Pontifícia Universidade Católica de Campinas

Faculdade de Engenharia de Computação - FECOMP

Sistemas Operacionais A – Relatório Experimento 4

Beatriz Morelatto Lorente RA: 18071597

Cesar Marrote Manzano RA: 18051755

Fabricio Silva Cardoso RA: 18023481

Pedro Ignácio Trevisan RA: 18016568

Sumário

Introdução

O experimento realizado permitiu o entendimento do uso de threads na linguagem de programação C, como forma de se obter concorrência em um programa. O experimento foi divido em duas tarefas.

Na primeira tarefa, foi executado um programa exemplo, ilustrando o problema do produtor consumidor. Havia 10 threads produtoras e 10 consumidoras, que compartilham um buffer, percorrido de maneira circular. As threads produtoras armazenam o valor 10 no buffer, e as consumidoras retiram esse valor. O problema dessa tarefa está em como manipulamos os acessos a esse buffer, de modo que as condições de corrida ocorram o menor número de vezes possível.

Na segunda tarefa, é proposto a solução do problema do jantar das filósofas (adaptação do problema clássico do jantar dos filósofos). No problema, cada filósofa é uma thread, e cada uma irá comer no máximo 365 vezes. No lugar do estado de “pensando”, foi programado uma espera de 25 microssegundos. A tarefa foi importante para observarmos o uso do mutex, para garantir exclusão mútua.

Apresentação dos erros do programa exemplo e suas soluções

Ao compilar o programa exemplo foi possível perceber alguns erros de sintáxe e lógica do programa. Os problemas estão listados abaixo, seguidos de suas soluções (as correções estão destacadas em negrito e itálico).

Problema 1

#include <pthread.h>

#include <stdio.h>

Problema corrigido:

Falta algumas bibliotecas para a compilação do programa.

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

Problema 2 e 3 (na função main)

tc = pthread_create(&consuers[i], NULL, consume, (void *) i + 1);

tc = pthread_create(&consuers[i], NULL, produce, (void *) i + 1);

Problema corrigido:

Era necessário copiar o valor de i + 1 para um vetor. Assim iniciamos declarando dois vetores, um para cada criação da thread.

int temp1[NUM_THREADS], temp2[NUM_THREADS];

Depois jogamos o valor de i + 1 para a posição correspondente e passamos a posição do vetor como parâmetro da função.

temp1[i] = i + 1;
tc = pthread_create(&consuers[i], NULL, consume, (void *)&temp1[i]);

temp2[i] = i + 1;
tc = pthread_create(&consuers[i], NULL, produce, (void *)&temp2[i]);

Desse jeito foi possível eliminar o ‘warning’ existente ao compilar o programa exemplo.

O problema 2 consiste na falta do uso da função ‘pthread_join()’, fazendo com que a função main só termine a sua execução depois que todas as threads produtoras e consumidoras terminarem.

```
for(i = 0; i < NUM_THREADS; i++){
        pthread_join(producers[i], NULL);
        pthread_join(consumers[i], NULL);
}
```

Problema 4 e 5 (na função consume)

printf("Soma do que foi consumido pelo Consumidor #%d : %d\n", threadid, sum);

while(cont_c > NO_OF_ITERATIONS)

Problemas Corrigidos:

No primeiro problema a variável 'threadid', estava sendo passada como parâmetro para a função, porém não estava sendo convertida em inteiro.

int *t_id = threadid;
printf("Soma do que foi consumido pelo Consumidor #%d : %d\n", *t_id, sum);

Para o segundo problema foi necessário trocar apenas o sinal de '>' para '<'.

while(cont_c < NO_OF_ITERATIONS)

Problema 6 ,7 e 8 (na função produce)

printf("Soma do que foi consumido pelo Consumidor #%d : %d\n", threadid, sum);

```
if(ret){
        cont_p+-;
        sum - = 10;
}
```

Problemas corrigidos:

A correção do problema 6 é similar ao do problema 4.

int *t_id = threadid;
printf("Soma do que foi consumido pelo Consumidor #%d : %d\n", *t_id, sum);

Para a correção do problema 7 e 8, foi alterado o sinal de '-' para o sinal de '+'.

```
if(ret){
        cont_p++;
        sum + = 10;
}
```

Problema 9 (na função myremove)

```
if(wp != rp) {
        //Código
}
```

Problema Corrigido:

A verificação para o buffer não estar vazio, não está correta. Por se tratar de um buffer circular, wp e rp podem estar na mesma posiçao, porém há elementos para serem consumidos. Para solucionar, usamos os contadores de produção e consumo (cont_p e cont_c, respectivamente) existentes no programa, para controlarmos o consumo no buffer.

```
int delta = (cont_c – cont_p);
if(SIZEOFBUFFER > delta) {
        //Código
}
```

Problema 10 (na função myadd)

```
if( (rp != (wp + 1)) && (rp + SIZEOFBUFFER – 1 != wp) ){
        //Código
}
```

Problema Corrigido:

Assim como no problema 9, a verificação para o buffer estar cheio não está correta. Novamente, por se tratar de um buffer circular, as condições não são válidas. A solução foi feita de maneira parecida, ao problema 9.

```
int delta = (cont_p – cont_c);
if(SIZEOFBUFFER > delta) {
        //Código
}
```

Respostas das perguntas

Perguntas do relatório

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

Resposta: Diferentemente dos processos, se caso ocorra algum problema um uma thread, todas as outras irão parar de funcionar. Portanto, por mais que as threads ofereçam um parelelismo maior do que os processos, elas possuem uma robustez menor.

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

Resposta:

Processos	Threads
Mais robustez	Menos robustez
Não compartilha recursos com outros processos	Compartilha recursos com outras threads
Troca de contexto mais lenta	Troca de contexto rápido

Pergunta 3: O que é a área de heap?

Resposta: É um espaço reservado para variáveis e dados criados durante a execução do programa (runtime).

Pergunta 4: Quais são as funções do dispatcher?

Resposta: 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?

Resposta: A memória cache é um tipo de memória ultra rápida que armazena os dados e instruções mais utilizadas pelo processador, permitindo que estas sejam acessadas rapidamente.

Perguntas do programa

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

Resposta: Pois a função ‘myadd()’ é responsável por validar o parâmetro passado, no caso do programa, o 10. Esse controle é feito a partir do retorno da função, sendo 1 para quando deu certo e 0 para quando deu errado. Se o valor de ret for 1 o programa entra na função, não sendo necessário nenhuma comparação.

Pergunta 2: Porque não há necessidade de um cast?

Resposta: Pois o retorno da função é do tipo inteiro (0 ou 1), não sendo necessário indicar para a variável um novo tipo.

Pergunta 3: Para que serve cada um dos argumentos usados com pthread_create?

Resposta:

• O primeiro parâmetro (&consumers[i]) é usado para guardar as informações sobre a thread criada;

• O segundo parâmetro (NULL) é usado para indicar algumas especificações sobre a thread criada. Quando definido como NULL a thread é criada com os atributos padrões definidos pelo sistema;

• O terceiro parâmetro (consume) é a função que a thread irá executar;

• O quarto parâmetro ((void *) i+1) são os argumentos de início da thread.

