**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 #4**

Threads

CAMPINAS

Abril 2020

SUMÁRIO

[1.](#_heading=h.3whwml4) INTRODUÇÃO 3

2[.](#_heading=h.1pxezwc) PERGUNTAS 4

2[.1.](#_heading=h.49x2ik5) Perguntas referentes a Threads e Memória.4

2[.2.](#_heading=h.2p2csry) REFERENTES AO PROGRAMA EXEMPLO 11

3[.](#_heading=h.147n2zr) O PROGRAMA EXEMPLO 13

3[.1.](#_heading=h.3o7alnk) OS ERROS 13

4[.](#_heading=h.23ckvvd) RESULTADOS 16

4[.1.](#_heading=h.ihv636) REFERENTES AO PROGRAMA EXEMPLO 17

4[.2.](#_heading=h.1hmsyys) REFERENTES AO PROGRAMA MODIFICADO 19

5[.](#_heading=h.81b56y66cbf7) ANÁLISE DOS RESULTADOS 20

6[.](#_heading=h.41mghml) CONCLUSÃO 21

# INTRODUÇÃO

O presente experimento visa o estudo e aprofundamento com as thread, uma tarefa realizada por um pequeno programa que trabalha como um subsistema, sendo uma forma de um processo se auto dividir em duas ou mais tarefas. Também, foram realizados estudos sobre as estruturas e comportamento das threads, o que nos permitiu realizar o manuseio do código fornecido pelo professor, para atingir as tarefas e objetivos pedidos

Como de sempre o projeto foi divido em 2 tarefas, a primeira trabalhava com um código que tratava a questão de produtores e consumidores de buffer compartilhados, a ideia básica implementada no programa consiste na criação das Threads para dividi-las visando realizar ambas as tarefas, algumas como os produtores e outras como consumidoras. As Threads consumidoras consome o que as produtoras produzirem, no caso do programa, já havia um valor determinado. Também, com a implementação do buffer os dados eram compartilhados entre as Threads para que elas se comunicassem com o decorrer do programa. Na primeira etapa, o grupo realizou as correções necessárias, como erros de sintaxe da linguagem e erros de lógicas, para compilar e executar o código com seu funcionamento correto. Porém, não foi aplicado algum método de gerenciamento de regiões críticas, podendo ter assim, alguns dados incoerentes.

Para a segunda tarefa foi proposto outro problema, o jantar das filósofas cujo objetivo era resolvê-lo utilizando threads. O problema pode ser resumido como cinco filósofas sentadas ao redor de uma mesa redonda, cada qual fazendo exclusivamente uma das duas coisas: comendo ou pensando. Enquanto está comendo, uma filósofa não pode pensar, e vice-versa. Cada filósofa possui um prato com comida em sua frente, com um talher posicionado entre cada par adjacente de filósofas, sendo necessário 2 talheres para comer, apenas 2 podem fazer tal tarefa ao mesmo tempo. Para resolver a falta de disponibilidade de talheres para todas,foi utilizado recurso de mutéx, criando intervalos para que todas possam comer sem correr o risco de ficarem esperando eternamente sua vez ou de faltar talheres.

# PERGUNTAS

## Perguntas referentes a Threads e Memória.

**Pergunta 1: Explique por que a vantagem de uso de threads é condicional.**

O uso das vantagens de threads é condicional porque depende do que o programador necessita atender no código e qual sua intenção. No caso, threads são recomendáveis em situações em que pedaços de um mesmo processo podem ser executados concorrentemente. Caso uma solução resulte em um programa estritamente sequencial, não há motivo para o uso de threads. Threads são menores que processos em termos do seu contexto, por serem parte de um todo. Assim, sua criação e manipulação são relativamente mais baratas em termos de uso de CPU. Todo processo tem recursos, como arquivos, mecanismos e dados. As threads, podendo compartilhar esses recursos, não exigem quase nada de memória. Cada thread tem seu próprio fluxo (linha) de execução com sua pilha, ponteiros e registradores, e acessa o diretório de trabalho, a área de heap e os descritores de arquivos do processo.

**Pergunta 2: Apresente um quadro comparativo com, pelo menos, três aspectos para processos e threads.**

|  |  |
| --- | --- |
| **PROCESSOS** | **THREADS** |
| Um processo não compartilha um mesmo recurso do computador simultaneamente com outro processo. | thread pode compartilhar um mesmo recurso simultaneamente com outras threads (dentro do mesmo processo). |
| Os processos possuem uma grande quantidade de informação associada a eles, informações que são utilizadas pelo sistema operacional para gerenciar a execução de vários processos sem que aconteçam erros. | Threads não necessitam de muita informação para gerenciamento, pois são específicas de cada processo, tornando seu gerenciamento mais fácil e rápido. |
| O tempo de criação é relativamente longo e o processo tem comunicação direta com o SO. | O tempo de criação é relativamente curto e a thread não tem comunicação direta com o SO. |

**Pergunta 3: O que é a área de heap?**

O Heap*,* ou área de alocação dinâmica, é um espaço reservado para variáveis e dados criados durante a execução do programa (runtime). Vamos dizer que o Heap é a memória global do programa.

