



PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS
ENGENHARIA DE COMPUTAÇÃO
SISTEMAS OPERACIONAIS A

Carlos Henrique Vieira Marques – 18720367

Erick Matheus Lopes Pacheco – 18711630

Hiago Silva Fernandes – 18726455

João Henrique Pereira – 18712919

Leonardo Sanavio – 18054395

EXPERIMENTO #2

**COMUNICAÇÃO ENTRE PROCESSOS (IPC) E TROCA DE DADOS
ATRAVÉS DO SYSTEM V**

CAMPINAS
MARÇO 2020

SUMÁRIO

1.	INTRODUÇÃO	3
2.	AMBIENTE DOS TESTES	4
3.	PERGUNTAS	4
3.1.	REFERENTES AOS SISTEMAS UNIX	4
3.2.	REFERENTES AO PROGRAMA EXEMPLO	11
4.	O PROGRAMA EXEMPLO	13
4.1.	OS ERROS	13
5.	RESULTADOS	16
5.1.	REFERENTES AO PROGRAMA EXEMPLO	17
5.2.	REFERENTES AO PROGRAMA MODIFICADO	19
6.	ANÁLISE DOS RESULTADOS	20
7.	CONCLUSÃO	21

1. INTRODUÇÃO

O presente experimento visa o estudo e aprofundamento nas chamadas de IPC (comunicação entre processos), existentes no Sistema UNIX, através do método System V. Além disso, foram realizados estudos sobre fila de mensagens a respeito de suas estruturas e lógica. Dessa forma, foi possível fazer a análise sobre a troca de conteúdos entre processos e também a medição de tempo em que um processo gasta para transferir dados de um para outro.

Para realização do projeto, na primeira tarefa foi utilizado um programa exemplo fornecido pelo professor que trabalha com recursos compartilhados entre processos e procura calcular quanto tempo demora para se transferir um dado (mensagem) de um processo para outro através de uma fila de mensagens. Para isso, ocorre a criação de dois processos filhos onde um é o receptor (segundo) da mensagem e o outro é emissor da mensagem (primeiro). Por fim, o emissor recebe uma mensagem da fila e em seguida armazena o tempo em que é iniciado, repete o processo por um número de vezes determinado e depois envia a mensagem e o tempo pela fila. O receptor, realiza o cálculo da diferença de tempo que o processo emissor gasta para realizar todo o processo de troca de dados, guarda este tempo na fila e aguarda até que a finalização da troca de dados.

Para a segunda tarefa ocorreu diversas alterações, o tamanho da mensagem sofria variação determinada entre 512b a 5Kb, também foi criada uma segunda fila de mensagens no processo receptor (segundo filho), essa fila é usada por um terceiro filho que é criado visando receber e exibir os cálculos feitos pelo irmão receptor (segundo filho). Outra modificação, são os métodos de remover as filas de mensagens, nesse caso quando o segundo filho manipula os dados da primeira fila, ela é deletada, e utiliza a segunda fila para transferir os dados para o terceiro dado, em seguida o terceiro filho depois de exibir todos os dados deleta a segunda fila.

2. AMBIENTE DOS TESTES

Antes de partirmos para os resultados dos testes, é válido que os façamos sob um ambiente de trabalho com a maior equivalência de hardware possível e livre de interferências (de softwares externos, por exemplo), para que os testes feitos nos forneçam um resultado coerente.

Foi utilizada apenas 1 máquina:

- Notebook i7 4790 **4.0GHz** & 16GB de RAM DDR3 1600MHz;