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()?

Resposta: Se a thread responsável por criar outros threads for finalizada, as threads criadas também serão finalizadas. Isso também ocorre caso ocorra um erro em alguma thread, assim todas as outras irão dar erro e consequentemente o programa irá finalizar sua execução. Pthread_exit() é responsável por liberar todos os recursos ligados a thread finalizada.

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

Resposta: Se o término ocorre pelo exit(), a thread criadora terá um valor de retorno que será passado para as outras threads que ela mesmo criou, não gerando erro e esperando o término das outras threads.

Resultados da execução do programa exemplo

Abaixo, são mostrados alguns prints dos resultados do programa exemplo. Foram feitos dois testes diferentes. O primeiro consiste em iniciar (zerar) o valor dos contadores de produção e consumo apenas quando estas eram declaradas. Já o segundo teste zerava o valor do contador de produção na função “produce” e o de consumo na função “consume”.

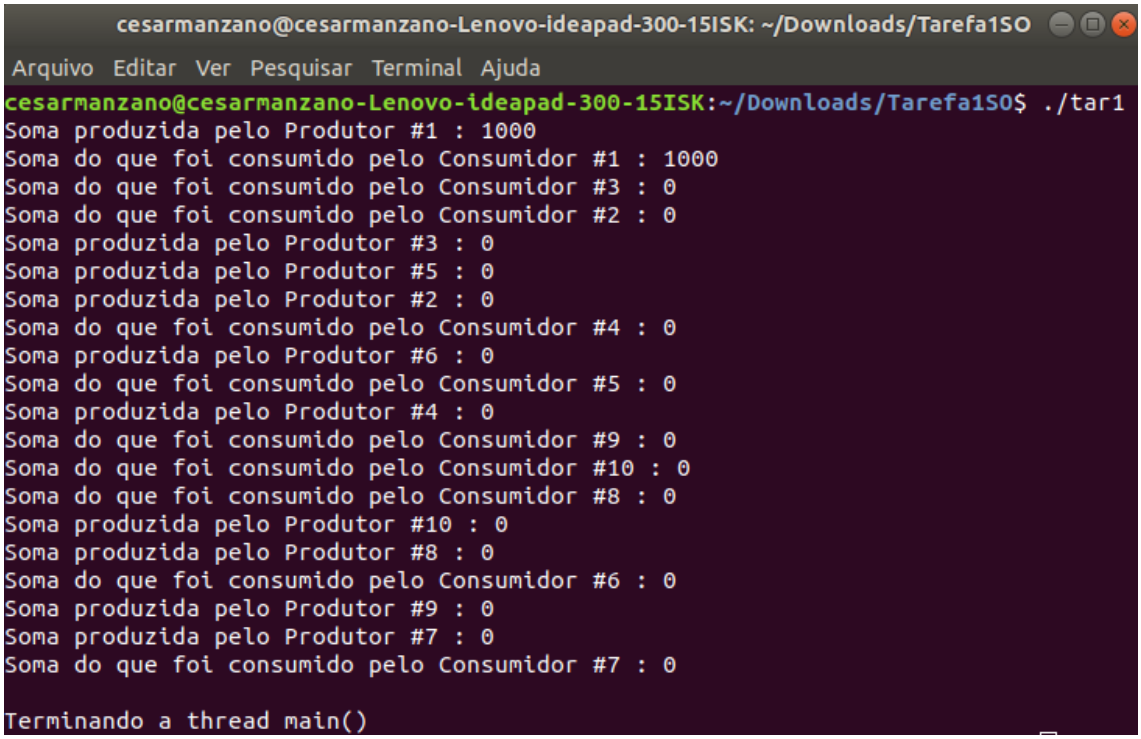

Figura 1 - Resultado do primeiro teste do programa exemplo

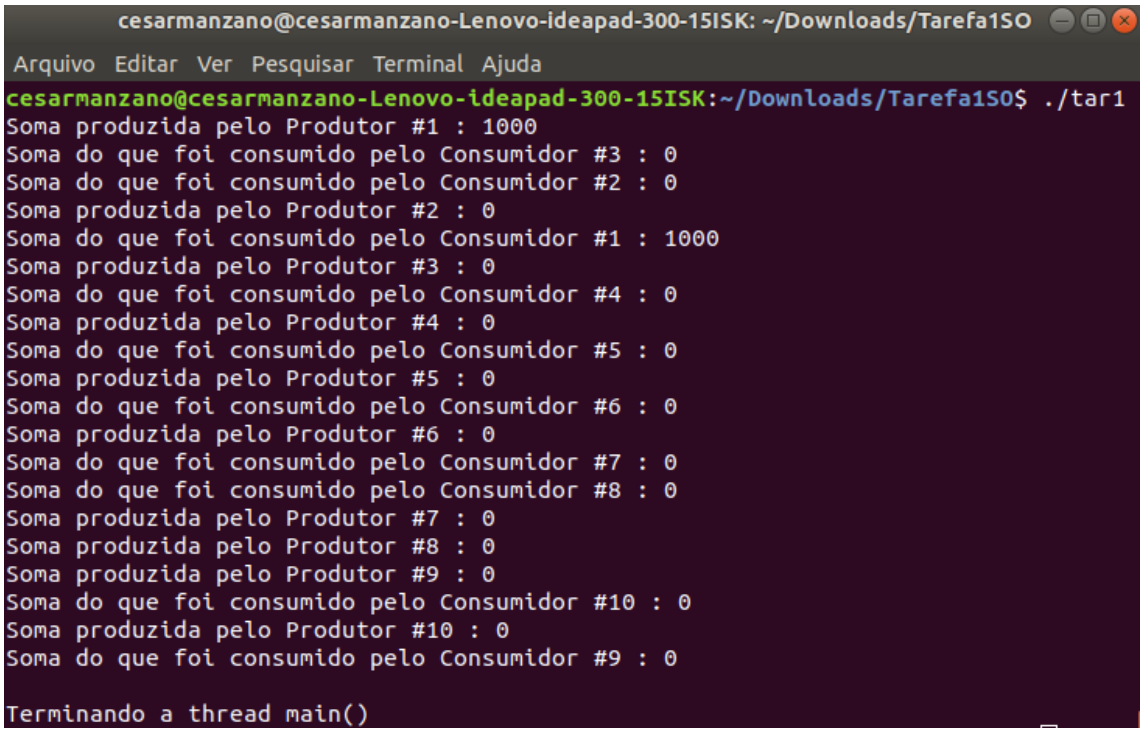

Figura 2 - Resultado do primeiro teste do programa exemplo

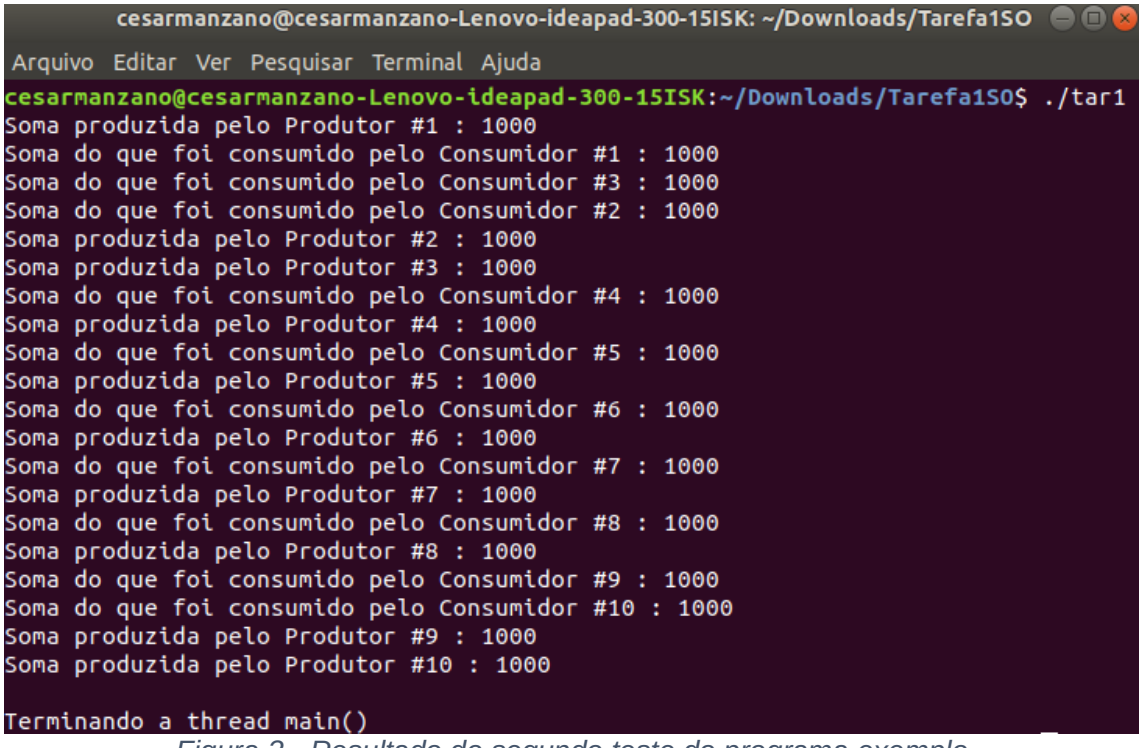

Figura 3 - Resultado do segundo teste do programa exemplo

Figura 4 - Resultado do segundo teste do programa exemplo

Também foram feitos teste para mostrar quantas vezes um produtor deixou de consumir e um consumidor deixou de consumir. Esses testes foram feitos zerando os contadores globalmente, quando foram declarados.

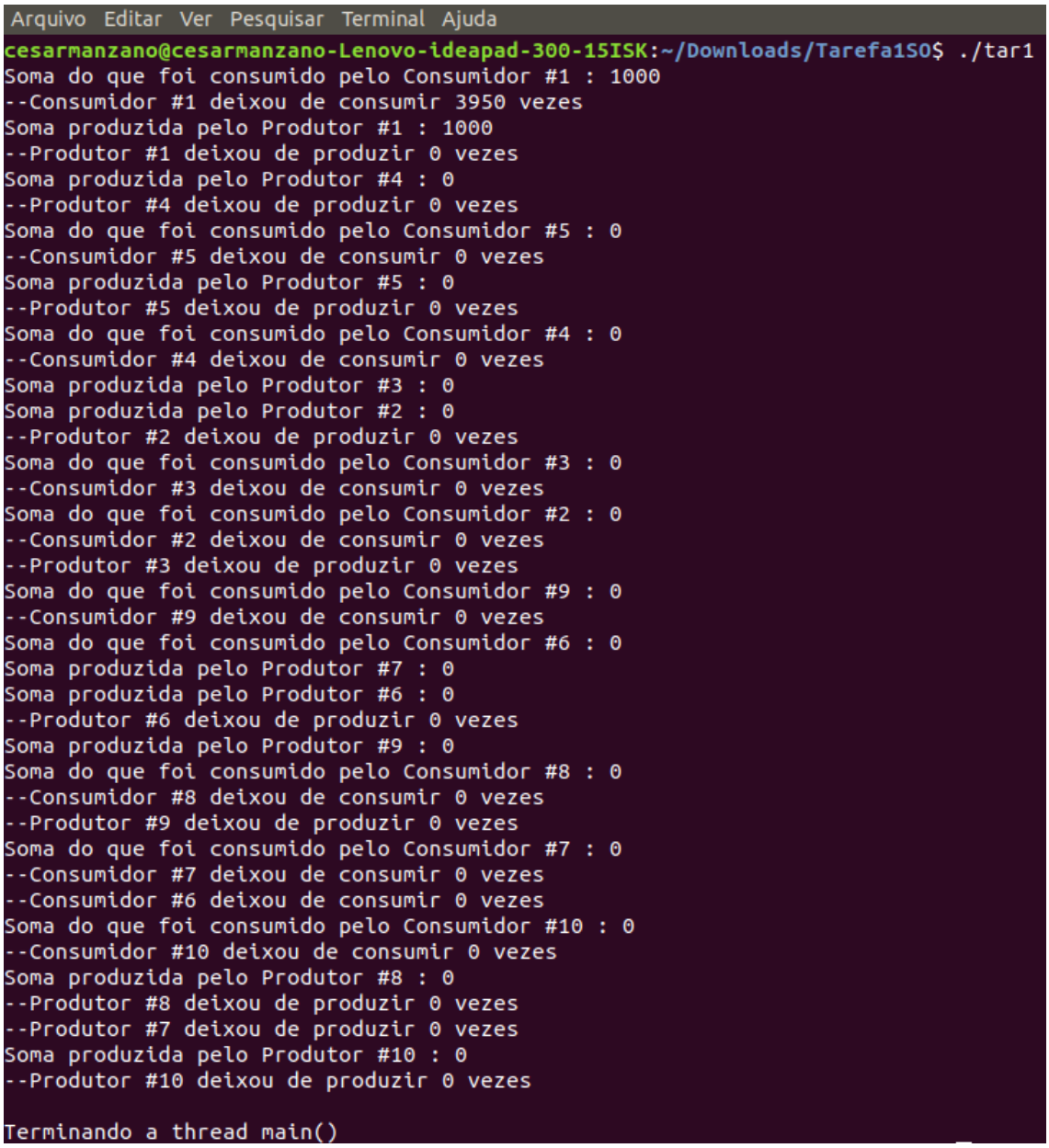

Figura 5 - Resultado mostrando quantas vezes um produtor não consumiu e um consumidor não consumiu