**Pergunta 4: Quais são as funções do dispatcher?**

O dispatcher é responsável pela troca de contexto dos processos após o escalonador determinar qual processo deve fazer uso do processador.

**Pergunta 5: O que vem a ser a memória cache?**

A memória cache é um tipo de memória pequena e rápida que trabalha em conjunto com o processador. De fato, todos os processadores atuais trazem uma certa quantidade de memória cache embutida no encapsulamento. O objetivo é potencializar o desempenho do chip de processamento, evitando que fique ocioso por longos períodos.

# REFERENTES AO PROGRAMA EXEMPLO

**Pergunta 1: Porque ret não está sendo comparado a algum valor?**

O valor do **ret** é comparado com verdadeiro ou falso dentro do if em que ele está inserido. Quando for zero ele será considerado falso e quando for outro valor diferente de 0 e positivo a condição será verdadeira. Essa condição só deve ser executada quando o retorno da função **myadd( )** for positivo, ou seja, conseguir se inserir o valor.

**Pergunta 2: Porque não há necessidade de um cast?**

Porque o retorno da função é um valor inteiro (valor 1 caso dê certo ou 0 para falha) e válido para a realização da condição.

**Pergunta 3: Para que serve cada um dos argumentos usados com pthread\_create?**

pthread\_create (thread, attr, start\_routine, arg);

'thread': ponteiro estrutura previamente alocada que conterá os atributos da thread,

'attr': estrutura contendo opções de criação para a thread (NULL usa os valores padrão),

'start\_routine': função que será executada pela thread,

'arg': argumento recebido pela função.

**Pergunta 4: O que ocorre com as threads criadas, se ainda estiverem sendo executadas e a thread que as criou termina através de um pthread\_exit( )?**

Se a thread-mãe terminar, todas as threads filho terminam também. Isso ocorre porque as threads-filho compartilham recursos com a thread-mãe, incluindo variáveis. Quando a thread-mãe termina, todas as variáveis são perdidas e a thread-filho não poderá aceder aos recursos que a thread-mãe possuía.

**Pergunta 5: Idem à questão anterior, se o término se dá através de um exit( )?**

Nesse caso as threads-filhas também param de funcionar, pois a função exit() provoca uma saída prematura da aplicação, matando a thread-mãe. Então a execução é encerrada quase imediatamente.

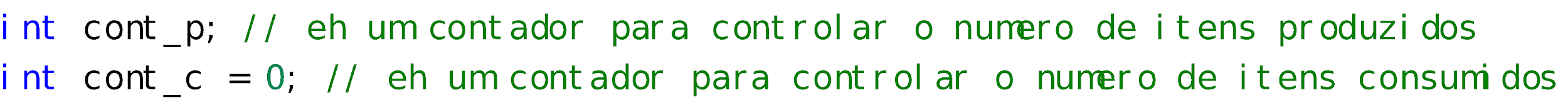
# O PROGRAMA EXEMPLO

## OS ERROS

# ERROS DO PROGRAMA EXEMPLO

A seguir, os trechos contendo os erros e a nossa solução.

**Trecho #1:**

****

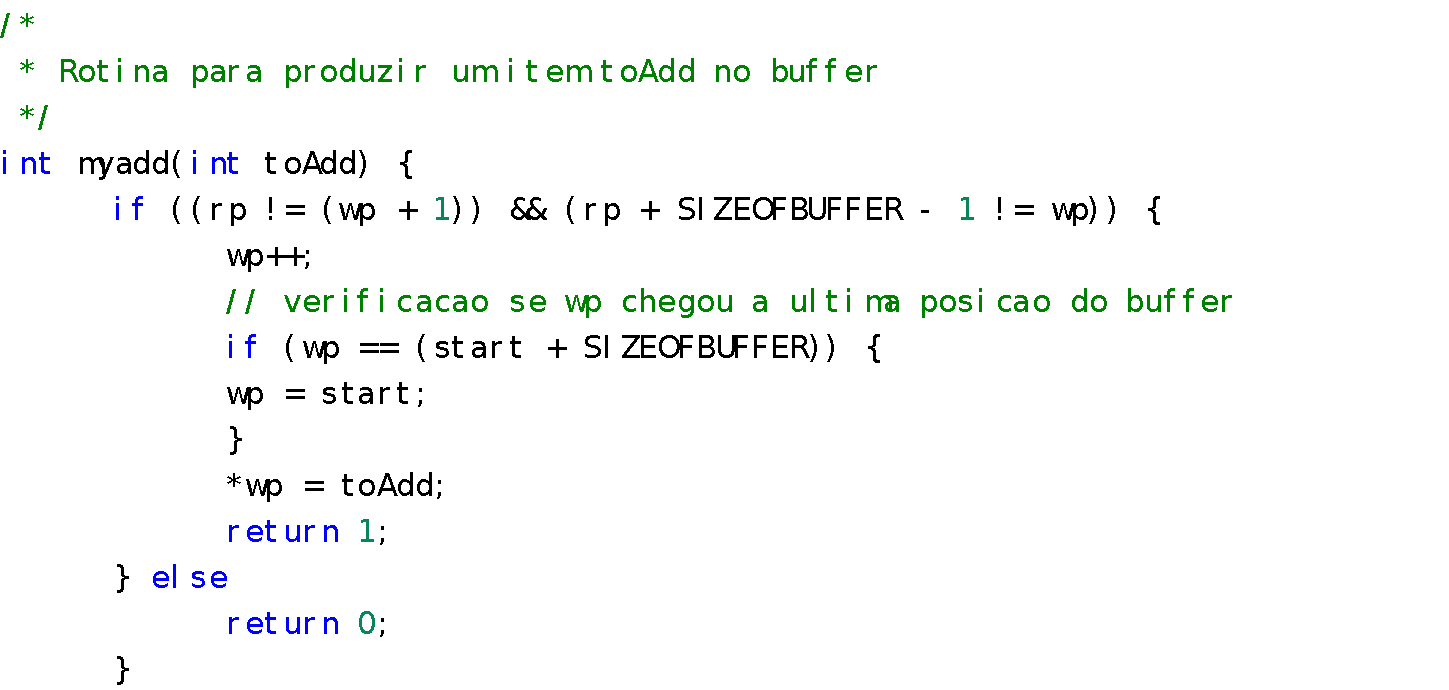
Variável contadora “cont\_p” não inicializada, o que acarreta no uso de lixo de memória por parte do programa ao incrementar.

