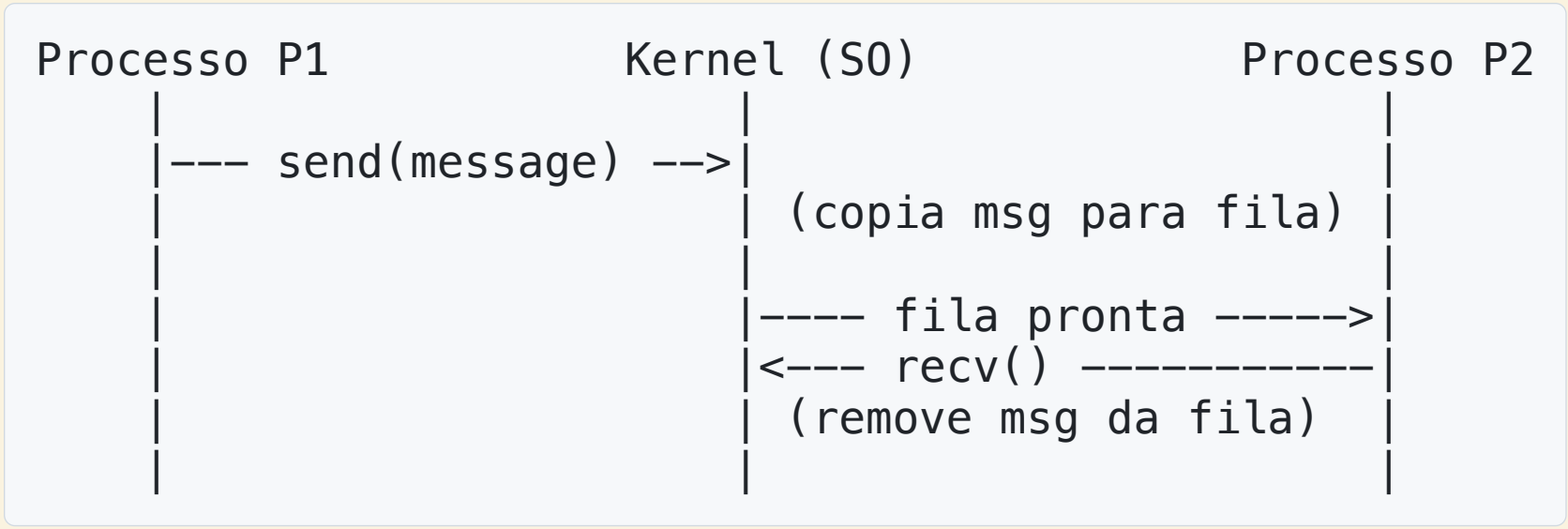
```python
import os
from multiprocessing import Process, Queue

def processo_filho(queue):
    queue.put("** Olá do processo filho! **")

if __name__ == "__main__":
    print("PID do processo pai:", os.getpid())
    queue = Queue()
    process = Process(target=processo_filho, args=(queue,))
    process.start()
    print("PID do processo filho:", process.pid)
    print(queue.get())  # Saída: Olá do processo filho!
    process.join()
```

Mecanismos de IPC: Fila de Mensagens (cont.)

O Kernel **intermedia** a comunicação, gerenciando a fila de mensagens.

