Maximizando a Eficiência na Comunicação entre Processos: Explorando uma Abstração de Zero-Cópia

Vinicius Francisco da Silva, Henrique Cota de Freitas, Pedro Henrique Penna

[1]Instituto de Ciências Exatas e Informática – PUC Minas – Belo Horizonte – MG – Brasil

viniciusfrancisco@pucminas.br, cota@pucminas.br, pedrohenriquepenna@gmail.com

Abstract. *Operating systems with a microkernel architecture aim for an approach where essential functions are maintained in a minimalist core. Additional functionalities run as user mode processes. Since these services are separated, communication requires indirect message passing, which poses a challenge. Zero-copy communication optimizes data transfer between processes by eliminating intermediate memory copies between the sender and the recipient of a message. This reduces processing overhead and improves system efficiency. In this context, the present work aims to propose an abstraction of interprocess communication using the zero-copy technique, involving the direct transfer of data between processes or components without the need for intermediate data copying.*

Resumo. *Sistemas operacionais com arquitetura microkernel visam uma abordagem em que funções essenciais são mantidas em um núcleo minimalista, enquanto funcionalidades adicionais executam como processos em modo usuário. Como esses serviços estão separados, a comunicação requer a passagem de mensagens de forma indireta, tornando-se um desafio. A comunicação zero cópia otimiza a transferência de dados entre processos, eliminando cópias de memória intermediárias entre o remetente e o destinatário de uma mensagem. Isso reduz a sobrecarga de processamento e melhora a eficiência do sistema. Nesse contexto, o presente trabalho tem como objetivo propor uma biblioteca de comunicação entre processos, usando a técnica de zero cópia, que envolve a transferência direta de dados entre processos ou componentes sem a necessidade de copiar os dados intermediariamente.*

1. Introdução

A arquitetura *microkernel* é um modelo de sistema operacional que implementa apenas funções essenciais no *kernel*, como gerenciamento de processos, memória e comunicação entre processos, enquanto serviços não essenciais são executados em espaços de usuário separados, chamados servidores [Liedtke 1995]. Isso resulta em um *kernel* menor, mais seguro, flexível e escalável, permitindo a substituição de serviços sem afetar o núcleo do sistema, promovendo modularidade. No entanto, a extração de componentes como *device drivers* para o espaço de usuário aumenta a complexidade e a sobrecarga na comunicação entre processos.

A comunicação entre processos em um *microkernel* enfrenta desafios como sobrecarga, desempenho, gerenciamento de memória compartilhada, segurança, sincronização, escalabilidade, tolerância a falhas e gerenciamento de recursos. Este trabalho propõe uma

biblioteca de comunicação entre processos utilizando a técnica de *zero-copy*, que permite a transferência direta de dados entre processos sem replicação, aumentando a eficiência e desempenho na transmissão de informações.

2. Fundamentação Teórica

Nesta seção, são apresentados os conceitos de *microkernel* e os mecanismos de comunicação entre processos.

2.1. Sistemas Operacionais *Microkernel*

Um sistema operacional *microkernel* é caracterizado por possuir um conjunto mínimo de abstrações de *hardware* em sua camada principal, que opera em modo *kernel*, como o escalonador, suporte básico à comunicação entre processos e memória virtual [Herder 2005]. Outras funcionalidades, são implementadas em servidores independentes e executadas em espaço de usuário, contrastando com sistemas operacionais monolíticos, que incorporam todas essas funcionalidades dentro de um único núcleo.

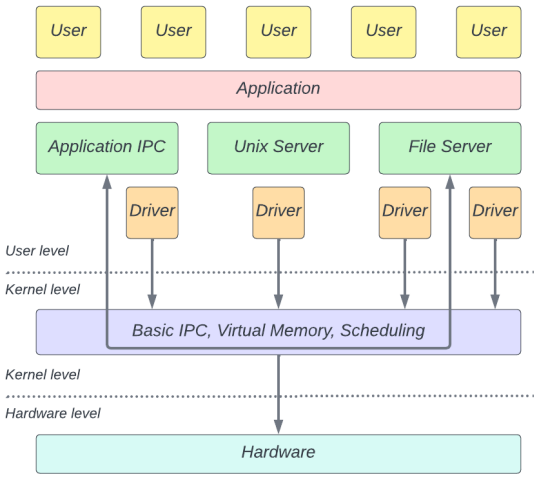

Figura 1. Arquitetura de um sistema operacional *microkernel*.

A Figura 1 mostra a disposição dos dispositivos de abstração de *hardware* em um sistema operacional *microkernel*. Para o funcionamento básico do sistema, existe uma comunicação direta entre o *kernel* e dois componentes que são executados no modo usuário: o *Application IPC*, que é um mecanismo que permite a comunicação entre processos a nível de usuário, e o *File Server*, um serviço que gerencia o acesso a arquivos e sistemas de arquivos, podendo ser implementado como um processo de usuário.