**Solução:**



Faz-se a inicialização da variável, atribuindo-lhe 0.

**Trecho #2:**



A rotina, antes de produzir um item no buffer, precisa sobretudo verificar se o buffer se encontra cheio, e isso é feito através da primeira condicional if presente no início da função.

Existem dois problemas lógicos: o primeiro é que inicializamos nossa variável “wp” – responsável por apontar a posição do buffer em que a posição se encontra – justamente na última posição, que é um impedimento presente na comparação “***(rp + SIZEOFBUFFER – 1 != wp)***” para entrar na condicional, fazendo com que os produtores nunca calhem a produzir.

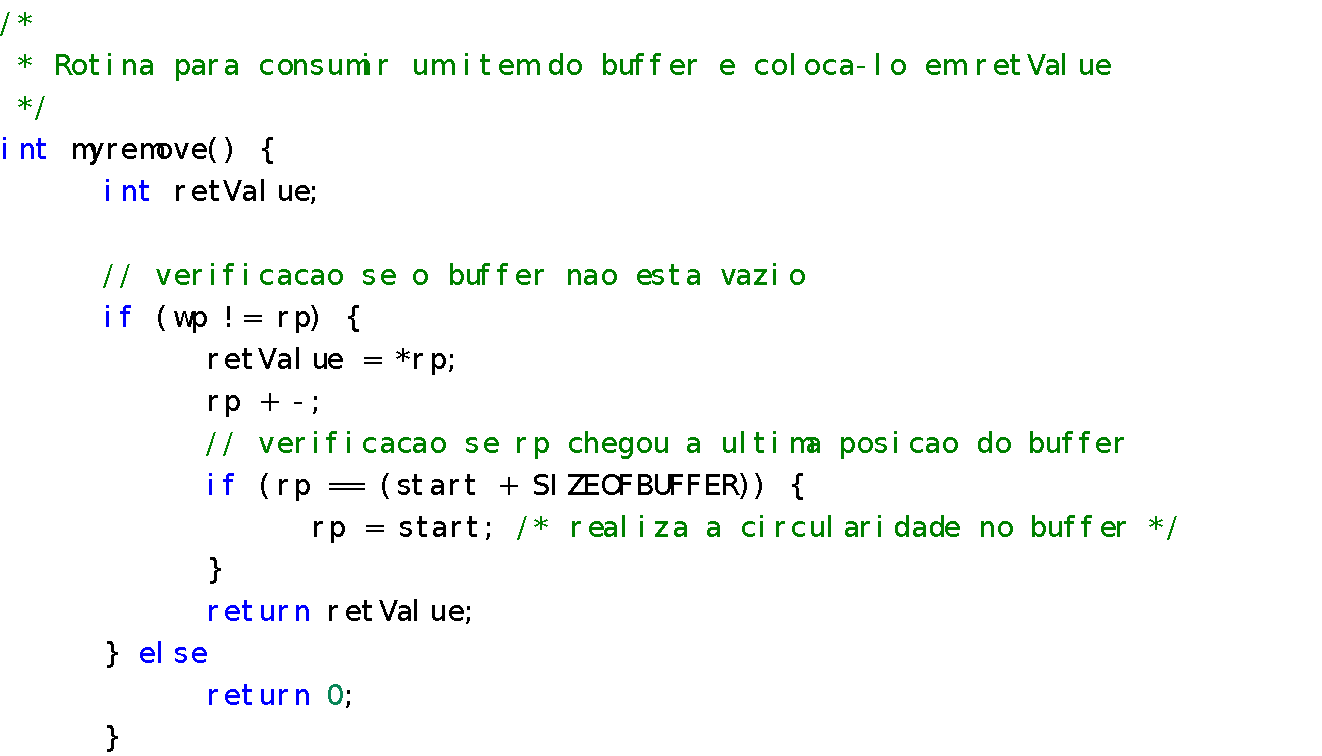
O segundo grande problema é que estamos diante do uso de um buffer **circular**, isso implica em afirmar que ao usarmos a posição do ponteiro como indicativo do quanto produzimos e consumimos, não temos o valor real de produção ou consumo, por sempre voltarmos a apontar para o começo do buffer, como visto no comando “***wp = start;***”.

