**PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS – PUC-CAMPINAS**

***Experimento 2***

***Sistemas Operacionais A***

|  |  |
| --- | --- |
| **ALUNO** | **RA** |
|
| Beatriz Morelatto Lorente | 18071597 |
| Cesar Marrote Manzano | 18051755 |
| Fabricio Silva Cardoso | 18023481 |
| Pedro Ignácio Trevisan | 18016568 |

**-= Sumário =-**

**1. Introdução.......................................................................................................................3**

**2. Apresentação dos erros do programa exemplo e suas soluções..............................4**

**3. Resultados da execução do programa exemplo..........................................................6**

**4. Resultados da execução do programa modificado....................................................28**

**5. Respostas das perguntas............................................................................................29**

**6. Análise dos Resultados................................................................................................32**

**7. Conclusão.....................................................................................................................39**

**-= Introdução =-**

O experimento realizado permitiu o entendimento da comunicação entre processos ou IPC (Inter-Process Communication). O experimento foi dividido em duas tarefas, com o objetivo de mostrar os tempos de transferência entre mensagens (enviadas através do mecanismo de fila de mensagens).

Na primeira tarefa, foi executado um programa simples, no qual um processo filho (Sender) mandava uma mensagem para outro (Receiver). O Receiver calculava o tempo médio e máximo de transferência e imprimia as informações.

Na segunda tarefa, havia três processos filhos, sendo que um era responsável por mandar as informações (Sender) para um outro processo (Receiver1) e este por um breve instante mandava uma mensagem para outro filho (Receiver2). O Receiver1 calculava o tempo total, médio, máximo e mínimo de transferência e envia essas informações para o Receiver2 imprimir. O usuário também poderia escolher o tamanho da mensagem que seria enviada, sendo que o usuário escolhia um número de 1 a 10, e este era multiplicado por 512, (representando 512kB).

**-= Apresentação dos erros do programa exemplo e suas soluções =-**

Ao compilar o programa pela primeira vez, foi mostrado no prompt alguns erros de sintaxe e lógica de programação. Os problemas estão listados abaixo, seguidos de suas soluções (as correções estão destacadas em negrito e itálico).

**Problema nas bibliotecas**

#include <sys/time.h>

#include <stdio.h>

#include <unistd.h>

#include <types.h>

#include <wait.h>

#include <ipc.h>

#include <msg.h>

**Programa corrigido:**

Algumas bibliotecas usam o recurso do SO, assim precisando do “sys/...” na frente.

#include <sys/time.h>

#include <stdio.h

#include <unistd.h>

***#include <sys/types.h>***

#include <wait.h>

***#include <sys/ipc.h>***

***#include <sys/msg.h>***

**Problema 1,2 e 3**

for( count = 0; count< NO\_OF\_CHILDREN; count- ) {

if( 0 !=rtn) {

rtn == fork;

} else {

exit(NULL);

}

}

**Programa Corrigido:**

Nessa parte do código percebe-se quatro erros. A não declaração e o valor do rtn, a incrementação da incorreta do valor da variável “count”, a falta do parênteses no fork() e a condição do else.

**rtn = 1;**

for( count = 0; count < NO\_OF\_CHILDREN; ***count++*** ) {

if( rtn != 0) {

***rtn == fork();***

} else {

***break;***

}

}

**Problema 4**

if( msgctl(queue\_id,IPC\_RMID,NULL) == 0 ) {

fprintf(stderr,"Impossivel remover a fila!\n");

exit(1);

}

**Programa corrigido:**

O valor usado na comparação para verificar se deu certo para remover a fila de mensagens foi alterado.

if( ***msgctl(queue\_id,IPC\_RMID,NULL) == -1*** ) {

fprintf(stderr,"Impossivel remover a fila!\n");

exit(1);

}

**Problema 5**

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;

**Problema corrigido:**

A forma de calcular o delta e o total estão escritos de forma errada.

***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;***

**-= Resultados da execução do programa exemplo =-**

Primeiramente executou-se o programa exemplo (Tarefa 1) no mesmo console (prompt de comando) que um programa para roubar tempo de CPU (chamado carga) e os seguintes resultados foram obtidos:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Rodada | Médio (seg) | Máximo (seg) | Carga | Mesmo Console |
| 1 | 0.0000081880 | 0.0000690000 | 0 | SIM |
| 2 | 0.0000055520 | 0.0003090000 | 5 | SIM |
| 3 | 0.0000063880 | 0.0007540000 | 10 | SIM |
| 4 | 0.0000048460 | 0.0006150000 | 15 | SIM |
| 5 | 0.0000197438 | 0.0079910001 | 20 | SIM |
| 6 | 0.0000038660 | 0.0001090000 | 25 | SIM |
| 7 | 0.0000037460 | 0.0001030000 | 30 | SIM |
| 8 | 0.0000032820 | 0.0000170000 | 35 | SIM |
| 9 | 0.0000083560 | 0.0023020001 | 40 | SIM |
| 10 | 0.0000667123 | 0.0055209999 | 45 | SIM |

Tabela - Execução no mesmo console

Fonte do programa usado para roubar tempo da CPU (carga.c):

#include <stdio.h>

int main ()

{

int i = 0;

while (1)

{

i++;

}

return 0;

}

O programa foi usado a partir da segunda rodada de teste, já iniciando com 5 cargas sendo executados ao mesmo tempo. A cada rodada foi sendo acrescentado mais 5 cargas para verificarmos o efeito no desvio.

A seguir mostra-se os prints de cada rodada de execução do programa no mesmo console.