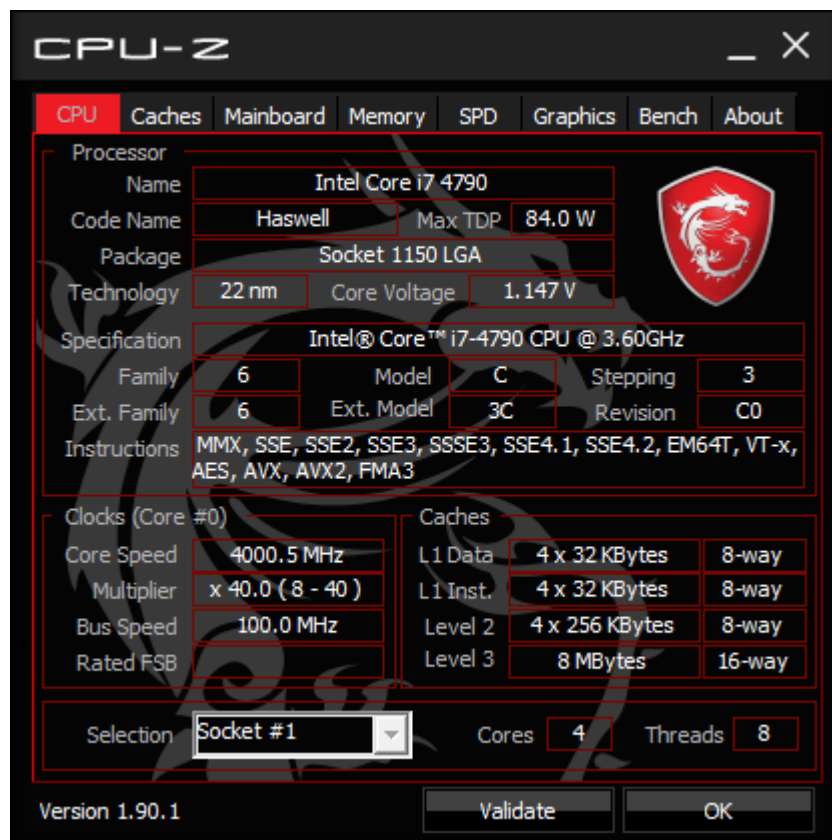


Figura 1 - Detalhes da CPU

3. PERGUNTAS

3.1. REFERENTES AOS SISTEMAS UNIX

Pergunta 1: Esclarecer o que são: Berkeley Unix, System V, POSIX, AT&T, socket, fila de mensagens, memória compartilhada e pipes.

Resposta: Berkeley Software Distribution (BSD) é um sistema operacional Unix com desenvolvimento derivado e distribuído pelo Computer Systems Research Group (CSRG) da Universidade da Califórnia em Berkeley, de 1977 a 1995. Hoje o termo "BSD" é frequentemente usado de forma não específica para se referir a qualquer descendente que juntos, formam uma ramificação da família Unix-like de sistemas operacionais. Sistemas operacionais derivados do código do BSD original ainda são ativamente desenvolvidos e largamente utilizados.

Historicamente, o BSD ou Berkeley Unix foi considerado uma ramificação do Unix da AT&T, porque compartilhou a base de dados inicial e o projeto original. Nos anos de 1980, o BSD foi largamente adotado por fornecedores de sistemas de estação de trabalho na forma de variantes proprietárias de Unix como DEC, ULTRIX e o SunOS da Sun Microsystems. Isto pode ser atribuído à facilidade com que ele pode ser licenciado, além da familiaridade que os fundadores de muitas empresas de tecnologia da época tinham com ele.

-

O Unix System V, normalmente abreviado como SysV, é uma das primeiras versões comerciais do sistema operacional Unix. Os mecanismos de comunicação inter-processos do System V estão disponíveis em sistemas operacionais tipo Unix não derivados do System V. Em particular, no Linux (uma reimplementação do Unix), bem como o derivado do BSD, o FreeBSD.

Os mecanismos de comunicação interprocessos do System V estão disponíveis em sistemas operacionais tipo Unix não derivados do System V. Em particular, no Linux (uma reimplementação do Unix), bem como o derivado do BSD, o FreeBSD. O POSIX 2008 especifica uma substituição para estas interfaces.

-

POSIX é uma família de normas definidas pelo IEEE para a manutenção de compatibilidade entre sistemas operacionais (sistemas operativos em PT-PT), e designada formalmente por IEEE 1003. POSIX define a interface de programação de aplicações (API), juntamente com shells de linha e comando e interfaces utilitárias, para compatibilidade de software com variantes de Unix e outros sistemas operacionais.

-

A AT&T é um dos maiores e mais tradicionais conglomerados de mídia e telecomunicações do mundo. Sua origem data da época da própria invenção do telefone. Foi fundada em 1885 por Alexander Graham Bell e Gardiner Greene Hubbard como American Telephone and Telegraph.

Até 1982, ela detinha o monopólio de telefonia nos Estados Unidos. Nesta época, as leis anti-trust dividiram a companhia. Em 2005, a AT&T Corporation foi adquirida pela Bell Operating Company e nova companhia combinada passou a se chamar apenas AT&T.

-

Um soquete de rede (em inglês: network socket) é um ponto final de um fluxo de comunicação entre processos através de uma rede de computadores. Hoje em dia, a maioria da comunicação entre computadores é baseada no Protocolo de Internet, portanto a maioria dos soquetes de rede são soquetes de Internet.

Uma API de soquetes (API sockets) é uma interface de programação de aplicativos (API), normalmente fornecida pelo sistema operacional, que permite que os programas de aplicação controlem e usem soquetes de rede. APIs de soquete de Internet geralmente são baseados no padrão Berkeley sockets.

Um endereço de soquete (socket address) é a combinação de um endereço de IP e um número da porta, muito parecido com o final de uma conexão telefônica que é a combinação de um número de telefone e uma determinada extensão. Com base nesse endereço, soquetes de internet entregam pacotes de dados de entrada para o processo ou thread (computação) de aplicação apropriado.

-

Uma fila de mensagens é uma forma de comunicação assíncrona entre serviços usada em arquiteturas sem servidor e de microsserviços. As mensagens são armazenadas na fila até serem processadas e excluídas. Cada mensagem é processada uma única vez, por um único consumidor. As filas de mensagens podem ser usadas para dissociar processamento pesado, para armazenar trabalho em buffers ou lotes e para processar uniformemente picos de cargas de trabalho.

As filas de mensagens disponibilizam recursos de comunicação e coordenação para esses aplicativos distribuídos. As filas de mensagens podem

simplificar bastante a codificação de aplicativos dissociados e aumentar a performance, a confiabilidade e a escalabilidade.