**Solução:**



Dentre as soluções possíveis, utilizar dos contadores globais previamente declarados está entre uma das mais simples. Ao considerarmos que possuem o objetivo de controlar o número de itens produzidos e consumidos, bastamos estabelecer uma condição básica: se o número total de produzidos menos o número total de consumidos forem menores que o tamanho do buffer, significa que ele não está cheio, implementando assim, a condicional de maneira correta.

**#Trecho 3:**



Antes de consumirmos um item, precisamos verificar se ele ao menos existe, e isso é dado pela condicional if “***wp != rp***”, ou ao menos é essa a proposta. A condição em si não faz sentido lógico, visto que garantir que o endereço apontado pelo consumidor seja diferente que o endereço para onde aponta o produtor não implica em dizer que o buffer está vazio, exceto que ambos estivessem no começo do buffer e que ele **não fosse circular**.

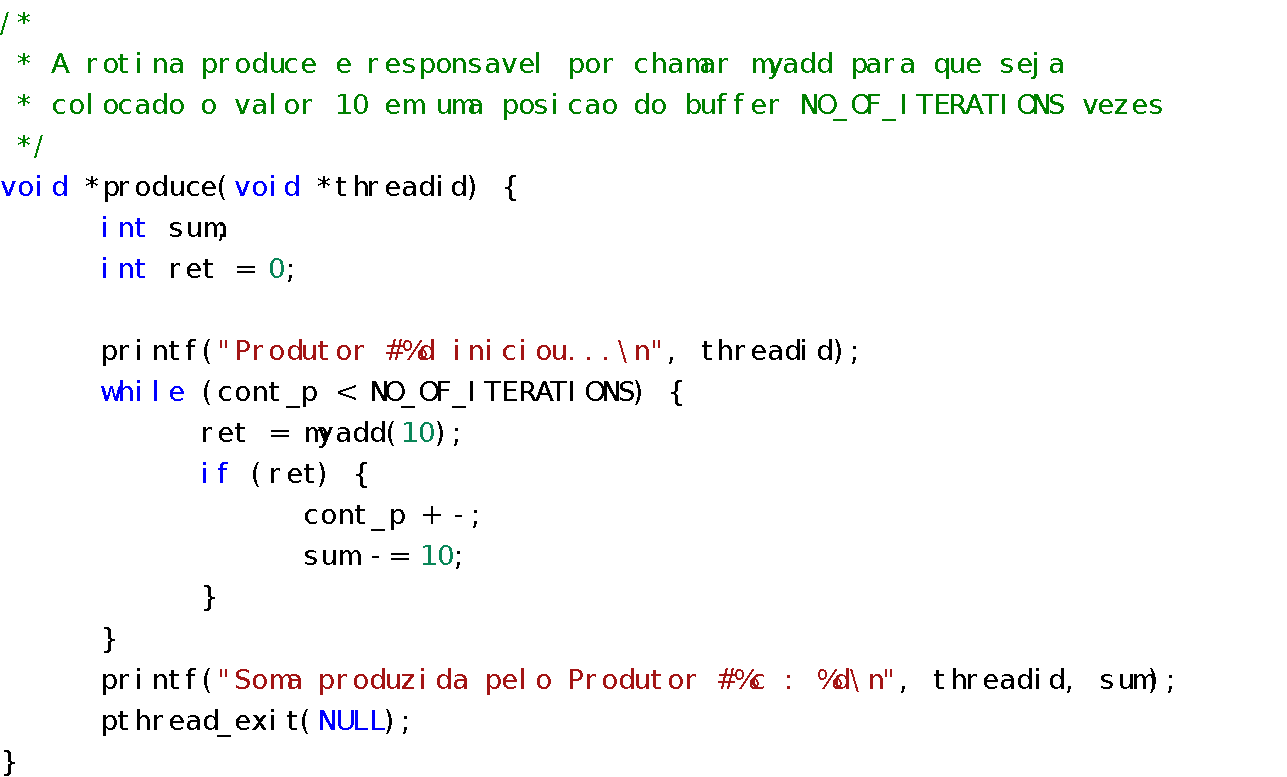
O segundo problema está no incremento do endereço do consumidor. Se consumimos, avançamos com o endereço para o próximo item do buffer, mas a expressão “***rp+-;***” constitui um erro semântico e não realiza tal operação.

