

RELATÓRIO DE LABORATÓRIO 4

SISTEMAS OPERACIONAIS

VICTOR MONEGO

ENGENHARIA ELETRÔNICA – UTFPR

16 DE ABRIL DE 2024

1.Introdução Geral

O seguinte relatório diz respeito ao Laboratório 04 de Sistemas Operacionais, focado na Comunicação Inter Processos por Filas de Mensagens POSIX.

Para a realização das análises a seguir, foi utilizada a plataforma WSL(Ubuntu) no Windows 11, e os códigos foram feitos usando o programa Notepad++.

2.Comunicação simples entre processos A e B

A figura 01 abaixo apresenta o código “processA_transmitter.c”, que é o código do processo A que enviará as mensagens.

```
// UNIVERSIDADE TECNOLOGICA FEDERAL DO PARANA
// DEPARTAMENTO ACADEMICO DE ENGENHARIA ELETRONICA
// VICTOR AUGUSTO DEL MONEGO - 2378345
// Codigo processA_transmitter.c : gera numeros aleatorios e envia mensagens para uma fila
POSIX
// No prompt do Linux: compilar usando: gcc -o processA_transmitter processA_transmitter.c -
lrt

#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <unistd.h>

// operating system check
#if defined(_WIN32) || (!defined(__unix__) && !defined(__unix) && (!defined(__APPLE__) ||
!defined(__MACH__)))
#warning Este codigo foi planejado para ambientes UNIX (Linux, *BSD, MacOS). A compilacao e
execucao em outros ambientes e responsabilidade do usuario.
#endif
#define QUEUE "/my_queue"

int main (int argc, char *argv[]){

    mqd_t queue;//descricao da fila
    struct mq_attr attr;//define os atributos da fila
    int message;//declara a variavel que irá receber a mensagem
    attr.mq_maxmsg = 10;//isto representa a capacidade do vetor de mensagens
    attr.mq_msgsize = sizeof(message);//especifica o tamanho de cada mensagem
```

```

attr.mq_flags    = 0;
umask(0); //mascara de permissão

// esta seção do código é referente a criação da fila com permissões 0666
if((queue = mq_open(Queue, O_RDWR|O_CREAT, 0666, &attr)) < 0) {
    perror("mq_open"); //imprime mensagem de erro
    exit(1);
}
for(;;){
    message = random() % 100; //message recebe um valor aleatório entre 0 e 99

    if (mq_send (queue, (void*) &message, sizeof(message), 0) < 0) { //verifica
se houve erro no envio
        perror("mq_send"); //caso erro, envia mensagem de erro
        exit(1);
    }
    printf("Mensagem enviada com valor %d\n", message); //imprime a mensagem
transmitida
    sleep(1);
}
}

```

Figura 1: Código processA_transmitter.c

A figura 02 abaixo apresenta o código “processB_receiver”, que é o código do processo receptor das mensagens.

```

// UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
// DEPARTAMENTO ACADEMICO DE ENGENHARIA ELETRÔNICA
// VÍCTOR AUGUSTO DEL MONTEGO - 2378345
// Código processB_receiver.c : recebe mensagens de uma fila de mensagens POSIX
// No prompt do Linux: compilar usando: gcc -o processB_receiver processB_receiver.c -lrt

#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <unistd.h>

// operating system check
#if defined(_WIN32) || (!defined(__unix__) && !defined(__unix) && (!defined(__APPLE__) ||
!defined(__MACH__)))
#warning Este código foi planejado para ambientes UNIX (Linux, *BSD, MacOS). A compilação e
execução em outros ambientes é responsabilidade do usuário.
#endif
#define QUEUE "/my_queue"
int main (int argc, char *argv[])
{
    mqd_t queue;

    //descrição da fila

    struct mq_attr attr;

    //define os atributos da fila