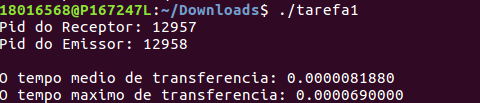
****

Figura - Execução 1: Print PID e resultados

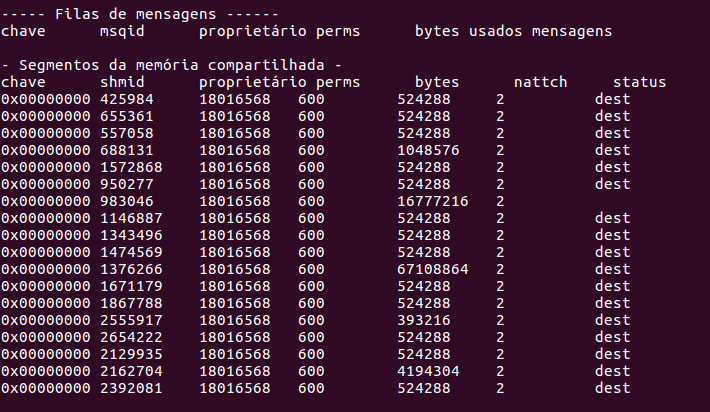
****

Figura - Execução 1: Print do comando ipcs

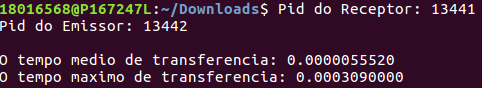
****

Figura - Execução 2: Print PID e resultados

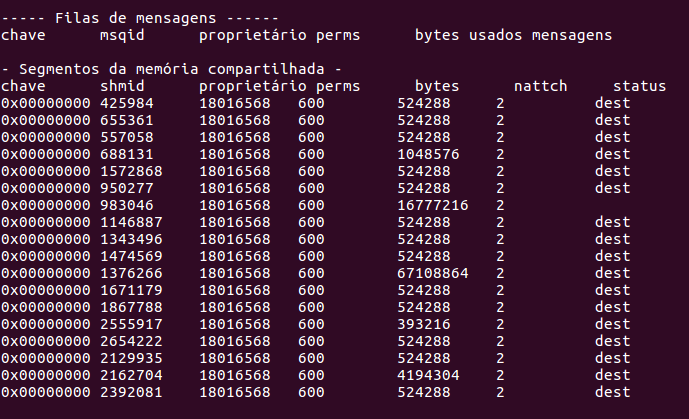
****

Figura - Execução 2: Print do comando ipcs

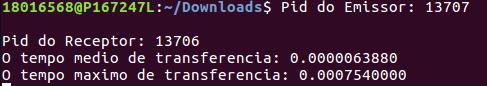
****

Figura - Execução 3: Print PID e resultados

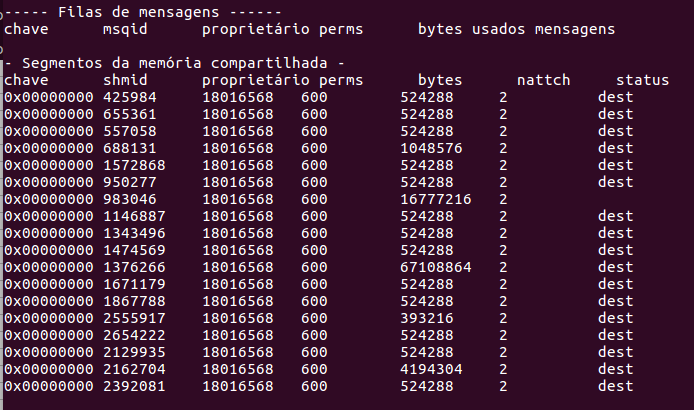
****

Figura - Execução 3: Print do comando ipcs

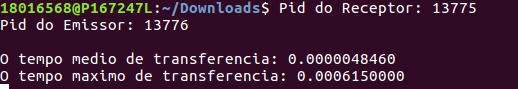
****

Figura - Execução 4: Print PID e resultados

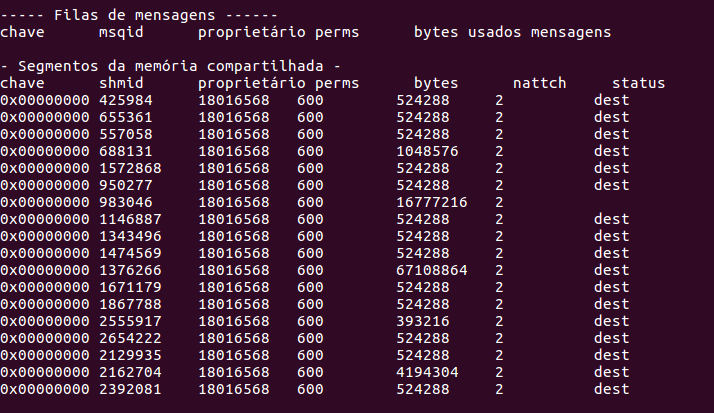
****

Figura - Execução 4: Print do comando ipcs

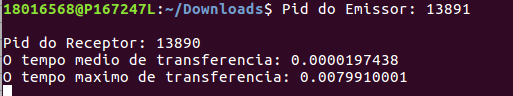
****

Figura - Execução 5: Print PID e resultados

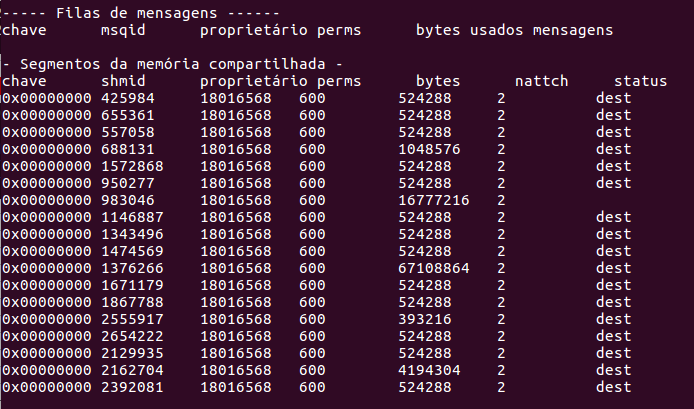
****

Figura - Execução 5: Print do comando ipcs

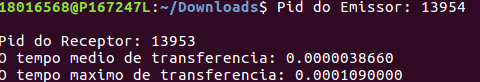
****

Figura - Execução 6: Print PID e resultados

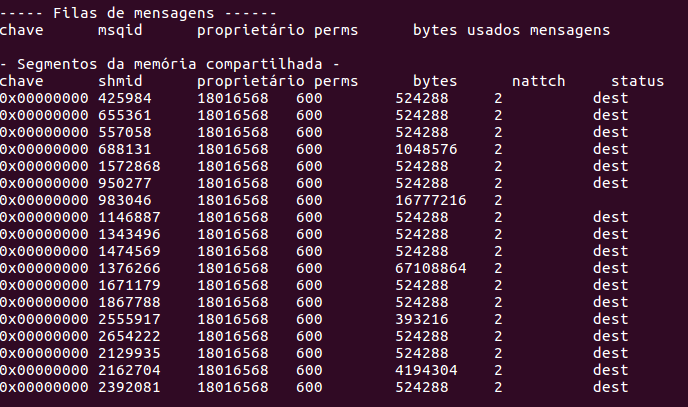
****

Figura - Execução 6: Print do comando ipcs

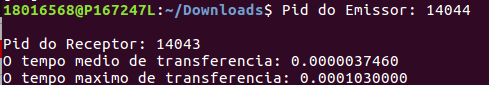
****

Figura - Execução 7: Print PID e resultados

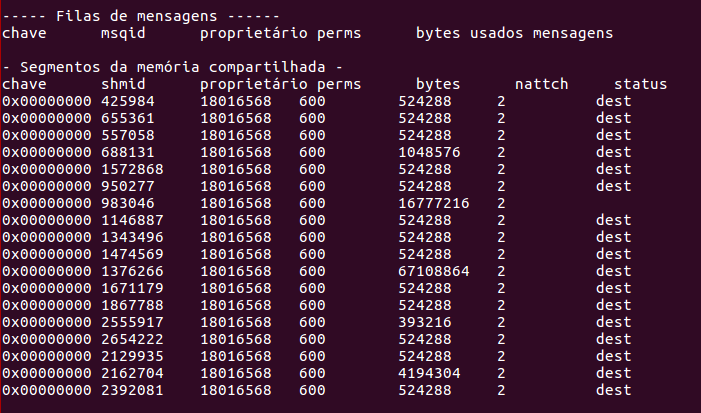
****

Figura - Execução 7: Print do comando ipcs

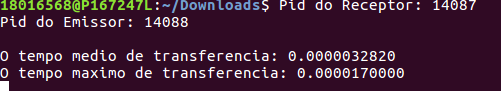
****

Figura - Execução 8: Print PID e resultados

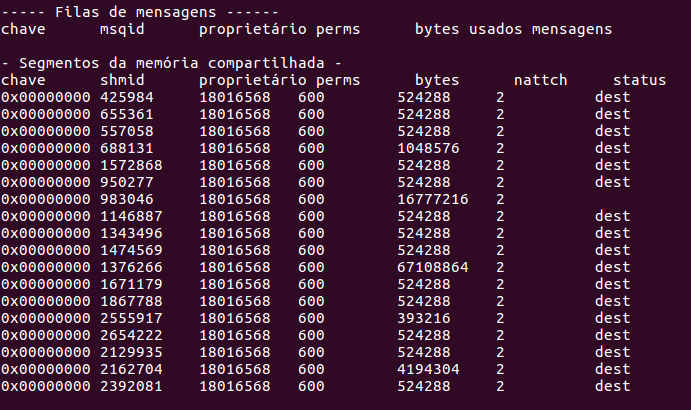