**Solução:**



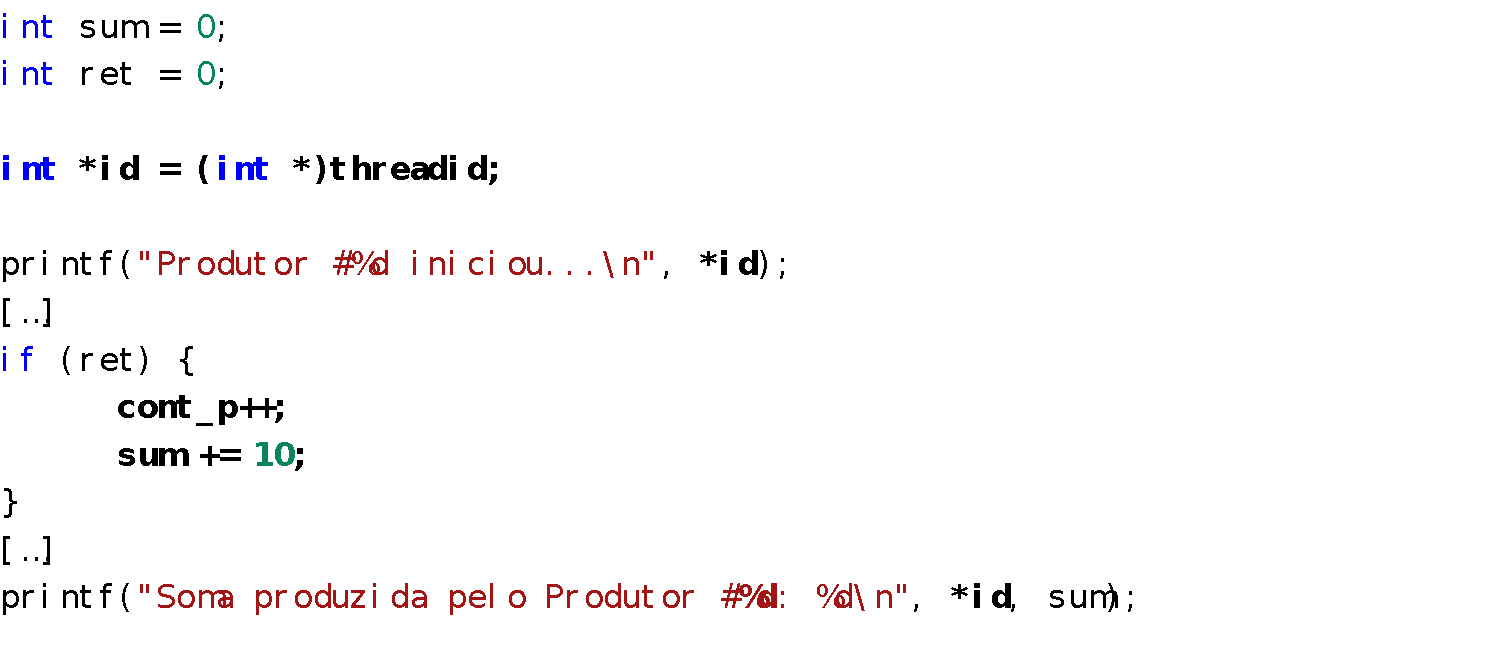
Utilizamos novamente os contadores absolutos de quanto produzimos e quanto consumimos, e desta vez, garantimos que sempre o número de consumidos seja *menor* que ao total de produzidos, visto que assim impedimos que haja consumo quando a quantidade de produzidos seja *igual a 0*, além de evitar que se consuma mais do que foi produzido, devido ao contador de consumidos sempre parar quando este é *igual* ao de produzidos, impedindo também que seja maior. O segundo erro, apenas corrige-se a aritmética, incrementando a variável rp.