```

```

int message;

//declara a variavel que irá receber a
mensagem
attr.mq_maxmsg = 10;

//isto representa a capacidade do vetor de mensagens
attr.mq_msgsize = sizeof(message);

//especifica o tamanho de cada mensagem
attr.mq_flags = 0;

umask(0);

//mascara de permissão

// esta seção do código é referente a criação da fila com permissões 0666
if((queue = mq_open(Queue, O_RDWR|O_CREAT, 0666, &attr)) < 0)
{
    perror("mq_open");

    //imprime mensagem de erro
    exit(1);
}
//esta seção se refere à recepção de mensagens
for(;;)
{
    if((mq_receive(queue, (void*) &message, sizeof(message), 0)) < 0)
        //verifica se houve erro de
recepção
    {
        perror("mq_receive:");

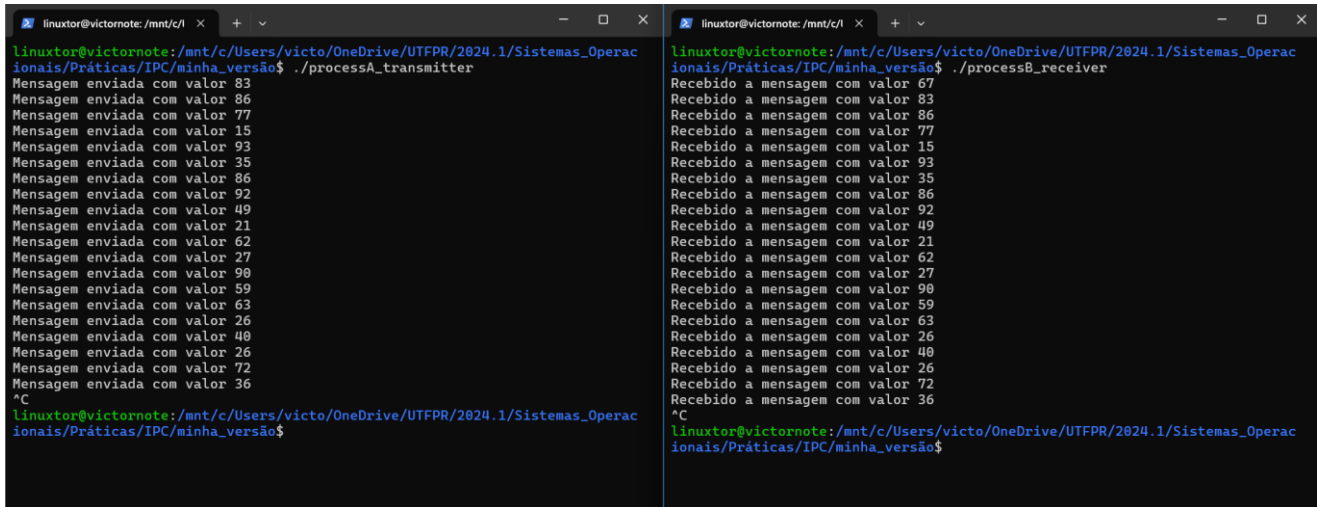
        //imprime mensagem de erro
        exit(1);
    }
    printf("Recebido a mensagem com valor %d\n", message);
    //imprime um
aviso de recebimento de mensagem
}
}

```

Figura 2: Código processB_receiver.c

O princípio de funcionamento é de que o processo A cria um buffer de mensagens de um tamanho estipulado no próprio código, e gera números aleatórios para usar como mensagens, que preenchem o vetor. O processo B então “abre” este buffer e lê as mensagens. Caso já existam mensagens no buffer, ele lê todas sequencialmente. Caso o processo B não seja executado, o processo A envia mensagens até encher o buffer, e após isso não consegue mandar mais mensagens, portanto o algoritmo indiretamente “pausa” até o processo B começar a ler as mensagens.