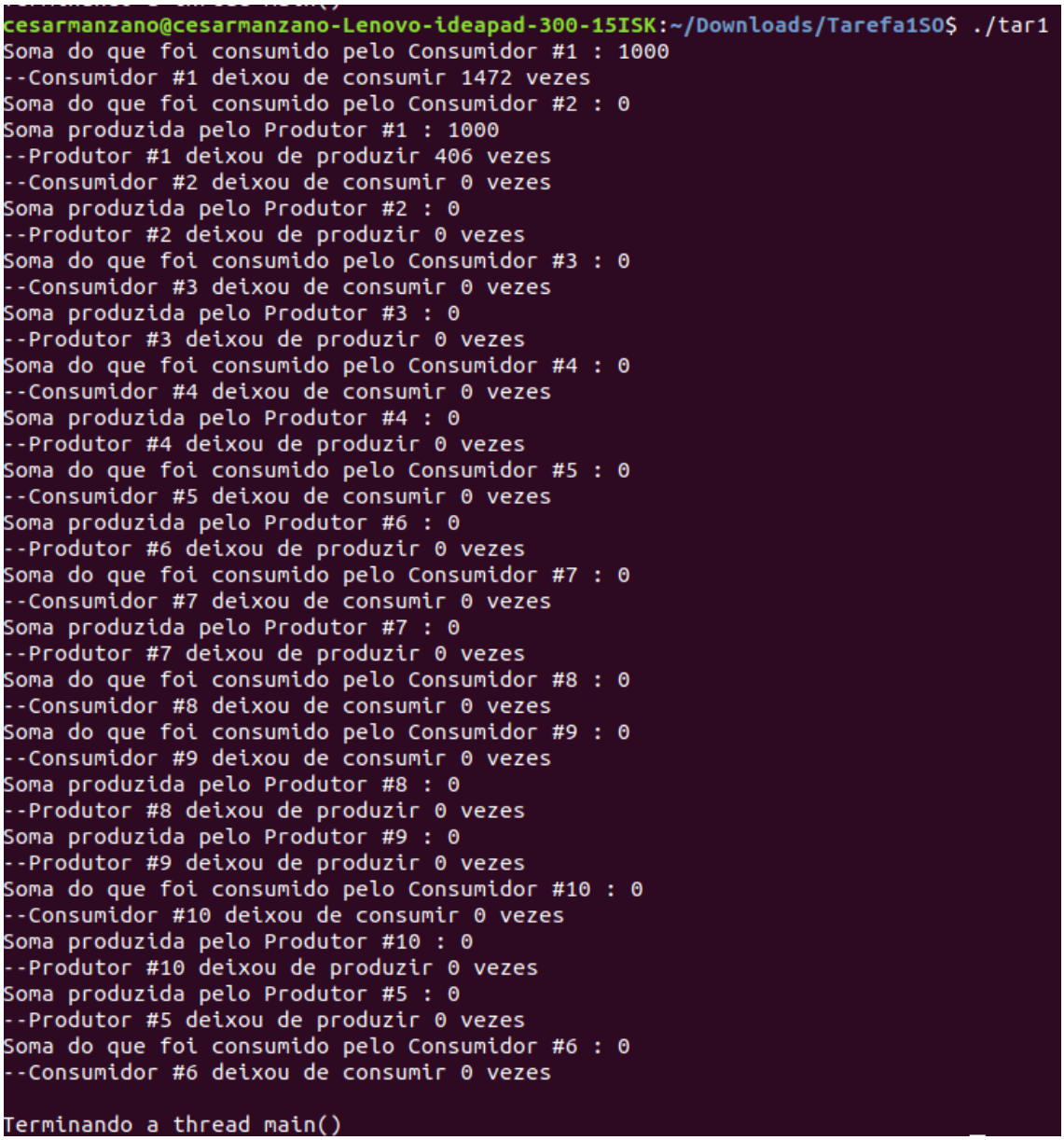

Figura 6 - Resultado mostrando quantas vezes um produtor não consumiu e um consumidor não consumiu

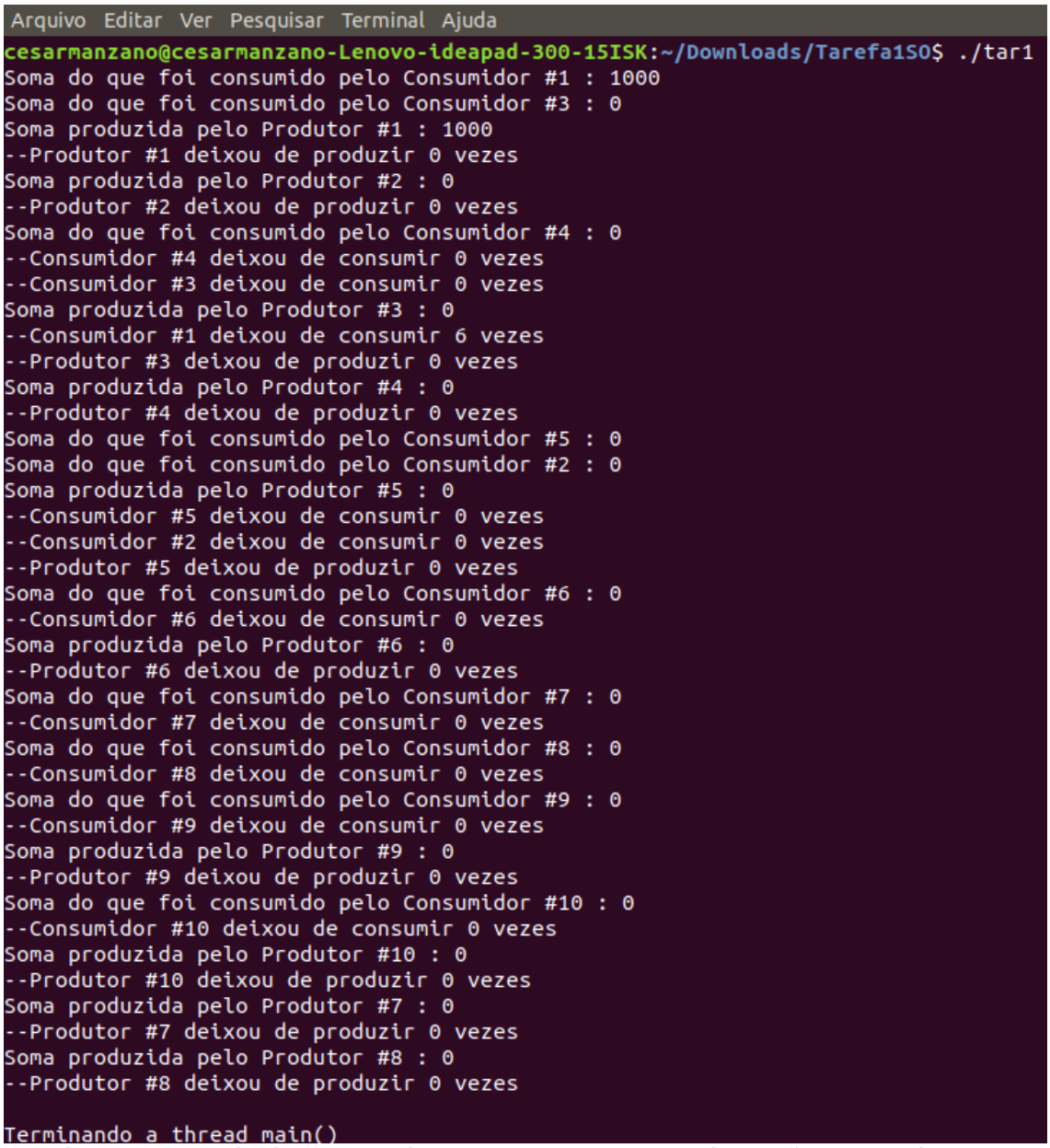

Figura 7 - Resultado mostrando quantas vezes um produtor não consumiu e um consumidor não consumiu

Resultados da execução do programa modificado

Abaixo, são mostrados alguns prints dos resultados do programa modificado.

