Experimento #4 Sistemas Operacionais A

PUC
CAMPINAS
PONTIFÍCIA UNIVERSIDADE CATÓLICA

Adriano de Oliveira Munin	17066960
Fábio Seiji Irokawa	17057720
Lucas Rodrigues Coutinho	17776501
Marcos Lelis de F. Oliveira	16248387
Paulo M. Birocchi	16148363

Introdução

O objetivo do experimento é a correção e a compreensão do programa apresentado que utiliza as funções que controlam o funcionamento dos threads. Neste caso os threads são produzidos e consumidos através de um buffer circular com dois apontadores, o ‘writing pointer’ (wp) e o ‘reading pointer’ (rp) ou o apontador de escrita e leitura respectivamente, que apontarão a posição de ambos no buffer.
No programa foi necessário a elaboração de condições que possibilitam tanto a escrita e leitura do conteúdo no buffer, quanto também a interrupção do apontador de escrita quando chegar ao apontador de leitura, evitando a sobrescrita, pois a informação deve ser lida além de evitar a perda de dados.
O código do jantar dos filósofos consiste no uso da exclusão mútua, onde cada filósofo é considerado um thread e a ‘memória compartilhada’ entre os filósofos são os garfos que são a seção crítica da exclusão. Foi necessário o uso das funções de ‘pthread_mutex’ para que, caso um filósofo tenha em mãos os garfos, os outros em sua volta consequentemente não podem ter os dois garfos necessário para comer. Por isso o uso da exclusão mútua serve para ‘bloquear’ o garfo caso esteja em uso e desbloquear quando estiver disponível para outro filósofo usá-lo.
Este experimento conclusivamente teve como o objetivo o aprendizado e como usar as funções que permitem o tratamento de threads e o uso de exclusão mútua para evitar a *race condition* ou quando duas ou mais threads tentam acessar a mesma memória compartilhada simultaneamente.

Resultados das Execuções

A imagem abaixo mostra um dos testes feitos na primeira etapa do experimento 4, nela é possível ver a soma produzida e consumida por cada um dos threads, além de quantas vezes não foi possível produzir (Buffer cheio) e quantas não foi possível consumir (Buffer vazio).