2.2. Mecanismos de Comunicação entre Processos

A comunicação entre processos possibilita a troca de informações entre processos. Porém, em um sistema *microkernel*, os servidores estão em modo usuário e são separados entre si, ou seja, cada servidor possui um espaço de memória exclusivo. Assim, a comunicação direta entre processos fica descentralizada, sendo necessário utilizar outras abordagens. Para uma comunicação entre processos síncrona, existem duas abordagens principais: comunicação por memória compartilhada e por troca de mensagens.

2.3. Comunicação por Memória Compartilhada

Regiões de memória compartilhadas (SM) são regiões de memória disponíveis geralmente para mais de uma entidade, podendo ser programas, *threads* ou processos. Esse acesso cria um espaço em que entidades podem compartilhar dados sem a necessidade de cópias adicionais e trocas de contexto (TC).

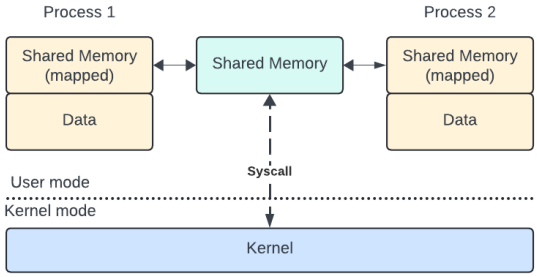

Figura 2. Mapeamento de uma região de memória compartilhada entre dois processos.

Para a comunicação entre processos, esta região é utilizada para disponibilização de informações. A memória compartilhada é criada pelo *kernel* e mapeada para o segmento de dados do espaço de endereço do processo solicitante. Supondo uma região de memória compartilhada mapeada para dois processos, o processo P1 grava dados no segmento de memória compartilhada. A Figura 2 mostra o mapeamento de uma região de memória compartilhada entre dois processos. Assim que os dados são escritos, ficam disponíveis para o processo P2. Esta forma de comunicação possui problemas de sincronização, pois em um espaço onde cada entidade tem permissão para ler e escrever, é necessário garantir exclusão mútua nesta seção crítica.

2.4. Comunicação por Troca de Mensagens

A comunicação feita via *kernel* aproveita recursos disponíveis no próprio *kernel* para estabelecer e gerenciar a comunicação entre diferentes processos. No entanto, ela necessita de um canal indireto para o estabelecimento da comunicação. Existem vários recursos que podem ser utilizados como canal de comunicação, tais como: *pipes*, filas de mensagens ou *sockets*. Suponha a passagem de dados entre dois processos, P1 e P2. O processo P1 faz uma chamada de sistema para enviar dados ao processo P2. A mensagem é copiada do espaço de endereço do processo P1 para o espaço do *kernel* durante a chamada de sistema para enviar a mensagem. Em seguida, o processo P2 faz uma chamada de sistema para receber a mensagem. A mensagem é copiada do espaço do *kernel* para o espaço de endereço do processo P2.

A passagem de dados entre processos com cópia ocorre com a mediação do *kernel*. O *microkernel* facilita essa operação copiando os dados do espaço de memória do processo de origem para um espaço compartilhado gerenciado pelo *kernel*. Posteriormente, o processo destinatário solicita ao *kernel* a cópia desses dados para o seu próprio espaço de memória. A Figura 3 mostra o funcionamento da passagem de dados entre os *buffers* de cada processo e do *kernel*, utilizando a abordagem de cópia da mensagem. A

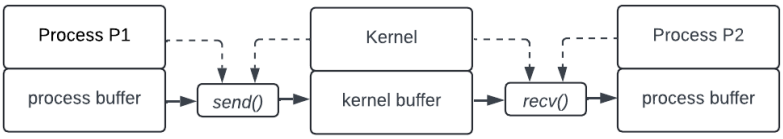

Figura 3. Passagem de mensagem com cópia tradicionalmente usada em sistemas operacionais *microkernel*.

transferência de dados em sistemas de arquitetura *microkernel* é descrita por Herder et al. [Herder et al. 2006].

3. Trabalhos Correlatos

Nesta seção, são apresentados estudos correlatos que contribuem para embasar o tema abordado no artigo.

3.1. Zerializer: Towards Zero-Copy Serialization

O artigo *"Zerializer: Towards Zero-Copy Serialization"* [Wolnikowski et al. 2021] discute a importância de alcançar entrada/saída (E/S) sem cópias (*zero-copy*) entre diferentes computadores. No entanto, a prática de serialização de dados para transmissão pode anular os benefícios do *zero-copy*, pois exige que a unidade central de processamento (CPU) leia, transforme e escreva os dados da mensagem, resultando em cópias adicionais. Para superar esse desafio, os autores propõem transferir a lógica de serialização para o caminho de *direct memory access* (DMA) usando *hardware* especializado. O estudo avaliou os benefícios e a viabilidade da utilização do Zerializer em aplicações do mundo real, focando no *software* de análise de rede *Deep Insight* da Intel. Os resultados indicaram que a inclusão desse módulo em um SmartNIC baseado em FPGA permitiria um processamento de aproximadamente 19 Gbps, enquanto um ASIC poderia alcançar 128 Gbps.