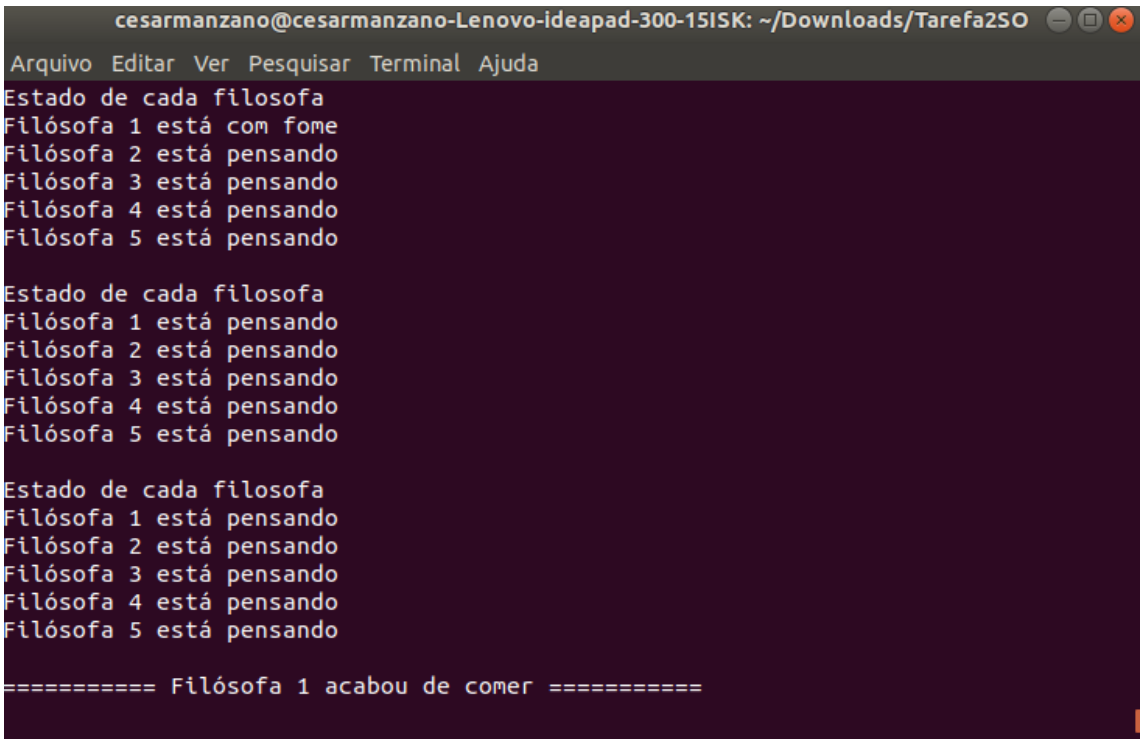

Figura 8 - Resultado do programa modificado, no qual a filósofa 1 acabou de comer

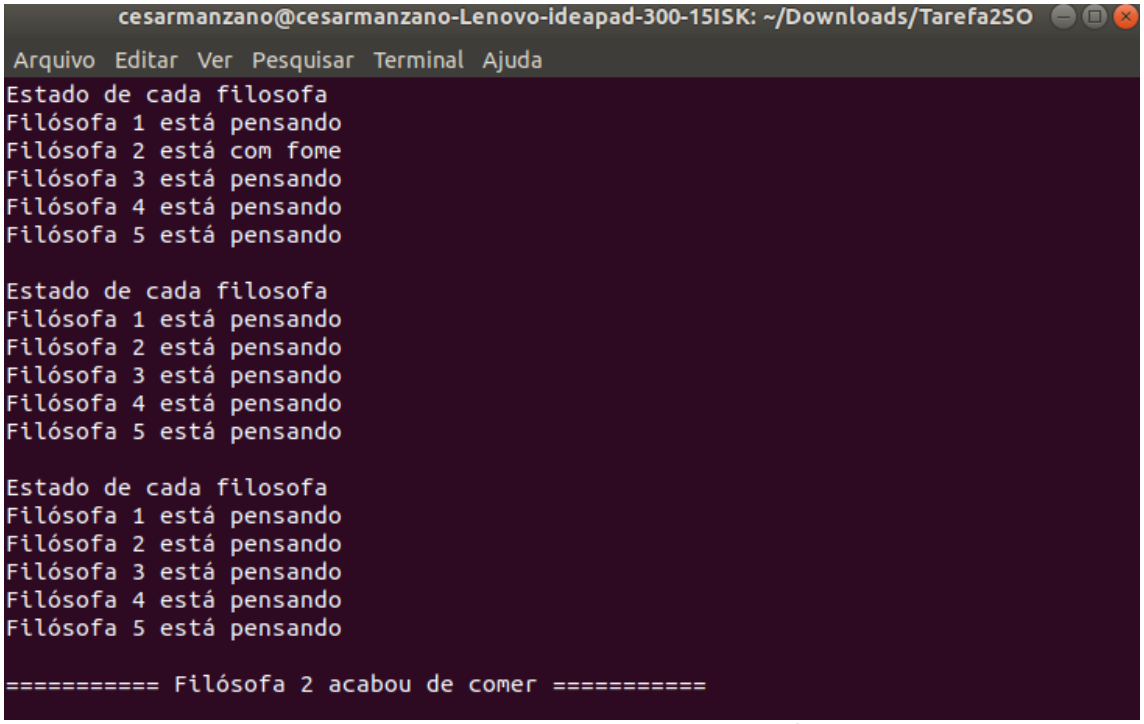

Figura 9 - Resultado do programa modificado, no qual a filósofa 2 acabou de comer