As filas de mensagens permitem que diferentes partes de um sistema se comuniquem e processem operações de forma assíncrona. Uma fila de mensagens disponibiliza um buffer leve que as armazena temporariamente, além de endpoints que permitem que os componentes de software estabeleçam conexão com a fila para o envio e o recebimento de mensagens.

-

No hardware, a memória compartilhada se refere tipicamente a grandes blocos de RAM que podem ser acessados por diferentes unidades centrais de processamento (CPU) em um sistema de multiprocessamento. Um sistema de memória compartilhada é simples de ser programado já que todos os processadores compartilham a mesma visão dos dados, e a comunicação entre processadores pode ser tão rápida quanto o acesso à memória na mesma posição.

O problema com sistema de memória compartilhada é que várias CPU necessitam acesso rápido à memória e por isso utilizam sistemas de cache na própria CPU, o que possui duas complicações. A primeira é que conexão da CPU para a memória se torna um gargalo no sistema. Computadores com memória compartilhada não possuem boa escalabilidade. A segunda diz respeito à integridade do cache, a condição de que ele esteja atualizado com as informações corretas, mais atuais. Mudanças no cache de um processador devem ser replicadas nos outros.

Em software, a memória compartilhada é tanto um método de comunicação entre processos quanto um método de conservação do espaço de memória. Para o primeiro, um processo pode criar uma área na RAM que outros processos sendo executados simultaneamente pode acessar. Para o segundo, ao invés de replicar cópias de blocos de memória entre diferentes processos, há somente uma instância.

Já que ambos os processos podem acessar a memória compartilhada como outra memória qualquer, este é um método muito rápido de comunicação entre processos (em contraste de outros métodos como pipe nomeado, socket Unix ou

sistemas como o CORBA). Por outro lado, não é tão robusto dado que, por exemplo, os processos devem estar sendo executados numa mesma máquina, enquanto há métodos de comunicação entre processos que podem interagir numa rede de computadores. Deve-se tomar cuidado também com a questão da integridade do cache citada acima, caso a arquitetura já não o faça.

-

Na ciência da computação, um pipe nomeado (também chamado de named pipe ou FIFO) é uma extensão do conceito de encadeamento do sistema Unix e dos seus métodos de comunicação entre processos. O mesmo conceito é encontrado no Microsoft Windows, apesar da sua semântica ser razoavelmente diferente. Uma canalização tradicional (anônima) dura somente o tempo de execução do processo e, por outro lado, o pipe nomeado persiste além da vida do processo e precisa ser "desligado" ou apagado quando não é mais usado. Os processos geralmente se conectam a um pipe nomeado (normalmente um arquivo) quando necessitam realizar alguma comunicação com outro processo (IPC). O pipe possui diversas aplicações tanto no Windows quanto no Linux e cada um possui comandos diferentes e únicos.

Pergunta 2: As chamadas ipcs e ipcrm apresentam informações sobre quais tipos de recursos?

Resposta: O comando ipcrm no Linux é usado para remover alguns recursos do IPC (Inter-Process Communication). Elimina os objetos IPC e sua estrutura de dados associada a forma do sistema. É preciso ser um criador ou superusuário ou o proprietário do objeto para remover esses objetos. Existem três tipos de objetos System V IPC, como semáforos, memória compartilhada e filas de mensagens. Sua sintaxe é a seguinte: ipcrm [sem|shm|msg] <id>

O comando ipcs -<recurso> fornece informações atualizadas de cada um dos recursos IPC implementados no sistema. O tipo de recurso pode ser especificado da seguinte forma:

- ipcs -m informações relativas aos segmentos de memória compartilhada
- ipcs -s informações sobre os semáforos

- `ipcs -q` informações sobre as filas de mensagens
- `ipcs -a` todos os recursos (opção *par défaut* se nenhum parâmetro for especificado).

Pergunta 3: Qual a diferença entre o handle devolvido pela chamada `msgget` e a chave única?

Resposta: A função `msgget` é utilizada para criar uma nova fila de mensagens, ou para obter o identificador da fila `msqid` de uma fila de mensagens existente no sistema. Esta função recebe dois parâmetros: `key` é a chave indicando uma constante numérica representando a fila de mensagens; `msgflg` é um conjunto de flags especificando as permissões de acesso sobre a fila.

O parâmetro `key` pode conter os seguintes valores:

- `IPC_PRIVATE` (`=0`): a fila de mensagens não tem chave de acesso e somente o proprietário ou o criador da fila poderão ter acesso à fila;
- um valor desejado para a chave de acesso da fila de mensagens.

Pergunta 4: Há tamanhos máximos para uma mensagem? Quais?

Resposta: A estrutura da mensagem é limitada de duas maneiras: primeiramente, ela deve ser menor que o limite estabelecido pelo sistema; depois, ela deve respeitar o tipo de dado estabelecido pela função que receberá a mensagem. Dessa forma, a mensagem tem um tamanho definido de 1 byte, porém no experimento modificado a mensagem passa a variar de acordo com o tamanho da struct que pode chegar a um tamanho determinado de no máximo 512Kb.

Pergunta 5: Há tamanhos máximos para uma fila de mensagens? Quais?