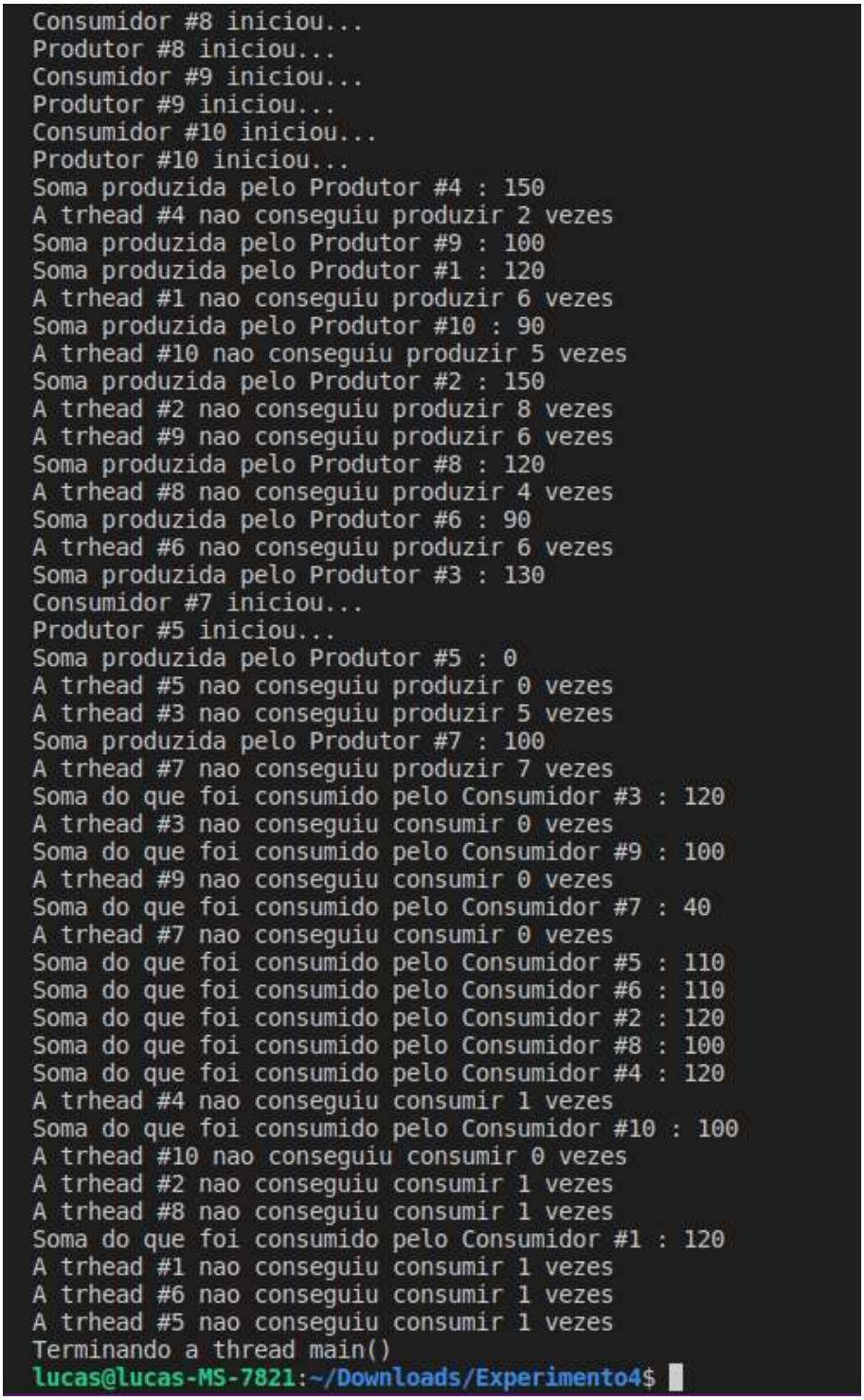

Figura 1- Teste primeira parte Experimento4

A seguir vemos os dados da imagem organizados um duas tabelas, sendo uma delas para os produtores e outra para os consumidores.

Tabela 1- Dados dos threads produtores

Thread produtora N	Soma do que foi produzido	N de vezes que não conseguiu produzir
1	120	6
2	150	8
3	130	5
4	150	2
5	0	0
6	90	6
7	100	7
8	120	4
9	100	6
10	90	5

Tabela 2-Dados dos threads consumidores

Thread Consumidora N	Soma do que foi consumido	N de vezes que não conseguiu consumir
1	120	1
2	120	1
3	120	0
4	120	1
5	110	1
6	110	1
7	40	0
8	100	1
9	100	0
10	100	0

Na imagem abaixo é possível ver a execução do programa, que demonstra o problema do jantar dos filósofos nas últimas instruções, quando os 5 filósofos estão prestes a atingir as 365 porções consumidas, que é quando o thread finaliza.