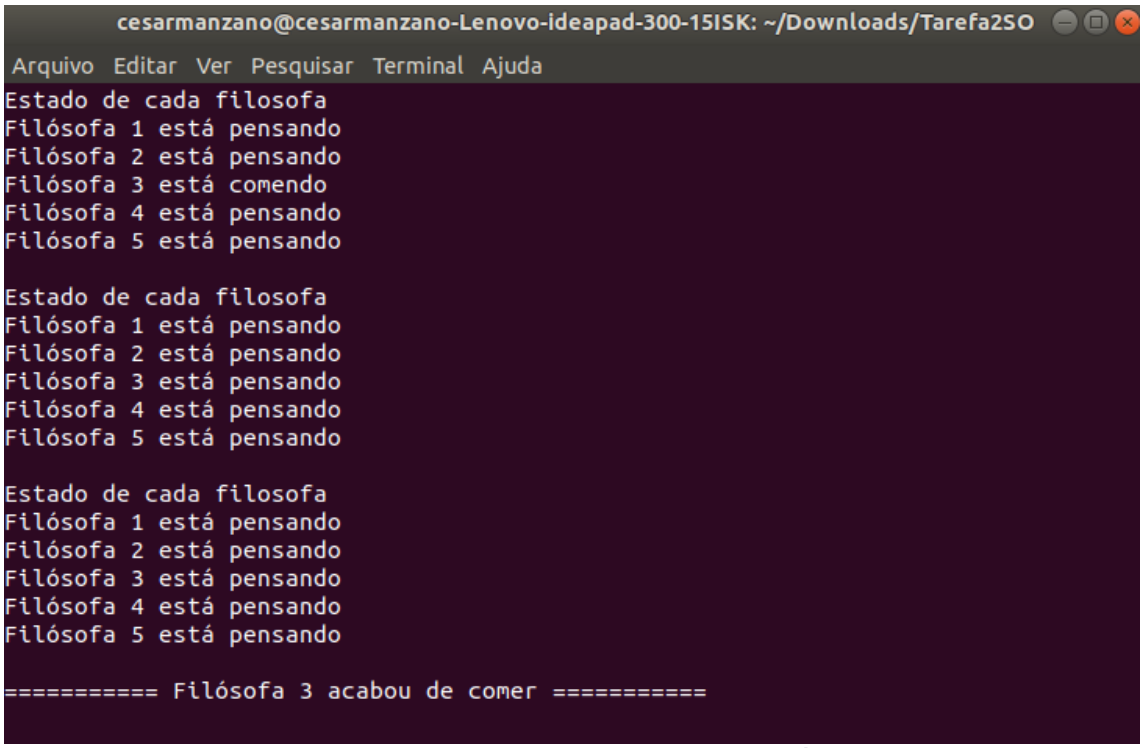

Figura 10 - Resultado do programa modificado, no qual a filósofa 3 acabou de comer

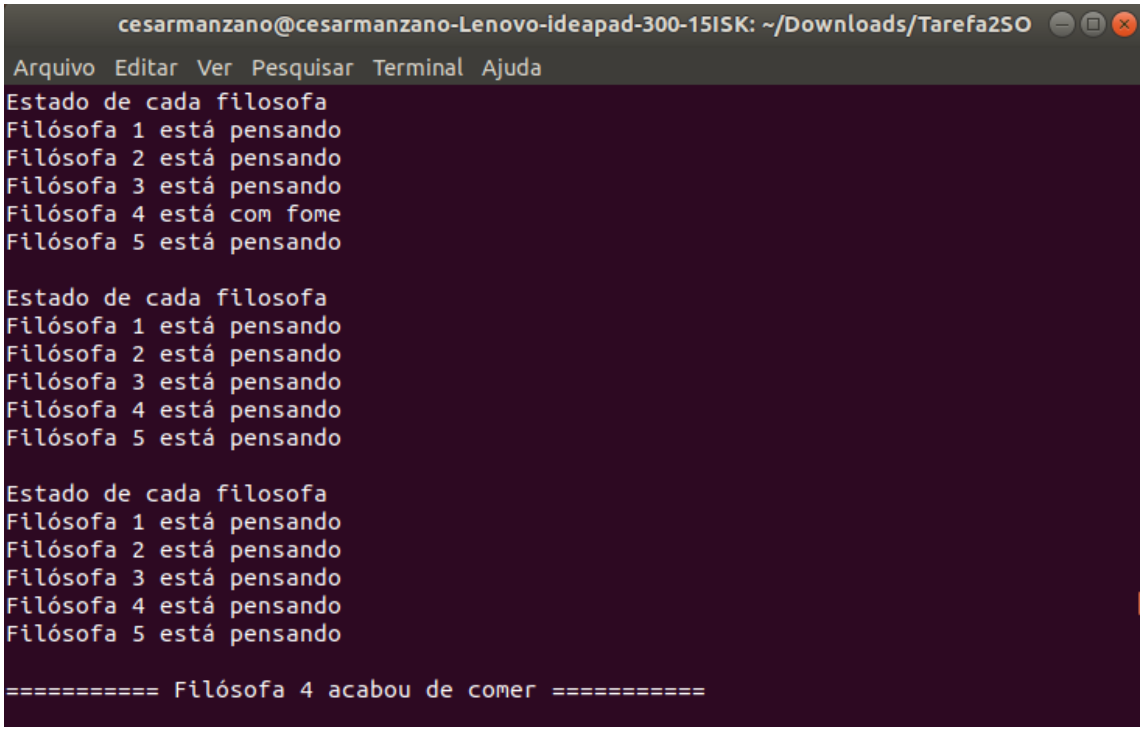

Figura 11 - Resultado do programa modificado, no qual a filósofa 4 acabou de comer

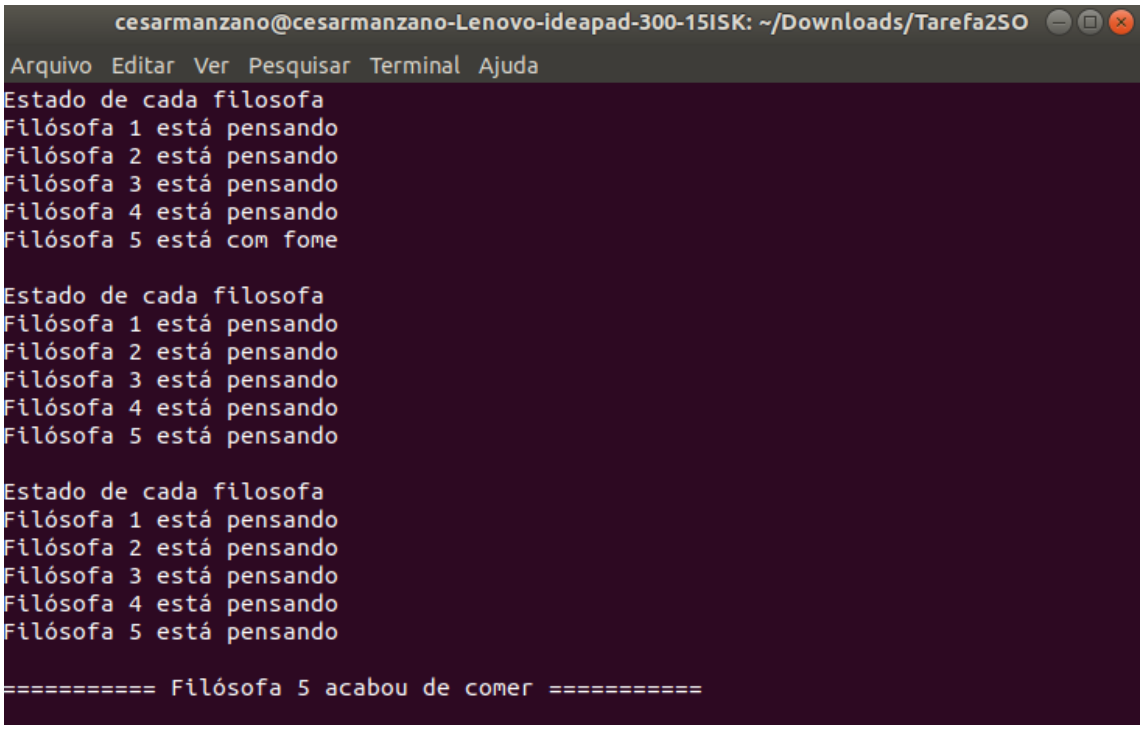

Figura 12 - Resultado do programa modificado, no qual a filósofa 5 acabou de comer