A figura 3 abaixo representa a execução de ambos os processos simultaneamente.



```
linuxor@victornote: /mnt/c/l \n linuxor@victornote: /mnt/c/l \n\nlinuxor@victornote: /mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/IPC/minha_versão$ ./processA_transmitter\nMensagem enviada com valor 83\nMensagem enviada com valor 86\nMensagem enviada com valor 77\nMensagem enviada com valor 15\nMensagem enviada com valor 93\nMensagem enviada com valor 35\nMensagem enviada com valor 86\nMensagem enviada com valor 92\nMensagem enviada com valor 49\nMensagem enviada com valor 21\nMensagem enviada com valor 62\nMensagem enviada com valor 27\nMensagem enviada com valor 90\nMensagem enviada com valor 59\nMensagem enviada com valor 63\nMensagem enviada com valor 26\nMensagem enviada com valor 40\nMensagem enviada com valor 26\nMensagem enviada com valor 72\nMensagem enviada com valor 36\n^C\nlinuxor@victornote: /mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/IPC/minha_versão$ \n\nlinuxor@victornote: /mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/IPC/minha_versão$ ./processB_receiver\nRecebido a mensagem com valor 67\nRecebido a mensagem com valor 83\nRecebido a mensagem com valor 86\nRecebido a mensagem com valor 77\nRecebido a mensagem com valor 15\nRecebido a mensagem com valor 93\nRecebido a mensagem com valor 35\nRecebido a mensagem com valor 86\nRecebido a mensagem com valor 92\nRecebido a mensagem com valor 49\nRecebido a mensagem com valor 21\nRecebido a mensagem com valor 62\nRecebido a mensagem com valor 27\nRecebido a mensagem com valor 90\nRecebido a mensagem com valor 59\nRecebido a mensagem com valor 63\nRecebido a mensagem com valor 26\nRecebido a mensagem com valor 40\nRecebido a mensagem com valor 26\nRecebido a mensagem com valor 72\nRecebido a mensagem com valor 36\n^C\nlinuxor@victornote: /mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/IPC/minha_versão$
```

Figura 3: Execução da comunicação A - B

Observando os dados enviados pelo processo A e recebidos pelo processo B, percebemos que todos os dados escritos são lidos, logo nenhuma informação que foi transmitida se perde, e é lida normalmente. O sistema utilizado, de criar um buffer de mensagens e preenchê-lo, é eficiente em garantir com segurança que as mensagens não se percam na pipeline, e a transmissão ocorra com êxito.

Tendo isso em mente, existe outra implementação possível.

3. Comunicação entre processos por memória compartilhada

A figura 4 abaixo representa o código “sharedmem_transmitter.c”, onde um processo aloca, cria e mapeia uma memória compartilhada e envia mensagens de forma similar ao processo A.

```
// UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ\n// DEPARTAMENTO ACADEMICO DE ENGENHARIA ELETRÔNICA\n// VICTOR AUGUSTO DEL MONEGO - 2378345\n// Arquivo sharedmem_transmitter.c: cria e usa uma área de memória compartilhada. Apenas\n// escreve. Ademais utiliza semaforos para sincronizar parmissões de escrita\n// No prompt do Linux: compilar usando: gcc -o sharedmem_transmitter sharedmem_transmitter.c\n// -lrt\n\n#include <stdio.h>\n#include <stdlib.h>\n#include <fcntl.h>\n#include <sys/stat.h>\n#include <sys/mman.h>\n\n#include <unistd.h>\n\nint main (int argc, char *argv[])\n{\n    int fd, value, *ptr;\n    int s_fd, s_value, *s_ptr; // usados para o semaforo
```

```

// Passos 1 a 3: abre/cria uma area de memoria compartilhada
fd = shm_open("/sharedmem", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
if(fd == -1) {
    perror ("shm_open");
    exit (1) ;
}

// Abre/cria uma área de memória compartilhada para utilizar como semáforo
s_fd = shm_open("/sharedmem_s", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
if(s_fd == -1) {
    perror("shm_s_open");
    exit (1);
}

// Passos 1 a 3: ajusta o tamanho da area compartilhada
if (ftruncate(fd, sizeof(value)) == -1) {
    perror ("ftruncate");
    exit (1) ;
}

// Faz o mesmo para a area de memoria do semáforo
if (ftruncate(s_fd, sizeof(s_value)) == -1) {
    perror ("ftruncate_s");
    exit (1);
}

// Passos 2 a 4: mapeia a area no espaco de enderecamento deste processo
ptr = mmap(NULL, sizeof(value), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
if(ptr == MAP_FAILED) {
    perror ("mmap");
    exit (1);
}

// Faz o mesmo para a área de memória do semáforo
s_ptr = mmap(NULL, sizeof(s_value), PROT_READ|PROT_WRITE, MAP_SHARED, s_fd, 0);
if(s_ptr == MAP_FAILED) {
    perror ("mmap_s");
    exit (1);
}

for (;;) {
    // Passo 5: escreve um valor aleatorio na area compartilhada
    if ((*s_ptr) == 1)
        sleep(1);
    else {
        value = random () % 1000 ;
        (*ptr) = value ;
        (*s_ptr) = 1;          // seta um flag para fazer o transmissor aguardar
antes de escrever
        printf ("Wrote value %i\n", value) ;
        sleep (1);
    }
}
}