**Trecho #4:**

****

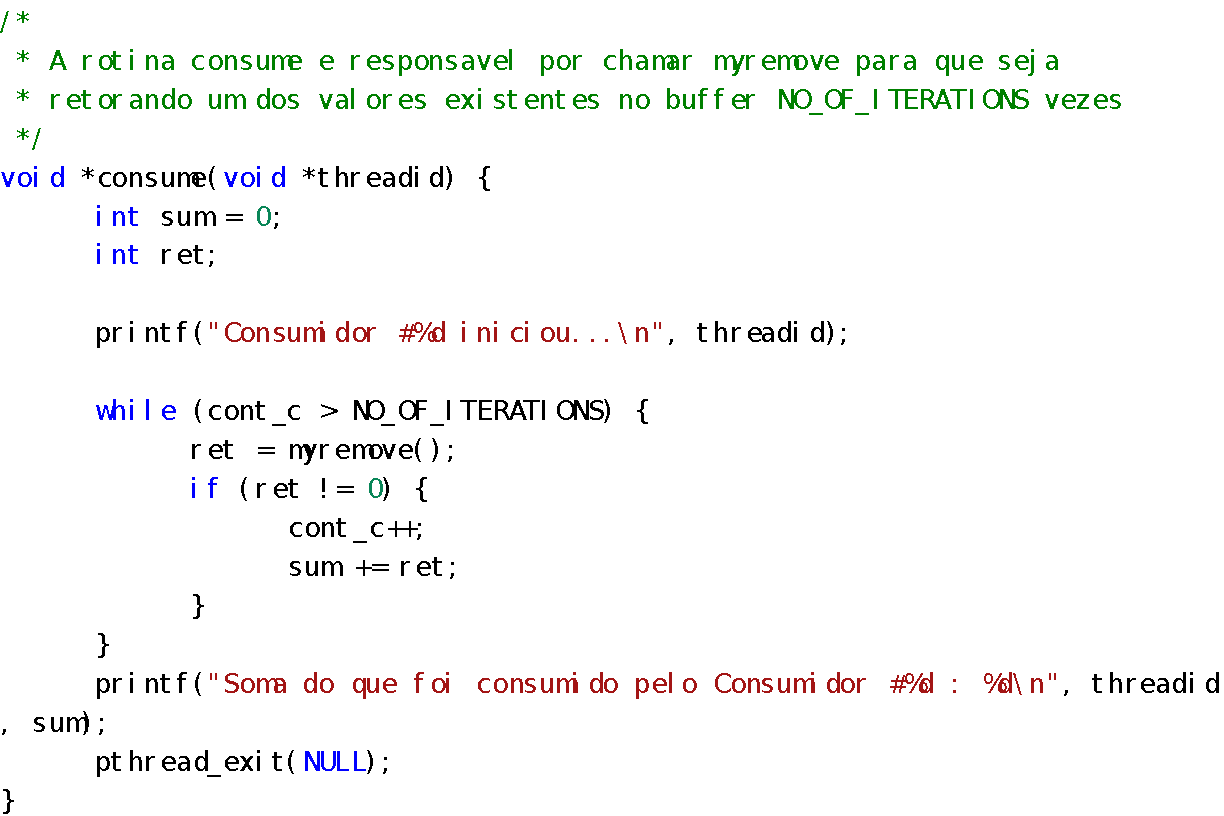
Encontramos alguns erros lógico-aritméticos, como a falta de inicialização em sum, e dois erros dentro da condicional if, um em cont\_p, que desejamos incrementar, e outro em sum, que não faz sentido subtrair se o próprio nome da variável significa soma. Por fim, temos um erro nos prints, com a especificação do formato errado e tratando o dado de maneira errada ao tentarmos printar um ponteiro void como um inteiro comum.

**Solução:**

****

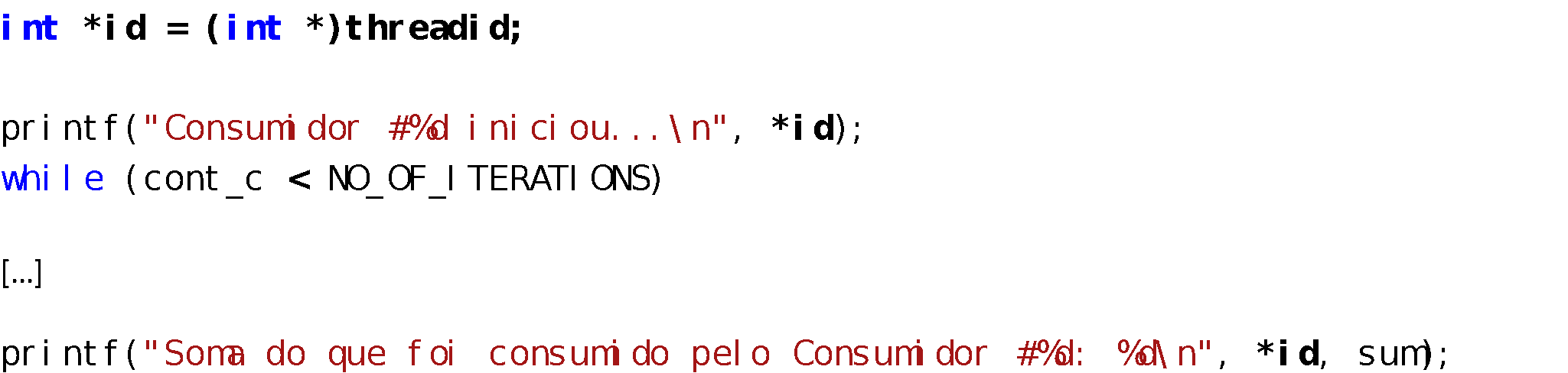
Apenas atribuição e correção dos operadores aritméticos, além disso, corrigimos os prints arrumando a especificação do tipo de print para “%d” e utilizando de uma variável ponteira do tipo inteiro “***id***” para recebermos o endereço do id do thread passado por parâmetro pela função main, assim, printamos o conteúdo desse endereço.

**Trecho #5:**



Mesmos erros do trecho anterior no que diz respeito aos prints, porém um erro distinto aparece: a condição do while faz com que nunca entremos neste, visto que nossa variável “***cont\_c***” é inicializada com 0 e só é incrementada dentro do while, o qual tendo como condição “0 > NO\_OF\_ITERATIONS”, sempre resultará em falso.

