**PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS**

**CENTRO DE CIÊNCIAS EXATAS, AMBIENTAIS E DE TECNOLOGIA**

**CHRISTOPHER OLIVEIRA RA:18726430**

**GIULIANO SANFINS RA:17142837**

**MATHEUS MORETTI RA:18082974**

**MURILO ARAUJO RA:17747775**

**VICTOR REIS RA:18726471**

**SISTEMAS OPERACIONAIS A - EXPERIMENTO 2**

**CAMPINAS**

**2020**

**SUMÁRIO**

[1. INTRODUÇÃO 2](#_Toc34838030)

[2. DISCUSSÃO 3](#_Toc34838031)

[3. SOLUÇÃO DE ERROS DE SINTAXE 5](#_Toc34838032)

[3.1 Erro no count, função fork e sintaxe “exit” usada errada 5](#_Toc34838033)

[3.2 Sintaxe do rtn no if errada, wait faltando null e if errado 5](#_Toc34838034)

[3.3 Variável max não inicializada, e lógica do count errada 6](#_Toc34838035)

[3.4 Sinal do if ao contrário 7](#_Toc34838036)

[3.5 Variáveis de lógicas erradas 7](#_Toc34838037)

[4 PERGUNTAS 8](#_Toc34838038)

[4.1 Pergunta 1 8](#_Toc34838039)

[4.2 Pergunta 2 9](#_Toc34838040)

[4.3 Pergunta 3 9](#_Toc34838041)

[4.4 Pergunta 4 9](#_Toc34838042)

[4.5 Pergunta 5 10](#_Toc34838043)

[4.6 Pergunta 6 10](#_Toc34838044)

[4.7 Pergunta 7 10](#_Toc34838045)

[4.8 Pergunta 8 10](#_Toc34838046)

[5. ANÁLISE DOS RESULTADOS 11](#_Toc34838047)

[5.1 Parte 1 11](#_Toc34838048)

[5.2 Parte 2 17](#_Toc34838049)

[6. CONCLUSÃO 22](#_Toc34838050)

# INTRODUÇÃO

O experimento tem como objetivo aprender e se familiarizar com ambiente Linux, como usar o compilador GCC, corrigindo o código que foi entregue tanto logicamente, quanto sintaticamente e o executando. Tendo seu foco na compreensão do funcionamento de métodos que permitem o compartilhamento de dados entre processos.

Existem dois tipos de processos: -Independentes; -Cooperantes. Um processo independente não é afetado pela execução de outros processos, enquanto processos cooperativos podem ser afetados. Pode-se pensar que processos que estão rodando independentemente são executados de forma mais eficiente, porém na prática existem situações onde o uso de processos cooperativos pode ser utilizado de forma a melhorar a velocidade computacional, conveniência e modularidade. De forma resumida, IPCs são mecanismos que permitem processos se comunicam e sincronizam suas ações.

# DISCUSSÃO

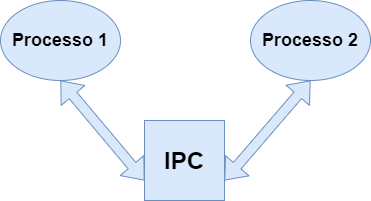
****

Figura 1: Explicação do IPC (Inter Process Communication)

Se 2 processos P1 e P2 querem se comunicar eles precisam prosseguir dessa maneira:

- Estabelecer um link de comunicação (se um link já existe, não é necessário criá-lo).

- Para efetuar a troca de mensagens precisamos de duas primitivas básicas:

•Sender(int queue\_id) - struct enviada com conteúdo da mensagem.

•Receiver(int queue\_id) – struct recebida com conteúdo da mensagem.

Exemplo da struct utilizado no experimento:

typedef struct {

unsigned int msg\_no;

struct timeval send\_time;

} data\_t;

typedef struct {

long mtype;

char mtext[sizeof(data\_t)];

} msgbuf\_t;

O tamanho da mensagem pode ser variável ou fixo. Uma mensagem padrão tem dois campos principais: ID e conteúdo. O ID é o identificador da fila desejada.