Resposta: Filas de mensagens provêm um protocolo de comunicação assíncrona, de forma que o remetente e o destinatário da mensagem não precisam interagir ao mesmo tempo. As mensagens são enfileiradas e armazenadas até que o destinatário as processe. A maioria das filas de mensagens definem limites ao

tamanho dos dados que podem ser transmitidos numa única mensagem. Aquelas que não possuem tal limite são chamadas caixas de mensagens. No projeto é solicitado uma mensagem de tamanho entre 512b e 512Kb, realizando os testes com tamanho de mensagem superior, notamos que a fila possui um espaço de 8109 bytes que aproximadamente é 8Kb

Pergunta 6: Para que serve um typedef?

Resposta: O comando typedef é usado para criar “sinônimo” ou um “alias” para tipos de dados existentes. Então na prática podemos dizer que estamos renomeando um tipo de dados.

A renomeação de tipos facilita a organização e o entendimento de códigos.

Pergunta 7: Onde deve ser usado o que é definido através de um typedef?

Resposta: É importante entender que o comando typedef **não cria** um novo tipo. Ele apenas permite que um tipo existente seja denominado de uma forma diferente, de acordo com a especificação desejada pelo programador. É muito frequente o uso de typedef para redefinir tipos como estruturas a fim de tornar os nomes mais curtos, desta forma podemos representar uma estrutura usando apenas seu sinônimo.

Pergunta 8: Na chamada msgsnd há o uso de cast, porém agora utiliza-se um “&” antes de message_buffer. Explicar para que serve o “&” e o que ocorreria se este fosse removido.

Resposta: O uso do & funciona para buscar o endereço do message_buffer, pois o cast é utilizado para transformar o message_buffer no tipo da struct de mensagem. Dessa forma, o tamanho com o que a variável será interpretada pelo compilador muda e assim sem o & ocorre erro na compilação pois o compilador não vai possuir o endereço para tratar essa variável de tipo trocado.

3.2. REFERENTES AO PROGRAMA EXEMPLO

Pergunta 1: O que é um protótipo? Por qual motivo é usado?

Resposta: Um protótipo de função ou declaração de função, nas linguagens de programação C e C++, é uma declaração de uma função que omite o corpo mas especifica o seu nome, aridade, tipos de argumentos e tipo de retorno. Um protótipo de função pode ser entendido como a especificação da sua interface.

Como um exemplo, pode-se considerar o seguinte protótipo de função:

```
int fac(int n);
```

Este protótipo especifica que neste programa, existe uma função de nome "fac" que utiliza um único parâmetro "n" que está declarado como tipo inteiro "int". A definição da função precisa ser disponibilizada para que a função possa ser utilizada.

Pergunta 2: O que significa cada um dos dígitos 0666?

Resposta: 0666 é a permissão de acesso usual no linux no formato octal rwx e com a sequência (user-group-user) IPC_CREAT possui o valor de 512 em decimal, conforme definido no arquivo de cabeçalho sys / pc.h.

Quando um novo arquivo ou diretório é criado, o processo de criação especifica a permissão que o novo arquivo ou diretório deverá ter.

As permissões padrões para novos arquivos ou diretórios criados são aplicadas nas configurações da conta de cada usuário individualmente.

Pergunta 3: Para que serve o arquivo stderr?

Resposta: Os fluxos padrão são canais de entrada/saída entre um programa de computador e o seu ambiente (tipicamente um terminal de texto) que são pré-conectados no início da execução. Estas conexões padrão são disponibilizadas nos sistemas operacionais do tipo Unix, ambientes de execução das linguagens C e C++ e seus sucessores. O *stderr* (*STandarD ERRor*, ou *Erro Padrão*), é um outro tipo de saída padrão, é utilizada pelos programas para envio de mensagens de erro ou de

diagnóstico. Este fluxo é independente da saída padrão e pode ser redirecionado separadamente. O destino usual é o terminal de texto onde o programa foi executado, para que haja uma grande chance da saída ser observada mesmo que a "saída padrão" tenha sido redirecionada, dessa forma pode ser apresentado no monitor.

Pergunta 4: Caso seja executada a chamada `fprintf` com o handler `stderr`, onde aparecerá o seu resultado?

Resposta: O resultado é apresentado no monitor se o `fprintf` for executado sem erros, caso contrário, será apresentado no lugar do resultado o erro, e esse resultado é apresentado graças ao `stderr`.

Pergunta 5: Onde `stderr` foi declarado?

Resposta: O `stdin`, `stdout` e `stderr` não são declarados no código apesar de serem arquivos eles são parecidos com "streams de E / S" em vez de arquivos. Por isso, essas entidades não vivem no sistema de arquivos. Dessa forma, O sistema de arquivos Linux `/proc`, não são arquivos reais, apenas gateways rigidamente controlados para informações do kernel e como a maioria dos programas precisa ler input, gravar saída e registrar erros, assim `stdin`, `stdout` e `stderr` são predefinidos na biblioteca `<stdio.h>`.

Pergunta 6: Explicar o que são e para que servem `stdin` e `stdout`.

Resposta: Assim como `stderr` tanto o `stdin` e o `stdout` são fluxos padrões. Porém cada um possui uma função diferente.