****

Figura - Execução 8: Print do comando ipcs

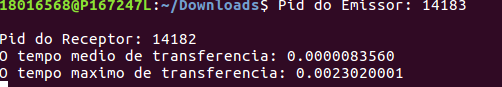
****

Figura - Execução 9: Print PID e resultados

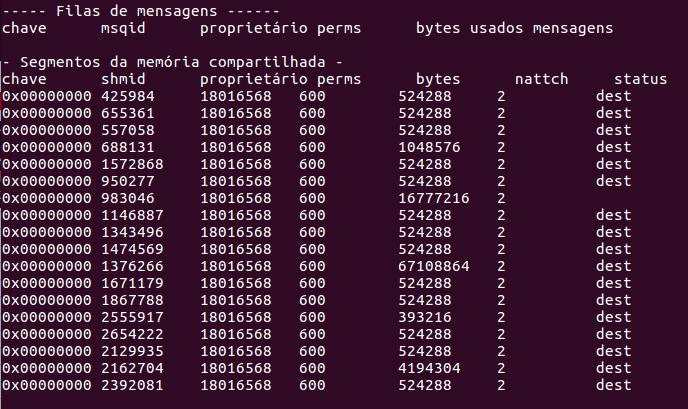
****

Figura - Execução 9: Print do comando ipcs

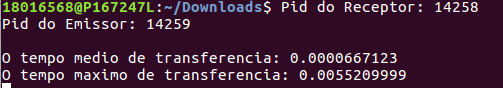
****

Figura - Execução 10: Print PID e resultados

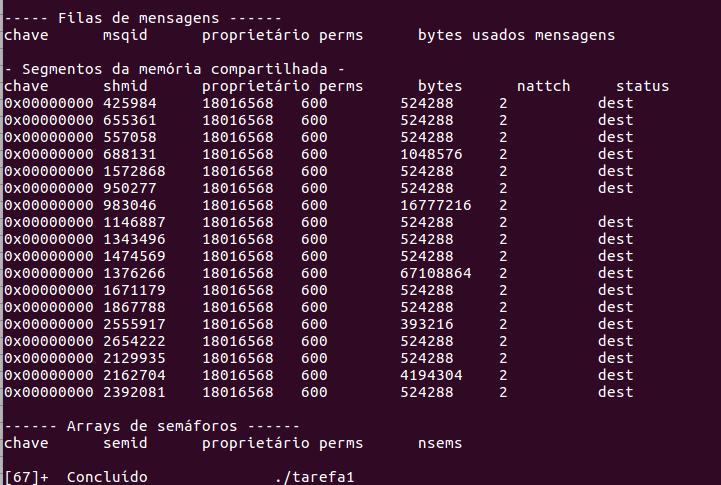
****

Figura - Execução 10: Print do comando ipcs

Após executar o programa exemplo e o programa carga no mesmo console, os mesmos foram executados, porém em consoles diferentes:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Rodada | Médio (seg) | Máximo (seg) | Carga | Mesmo Console |
| 1 | 0.0000030680 | 0.0000490000 | 0 | NÃO |
| 2 | 0.0000233219 | 0.0007560000 | 5 | NÃO |
| 3 | 0.0000035400 | 0.0000290000 | 10 | NÃO |
| 4 | 0.0003147016 | 0.0043890001 | 15 | NÃO |
| 5 | 0.0001015843 | 0.0066189999 | 20 | NÃO |
| 6 | 0.0000111439 | 0.0038999999 | 25 | NÃO |
| 7 | 0.0000585121 | 0.0039039999 | 30 | NÃO |
| 8 | 0.0002504069 | 0.0039049999 | 35 | NÃO |
| 9 | 0.0000347140 | 0.0017930000 | 40 | NÃO |
| 10 | 0.0000038800 | 0.0000180000 | 45 | NÃO |

Tabela - Execução em consoles diferentes

Fonte do programa usado para roubar tempo da CPU (carga.c):

#include <stdio.h>

int main ()

{

int i = 0;

while (1)

{

i++;

}

return 0;

}

O programa foi usado de maneira idêntica à execução do programa no mesmo console.