A primeira Struct contém o número da mensagem a ser enviada/recebida e o seu contéudo(retorno do gettimeofday).

A segunda Struct contém o tipo da mensagem a ser enviada/recebida, que pode ser tratada posteriormente e o vetor que será armazenado os dados.

Fora criada uma terceira Struct para enviar os dados calculados pelo segundo filho ao terceiro filho para que esse possa imprimi-los.

O programa abaixo tem por objetivo pré-definido agir como carga para aumentar o consumo do processador. Foi utilizado na primeira parte do exercício, onde era necessário colocar até 45 cargas simultâneas, fazendo com que elas competem entre si pelo tempo de CPU.

#include <stdio.h>

int main()

{

long long int count=0;

while(1){

count++;

}

return 0;

}

# SOLUÇÃO DE ERROS DE SINTAXE

## 3.1 Erro no count, função fork e sintaxe “exit” usada errada

• count- -> count++;

• fork -> fork();

• exit(NULL) -> break;

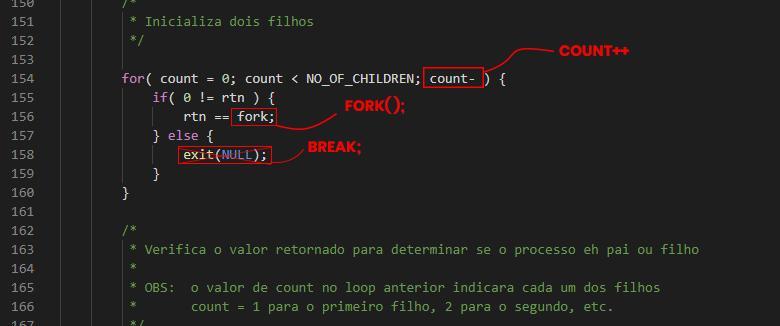


Figura 2: Erro no count, função fork() e de sintaxe no exit

## 3.2 Sintaxe do rtn no if errada, wait faltando null e if errado

• else if( rtn = 0) -> else if(rtn == 0);

• wait() -> wait(null);

• (msgctl == 0) -> (msgctl == -1);

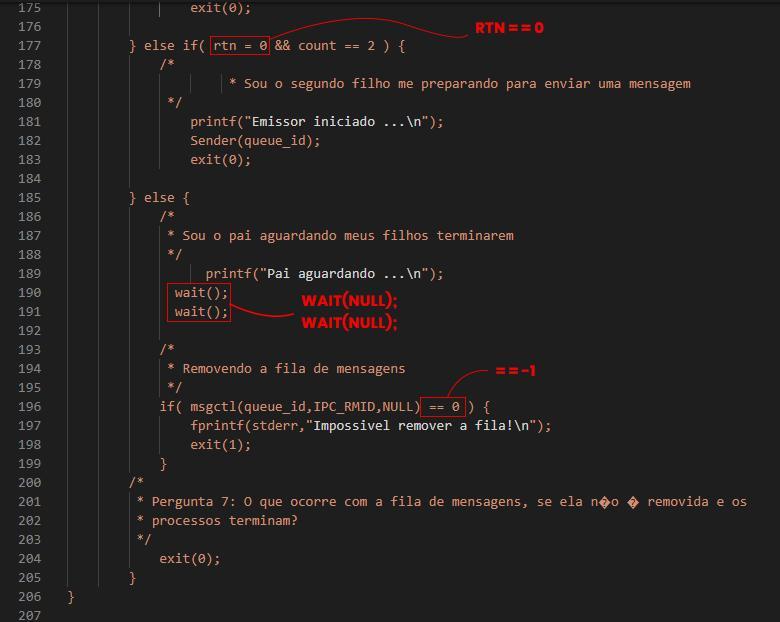


Figura 3: Sintaxe do rtn no if errada, wait faltando null e if errado

## 3.3 Variável max não inicializada, e lógica do count errada

• max; -> max=0;