O `stdin` conhecido como Entrada padrão indica que o dado (frequentemente texto) está indo para um programa. O programa requisita a transferência de dados através de uma operação de "leitura". Porém nem todos programas precisam de entrada. Se a entrada não for "redirecionada", a origem da entrada de um programa é o terminal de texto de onde o programa foi executado.

O `stdout` conhecido como Saída Padrão, é o inverso do `Stdin`, ele é um fluxo onde o programa escreve dados de saída. O programa requisita a transferência de dados através de uma operação de "escrita". E assim como no `stdin` nem todos os programas precisam de saída. e se a saída não for "redirecionada", o destino da saída padrão é o terminal de texto de onde o programa foi executado.

Pergunta 7: O que ocorre com a fila de mensagens, se ela não é removida e os processos terminam?

Resposta: A Fila continua presente no processo e também ocupando espaço de memória. Dessa forma, é recomendado o uso da função `ipcrm`.

Pergunta 8: Qual será o conteúdo de `data_ptr`?

Resposta: `data_ptr` é um ponteiro de struct, que aponta para o buffer da mensagem. Dessa forma, seu conteúdo é o endereço desse buffer.

4. O PROGRAMA EXEMPLO

O programa exemplo, já explicado superficialmente na introdução, continha uma série de erros propositais que eram parte do desafio, dentre eles, erros de lógica e sintaxe. Antes de seguirmos para a identificação e solução de tais erros, trataremos o conceito de algumas funções implementadas no código original e o seu propósito.

4.1. OS ERROS

A seguir, os erros contidos no programa-exemplo junto a nossa solução.

Trecho #1:

```
for (count = 0; count < NO_OF_CHILDREN; count -) {
    if (0 != rtn) {
        rtn == fork;
    } else {
        exit(NULL);
    }
}
```

Erro sintático/lógico no escopo do **for**, na verificação do **if** e na atribuição de **rtn** e chamada do **fork**, além de um erro crítico de lógica no **else**, fazendo com que todo processo filho que cairá nesta condicional sofra um `exit` sem executar as tarefas desejadas por nós.

Solução:

```

for (count = 0; count < NO_OF_CHILDREN; count++) {
    if (rtn != 0) {
        rtn = fork();
    } else {
        break;
    }
}

```

Trecho #2:

```

/*
 * Sou o primeiro filho me preparando para receber uma mensagem
 */
printf("Receptor iniciado ...\n");
Receiver(queue_id);
exit(0);

```

Erro sintático, basta fechar a aspas do printf.

Trecho #3:

```

/*
 * Sou o pai aguardando meus filhos terminarem
 */
printf("Pai aguardando ...\n");
wait();
wait();

```

Ausência de parâmetro, soluciona-se utilizando "NULL", visto que não queremos guardar o retorno da função wait() que refere-se ao status do processo filho.

Trecho #4:

```

/*
 * Removendo a fila de mensagens
 */
if (msgctl(queue_id, IPC_RMID, NULL) == 0) {
    fprintf(stderr, "Impossível remover a fila!\n");
    exit(1);
}

```

Teste da condicional incorreto, se nosso intuito é entrar na condicional apenas em caso de erro na remoção da fila (especificado pelo comando IPC_RMID) utilizando a função msgctl, o teste deve ser comparando o valor de retorno com “-1”, que é o retorno da função em caso de erro.

Trecho #5:

```

/*
 * Calcula a diferenca
 */
delta -= receive_time.tv_sec - data_ptr->send_time.tv_sec;
delta = (receive_time.tv_usec - data_ptr-
>send_time.tv_usec) / (float)MICRO_PER_SECOND;
total += -delta;

/*
 * Salva o tempo maximo
 */
if (delta < max) {
    max = delta;
}

```

Erros praticamente em todas as linhas de código. Na primeira linha, se trata de um erro de lógica, queremos *afirmar que delta é igual* ao valor da diferença entre o tempo de recebimento e o tempo de envio, todo o delta a fim de obtermos o valor total, e não subtrair. Na segunda linha, se trata de um erro de lógica, queremos *somar os microssegundos* com os segundos já salvos na linha anterior em **delta**, e não igualar (sobrescrevendo, assim, os segundos medidos). Já na condicional **if**, a lógica do comparador está incorreta, visto que se queremos valor mais alto, devemos comparar *delta (valor atual) > max (valor máximo anterior)*.

Solução:

```

/*
 * Calcula a diferenca
 */
delta = receive_time.tv_sec - data_ptr->send_time.tv_sec;
delta += (receive_time.tv_usec - data_ptr-
>send_time.tv_usec) / (float)MICRO_PER_SECOND;
total += delta;

/*
 * Salva o tempo maximo
 */
if (delta > max) {
    max = delta;
}