A seguir mostra-se os prints de cada rodada de execução do programa no mesmo console.

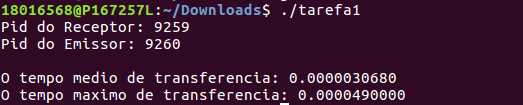
****

Figura - Execução 1: Print PID e resultados

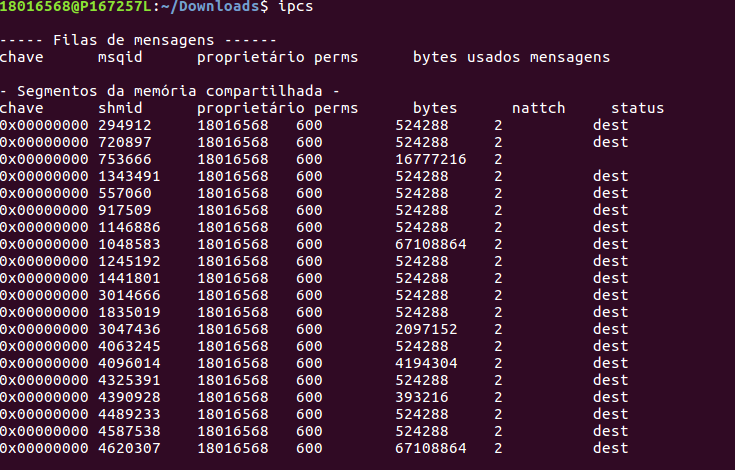
****

Figura - Execução 1: Print do comando ipcs

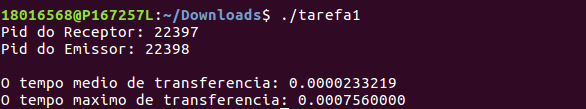
****

Figura - Execução 2: Print PID e resultados

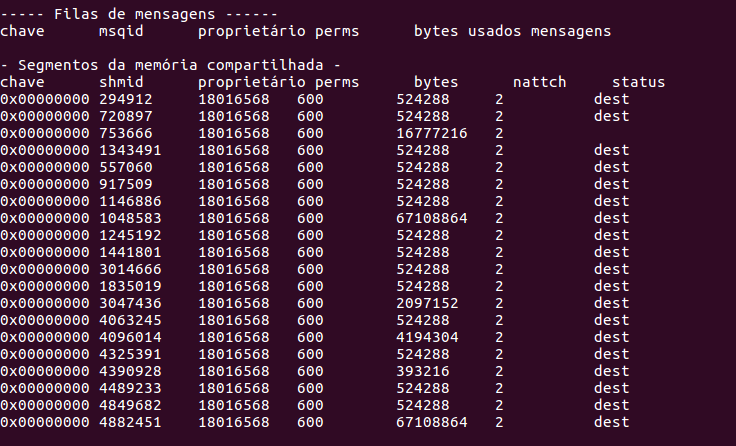
****

Figura - Execução 2: Print do comando ipcs

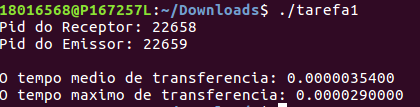
****

Figura - Execução 3: Print PID e resultados

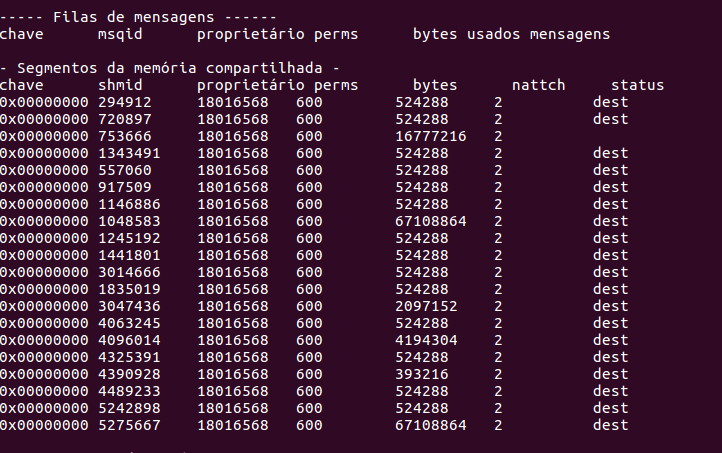
****

Figura - Execução 3: Print do comando ipcs

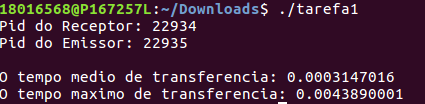
****

Figura - Execução 4: Print PID e resultados

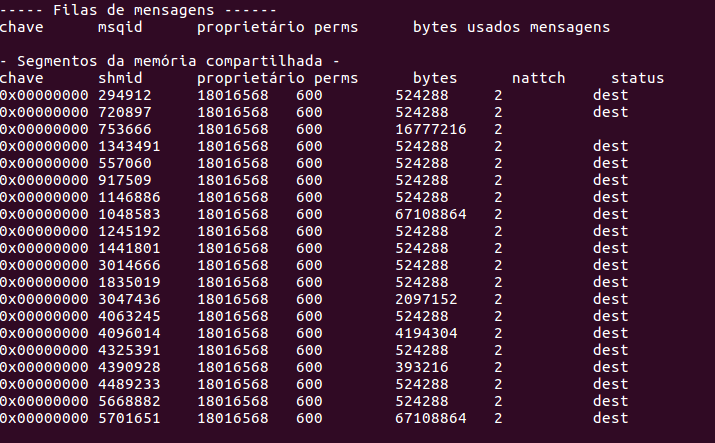
****

Figura - Execução 4: Print do comando ipcs

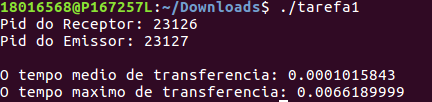
****

Figura - Execução 5: Print PID e resultados

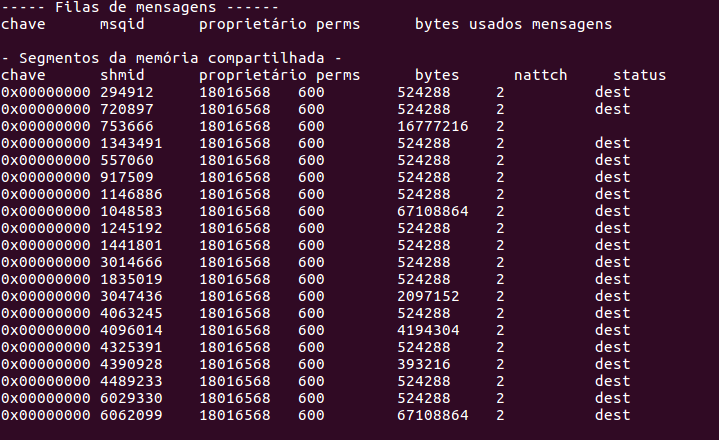
****

Figura - Execução 5: Print do comando ipcs

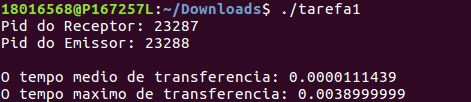
****

Figura - Execução 6: Print PID e resultados

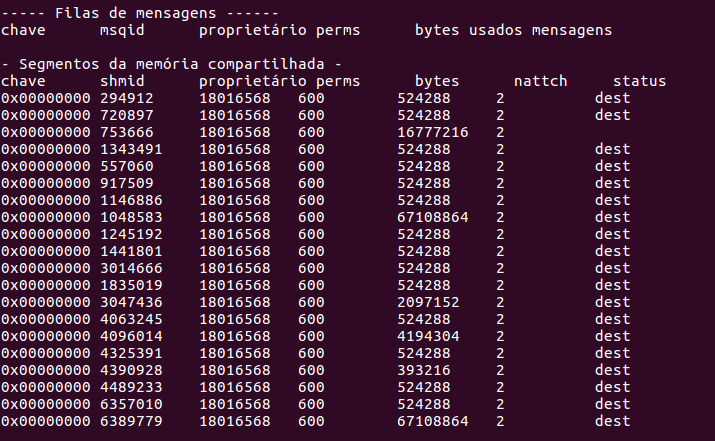
****

Figura - Execução 6: Print do comando ipcs

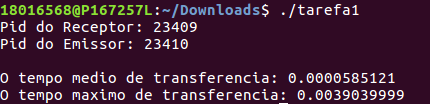
****

Figura - Execução 7: Print PID e resultados

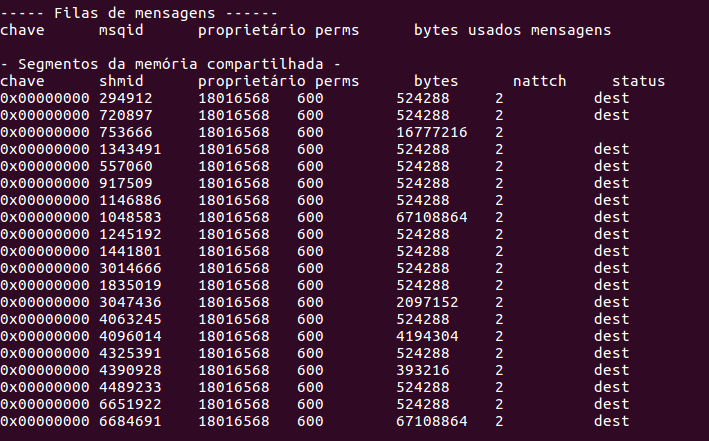
****

Figura - Execução 7: Print do comando ipcs

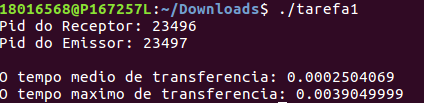
****

Figura - Execução 8: Print PID e resultados

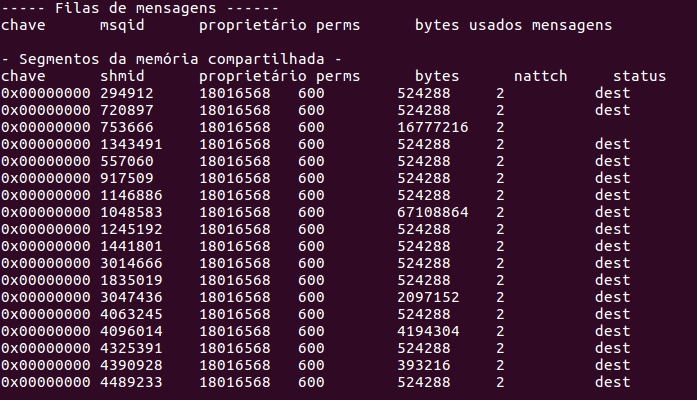
****

Figura - Execução 8: Print do comando ipcs

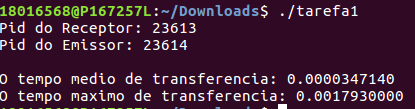
****

Figura - Execução 9: Print PID e resultados

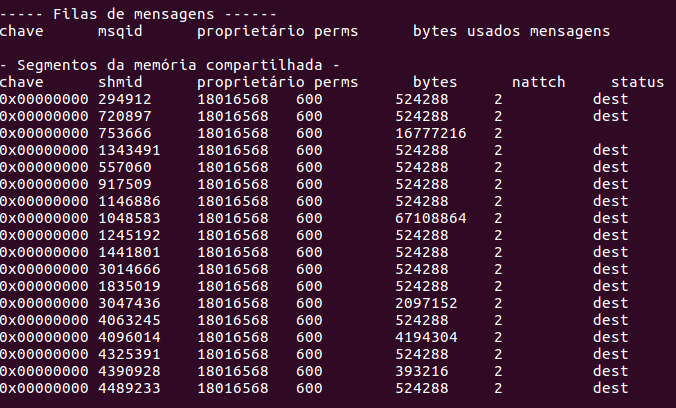
****

Figura - Execução 9: Print do comando ipcs

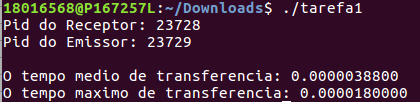
****

Figura - Execução 10: Print PID e resultados

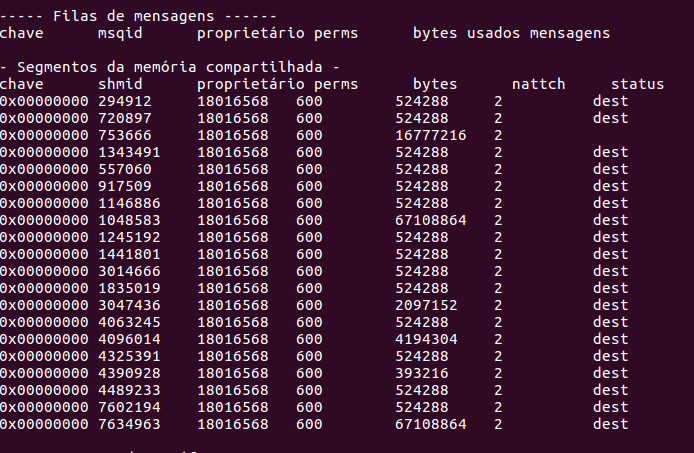
****

Figura - Execução 10: Print do comando ipcs

**-= Resultados da execução do programa modificado =-**

Após modificar o programa e executá-lo, os seguintes resultados foram obtidos (foram feitos dois testes em duas máquinas diferentes para melhor análise do experimento):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Rodada | Tamanho da mensagem | Total (seg) | Médio (seg) | Máximo (seg) | Mínimo (seg) |