```

Figura 4: código sharedmem_transmitter.c

A figura 5 abaixo representa o código “sharedmem_reciever.c”, onde um processo aloca, cria e mapeia uma memória compartilhada e recebe mensagens de forma similar ao processo B.

```
// UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
// DEPARTAMENTO ACADEMICO DE ENGENHARIA ELETRONICA
// VICTOR AUGUSTO DEL MONEGO - 2378345
// Arquivo sharedmem_receiver.c: cria e usa uma área de memória compartilhada. Apenas lê.
// Ademais também utiliza semaforos para liberar para a escrita.
// No prompt do Linux: compilar usando: gcc -o sharedmem_receiver sharedmem_receiver.c -lrt

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>

#include <unistd.h>

int main (int argc, char *argv[])
{
    int fd, value, *ptr;
    int s_fd, s_value, *s_ptr; // usados para o semaforo

    // Passos 1 a 3: abre/cria uma area de memoria compartilhada
    fd = shm_open("/sharedmem", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
    if(fd == -1) {
        perror ("shm_open");
        exit (1) ;
    }

    // Abre/cria uma área de memória compartilhada para utilizar como semáforo
    s_fd = shm_open("/sharedmem_s", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
    if(s_fd == -1) {
        perror("shm_s_open");
        exit (1);
    }

    // Passos 1 a 3: ajusta o tamanho da area compartilhada
    if (ftruncate(fd, sizeof(value)) == -1) {
        perror ("ftruncate");
        exit (1) ;
    }

    // Faz o mesmo para a area de memoria do semáforo
    if (ftruncate(s_fd, sizeof(s_value)) == -1) {
        perror ("ftruncate_s");
        exit (1);
    }

    // Passos 2 a 4: mapeia a area no espaco de enderecamento deste processo
    ptr = mmap(NULL, sizeof(value), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    if(ptr == MAP_FAILED) {
        perror ("mmap");
        exit (1);
    }

    // Faz o mesmo para a área de memória do semáforo
    s_ptr = mmap(NULL, sizeof(s_value), PROT_READ|PROT_WRITE, MAP_SHARED, s_fd, 0);
    if(s_ptr == MAP_FAILED) {
```

```

        perror ("mmap_s");
        exit (1);
    }
    for (;;) {
        // Passo 5: le e imprime o conteudo da area compartilhada.
        value = (*ptr);
        (*s_ptr) = 0; //libera para escrita, retirando o flag inserido pelo transmissor
        printf("Read value %i\n", value);
        sleep(1);
    }
}

```

Figura 5: código sharedmem_receiver.c

Por mais que seja mais versátil, existe um grande ponto fraco de implementar memórias compartilhadas para o compartilhamento de mensagens. Nos códigos acima, se for implementado apenas um espaço simples de memória compartilhada, não existe a segurança de que todas as informações que estão sendo transmitidas serão devidamente recebidas. Como visto anteriormente na comunicação simples, existe um “hard barrier” que evita que o processo A transmita mensagens em um momento em que o processo B não irá receber. Logo, para que a memória compartilhada, é necessário implementar um mecanismo de contingência. No caso desses códigos, é o semáforo.