```

Trecho #6:

```

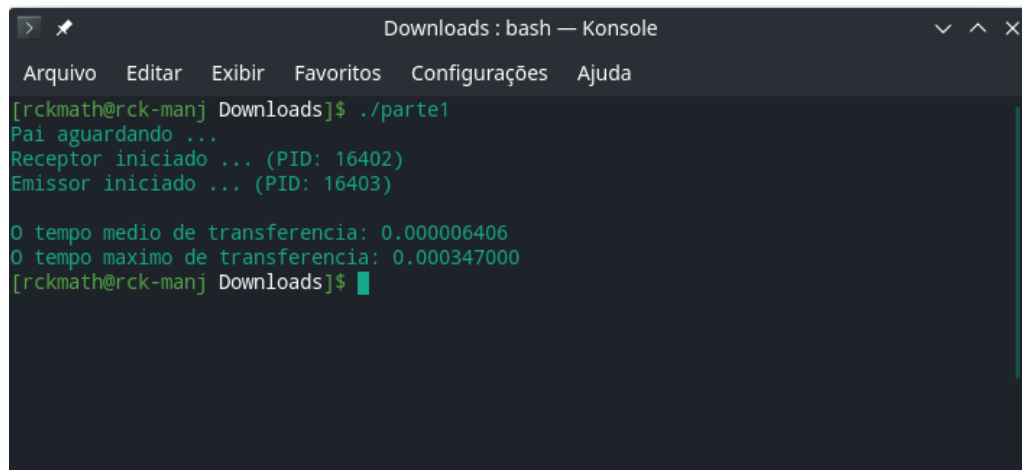
/*
 * Exibe os resultados
 */
printf("O tempo medio de transferencia: %.1f\n", total / NO_OF_ITERATI
ONS);
printf(stdout, "O tempo maximo de transferencia: %.1f\n", max);

```

Aumenta-se o número de casas decimais dos resultados (de 1 para 9) e retira-se o “stdout” do *segundo printf*, visto que este é utilizado apenas no **fprintf**.

5. RESULTADOS

5.1. REFERENTES AO PROGRAMA EXEMPLO



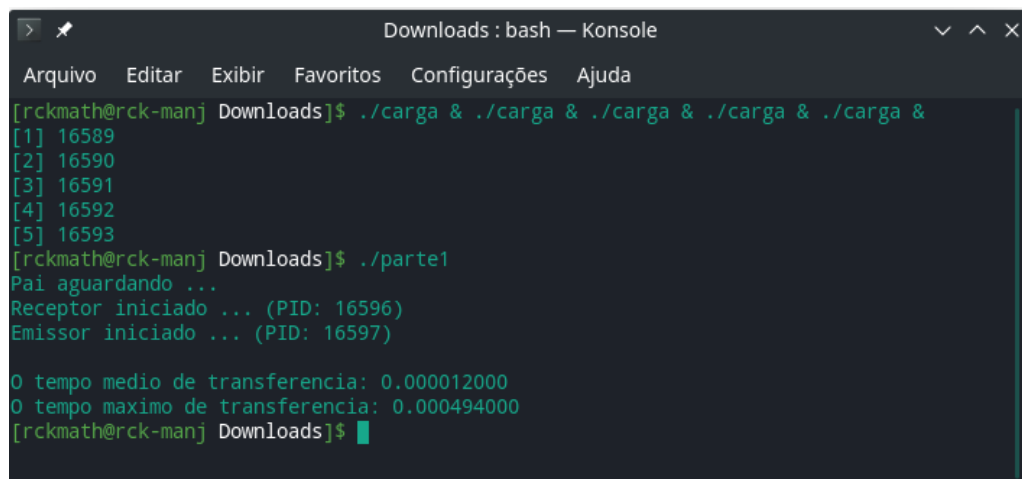
```

Downloads : bash — Konsole
Arquivo  Editar  Exibir  Favoritos  Configurações  Ajuda
[rckmath@rck-manj Downloads]$ ./parte1
Pai aguardando ...
Receptor iniciado ... (PID: 16402)
Emissor iniciado ... (PID: 16403)

0 tempo medio de transferencia: 0.000006406
0 tempo maximo de transferencia: 0.000347000
[rckmath@rck-manj Downloads]$

```

Figura 2 - 1º teste (carga normal do sistema)



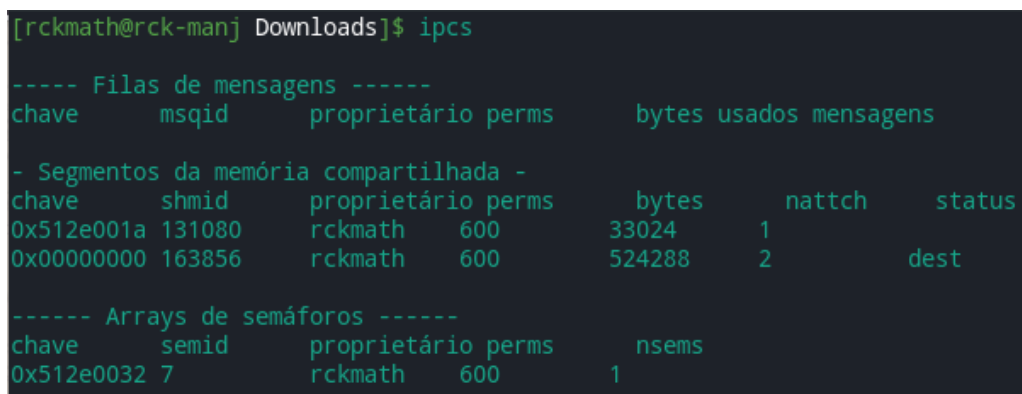
```

Downloads : bash — Konsole
Arquivo  Editar  Exibir  Favoritos  Configurações  Ajuda
[rckmath@rck-manj Downloads]$ ./carga & ./carga & ./carga & ./carga & ./carga &
[1] 16589
[2] 16590
[3] 16591
[4] 16592
[5] 16593
[rckmath@rck-manj Downloads]$ ./parte1
Pai aguardando ...
Receptor iniciado ... (PID: 16596)
Emissor iniciado ... (PID: 16597)

0 tempo medio de transferencia: 0.000012000
0 tempo maximo de transferencia: 0.000494000
[rckmath@rck-manj Downloads]$