| 1 | 1 \* 512 | 0.0026680126 | 0.0000053360 | 0.0000710000 | 0.0000030000 |
| 2 | 2 \* 512 | 0.0024850108 | 0.0000049700 | 0.0000570000 | 0.0000030000 |
| 3 | 3 \* 512 | 0.0026250121 | 0.0000052500 | 0.0000770000 | 0.0000030000 |
| 4 | 4 \* 512 | 0.0025220104 | 0.0000050440 | 0.0000570000 | 0.0000030000 |
| 5 | 5 \* 512 | 0.0026990112 | 0.0000053980 | 0.0000590000 | 0.0000030000 |
| 6 | 6 \* 512 | 0.0026640114 | 0.0000053280 | 0.0001220000 | 0.0000030000 |
| 7 | 7 \* 512 | 0.0025610109 | 0.0000051220 | 0.0000560000 | 0.0000030000 |
| 8 | 8 \* 512 | 0.0028090125 | 0.0000056180 | 0.0000700000 | 0.0000040000 |
| 9 | 9 \* 512 | 0.0030410120 | 0.0000060820 | 0.0000610000 | 0.0000040000 |
| 10 | 10 \* 512 | 0.0028020116 | 0.0000056040 | 0.0000610000 | 0.0000040000 |

Tabela - Execução 1: Resultados do programa modificado

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Rodada | Tamanho da mensagem | Total (seg) | Médio (seg) | Máximo (seg) | Mínimo (seg) |
| 1 | 1 \* 512 | 0,0413350053 | 0,0000826700 | 0,0015330000 | 0,0000060000 |
| 2 | 2 \* 512 | 0,0279500186 | 0,0000559000 | 0,0018160000 | 0,0000160000 |
| 3 | 3 \* 512 | 0,0323980413 | 0,0000647961 | 0,0021060000 | 0,0000100000 |
| 4 | 4 \* 512 | 0,0311090201 | 0,0000622180 | 0,0010340000 | 0,0000120000 |
| 5 | 5 \* 512 | 0,0104360376 | 0,0000208721 | 0,0000670000 | 0,0000170000 |
| 6 | 6 \* 512 | 0,0073929941 | 0,0000147860 | 0,0000440000 | 0,0000060000 |
| 7 | 7 \* 512 | 0,0159809887 | 0,0000319620 | 0,0004070000 | 0,0000120000 |
| 8 | 8 \* 512 | 0,0165020209 | 0,0000330040 | 0,0004350000 | 0,0000190000 |
| 9 | 9 \* 512 | 0,0250889901 | 0,0000501780 | 0,0020099999 | 0,0000140000 |
| 10 | 10 \* 512 | 0,0278699677 | 0,0000557399 | 0,0030330000 | 0,0000110000 |

Tabela - Execução 2: Resultados do programa modificado

O tamanho da mensagem foi sendo aumentado, com o intuito de observar o impacto nos desvios. Como é observado na tabela, o tamanho da mensagem se inicia em 512 (1 \* 512) e vai até 5120 (10 \* 512), aumentando 512 a cada rodada

**-= Respostas das perguntas =-**

**Perguntas do relatório**

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

**Resposta:**

* **Berkeley Unix:** Sistema Unix desenvolvido pela Universidade da Califórnia em Berkeley, usado no período de 1977 até 1955.