• ++count -> count++;

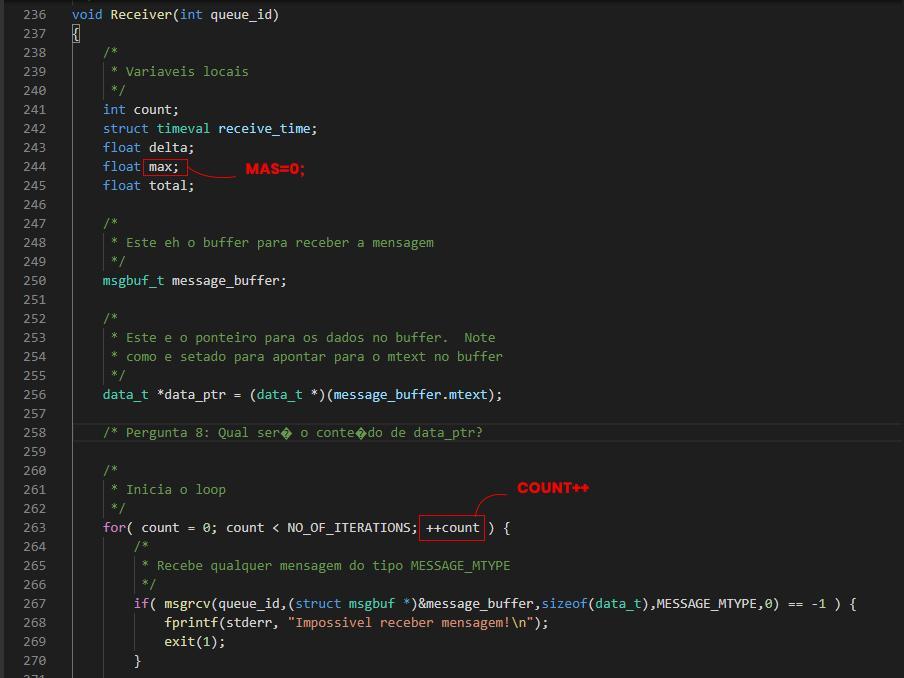


Figura 4: Variável max não inicializada, e lógica do count errada

## 3.4 Sinal do if ao contrário

• delta < max; -> delta > max;

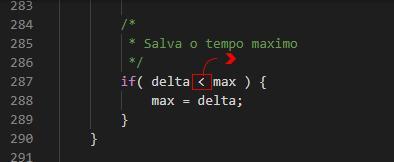


Figura 5: Sinal do if ao contrário

## 3.5 Variáveis de lógicas erradas

• delta-= -> delta==;

• delta = -> delta+=;

• total+=- -> delta+=;

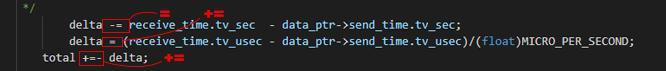


Figura 6: Variáveis de lógicas erradas

# PERGUNTAS

## 4.1 Pergunta 1

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

R: Berkeley Software Distribution (BSD) é um sistema operacional Unix. Este sistema é reconhecido pela facilidade com que ele pode ser licenciado. O Unix System V, normalmente abreviado como SysV, é uma das primeiras versões comerciais do sistema operacional Unix. Foi originalmente desenvolvido pela American Telephone & Telegraph (AT&T).

POSIX (Interface Portável entre Sistemas Operativos) é uma família de normas definidas pelo IEEE para a manutenção de compatibilidade entre sistemas operacionais, 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.

Tem como objetivo garantir a portabilidade do código-fonte de um programa a partir de um sistema operacional que atenda às normas POSIX para outro sistema POSIX, desta forma as regras atuam como uma interface entre sistemas operacionais distintos, enfim, de modo informal "programar somente uma vez, com implementação em qualquer sistema operacional".

AT&T (American Telephone and Telegraph) é uma companhia americana de telecomunicações que provê serviços de telecomunicação de voz, vídeo, dados e Internet para empresas, particulares e agência governamentais.