```

Figura 3 - 2º teste (5 cargas no sistema)



```

[rckmath@rck-manj Downloads]$ ipcs

----- Filas de mensagens -----
chave      msqid      proprietário perms      bytes usados mensagens
- Segmentos da memória compartilhada -
chave      shmid      proprietário perms      bytes      nattch      status
0x512e001a 131080     rckmath    600      33024      1
0x00000000 163856     rckmath    600      524288     2          dest

----- Arrays de semáforos -----
chave      semid      proprietário perms      nsems
0x512e0032 7          rckmath    600      1

```

Figura 4 – Resultado do comando IPSCS após a execução da parte 1

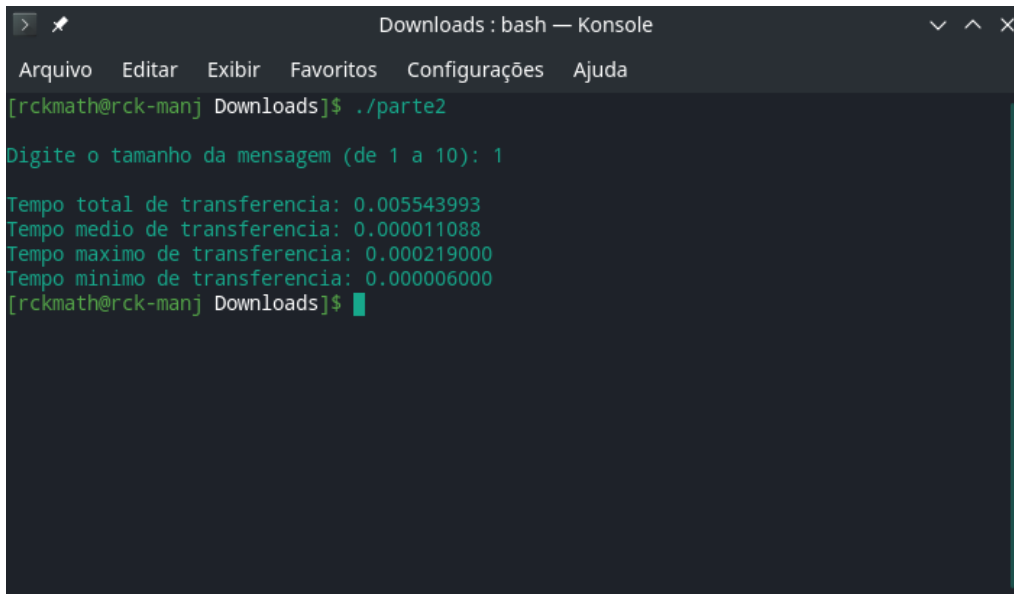
Manjaro KDE Plasma 19.0.2 - CPU i7 4790 4.0GHz			
Rodada	Médio (seg)	Máx (seg)	Carga
1	0,000006406	0,000347000	0
2	0,000012000	0,000494000	5
3	0,000589605	0,007036000	10
4	0,000498233	0,002924000	15
5	0,000017702	0,006678000	20
6	0,000013080	0,004218000	25
7	0,001975687	0,010044000	30
8	0,000022660	0,008234000	35
9	0,003172487	0,013335000	40
10	0,001809394	0,013005000	45

Figura 5 – Tabela de resultados de 10 execuções do programa exemplo



Figura 6 – Gráfico de resultados de 10 execuções do programa exemplo

5.2. REFERENTES AO PROGRAMA MODIFICADO



```
Downloads : bash — Konsole
Arquivo  Editar  Exibir  Favoritos  Configurações  Ajuda
[rckmath@rck-manj Downloads]$ ./parte2
Digite o tamanho da mensagem (de 1 a 10): 1
Tempo total de transferencia: 0.005543993
Tempo medio de transferencia: 0.000011088
Tempo maximo de transferencia: 0.000219000
Tempo minimo de transferencia: 0.000006000
[rckmath@rck-manj Downloads]$
```

Figura 7 – 1º teste da segunda tarefa (carga normal do sistema)



```
Downloads : bash — Konsole
Arquivo  Editar  Exibir  Favoritos  Configurações  Ajuda
[rckmath@rck-manj Downloads]$ ./carga & ./carga & ./carga & ./carga & ./carga &
[41] 17813
[42] 17814
[43] 17815
[44] 17816
[45] 17817
[rckmath@rck-manj Downloads]$ ./parte2
Digite o tamanho da mensagem (de 1 a 10): 10
Tempo total de transferencia: 0.006139941
Tempo medio de transferencia: 0.000012280
Tempo maximo de transferencia: 0.001828000
Tempo minimo de transferencia: 0.000004000
[rckmath@rck-manj Downloads]$
```