3.2. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems

O artigo *"The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems"* [Zhang et al. 2021] propõe o Demikernel, um sistema operacional projetado para dispositivos de baixa latência em centros de dados, alcançando latências na ordem de microssegundos (µs). Utilizando dispositivos que contornam o *kernel* (*kernel-bypass*), o Demikernel permite que aplicativos realizem E/S diretamente, eliminando o *kernel* do caminho de dados e, assim, alterando a arquitetura tradicional do SO. O objetivo é preencher a lacuna deixada pela remoção do *kernel*, fornecendo uma substituição de SO de caminho de dados geral para sistemas de microssegundos. Experimentos avaliaram o desempenho do Demikernel, incluindo latências de operações de eco (*echo*), execução de um servidor de relé UDP, testes de desempenho do Redis [Sanfilippo 2009] e avaliação do TxnStore, um sistema de armazenamento transacional distribuído. Em todos os testes, o Demikernel demonstrou latências competitivas, mostrando resultados positivos em comparação com outras soluções.

3.3. TAS - TCP Acceleration as an OS Service

O artigo *"TAS - TCP Acceleration as an OS Service"* [Kaufmann et al. 2019] apresenta um desafio crescente nos centros de dados, onde o processamento de pacotes *transmission control protocol* (TCP) consome uma parcela do ciclo de processamento CPUs dos servidores, especialmente para chamadas de procedimento remoto (RPCs). Existem abordagens que buscam otimizar esses processamentos, como evitar o *kernel* do sistema operacional, personalizar a pilha TCP para uma aplicação específica ou transferir o processamento de pacotes para *hardware* dedicado. No entanto, essas abordagens muitas vezes sacrificam segurança, agilidade ou generalidade em troca de eficiência. O artigo propõe uma solução chamada *acceleration as a service* TAS, que divide o processamento TCP comum para RPCs em *data centers* do *kernel* do SO e o executa como um serviço de caminho rápido em CPUs dedicadas, mantendo todas as funcionalidades de uma pilha TCP padrão, incluindo segurança, agilidade e generalidade. O estudo analisa a sobrecarga do processamento de pacotes TCP, considerando a arquitetura moderna de processadores. Em comparação com sistemas de transporte de *software* tradicionais, o TAS demonstrou melhorias no desempenho das operações de chamada de procedimento remoto (RPC). A latência das RPCs foi significativamente reduzida com o uso do TAS.

3.4. zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO

O artigo *"zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO"* [Stamler et al. 2022] propõe zIO. Um acelerador que executa em modo usuário e utiliza mecanisco *zero copy* para aplicações que possuem uso intenso de entrada e saída E/S. O zIO faz rastreamento de dados de E/S eliminando as cópias desnecessárias desses dados, garantindo também a consistência dos mesmos. Para isso em chamadas de recebimento da pilha IO a localização do *buffer* é registrado pelo zIO a partir de chamadas *Portable Operating System Interface* (POSIX). Com este registro o zIO filtra e localiza *buffers* intermediários através de *skiplists* que gravam informações que darão a localização dos mesmos. Após a cópia de dados em *buffers* intermediários e com a localização a partir das chamadas *memcpy* e *memmove*, o zIO registra o *buffer* de destino fornecido pela aplicação como o *buffer* original, juntamente com um *buffer* intermediário. Este registro filtra os *buffers* de E/S para rastreamento e eliminação de cópias. Os resultados da avaliação demonstram que o zIO melhora o desempenho de aplicativos em até 1,8 vezes no Linux e até 2,5 vezes nas pilhas de E/S que evitam o *kernel*, quando usado em conjunto com otimizações de persistência do receptor de rede.

3.5. RedLeaf: Isolation and Communication in a Safe Operating System

O artigo *"RedLeaf: Isolation and Communication in a Safe Operating System"* [Narayanan et al. 2020] propõe um sistema operacional chamado RedLeaf, construído completamente do zero usando a linguagem de programação Rust. O foco principal é entender como a segurança embutida na linguagem Rust pode impactar a estrutura de um sistema operacional. Uma das novidades trazidas pelo RedLeaf é uma abstração chamada "domínio de isolamento leve baseado em linguagem", que oferece uma unidade de ocultação de informações e isola falhas por meio de 5 princpios: *heap isolation*, *exchangeable types*, *ownership tracking*, *interface validation* e *cross-domain call proxying*. O estudo RedLeaf realizou uma série de testes de desempenho em diversos cenários para avaliar a eficácia do uso da linguagem Rust em sistemas de alto desempenho com isolamento de falhas. Alguns dos testes e seus resultados.

3.6. Userspace Bypass: Accelerating Syscall-intensive Applications

O artigo *"Userspace Bypass: Accelerating Syscall-intensive Applications"* [Zhou et al. 2023] propõe o Userspace Bypass (UB) para acelerar aplicações que fazem uso intensivo de chamadas ao sistema. A principal estratégia do UB é transferir instruções do espaço do usuário para o espaço do *kernel*. Ele monitora a execução de um aplicativo, rastreando as chamadas de sistema feitas por um *thread* específico. Quando detecta várias chamadas de sistema consecutivas em um curto período, identifica essas chamadas como "quentes". O UB analisa e registra as instruções de espaço de usuário executadas entre essas chamadas, traduzindo-as para o formato *Binary Translation Cache* (BTC). Os resultados mostram que o UB melhora significativamente o desempenho de aplicativos como Redis, com aumentos de 4,4% a 10,8% para tamanhos de dados de 1B a 4KiB, em uma máquina virtual com KPTI ativado. O UB também acelera operações de E/S em um *micro benchmark*, com melhorias de 30,3% a 88,3%.