**Solução:**

****

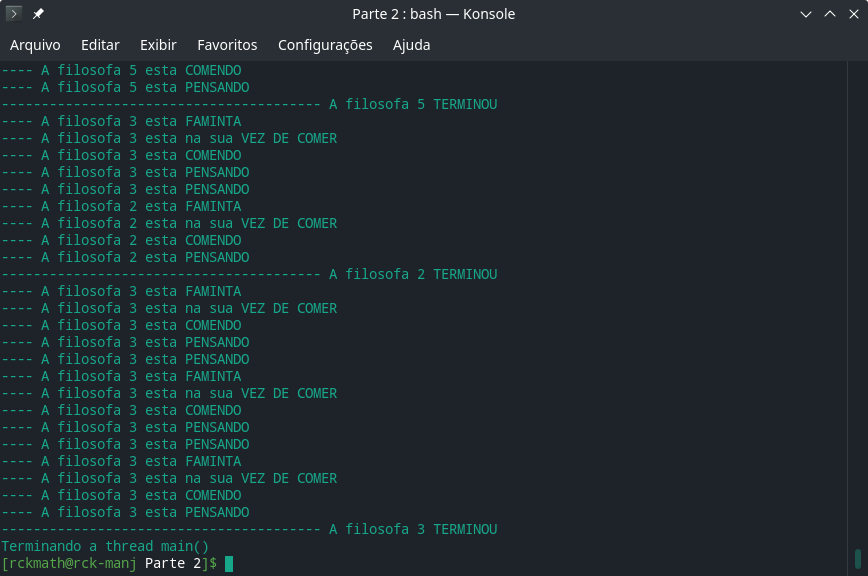
Inverte-se a condição do while para *menor*, visto que nosso propósito é que enquanto não se tenha consumido 100 (valor de NO\_OF\_ITERATIONS), continue tentando consumir. No mais, apenas as mesmas correções presentes no trecho anterior.

# RESULTADOS

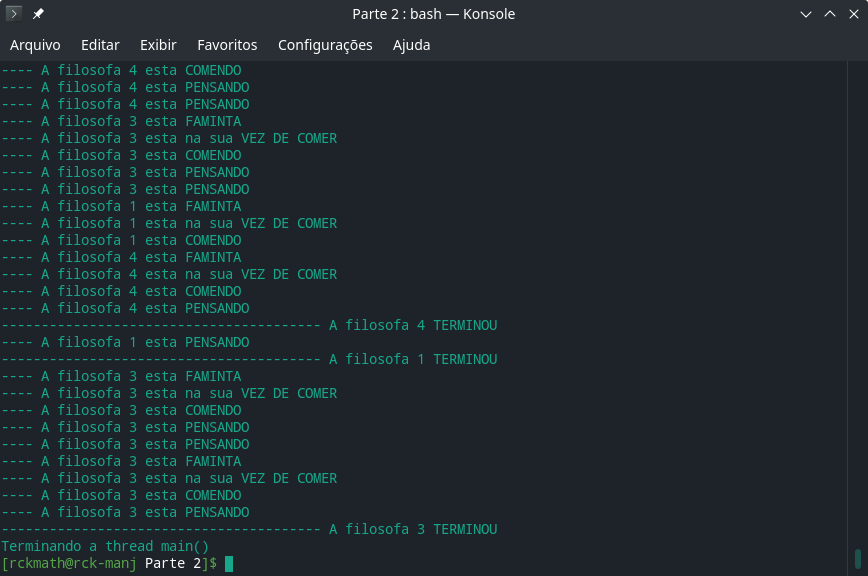
## REFERENTES AO PROGRAMA EXEMPLO



## REFERENTES AO PROGRAMA MODIFICADO



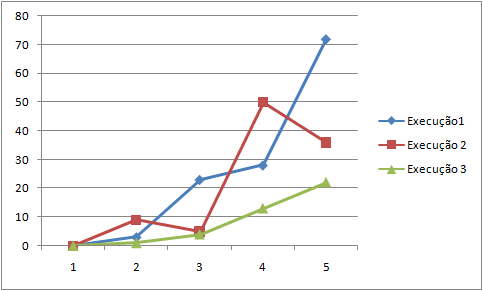
**Legenda: As Filósofas quando terminam de comer 365x sempre terminam “PENSANDO” como seu último estado.**



**Legenda: As Filósofas quando terminam de comer 365x sempre terminam “PENSANDO” como seu último estado.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Execução 1** | | **Execução 2** | | **Execução 3** | |
| **Ordem de término** | **Consumos após 1º término** | **Ordem de término** | **Consumos após 1º término** | **Ordem de término** | **Consumos após 1º término** |
| **thread 2** | **-** | **thread 5** | **-** | **thread 4** | **-** |
| **thread 3** | **3** | **thread q** | **9** | **thread 5** | **1** |
| **thread 1** | **23** | **thread 4** | **5** | **thread 1** | **4** |
| **thread 5** | **28** | **thread 2** | **50** | **thread 3** | **13** |
| **thread 4** | **72** | **thread 3** | **36** | **thread 2** | **22** |