* **System V:** System V, ou SysV, é uma das primeiras versões comerciais do sistema Unix, desenvolvida pela American Telephone & Telegraph (AT&T).
* **POSIX:** Portable Operation System Interface, ou de forma abreviada, POSIX é um padrão definido para a compatibilidade entre os diversos sistemas distribuídos baseados no Unix. Ele é responsável por definir algumas variáveis que são obrigatórias além de alguns diretórios padrões dos sistemas Unix.
* **AT&T:** Empresa americana responsável pela prestação de serviços de telecomunicações. Ficou famosa no ramo da computação por conta do desenvolvimento e da distribuição do famoso System V.
* **Socket:** Mecanismo de comunicação baseado no Protocolo de Internet. Normalmente usado para implementar um modelo cliente/servidor, que permite a troca de mensagens entre processos de uma aplicação servidor e de uma aplicação cliente.
* **Fila de Mensagens:** Uma fila de mensagens é uma forma de comunicação assíncrona entre serviços. Na fila, as mensagens são armazenadas até serem processadas ou excluídas. O funcionamento se deve pela teoria do produtor/consumidor, onde um ou mais processos são responsáveis por produzir um conteúdo para a fila e um ou mais consumidores são responsáveis por ler (consumir) o que está na fila.
* **Memória Compartilhada:** Memória que pode ser acessada por mais de um processo de forma simultânea, compartilhando dados entre si.
* **Pipes:** Comunicação entre dois ou mais processos relacionados. Pode ser a comunicação com um processo ou entre Pai e Filho.

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

**Resposta:** *ipcs* é responsável por mostrar quais são as filas de mensagens que estão disponíveis, quais são os segmentos de memória compartilhados e todas as informações dos próprios, como por exemplo o quanto de memoria está sendo utilizado, além dos semáforos que estão sendo executados. Já o comando *ipcrm* é responsável por excluir Inter processos de comunicação (IPC) e todas as estruturas de dados a ela associada do sistema.

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

**Resposta:** A diferença entre o handle devolvido pelomsgget() e a chave única é que o gandle é um inteiro positivo que permite acesso a informações futuras, já a chave única devolve o ID da fila de mensagens.