Vamos observar as figuras 6 e 7 abaixo.

```

        // Abre/cria uma área de memória compartilhada para utilizar como semáforo
        s_fd = shm_open("/sharedmem_s", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
        if(s_fd == -1) {
            perror("shm_s_open");
            exit (1);
        }

        // Faz o mesmo para a area de memoria do semáforo
        if (ftruncate(s_fd, sizeof(s_value)) == -1) {
            perror ("ftruncate_s");
            exit (1);
        }

        // Faz o mesmo para a área de memória do semáforo
        s_ptr = mmap(NULL, sizeof(s_value), PROT_READ|PROT_WRITE, MAP_SHARED, s_fd, 0);
        if(s_ptr == MAP_FAILED) {
            perror ("mmap_s");
            exit (1);
        }
    }
}

```

Figura 6: declarações do semáforo

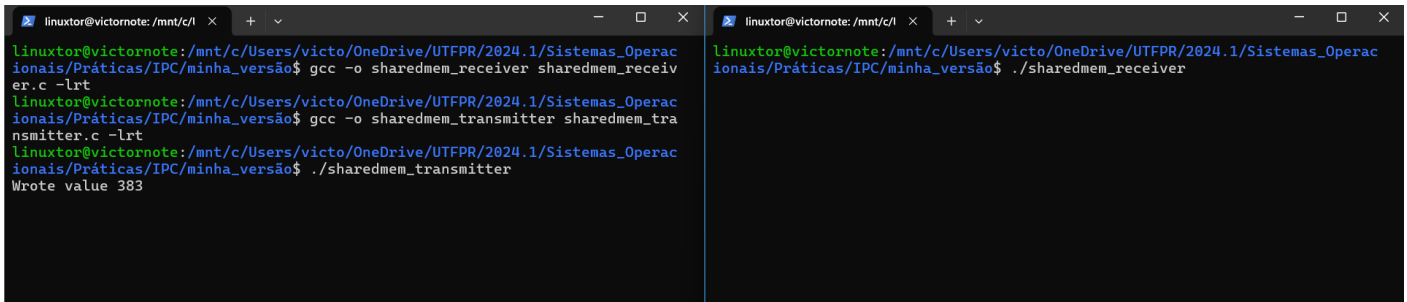
```

    for (;;) {
        // Passo 5: le e imprime o conteudo da area compartilhada.
        value = (*ptr);
        (*s_ptr) = 0; //libera para escrita, retirando o flag inserido
        pelo transmissor
        printf("Read value %i\n", value);
        sleep(1);
    }
}

```

Figura 7: loop de execução com semáforo incluso

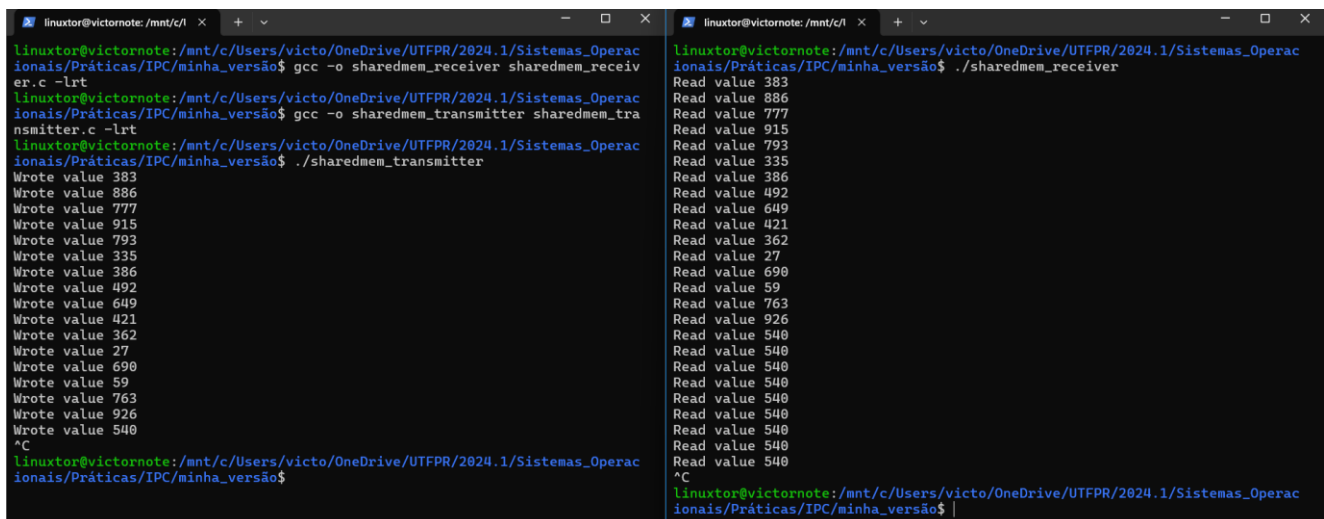
Os trechos de código nos demonstram que o semáforo se trata de um espaço de memória compartilhada adicional, com uma flag. Esse flag é utilizado com o novo “hard barrier” para garantir a sincronização de ambos os processos. O processo transmissor configura o flag para 1 e aguarda. Quando o processo receptor lê a mensagem, configura o flag para 0, e só então o transmissor envia outra mensagem. Essa lógica é exemplificada nas figuras 8 e 9.



```
linuxor@victornote: /mnt/c/l \
linuxor@victornote: /mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/IPC/minha_versão$ gcc -o sharedmem_receiver sharedmem_receiver.c -lrt
linuxor@victornote: /mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/IPC/minha_versão$ gcc -o sharedmem_transmitter sharedmem_transmitter.c -lrt
linuxor@victornote: /mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/IPC/minha_versão$ ./sharedmem_transmitter
Wrote value 383

linuxor@victornote: /mnt/c/l \
linuxor@victornote: /mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/IPC/minha_versão$ ./sharedmem_receiver
```