3.7. z-READ: Towards Efficient and Transparent Zero-copy Read

O artigo *"z-READ: Towards Efficient and Transparent Zero-copy Read"* [Park et al. 2019] apresenta um esquema eficiente e transparente de leitura de (I/O) sem cópia. Ele permite que as aplicações obtenham os benefícios da leitura sem cópia através das interfaces tradicionais de leitura/escrita do (POSIX). O esquema é baseado em técnicas de remapeamento de páginas e *copy-on-write (CoW)* minimizando as sobrecargas de remapeamento de páginas, reduzindo o número de operações de *Translation lookaside buffer (TLB) shootdown* remotas. Os resultados experimentais mostram que o desempenho dos *workloads* intensivos em memória co-localizados pode ser afetado negativamente pelos workloads intensivos em (I/O) no caso de (I/O) baseado em cópia (até 1,96x de desaceleração nas configurações em memória), enquanto o z-READ incorre apenas em até 1,07x de desaceleração para a respectiva configuração.

3.8. Snap: a Microkernel Approach to Host Networking

O artigo *"Snap: a Microkernel Approach to Host Networking"* [Marty et al. 2019] propõe um novo sistema de rede *host* inspirado em *microkernels* chamado Snap. Um sistema de rede em espaço de usuário que suporta as necessidades em constante evolução do Google com módulos flexíveis que implementam uma gama de funções de rede, incluindo comutação de pacotes de borda, virtualização para nossa plataforma de nuvem, execução de políticas de modelagem de tráfego e um serviço de mensagens confiáveis de alto desempenho e RDMA-*like*. O Snap apresentou uma melhoria substancial na taxa de transferência e latência em comparação com a pilha de TCP/IP do *kernel* Linux, alcançando uma taxa de transferência de 82.2 Gbps usando apenas um núcleo, em comparação com os 22 Gbps do TCP. Além disso, a latência do Snap/Pony foi inferior a 10µs em configurações otimizadas.

3.9. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels

O artigo *"SkyBridge: Fast and Secure Inter-Process Communication for Microkernels"* [Mi et al. 2019] apresenta uma nova facilidade de comunicação projetada para comunicação inter-processos síncrona em *microkernels*, eliminando a necessidade do envolvimento do *kernel*. O SkyBridge permite que um processo mude diretamente para o espaço de endereço do processo de destino e invoque a função necessária, mantendo o

isolamento do espaço de endereço virtual, facilitando sua integração com *microkernels* existentes. O SkyBridge utiliza o recurso de virtualização *VMFUNC* para comunicação eficiente entre processos, reduzindo a latência de IPC. Em testes de IPC em um único núcleo, alcançou melhorias de 1.49x, 5.86x e 19.6x para os *microkernels* seL4, Fiasco.OC e Zircon, respectivamente. Para IPC entre núcleos, as melhorias foram ainda mais significativas, com aumentos de 16.08x, 20.31x e 49.76x.

3.10. Cornflakes: Zero-Copy Serialization for Microsecond-Scale Networking

O artigo *"Cornflakes: Zero-Copy Serialization for Microsecond-Scale Networking"* [Raghavan et al. 2023] apresenta o Cornflakes, um *framework* de serialização *zero-copy* para redes de microssegundos, visando minimizar a sobrecarga de cópia de dados na rede. Cornflakes utiliza um novo algoritmo de serialização que encapsula dados em um único *buffer*, permitindo envio direto pela rede. A biblioteca híbrida de serialização usa *scatter-gather* e é co-projetada com uma pilha de rede integrada, oferecendo uma API de serialização geral que gerencia acessos de *scatter-gather* à memória do aplicativo, compatível com NICs de *commodity* em servidores modernos. A avaliação do Cornflakes buscou responder a questões sobre desempenho e flexibilidade, comparando-o com abordagens tradicionais de serialização, analisando seu desempenho com diferentes distribuições de tamanhos de campos em mensagens e sua capacidade de integração em sistemas existentes, bem como sua compatibilidade com diferentes NICs.