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

**Resposta:** Segundo a estrutura definida pelo IPC, sim, onde o tamanho é igual apenas 1 byte. Esse problema é contornado com a declaração de uma struct, onde o tamanho máximo da mensagem passa a ser definido pelo programador e varia de acordo com a aplicação.

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

**Resposta:** O tamanho da fila de mensagens é definido pelo tamanho do seu buffer e pode ser divido em 3 categorias para limitar o tamanho máximo da fila, são eles:

* Zero – Nenhuma mensagem fica esperando. A próxima mensagem só é enviada quando a ultima é lida
* Limitada – Limitada por um numero finito N de mensagens.
* Ilimitada – Buffer não tem um tamanho limite definido.

***Pergunta 6: Para que serve um typedef?***

**Resposta:** typedef serve para definir algo como um tipo. Por exemplo ao usar o typedef antes de uma struct, é definido que o nome atribuído para aquela struct passa a ser um novo tipo, como int ou char, porem carregando as características definidas na struct.

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

**Resposta:** Deve ser usando antes da declaração da variável, criando uma referência ao tipo da variável declarada e permitindo o uso de sinônimo. Ex: typedef struct Nome passa a ser chamado no programa apenas por Nome, por conta da referência criada.

***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 “&” é utilizado para indicar o endereço de memória de uma variável. No programa, sem a utilização do “&” resultaria em um erro pois a função msgsnd espera um ponteiro no parâmetro, além da informação buscada ser errada, pois a função busca o endereço de memória e não o conteúdo da variável.

**Perguntas do programa**

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

**Resposta:** Protótipo é uma declaração de uma função, o tipo dela, o seu nome assim como o de seus parâmetros também. É usado para alocar espaço de memória previamente.

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

**Resposta:** Os números "0666" são as permissões de acesso do Unix (permissão de leitura e escrita para todos), juntamente com o “IPC\_CREATE”, são flags usadas para indicar alguma coisa para o sistema. Em relação a cada dígito desta permissão, o 0 no início define que o número é um octal, o segundo campo é o "suid" que disponibiliza uma permissão especial onde os arquivos executáveis que possuam a permissão suid serão executados em nome do dono do arquivo, e não em nome de quem os executou. No terceiro campo tempos o sgid que de maneira semelhante, a permissão atua em diretórios. A permissão sgid é uma permissão de grupo, portanto aparece no campo de permissões referente ao grupo. Num diretório com a permissão sgid, todos os arquivos criados pertencerão ao grupo do diretório em questão, o que é especialmente útil em diretórios com o qual trabalham um grupo de usuários pertencentes ao mesmo grupo. E o quarto campo, terceiro campo depois do 0, é a permissão sticky que inibe usuários de apagarem arquivos que não tenham sido criados por eles mesmos. O número 6 ativa as permissões suid e sgid e não ativa o sticky, e por isso temos 0666.

***Pergunta 3: Para que serve o arquivo stderr?***

**Resposta:** O arquivo stderr é responsável por receber os erros do arquivo executável, gerando uma saída de erro padrão. O erro padrão é um tipo de saída, que é utilizada pelos programas para envio de mensagens de erro ou de diagnóstico. Este fluxo é independente da saída padrão do programa 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 (e, portanto, não observável prontamente). Por exemplo, a saída de um programa em uma canalização Unix é redirecionada para a entrada do próximo programa, mas os erros de cada um deles continuam sendo direcionados ao terminal de texto. É aceitável, e até normal, que a "saída padrão" e o "erro padrão" sejam direcionados para o mesmo destino, como um terminal de texto.

As mensagens aparecem na mesma ordem em que o programa as escreve. O descritor de arquivo para o erro padrão é 2; a variável correspondente na biblioteca stdio.h é FILE \*stderr.

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

**Resposta:** Será apresentado no prompt de saída.

***Pergunta 5: Onde stderr foi declarado?***

**Resposta:** Stderr é algo já declarado e pode ser incluído através de bibliotecas, mais especificadamente na biblioteca <stdio. h>.

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

**Resposta:** Stdin são as entradas do programa, podendo ser arquivos ou comandos do teclado, e o Stdout são as saídas do programa, que regularmente é o monitor.

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

**Resposta:** A fila de mensagens fica retida, apenas ocupando espaço de memória, sendo necessário mandar o comando ipcrm para remover.

***Pergunta 8: Qual será o conteúdo de data\_ptr?***

**Resposta:** data\_ptr receberá o tamanho máximo da mensagem, definido dentro da struct message\_buffer pela variável mtext.

**-= Análise dos Resultados =-**

**Tarefa 1 (programa exemplo)**

Para melhor análise dos resultados, alguns gráficos foram feitos e estão sendo mostrados abaixo:

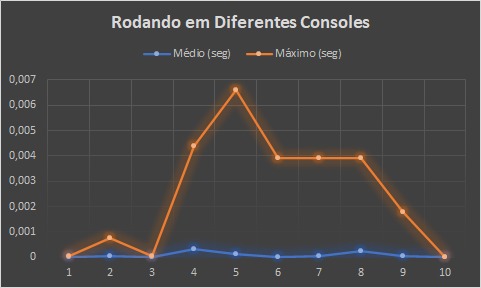


Gráfico - Programa exemplo e carga, rodando em diferentes consoles

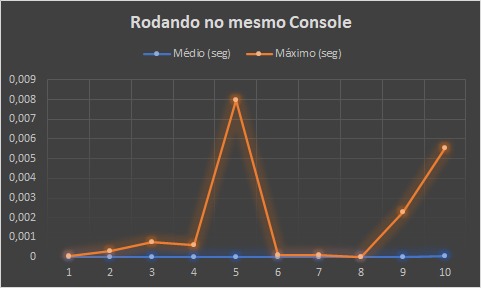


Gráfico 2 - Programa exemplo e carga, rodando no mesmo console

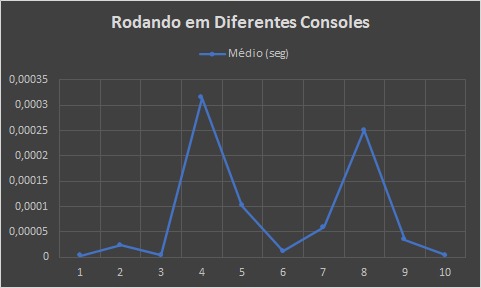


Gráfico 3 - Tempo médio, rodando em diferentes consoles

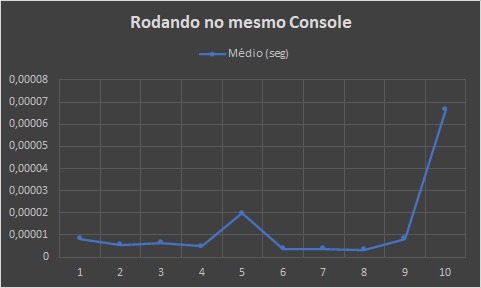


Gráfico 4 - Tempo médio rodando no mesmo console

Os gráficos mostram que apesar do aumento da carga (uso de CPU), o tempo de envio da mensagem não aumentou. A ideia de que o tempo de envio aumentaria se deve ao fato de que mais processos estão usando a CPU e consequentemente, os processos demoram mais para terminar. Porém esse fato não ocorreu e observa-se que:

- Rodando o programa exemplo e a carga em consoles diferentes, o tempo variou bastante;

- Rodando no mesmo console, o tempo foi praticamente igual em todas as rodadas, exceto na rodada 5 e 10.

O fato dos tempos não aumentarem com o tempo, provavelmente ocorreu devido ao frequency scaling. A técnica consiste na incrementação (feita pelo SO) da frequência do clock do processador, conforme a CPU é mais requisitada, aumentando assim a velocidade da CPU e consequentemente aumentando a velocidade com que os processos são executados. Isso pode justificar o fato dos tempos de envio não sofrerem grandes mudanças ao longo das rodadas de teste.

Observa-se também que quando os programas são executados no mesmo console, o tempo é menor e quando executamos em consoles diferentes, o tempo é maior. Quando executamos processos no mesmo console, estes são escalonados um em seguida do outro, diminuindo o tempo de execução de cada um. Quando executamos os processos em diferentes terminais, pelo fato de executarmos o programa da carga primeiro, o programa exemplo pode demorar mais a vir ser escalonado, pelo fato de a CPU já estar sendo utilizada. Isso resulta na maior variância do tempo de envio.

**Tarefa 2 (programa modificado)**

Para melhor análise dos resultados, alguns gráficos foram feitos e estão sendo mostrados abaixo:

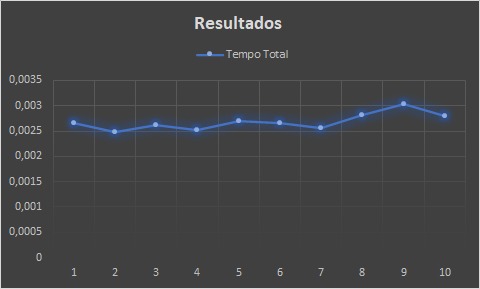


Gráfico 5 - Execução 1: Tempo total do programa modificado



Gráfico 6 - Execução 1: Tempo médio do programa modificado

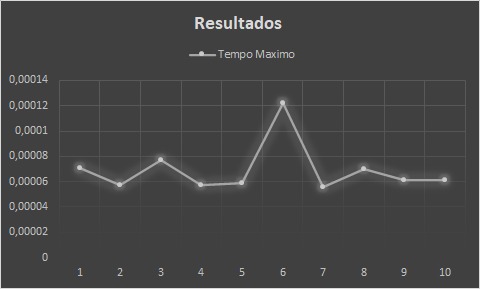


Gráfico 7 - Execução 1: Tempo máximo do programa modificado



Gráfico 8 – Execução 1: Tempo mínimo do programa modificado

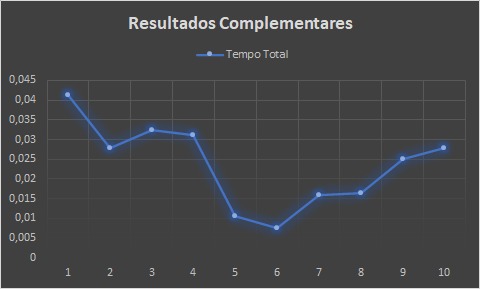


Gráfico 9 - Execução 2: Tempo total do programa modificado



Gráfico 0 - Execução 2: Tempo médio do programa modificado

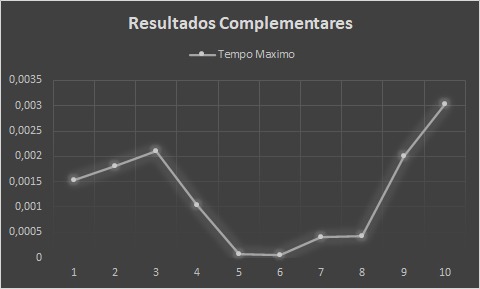


Gráfico 11 - Execução 2: Tempo máximo do programa modificado



Equação 1 - Execução 2: Tempo mínimo do programa modificado

No programa modificado, não houve o aumento no uso da CPU, apenas o aumento no tamanho da mensagem, portanto a análise gira em torno deste tamanho. Os tempos principais a serem observados são o tempo total e o tempo médio, uma vez que mostraram que o aumento no tamanho da mensagem não influenciou no aumento gradativo de envio da mensagem. As variações entre as rodadas de teste são resultado do escalonamento dos processos. Não sabemos quais processos estavam rodando na CPU no determinado instante do teste do programa, e por isso o processo do programa pode ter demorado um pouco mais a ser executado devido a algum outro processo que ficou mais tempo na CPU.

**-= Conclusão =-**

Através deste experimento foi discutido como acontece a comunicação entre processos (IPC), o conceito de fila de mensagens e o entendimento de cada um desses conceitos. Usando o Sistema Operacional Linux para testar os programas, observamos a relação de algumas variáveis que afetam o tempo de envio de uma mensagem. Chamadas de funções como msgsnd(), msgrcv(), além do tempo de transferir a mensagem e o tempo para troca de contexto, causam um desvio no tempo de envio da mensagem.

Também foi possível compreender a utilidade de novas funções como: msgget(), msgclt(), msgsnd() e msgrcv(). A primeira função cria a fila de mensagens, a segunda exclui a fila de mensagens, a terceira envia uma mensagem e a quarta recebe a mensagem.

Outro fato importante foi o estudo de novos comandos do terminal Linux como ipcs e ipcrm. O primeiro comando é responsável por mostrar quais são as filas de mensagens disponíveis e os segmentos de memória compartilhados. Já o segundo comando é responsável por excluir IPCs e todas as estruturas de dados associadas.