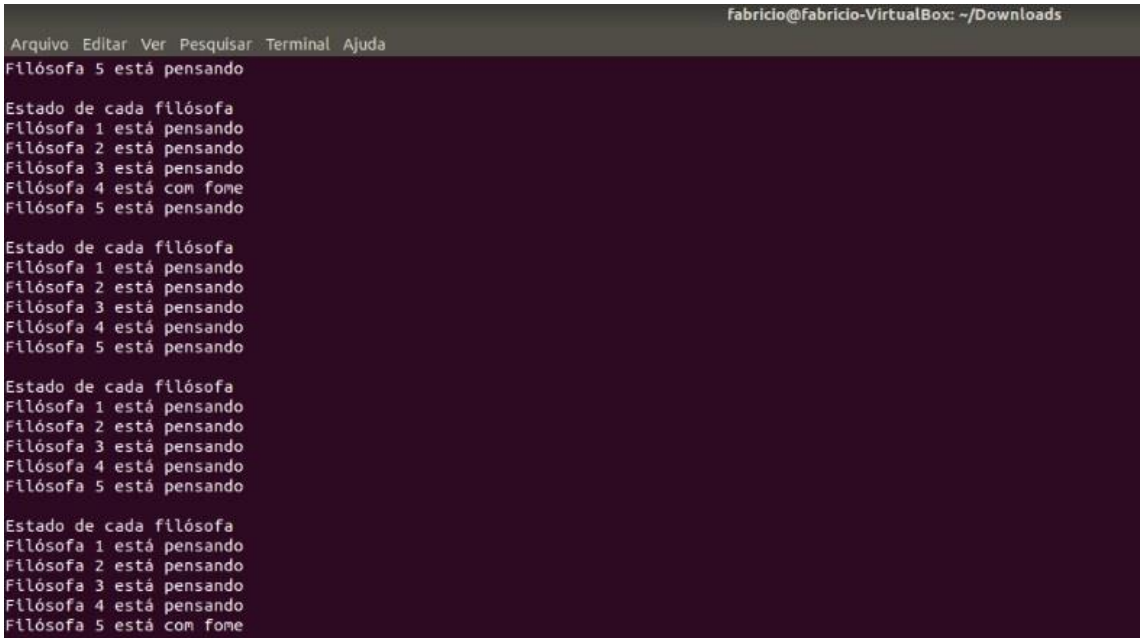

Figura 13 - Resultado do programa modificado, no qual percebe-se que apenas uma filósofa pode comer ao mesmo tempo

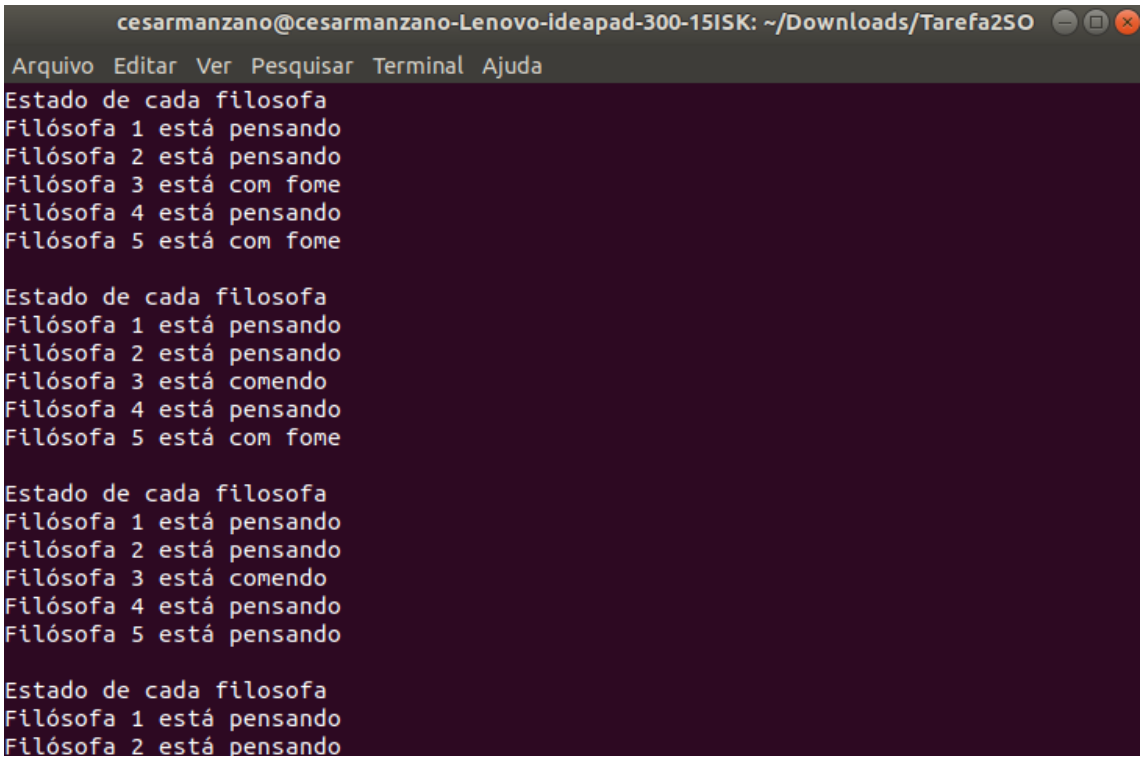

Figura 14 - Resultado do programa modificado, no qual percebe-se que apenas uma filósofa pode comer ao mesmo tempo

Análise dos resultados

Programa Exemplo

Para iniciar a análise do programa exemplo, vamos verificar o primeiro teste feito, no qual os contadores não são zerados localmente, ou seja, nas funções. Vamos pegar a imagem abaixo para podermos analisá-la.

Figura 15 - Resultado do primeiro teste do programa exemplo

Vale lembrar que cada thread repetia o acesso ao buffer 100 vezes, independentemente de ser produtora ou consumidora, colocando ou consumindo o valor 10 em cada posição, e por isso o valor máximo que elas poderiam produzir/consumir era 1000. Como cada contador era incrementado quando produzíamos ou consumíamos algo, as primeiras thread acessariam o vetor 100 vezes, já estourando o limite de acessos ao buffer. Como podemos ver na imagem isso ocorreu, visto que a primeira thread consumidora e a primeira produtora acessaram o buffer 100 vezes e a soma resultou no valor 1000. Uma vez que o contador não era reiniciado ao entrar nas funções “consume” e “produce”, era esperado que apenas as primeiras threads fizessem o acesso ao buffer.

Agora vamos analisar o resultado quando os contadores eram reiniciados, ou seja, zerados, no início das funções “consume” e “produce”. Vamos tomar a imagem abaixo como parâmetro para a análise.