4. Estudo comparativo

A Tabela 1 resume as principais características exploradas nos trabalhos relacionados e como a proposta do presente trabalho diferencia das demais.

	zero cópia	**comunicação sincrona**	**serialização**	**passagem de dado via memória comparilhada**	***kernel bypass***	**reduzir *overhead***
[Zhang et al. 2021]	X				X	
[Kaufmann et al. 2019]				X		
[Stamler et al. 2022]	X		X	X	X	
[Narayanan et al. 2020]	X	X	X			
[Zhou et al. 2023]						X
[Mi et al. 2019]	X	X			X	X
[Raghavan et al. 2023]	X		X	X	X	
[Marty et al. 2019]	X	X				
[Wolnikowski et al. 2021]	X		X			
[Park et al. 2019]	X					
[Penna 2021]	X	X	X	X	X	

Tabela 1. Comparação entre trabalhos relacionados e recursos utilizados

Na Tabela 1, a primeira coluna mostra os trabalhos que implementam zero cópia, uma técnica fundamental que a maioria dos trabalhos adota. Este artigo propõe uma biblioteca de comunicação utilizando zero cópia. A segunda coluna apresenta trabalhos que

buscam minimizar o *overhead* em chamadas de sistema, foco que não é abordado na proposta, pois o objetivo é remover as chamadas de sistema e implementar uma comunicação em espaço de usuário sem envolver o *kernel*. A terceira coluna apresenta projetos que utilizam comunicação síncrona entre processos, enquanto a proposta deste artigo utiliza memória compartilhada com semáforos e *mutex* para controle de acesso concorrente. A quarta coluna lista trabalhos que utilizam serialização para envio entre processos ou dispositivos; nossa proposta aplica serialização na comunicação entre processos via memória compartilhada. A quinta coluna mostra trabalhos que utilizam memória compartilhada para diversos problemas; nossa proposta visa usá-la para troca de mensagens entre processos. A sexta coluna apresenta projetos que utilizam *kernel bypass*, recurso também adotado neste artigo ao mover a comunicação entre processos para o espaço de usuário.

Nossa proposta utiliza quase todos os recursos listados, exceto a minimização do *overhead* nas chamadas de sistema, pois a abstração de comunicação em espaço de usuário elimina a necessidade frequente de chamadas *send* e *receive*, tornando essa minimização desnecessária. Outros recursos são utilizados para melhorar o desempenho da comunicação entre processos.

5. Medotologia

Esta seção apresenta as ferramentas adotadas para o desenvolvimento da proposta do artigo, assim como o método pelo qual o desenvolvimento será realizado.

Para obter resultados, foram utilizados sistemas operacionais baseados em UNIX, com implementação no padrão POSIX. A programação foi realizada em C, utilizando o GNU *Compiler Collection* (GCC) e o GNU *Debugger* (GDB). Os testes foram realizados em uma máquina GNU/Linux com 16GB de RAM e processador Intel Core i7-1165G7 @ 2.80GHz. A biblioteca proposta será desenvolvida para fácil implementação em sistemas *microkernel*, com testes de integração no sistema *microkernel* Nanvix, que suporta chamadas de sistema POSIX.

Para analisar o desempenho, foram definidos três experimentos. O primeiro experimento proposto neste estudo compreende a análise comparativa de desempenho entre a "biblioteca implementada", referida como a proposta do artigo, e a conexão *socket pipe*. Denominado como "*pingpong*", o experimento envolveu a troca de mensagens entre dois processos, variando o tamanho do *buffer* de 0,064KB a 8KB, dobrando o valor a cada execução. O segundo experimento compreendeu a taxa de injeção de mensagens por segundo em uma conexão *pipe* e a "biblioteca implementada", analisando como intervalos de injeção e tamanhos de mensagens afetam a eficiência da comunicação entre processos. O experimento foi realizado variando o tempo de 0,5 até 5 segundos, também variando o tamanho das mensagens entre 100, 500, 1000 e 2000, todas com tamanho fixo de 65KB. O terceiro experimento teve o objetivo de analisar e comparar o desempenho entre a conexão *pipe* e a "biblioteca implementada" em relação à latência na transferência de mensagens, variando o tamanho das mensagens de 64B até 8KB, dobrando o valor a cada execução.

6. Implementação

Para a implementação, foi utilizada a linguagem de programação C. A parte central da biblioteca está em uma estrutura implementada com o comando "*struct*" em C, chamada "*ipc*" como mostra a Listing 1.

```c
typedef struct ipc {
    __pid_t first_pid;
    __pid_t second_pid;
    shmseg *shm;
} ipc_t;
```

Listing 1. Implementação da estrutura principal da biblioteca denominada "*ipc*".

Ela cria a memória compartilhada por meio da função "*ipc_init()*". A Listing 2 mostra a implementação realizada.

```c
sem_t empty, mutex, full;
pthread_t writer, reader;
static void ipc_init(__pid_t p1, __pid_t p2){
    /* ipc library reference */
    ipc_t ipc;
    /* IPC receiving process */
    ipc.first_pid  = p1;
    ipc.second_pid = p2;
    /* Shared memory initialized */
    ipc.shm = shm_init();
    semaphore_init(mutex, 0);
    semaphore_init(empty, 0);
    semaphore_init(full, 0 );
    /* Semaphore initialized */
    phthread_create(&writer, NULL, ipc_send, NULL);
    pthread_create(&reader, NULL, ipc_receive, NULL);
    pthread_join(writer, NULL);
    pthread_join(reader, NULL);
}
```

Listing 2. Função que inicializa os agentes do processo de comunicação.