**(Tabela 1: Mostra quantas vezes uma filósofa comeu após a primeira terminar, no caso da primeira execução a filósofa 2 foi a primeira a consumir 365 vezes, e após isso a filósofa 1 teve que comer 23 vezes para terminar)**

****

**(Gráfico 1: (Ordem de término) x (Consumo posterior ao 1 término) referente a tabela 1)**

# ANÁLISE DOS RESULTADOS

O Programa Exemplo apresentou resultados bastante intrigante, isso ocorreu por causa da variabilidade dos resultados, que tornou imprecisa a nossa análise e a conclusão a respeito do processo de execução do programa. Vale ressaltar que o Programa Exemplo havia problemas de race condition que não seriam tratados e talvez este tenha sido o maior motivo para a falta de conclusão sólida. Devido o tamanho do buffer no experimento, geralmente a execução do programa ocorria de maneira que apenas as threads que fossem escalonadas primeiro produzia e consumia todo o buffer. Por outro lado, ocorria instantes em que duas threads Produtoras ou Consumidoras eram iniciadas e a segunda thread escalonada interrompia a primeira sem que ela tivesse terminado, fazendo com que ambas produzissem ou consumissem até que a primeira alcançasse a quantidade máxima exigida pelo programa.

No caso das condições de corrida(se maior o buffer), as threads iniciadas executam em paralelo uma com as outras tornando os dados manipulados simultaneamente por elas inconsistentes. Isso tornou os resultados imprevisíveis por não garantir que as condicionais executadas pelas threads fossem executadas com os valores esperados,tornando seus comportamentos inesperados e não sendo possível garantir o propósito para o qual o programa foi designado, ressaltando que, todas essas races conditions poderiam ser solucionadas com o uso de semáforos.

O programa modificado, onde se deveria resolver o problema do jantar das filósofas, mostrou-se de alta complexidade. Abordagens e algoritmos simplistas apresentaram-se ineficazes para a solução do problema, resultando em inconveniências como Deadlocks e Starvations.

Cada filósofa era representada por uma thread, sendo necessário garantir total sincronização no processo de alternância entre pensar e comer.

Por meio do recurso do mutex foi possível garantir a exclusão mútua nas regiões críticas, mais precisamente nos trechos de pegar e devolver os garfos, e coordenar o uso e desuso dos garfos pelas filósofas, desbloqueando-as enquanto comiam e bloqueando-as quando pensavam.

Uma abordagem feita pelo grupo na análise dos resultados, foi tentar mensurar, quanto tempo as demais threads demoravam para terminar após o primeiro término. A maneira encontrada para quantificar essa relação, foi estabelecer a ordem em que as threads terminaram e contar quantas vezes cada filosofa comeu após a primeira terminar. Foi contabilizado os valores em três execuções (Tabela 1) e os resultados obtidos foram um tanto quanto intrigantes. Apesar da tentativa de promover um acesso simétrico aos garfos, certas filósofas prevaleceram sobre as outras, comendo mais do que pensavam, o que pode ser visualizado pelo comportamento inconstante no gráfico 1. Tal instabilidade pode ser atribuída a forma em que se deu o escalonamento das threads, mas ainda que essa afirmação esteja correta, cabe a necessidade de aperfeiçoar o código. Algo a ser também ponderado, é que a ordem no término da execução está relacionado em como as filósofas estão dispostas na mesa, sendo as adjacentes da primeira a terminar, as que tendem a ser as próximas,o que condiz com o algoritmo usado.

Todas filósofas comeram 365 vezes e apesar de certas filósofas predominar sobre outras, o algoritmo se mostrou eficiente diante da imprevisibilidade do fator escalonabilidade.

O programa exemplo e o problema do jantar das filósofas tem fins distintos. Porém, compartilham alguns conceitos como o uso de threads e da relevância de adotar mecanismos de sincronização. O programa modificado tinha a integridade de seus dados a mercê do escalonamento, resultando em inúmeras race conditions. Já no problema do jantar das filósofas, o mutex garantiu com eficiência a exclusão mútua, além de ser o ponto central no controle dos garfos, evidenciando a importância de gerenciar recursos compartilhados.

# CONCLUSÃO

O experimento 4 nos apresentou muitos desafios, primeiramente foi necessário aprender acerca da estrutura e linguagem das threads. Em seguida, foi preciso compreender seu funcionamento quando não há tratamento de race condition o que gerou diversos debates e dúvidas entre a equipe. Contudo, o maior desafio foi a implementação do problema Jantar das filósofas, resolver a lógica do problema e tentar resolver seus erros de sintaxe e lógica foi o que mais cobrou da equipe. O grupo procurava aplicar métodos alternativos, porém semelhantes ao que estava dando errado, e o problema funcionava e isso aumentava o questionamento do grupo em relação a execução e o meios implementados para a busca da solução do problema. Threads são mais rápidas que processos e pudemos perceber isso comparado a execução desse experimento em relação aos outros, também são mais práticas e maleáveis a alterações, mas é essencial a aplicação de Exclusão Mútua, seja para threads ou para criação de processos filhos, pois é muito importante evitar a inconsistência dos dados para uma execução correta do programa.