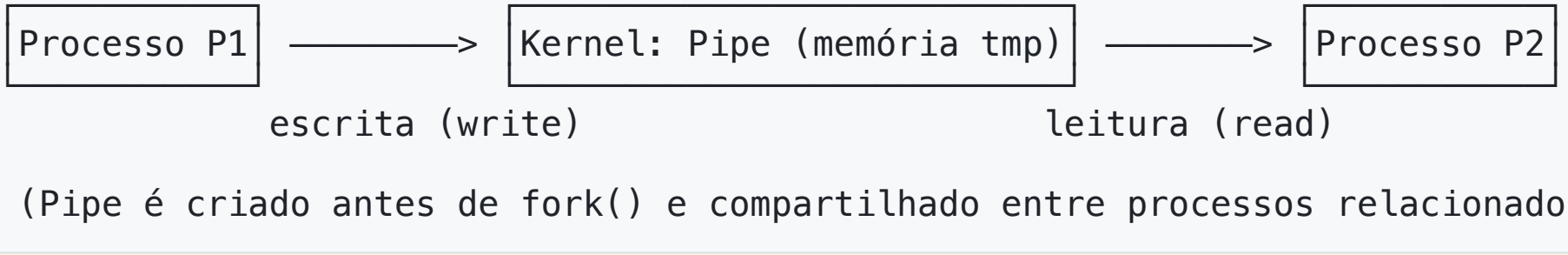


Mecanismos de IPC: Pipes e Named Pipes

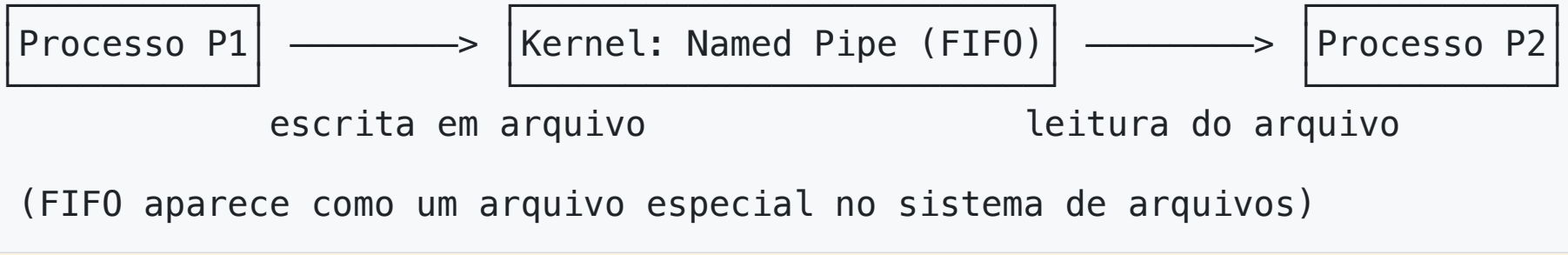
- **Descrição**: Canais de comunicação unidirecional, geralmente usados para comunicação entre processos relacionados (pai/filho).
  - **Named Pipes (FIFOs)**: Criados no sistema de arquivos, permitem comunicação entre processos independentes.
- **Vantagens**: Simples de implementar e eficientes para comunicação local.
- **Desvantagens**: Limitado a comunicação unidirecional.
- **Exemplo**: Um processo que gera dados e outro que os consome.
  - Comandos típicos: `pipe()`, `mkfifo()`

Mecanismos de IPC: Pipes e Named Pipes (cont.)

- Comunicação com **Pipe** Anônimo entre processos *relacionados* (pai-filho):
  - **Demo**: process_to_process-with-pipe.py por James Spurin.



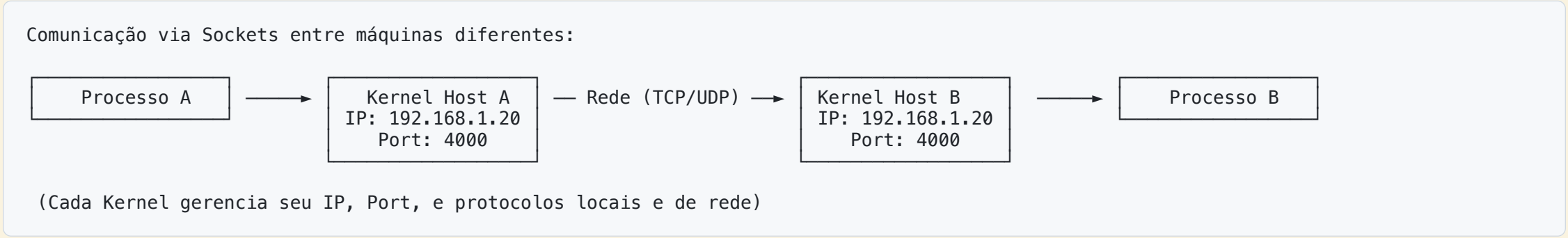
- Comunicação com **Named Pipe (FIFO)** entre processos *independentes*:
  - **Demo**: process_to_process-with-fifo.py por James Spurin.