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. Atualmente, a maioria da comunicação entre computadores é baseada no Protocolo de Internet, portanto a maioria dos soquetes de rede são soquetes de Internet.

## 4.2 Pergunta 2

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

R: O *ipcs* exibe informações relacionadas a comunicação entre processos, por padrão exibi informação sobre compartimento de memória compartilhada, fila de mensagens e vetor de semáforos.

O comando *ipcrm* permite que recursos IPC que tenham acidentalmente restado no sistema após a execução da aplicação possam ser destruídos via linha de comando. Esse comando exige um parâmetro especificando o tipo de recurso a ser destruído, assim como o identificador associado a esse recurso.

## 4.3 Pergunta 3

Qual a diferença entre o handle devolvido pela chamada msgget e a chave única?

R: Na chamada msgget é retornado o identificador msqid da fila, ou -1 em caso de erro. Enquanto em chamadas que utilizam chave única, como ftok o valor de retorno é de uma chave única para todo o sistema ou -1 em caso de erro.

## 4.4 Pergunta 4

Há tamanhos máximos para uma mensagem? Quais?

R: Tem sim defino como 1 byte, porém esse problema pode ser contornado através da declaração da struct.

## 4.5 Pergunta 5

Há tamanhos máximos para uma fila de mensagens? Quais?

R: Sim, 8192 Bytes = 8KB.

## 4.6 Pergunta 6

Para que serve um *typedef*?

R: O comando *typedef* permite ao programador definir um novo nome para um determinado tipo. A palavra reservada *typedef* nada mais é do que um atalho em C para que possamos nos referir a um determinado tipo existente com nomes sinônimos.

  Por exemplo, com o *typedef*, em vez de termos que nos referir como *'struct* *Aluno'*, poderíamos usar somente 'Aluno' para criar structs daquele tipo.

Em vez de escrever sempre *'struct* *Funcionario'*, poderíamos escrever apenas *'Funcionario'* e então declarar várias *structs* do tipo *'Funcionario'*.

## 4.7 Pergunta 7

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

R: O comando *typedef* também pode ser utilizado para dar nome a tipos complexos, como as estruturas.

## 4.8 Pergunta 8

Na chamada *msgsnd* há o uso de *cast*, porém agora utiliza-se um “&” antes   *message*\_*buffer*. Explicar para que serve o “&” e o que ocorreria se este fosse removido.

R: Que apenas use a fila e caso ela não exista não criar a fila não usando ip\_creat, mas posso fazer que caso não exista ela seja criada usando o ip\_create

# ANÁLISE DOS RESULTADOS

Medir o tempo que um determinada ação leva para ser realizada é de extrema importância para verificação de um código, se ele está executando de maneira condizente ao esperado, e tratando-se de paralelismo e comunicação entre processos, conseguir entender como e quando as coisas estão ocorrendo é de vital importância para manter a comunicação entre os processos coerente.

Geralmente a existência das medições em meios computacionais podem afetar os resultados da medição, podendo ser devido a forma como o fora escalonado o processo que está realizando a medição, flutuações de desempenho do sistema, falhas no software, sobre cargas, entre outros motivos.

A respeito da comunicação entre processos, foi necessário achar uma forma de garantir que um determinado processo filho recebe-se uma determinada mensagem, durante a primeira parte não houveram problemas desse tipo pois só havia um processo que enviava mensagem e outro que recebia, porém na segunda parte existia um processo que enviava, um que recebia e enviava, e outro que recebia um mensagem, e nessa situação, quando iniciada a execução do código, ambos os processos 2 e 3 já ficavam esperando por uma mensagem, porém o processo 3 deveria esperar que o 2 tivesse-se lido a mensagem e enviado outra (se tratava de mensagem diferentes que deveriam ler), para isso (como fora anteriormente relatado) foi criado dois canais de comunicação diferentes com structs de mensagem diferentes, assim não tem problema que o processo filho 3 fica aguardando uma mensagem pois só terá uma mensagem no canal que ele está esperando quando o filho 2 colocar nele sua mensagem.