Para que os processos se comuniquem, a biblioteca fornece as funções "*ipc_send(msg)*" e "*ipc_receive(msg)*". Ambas implementam um sistema de sincronização utilizando semáforos para acesso à sessão crítica (memória compartilhada). A Listing 3 mostra a implementação das funções.

```c
static void ipc_send(ipc_t ipc, message_buffering* mb){
    while(true){
        /* Check if message buffering's process is allowed to
           write the message */
        if(mb.pid == ipc.first_pid || mb.pid == ipc.second_pid){
                semaphore_wait(empty);
                semaphore_wait(mutex);
                /* Process writing from shared memory */
                shm_write(ipc.shm, message_buffering.message);
                semaphore_signal(mutex);
                semaphore_signal(full);
        }else{
                /* Process not allowed to send message */
                ipc_panic("[Permission Denied!]");
```

```
14            }
15        }
16    }
17
18    static void ipc_receive(ipc_t ipc, message_buffering* mb){
19        while(true){
20                /* Check if message buffering's process is allowed
                      to receive the message */
21            if(mb.pid == ipc.first_pid || mb.pid == ipc.second_pid){
22                    semaphore_wait(full);
23                    semaphore_wait(mutex);
24                    /* Process reading from shared memory */
25                    shm_read(ipc.shm, message_buffering.
                          received_buffer);
26                    semaphore_signal(mutex);
27                    semaphore_signal(empty);
28            }else{
29                    /* Process not allowed to receive message */
30                    ipc_panic("[Permission Denied!]");
31            }
32        }
33    }
```

Listing 3. Funções de envio e recebimento de mensagens.

Cada estrutura denominada "*ipc*" vincula dois processos pelos seus identificadores de processo (PIDs). Cada processo possui um *buffer* para recebimento de mensagens, denominado "*message buffering*". As funções "*ipc_send(msg)*" e "*ipc_receive(msg)*" transferem mensagens entre os buffers e a memória compartilhada, que é implementada como uma fila. A implementação utiliza semáforos para controle de acesso. A Figura 4 mostra a estrutura da implementação.

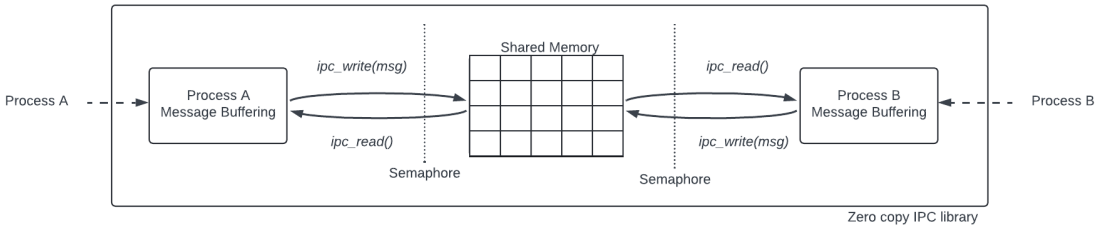

Figura 4. Implementação da biblioteca de comunicação entre processos.

Ao final da comunicação, a função "*ipc_destroy()*" libera a memória compartilhada e desvincula os processos como mostra a Listing 4.

```
1    static void ipc_destroy(ipc_t ipc){
2        /* unlink process of interprocess communication "ipc" */
3        process_ipc_destroy(ipc.first_pid);
4        process_ipc_destroy(ipc.second_pid);
5        message_buffering_ipc_destroy(ipc);
```

```
6       shm_destroy(ipc.shm);
7       /* Free ipc reference */
8       free(ipc);
9   }
```

Listing 4. Destruição do processo de comunicação.

7. Experimentos

Para a extração dos resultados, inicialmente foi executado um experimento com o objetivo de analisar o desempenho relacionado ao tempo de execução da proposta deste artigo, denominada "biblioteca implementada", em comparação com a conexão *socket pipe*. No primeiro experimento, as implementações foram executadas no formato *"ping-pong"*, onde um número x de mensagens é enviado em formato de *buffer* do processo 1 para o processo 2. O processo 2 realiza a leitura e envia novamente o *buffer* para que o processo 1 faça a leitura. Os testes são executados variando x, que é o número de mensagens presentes no *buffer*, começando com tamanho de 0,064KB até 8KB, dobrando o valor em cada execução.

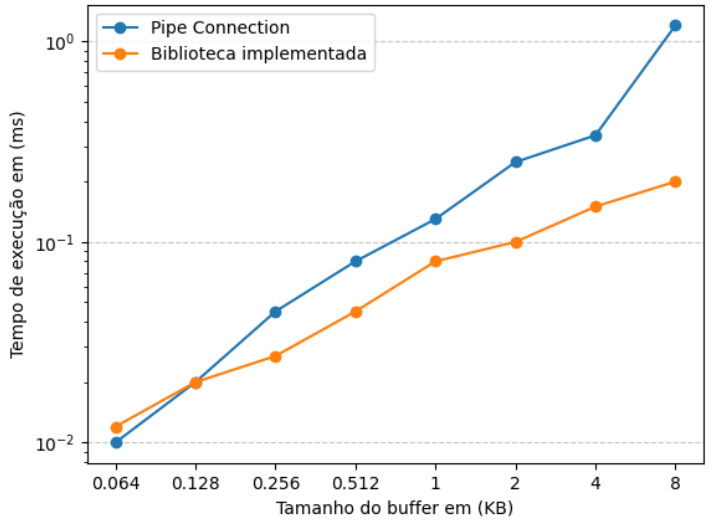

Figura 5. Desempenho no envio e recebimento de mensagem no formato "*ping-pong*".