Figura 8 – 2º teste da segunda tarefa (carga normal do sistema)

Manjaro KDE Plasma 19.0.2 - CPU i7 4790 4.0GHz						
Rodada	Min (seg)	Máx (seg)	Médio (seg)	Total (seg)	Carga	Tam. msg. (bytes)
1	0,000006000	0,000219000	0,000011088	0,005543993	0	512
2	0,000003000	0,000195000	0,000005040	0,002520007	5	1024
3	0,000003000	0,001566000	0,000055952	0,027975975	10	1536
4	0,000003000	0,006629000	0,000130771	0,065385416	15	2048
5	0,000004000	0,009825000	0,000245228	0,122614138	20	2560
6	0,000004000	0,000050000	0,000004626	0,002313007	25	3072
7	0,000004000	0,014143000	0,000176564	0,088282198	30	3584
8	0,000004000	0,009992000	0,000156456	0,078228056	35	4096
9	0,000003000	0,008866000	0,000114908	0,057453837	40	4608
10	0,000004000	0,001828000	0,000012280	0,006139941	45	5120

Figura 9 – Tabela de resultados das 10 execuções da segunda tarefa

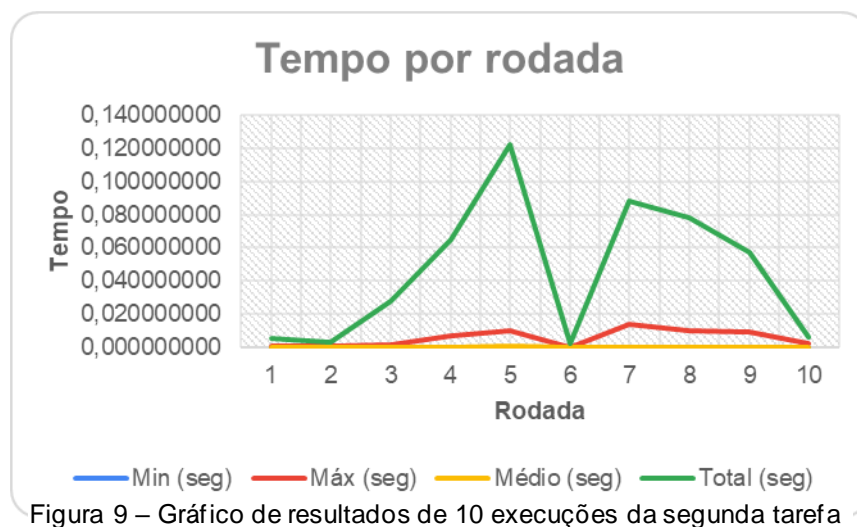


Figura 9 – Gráfico de resultados de 10 execuções da segunda tarefa

6. ANÁLISE DOS RESULTADOS

Os gráficos obtidos a partir dos dados do experimento, tanto do programa exemplo quanto do modificado, mostraram-se com grandes variações de difícil análise, apesar de alguns padrões observáveis. Tal instabilidade ocorre devido à inconsistência do método de medição de tempo do programa, que por si próprio influencia o valor a se obter, e é influenciado por atividades externas, como o escalonamento e demais chamadas do SO, além de ser fruto de um processo digital.

Além disso, o programa exemplo não sofreu um aumento proporcional no tempo medido conforme o aumento de cargas. Isso se deve ao fato de que o aumento de concorrência pela CPU, também aprimora o seu

desempenho. Tal técnica, se baseia em uma característica da CPU nomeada de *frequency scaling*, na qual se aumenta e se diminui a frequência de operação do processador com o intuito de economizar energia. Também foi observado a conjuntura da fila de mensagens após a execução do programa, por meio do comando `ipcs`. Foi possível constatar que a fila havia sido removida pelo próprio programa, já que não constava nenhuma fila de mensagens no sistema.

Já no programa modificado, se constatou que o aumento do tamanho da mensagem não repercute um aumento proporcional no tempo medido, o que possivelmente significa que a velocidade de transmissão não se afetou também. Entretanto, apresenta resultado curioso a respeito do tempo total, com uma variação um tanto quanto intrigante. Vale lembrar a fragilidade da ferramenta de medida, o que pode explicar tal fenômeno.

7. CONCLUSÃO

Apesar de ter a análise do experimento dificultada devido às inconsistências dos resultados, foi extremamente importante observar aspectos a respeito da comunicação interprocessos, assim como da fila de mensagens. Desde o processo de criação da fila com atribuição de uma key e a concessão de permissões específicas, a maneira de operação das funções `msgsnd()` e `msgrcv()`. Algo observado durante o experimento, foi o parâmetro *int msgflag* das funções `msgsnd()` e `msgrcv()` setado como 0 em ambas as tarefas. Essa flag controla o modo de execução da recepção da mensagem e tem papel fundamental no funcionamento do programa. Se estivesse, por exemplo, definida como `IP_NOWAIT` caso o receptor tentasse ler a fila vazia, ou o emissor tentasse escrever na fila cheia, as funções retornariam um erro, o que caracteriza a troca de mensagens do atual programa como bloqueante.