Figura 8: processo transmissor aguardando processo receptor



```
linuxor@victornote: /mnt/c/l \
linuxor@victornote: /mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/IPC/minha_versão$ gcc -o sharedmem_receiver sharedmem_receiver.c -lrt
linuxor@victornote: /mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/IPC/minha_versão$ gcc -o sharedmem_transmitter sharedmem_transmitter.c -lrt
linuxor@victornote: /mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/IPC/minha_versão$ ./sharedmem_transmitter
Wrote value 383
Wrote value 886
Wrote value 777
Wrote value 915
Wrote value 793
Wrote value 335
Wrote value 386
Wrote value 492
Wrote value 649
Wrote value 421
Wrote value 362
Wrote value 27
Wrote value 690
Wrote value 59
Wrote value 763
Wrote value 926
Wrote value 540
^C
linuxor@victornote: /mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/IPC/minha_versão$

linuxor@victornote: /mnt/c/l \
linuxor@victornote: /mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/IPC/minha_versão$ ./sharedmem_receiver
Read value 383
Read value 886
Read value 777
Read value 915
Read value 793
Read value 335
Read value 386
Read value 492
Read value 649
Read value 421
Read value 362
Read value 27
Read value 690
Read value 59
Read value 763
Read value 926
Read value 540
Read value 540
Read value 540
Read value 540
Read value 540
Read value 540
Read value 540
Read value 540
^C
linuxor@victornote: /mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/IPC/minha_versão$
```

Figura 9: transmissor e receptor sincronizados

Resumindo, o que podemos observar como diferença entre os dois tipos de comunicação é a maneira como transmitem dados. Na comunicação entre processos A e B, é utilizado um buffer simples de mensagens, e na comunicação entre transmissor e receptor, é utilizado um espaço de memória compartilhada entre ambos os processos, juntamente com um espaço adicional configurado como semáforo.

Referências:

- MAZIERO, C. A. **Construção de semáforos**. Disponível em: [https://wiki.inf.ufpr.br/maziero/doku.php?id=so:semaforos&s\[\]=ipc](https://wiki.inf.ufpr.br/maziero/doku.php?id=so:semaforos&s[]=ipc). Acesso em: abril de 2024
- MAZIERO, C. A. **Filas de mensagens**. Disponível em: [https://wiki.inf.ufpr.br/maziero/doku.php?id=so:filas_de_mensagens&s\[\]=mensagens](https://wiki.inf.ufpr.br/maziero/doku.php?id=so:filas_de_mensagens&s[]=mensagens). Acesso em: abril de 2024
- COPETTI, L. F. **ELF66-S12 - Sistemas Operacionais 2024-1**. Disponível em: <https://moodle.utfpr.edu.br/course/view.php?id=18965>. Acesso em: abril de 2024