O primeiro experimento buscou analisar o tempo de execução das conexões *pipe* e da "biblioteca implementada" em função do tamanho do *buffer*. O objetivo foi verificar se a proposta deste artigo mantém um desempenho superior à conexão *pipe* à medida que o tamanho do *buffer* aumenta. A Figura 5 mostra que, ao dobrar o tamanho do *buffer*, o tempo da conexão *pipe* dobra, enquanto a "biblioteca implementada" apresenta um aumento menos acentuado no tempo de execução, demonstrando melhoria no desempenho. A conexão *pipe* torna-se menos eficiente com o aumento do *buffer* devido à necessidade de copiar os dados para o espaço do *kernel*, o que consome tempo e recursos de forma desproporcional. Em contraste, a "biblioteca implementada" realiza operações diretamente no espaço do usuário, eliminando a cópia para o *kernel* e sendo mais eficiente para a transferência de dados.

No segundo experimento, mediu-se a taxa de injeção de mensagens por segundo em uma conexão *pipe* e na "biblioteca implementada". Foram enviadas mensagens de 64 *bytes* através de ambos os métodos, variando o número de mensagens por injeção e os intervalos de injeção, para compreender como esses fatores afetam a eficiência da comunicação entre processos.

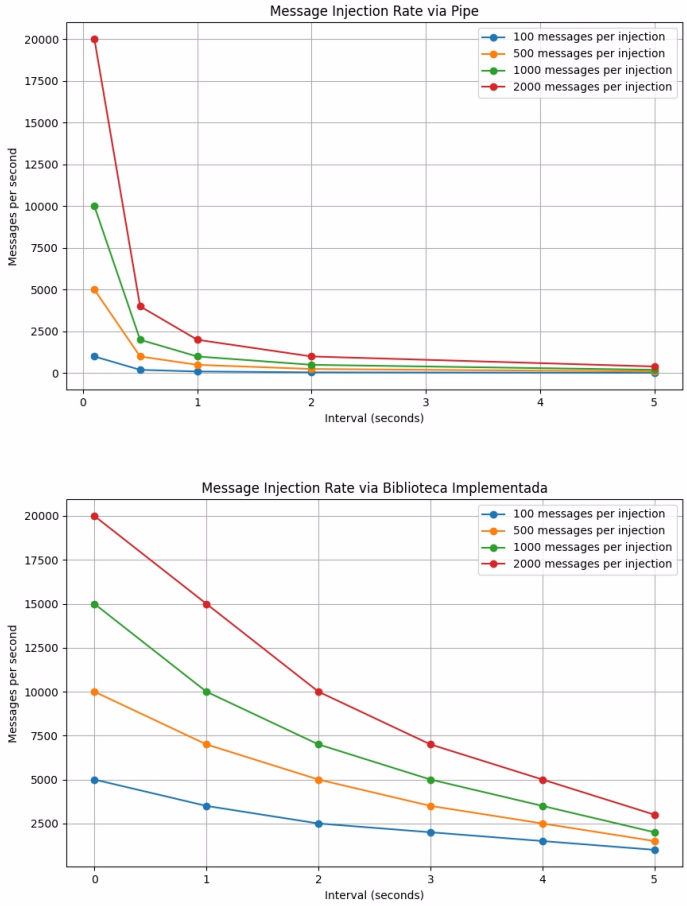

Figura 6. Taxa de injeção de envio de mensagens em segundos.

Na Figura 6, o gráfico superior mostra os resultados da comunicação via *pipe*. Observa-se uma queda acentuada na taxa de mensagens por segundo conforme o volume de mensagens aumenta, especialmente para volumes de 500 e 1000 mensagens por injeção. O sistema mantém uma alta taxa para 100 mensagens por injeção, mas a eficiência diminui com o aumento do volume. No gráfico inferior, que representa a comunicação via "biblioteca implementada", a taxa de mensagens por segundo é mais alta para todos os volumes de mensagens. Além disso, a taxa permanece estável mesmo com o aumento do volume, indicando que a "biblioteca implementada"lida melhor com grandes volumes de dados sem degradação de desempenho.

No terceiro experimento, o objetivo foi analisar e comparar o desempenho entre duas implementações de comunicação entre processos: *pipe* e a "biblioteca implementada", em termos de latência à medida que o tamanho da mensagem aumenta. A análise foi realizada medindo o tempo necessário para um processo enviar uma mensagem até

que o segundo processo a receba. Variamos o tamanho da mensagem de 64B a 8KB, dobrando o tamanho a cada execução.