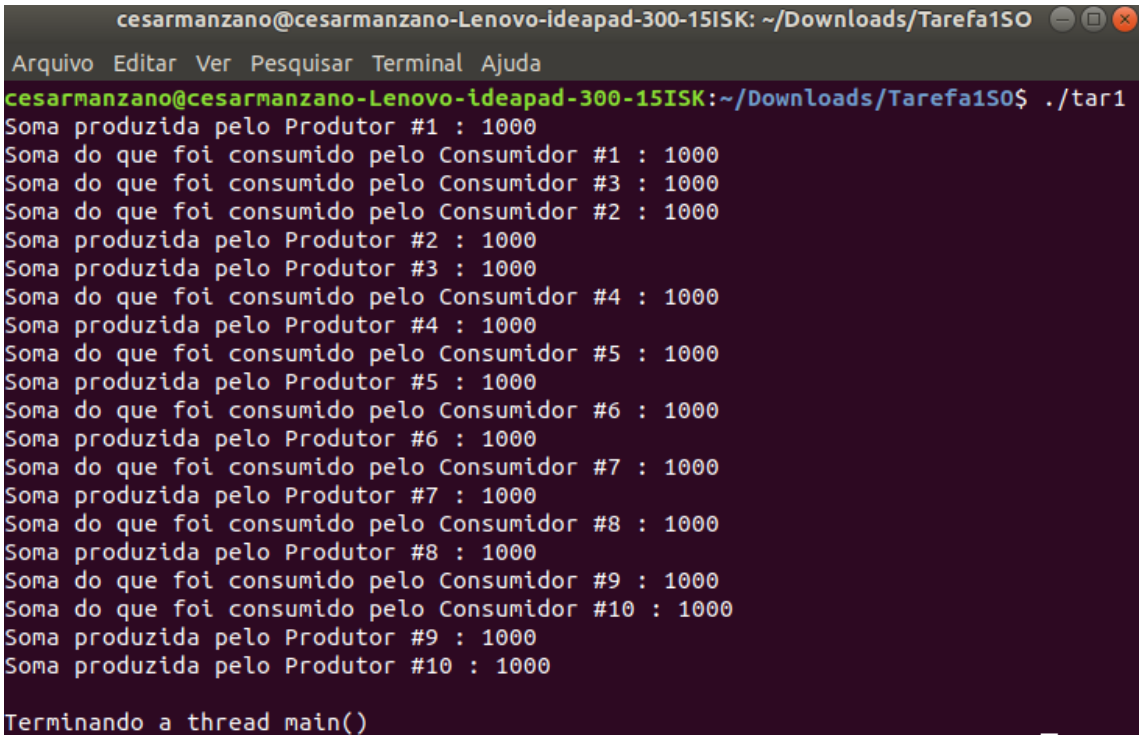

Figura 16 - Resultado do primeiro teste do programa exemplo

O resultado foi coerente com o esperado, uma vez que todas as threads deveriam produzir ou consumir valores do buffer 100 vezes, resultando em uma soma total no valor de 1000.

Por fim vamos analisar os resultados de quantas vezes uma thread produtora deixa de produzir ou uma thread consumidora deixa de consumir. Para a análise foi usado o primeiro cenário, no qual os contadores eram zerados apenas quando eram declarados. Para a análise foi feita uma tabela, mostrando os valores esperados de produção e consumo das threads e os valores reais.

Rodada	Thread	Valor esperado	Valor obtido
1	1	0	3950
	2	0	0
	3	0	0
	4	0	0
	5	0	0
2	1	0	406
	2	0	0
	3	0	0
	4	0	0
	5	0	0
3	1	0	0
	2	0	0
	3	0	0
	4	0	0
	5	0	0

Tabela 1 - Tabela mostrando o valor esperado de vezes que uma thread produtora não produz x valor obtido

Rodada	Thread	Valor esperado	Valor obtido
1	1	0	0
	2	0	0
	3	0	0
	4	0	0
	5	0	0
2	1	0	1472
	2	0	0
	3	0	0
	4	0	0
	5	0	0
3	1	0	6
	2	0	0
	3	0	0
	4	0	0
	5	0	0

Tabela 2 - Tabela mostrando o valor esperado de vezes que uma thread consumidora não consome x valor obtido

O valor esperado de 0 erros para cada thread, se deve à expectativa de termos um cenário ideal e de exclusão mútua entre as threads. Nesse cálculo esperado também inclui o fato de que a primeira thread produtora seria a primeira a ser executada seguida da primeira thread consumidora e assim por diante.

Como o algoritmo desenvolvido não se preocupava com as condições de corrida (race conditions), que poderiam existir no programa, os resultados esperados não foram obtidos.Outro fator que pode ter influenciado no resultado, é o fato de não sabermos como o SO irá escalonar os processos.. Mesmo não sabendo a ordem que as threads foram escalonadas, percebe-se que apenas as primeiras threads obtiveram erros, uma vez que estas foram as únicas a acessarem o buffer. Esse fato pode ter sido apenas uma coincidência, e que poderíamos ter mais certeza se fossem feitos milhares de testes.

Programa Modificado

O programa modificado era uma adaptação do problema do jantar dos filósofos. É um problema clássico de sincronização, proposto por Dijkstra em 1965. Consiste em uma mesa redonda de jantar, na qual há 5 filósofos sentados, com um prato de comida à sua frente e um garfo ao seu lado. Porém, para o filósofo poder comer, é necessário que este esteja segurando dois garfos ao mesmo tempo. Todos os filósofos alternam entre dois estados: pensar e comer. Quando um filósofo para de pensar, fica com fome, e tenta pegar os garfos tanto à direita quanto à esquerda. Se essa tentativa for possível, o filósofo come e depois devolve os garfos para a mesa. O problema do algoritmo consiste no fato de que vários filósofos podem tentar comer ao mesmo tempo, causando problemas de travamento, portanto o algoritmo necessita ser desenvolvido de maneira que nenhum filósofo fique travado.

Na versão programada, nomeada o jantar das filósofas, possuía algumas diferenças do problema clássico, listados abaixo:

- As filósofas não ficavam eternamente pensando e comendo. Quando uma comia um total de 365 vezes, esta não poderia comer mais.
- O tempo de pensar de cada filósofa era fixo em 25 microssegundos.
- Não possuía tempo para comer.

Como podemos ver pelos resultados obtidos, o programa foi feito de maneira correta. A garantia de exclusão mútua e não ocorrência de deadlock, permitiu com que apenas uma filósofa comesse por vez, assim todas as filósofas pudessem terminar de comer 365 vezes.

Como cada filósofa era uma thread, a exclusão mútua foi realizada utilizando-se as funções “pthread_mutex_lock” e “pthread_mutex_unlock”. As regiões nas quais poderiam ocorrer race conditions, ocorriam quando as filósofas verificavam se poderiam comer. Esse fato ocorre duas vezes ao longo do programa: quando a filósofa pega um garfo da mesa e quando a mesma colocava os garfos de volta na mesa.

Conclusão

Através desse experimento foi possível observar o funcionamento de threads e como se beneficiar do paralelismo gerado. Com a resolução do jantar dos filósofos, também foi possível entender melhor os conceitos de exclusão mútua, deadlock, starvation entre outros.

Também foi possível compreender o uso de algumas funções que manipulavam threads e mutex, e estas são mostradas a seguir: pthread_create(), pthread_join(), pthread_exit(), pthread_mutex_init(), pthread_mutex_destroy(), pthread_mutex_lock() e pthread_mutex_unlock(). Essas funções foram essenciais para que as tarefas fossem completadas com sucesso.