## 5.1 Parte 1

Os seguintes dados são referentes a primeira parte do experimento, onde foi coletado os tempos médios e máximos da comunicação entre os processos, em relação com a quantidade de cargas na CPU. Nesta primeira parte foi usado o Taskset para restringirmos o processamento das cargas e do programa para somente um núcleo, não se trata da melhor forma de fazê-lo, porém ele cumpre com o objetivo, além de ser uma forma, relativamente, simples de conter em apenas um núcleo de processamento todos os itens necessários para a análise.

Analisando os gráficos é possível perceber que tanto variação no tempo médio e máximo seguem de forma mui parecida, tendo leves variações entre os pontos de 1 a 4 e de 8 a 12 tendo, porém, uma variação abrupta nos pontos 5,6, e 7 e sendo a maior delas no ponto 6. Isso pode ter ocorrido pois até aquela quantidade de cargas o processador com uma determinada frequência conseguia seguir seu processamento sem muita variação (na temperatura do processador, quantidade de energia consumida), porém quando percebeu que, devido ao aumento no número de cargas, tal frequência talvez não fosse mais suficiente para suprir a necessidade, então a frequência do processador fora aumentada, isso ocorre devido a técnica da escala de frequência dinâmica, fazendo com que os próximos pontos (7, 8, 9 e 10) voltassem a uma variação entre eles mais branda. Na tabela abaixo, podemos verificar os dados que foram executados com o auxílio do comando *“Taskset”* no *linux*, onde foi possível isolar um núcleo do processador para essa determinada tarefa, *entretanTo*, esse isolamento não permite que o sistema operacional escalone outros processos para esse núcleo e nem que seja possível isolar os outros núcleos do CPU, mas de certa forma, foi uma maneira que encontramos para comparar o que aconteceria se isolarmos esse núcleo e também sem estar isolando, como podemos observar abaixo das tabelas comparativas que utilizamos o *“Taskset”.*

|  |  |  |  |
| --- | --- | --- | --- |
| **COM TASKSET** | | | |
|  | | | |
| **Execução** | **Médio (s)** | **Máximo (s)** | **Carga** |
| 1 | 0,000006782 | 0,00025248 | 0 |
| 2 | 0,000002434 | 0,00007 | 5 |
| 3 | 0,00000308 | 0,000082 | 10 |
| 4 | 0,00000314 | 0,000088 | 15 |
| 5 | 1,08178E-05 | 0,003818 | 20 |
| 6 | 1,84138E-05 | 0,007594 | 25 |
| 7 | 1,06238E-05 | 0,00385 | 30 |
| 8 | 0,000003042 | 0,00002 | 35 |
| 9 | 0,000002974 | 0,000021 | 40 |
| 10 | 0,000003374 | 0,000018 | 45 |

Tabela 1: Execuções com *Taskset*

Gráfico 1: Gráfico 1: Análise do tempo médio (segundos X execuções)

Gráfico 2: Análise do tempo médio com curvas (segundos X execuções)

Gráfico 3: Análise do tempo máximo (segundos X execuções)

Gráfico 4: Análise do tempo máximo com curvas (segundos X execuções)

|  |  |  |  |
| --- | --- | --- | --- |
| **SEM TASKSET** | | | |
|  | | | |
| **Execução** | **Médio (s)** | **Máximo (s)** | **Carga** |
| 1 | 0,000004832 | 0,000035 | 0 |
| 2 | 0,000005718 | 0,000351 | 5 |
| 3 | 0,001035124 | 0,007992003 | 10 |
| 4 | 1,09978E-05 | 0,003914 | 15 |
| 5 | 2,17578E-05 | 0,00649 | 20 |
| 6 | 0,00075412 | 0,003988999 | 25 |
| 7 | 0,000008082 | 0,000359 | 30 |
| 8 | 0,001649099 | 0,02004999 | 35 |
| 9 | 1,68139E-05 | 0,003978 | 40 |
| 10 | 0,000562899 | 0,012022 | 45 |