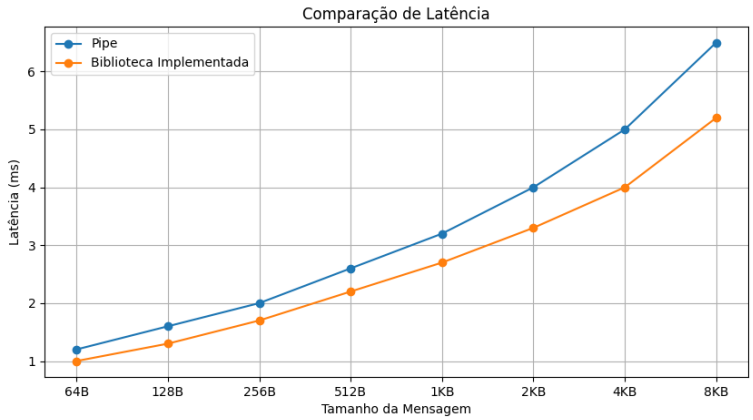

Figura 7. Latência na transferência de mensagens em milisegundo.

Os resultados apresentados na figura 7 revelam que a latência aumenta com o tamanho da mensagem para ambas as implementações. No entanto, a taxa de aumento é maior para a implementação *pipe* em comparação com a "biblioteca implementada" para todos os valores de tamanho de mensagem.

8. Conclusão e Trabalhos Futuros

Neste artigo, exploramos uma biblioteca de comunicação entre processos baseada na técnica de *zero-copy*, com o objetivo de otimizar a transferência de dados em sistemas operacionais com arquitetura *microkernel*. Através dos experimentos realizados, demonstramos que a "biblioteca implementada" apresenta um desempenho superior em comparação com a comunicação entre processos via *pipe* tradicional, especialmente à medida que o tamanho do *buffer* aumenta. A técnica de *zero-copy* mostrou-se eficaz na redução da sobrecarga causada pelas cópias intermediárias de dados, resultando em uma melhoria na eficiência e desempenho do sistema. Os resultados obtidos indicam várias direções para trabalhos futuros. Implementar a técnica de comunicação *remote procedure call* (RPC) em memória compartilhada, investigar a integração de otimizações de *hardware* explorando o *direct memory access* (DMA) pode aumentar ainda mais a eficiência da comunicação *zero-copy*. Avaliar o desempenho da "biblioteca implementada" em aplicações de tempo real, analisando seu comportamento sob diferentes cargas de trabalho. Essas direções futuras podem ampliar o impacto da técnica zero-copy proposta neste artigo e contribuir para a eficiência na comunicação entre processos em sistemas operacionais modernos.

Referências

Herder, J. N. (2005). Towards a true microkernel operating system. *Master's thesis, Vrije Universiteit Amsterdam*, (2005).

Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. (2006). Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89.

Kaufmann, A., Stamler, T., Peter, S., Sharma, N. K., Krishnamurthy, A., and Anderson, T. (2019). Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16.

Liedtke, J. (1995). On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 237–250, New York, NY, USA. Association for Computing Machinery.

Marty, M., de Kruijf, M., Adriaens, J., Alfeld, C., Bauer, S., Contavalli, C., Dalton, M., Dukkipati, N., Evans, W. C., Gribble, S., et al. (2019). Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413.

Mi, Z., Li, D., Yang, Z., Wang, X., and Chen, H. (2019). Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15.

Narayanan, V., Huang, T., Detweiler, D., Appel, D., Li, Z., Zellweger, G., and Burtsev, A. (2020). {RedLeaf}: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39.

Park, J., Min, C., Yeom, H. Y., and Son, Y. (2019). z-read: Towards efficient and transparent zero-copy read. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 367–371. IEEE.

Penna, P. H. (2021). *Nanvix: A Distributed Operating System for Lightweight Manycore Processors*. PhD thesis, Université Grenoble Alpes [2020-....]; Pontifícia universidade católica de ....

Raghavan, D., Ravi, S., Yuan, G., Thaker, P., Srivastava, S., Murray, M., Penna, P. H., Ousterhout, A., Levis, P., Zaharia, M., et al. (2023). Cornflakes: Zero-copy serialization for microsecond-scale networking. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 200–215.

Sanfilippo, S. (2009). Redis: An open source, in-memory data structure store.

Stamler, T., Hwang, D., Raybuck, A., Zhang, W., and Peter, S. (2022). {zIO}: Accelerating {IO-Intensive} applications with transparent {Zero-Copy}{IO}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 431–445.

Wolnikowski, A., Ibanez, S., Stone, J., Kim, C., Manohar, R., and Soulé, R. (2021). Zerializer: Towards zero-copy serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 206–212.

Zhang, I., Raybuck, A., Patel, P., Olynyk, K., Nelson, J., Leija, O. S. N., Martinez, A., Liu, J., Simpson, A. K., Jayakar, S., et al. (2021). The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 195–211.

Zhou, Z., Bi, Y., Wan, J., Zhou, Y., and Li, Z. (2023). Userspace bypass: Accelerating syscall-intensive applications. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 33–49.