Mecanismos de IPC: Sockets

- **Descrição**: Processos se comunicam como se estivessem em uma rede local.
  - Utilizam protocolos de comunicação (ex: TCP/IP).
  - Permitem comunicação entre processos locais ou remotos.
  - Base de sistemas distribuídos e cliente/servidor.
- **Vantagens**: Flexibilidade, pode ser usado para comunicação entre máquinas diferentes.
- **Desvantagens**: Mais complexo que outros mecanismos de IPC.
- **Exemplo**: Um servidor web e um cliente navegando na internet.
  - `SOCK_STREAM` (TCP) – conexão confiável.
  - `SOCK_DGRAM` (UDP) – comunicação mais leve.
  - **Comandos típicos:** `socket()`, `bind()`, `listen()`, `accept()`, `recv()`

Mecanismos de IPC: Sockets (cont.)



- Sockets locais usam o endereço 127.0.0.1 (loopback) com portas distintas para identificar conexões diferentes.
- Sockets remotos usam endereços IP reais e portas para estabelecer a comunicação sobre a rede.

Mecanismos de IPC: Sockets

Exemplo em Python:

```python
import socket
from multiprocessing import Process

def servidor():
    s = socket.socket()
    s.bind(("127.0.0.1", 5555))
    s.listen(1)
    conn, addr = s.accept()
    print("Conexão:", addr)
    conn.send(b"Oi, sou o servidor!")
    conn.close()

def cliente():
    s = socket.socket()
    s.connect(("127.0.0.1", 5555))
    print(s.recv(1024).decode())  # Saída: Oi, sou servidor!!
    s.close()

if __name__ == "__main__":
    server_process = Process(target=servidor)
    server_process.start()
    client_process = Process(target=cliente)
    client_process.start()
    server_process.join()
    client_process.join()
```

Conclusão

Conclusão: Resumo

- **IPC permite**:
  - Compartilhar dados com eficiência
  - Criar aplicações modulares e escaláveis, decompondo tarefas complexas em unidades menores que podem ser executadas simultaneamente por diferentes processos.
- A escolha do mecanismo de IPC depende fortemente dos **requisitos da aplicação**: frequência de acesso aos dados, necessidade de sincronização, arquitetura do sistema e restrições de desempenho.
- IPC introduz **complexidades adicionais**, como a necessidade de lidar com condições de corrida (race conditions) e deadlocks.

Conclusão:: Comparativo entre mecanismos de IPC

Mecanismo	Direção	Velocidade	Complexidade	Local/Remoto
Memória Compartilhada	Bidirecional	Alta	Alta	Local
Filas de Mensagem	Bidirecional	Média	Moderada	Local
Pipes	Unidirecional	Média	Baixa	Local
Sockets	Bidirecional	Variável	Alta	Ambos

Conclusão: Considerações de Projeto

Para escolher o mecanismo ideal:

- Volume de dados trocados
- Localização dos processos (mesma máquina ou rede)
- Necessidade de sincronização
- Facilidade de implementação
- Segurança e escalabilidade

Leitura adicional

- Capítulo 2 do livro **Sistemas Operacionais Modernos**, de A. TANENBAUM
- Capítulo 3 do livro: **Operating Systems Concepts**, de A. Silberchatz *et. al*.

Dúvidas e Discussão