Tabela 2: Execuções sem o *Taskset*

Gráfico 5: Análise do tempo médio sem utilizar *Taskset* (segundos X execuções)

Gráfico 6: Análise do tempo médio sem utilizar *Taskset* com curvas (segundos X execuções)

Gráfico 7: Análise do tempo máximo sem utilizar *Taskset* (segundos X execuções)

Gráfico 8: Análise do tempo médio sem utilizar *Taskset* com curvas (segundos X execuções)

## 5.2 Parte 2

Os seguintes dados são referentes a segunda parte do experimento, onde foram coletados os tempos médios, máximos, mínimos e totais da comunicação entre três processos, diferentemente da primeira parte desta vez não foram adicionadas cargas na CPU, mas foi feita uma relação do tempo com tamanho da mensagem (tamanho selecionados de 1 a 10 multiplicados por 512).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Nº** | **TEMPO MÉDIO (s)** | **TEMPO MÁXIMO (s)** | **TEMPO MÍNIMO (s)** | **TEMPO TOTAL (s)** |
|  | | | | |
| 1 | 0,0000111220 | 0,0002810000 | 0,0000040000 | 0,0055609858 |
| 2 | 0,0000089560 | 0,0001200000 | 0,0000060000 | 0,0044780094 |
| 3 | 0,0000103020 | 0,0000920000 | 0,0000070000 | 0,0051510017 |
| 4 | 0,0000065660 | 0,0000500000 | 0,0000050000 | 0,0032830143 |
| 5 | 0,0000091800 | 0,0001150000 | 0,0000070000 | 0,0045900140 |
| 6 | 0,0000104000 | 0,0001010000 | 0,0000070000 | 0,0051999968 |
| 7 | 0,0000101500 | 0,0001100000 | 0,0000060000 | 0,0050750095 |
| 8 | 0,0000062980 | 0,0000550000 | 0,0000040000 | 0,0031490126 |
| 9 | 0,0000114040 | 0,0001000000 | 0,0000080000 | 0,0057019917 |
| 10 | 0,0000118540 | 0,0000640000 | 0,0000070000 | 0,0059269960 |

Tabela 3: Execuções relacionada a parte 2 do experimento

Gráfico 9: Análise do tempo médio (segundos X execuções)

Gráfico 10: Análise do tempo médio com curvas (segundos X execuções)

Gráfico 11: Análise do tempo mínimo (segundos X execuções)

Gráfico 12: Análise do tempo mínimo com curvas (segundos X execuções)

Gráfico 13: Análise do tempo máximo (segundos X execuções)

Gráfico 14: Análise do tempo máximo com curvas (segundos X execuções)

Gráfico 15: Análise do tempo total (segundos X execuções)

Gráfico 16: Análise do tempo total com curvas (segundos X execuções)

# CONCLUSÃO

Este experimento teve como propósito o estudo e, portanto, o conhecimento a respeito de comunicação entre processos, neste utilizando-se de filas de mensagens, e também submetendo o programa a uma série de teste para que seja possível uma análise mais precisa.

Através das funções *msgrcv* (para receber) e *msgsnd* (para enviar), foi possível compreender o funcionamento da comunicação e todas as suas dificuldades, desde o conflito entre os processos, como exemplo, devido ao paralelismo, quem tem de receber uma mensagem, pode ocorrer de recebê-la muito antes de ser enviada, travando todo sistema, com este travamento pode concluir-se que as funções msgrcv e msgsnd implementa uma espera ocupada. Outro ponto é a chave que pode ser gerada para que seja possível abrir a mesma fila em processos diferentes, assim possibilitando a comunicação.

Destarte o objetivo almejado foi alcançado com êxito, correspondendo a todas expectativas de aprendizado sobre o assunto em questão, tornando nossos conhecimentos sobre processos cada vez mais diversificado, e nos ajudando a compreender como realmente tudo acontece no SO, mesclando ideias antes soltas, em algo muito mais consistente.