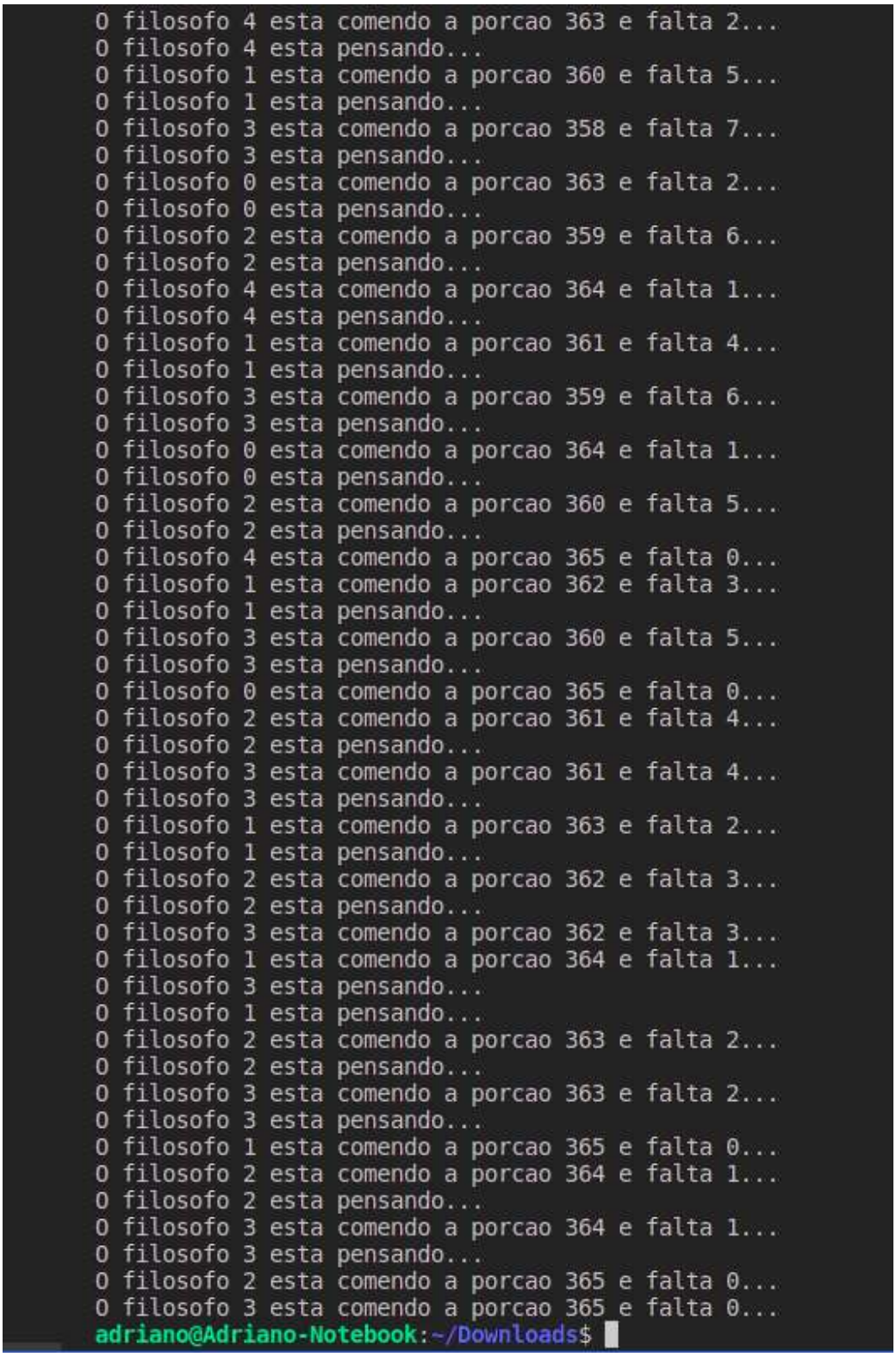

Figura 2- Teste do programa o jantar dos filósofos

Analise Dos Resultados

A tarefa 1 foi a parte mais trabalhosa do experimento, pois consistia em compilar e corrigir erros de logica e sintaxe de um programa que trabalhava com buffer circular. Foi tentada diversas abordagens para a resolução da lógica do programa, inclusive criamos um programa de buffer circular a parte para auxiliar no entendimento do problema, pois não queríamos criar novas variáveis globais, como um contador. Porém, a pesar de testarmos diversos algoritmos e pesquisar diversas informações na internet,
não conseguimos implementar o método que desejávamos, pois enfrentamos o problema de acabar gerando um “buraco” no buffer, com isso, implementamos a solução de utilizar um contador global que recebe dados tanto da função que insere dados, quanto da que remove para gerenciar possíveis conflitos
no buffer. Com está solução implementada, os resultados dos testes foram compatíveis com nossas expectativas.

A tarefa 2 consistia em elaborar uma solução para o problema do jantar dos filósofos, no qual cinco threads deveriam utilizar dois de cinco recursos compartilhados cada, sendo assim apenas 2 threads conseguem executar por vez, utilizando 4 recursos. A execução dos threads gera mensagens na tela informando quantas vezes já se passaram e faltam para acessar o recurso. O resultado que esperávamos de tal programa e de que ele gerasse as informações aleatoriamente, sendo a execução de cada thread e seus consumos dados randômicos. Os resultados obtidos foram compatíveis com os esperados, pois devido ao fato de threads serem processos concorrentes, não há uma ordem para a execução dos mesmos, sendo eles executados de acordo com a conveniência do sistema operacional.

Erros no Código do programa

***Global**

-Foi criada uma variável "elementos" inicializada com 0, que funciona como contador da quantidade de elementos no buffer, serve para verificar se o buffer está vazio ou não.

***Função myadd()**

-A condição que verifica se pode inserir um elemento no buffer foi alterada para "if(elementos < SIZEOFBUFFER)", onde verifica se o buffer está cheio.

***Função myremove()**

-A condição que verifica se pode removar um elemnto no buffer foi alterada para "if(elementos > 0)", onde verifica se o buffer está ou não vazio, se entrar na condição o contador "elementos" é decrementado.

- rp+- foi alterado para rp++;

***Função produce()**

-Foi criada uma variável contadora "buffCheio", inicialiazda com 0, para contar a quantidade de vezes que não foi possível inserir no buffer, para cada thread, já que o buffer estava cheio.
-sum -= alterado para sum +=

***Função consume()**

-Foi criada uma variável contadora "buffVazio", inicializada com 0, para contar a quantidade de vezes que não foi possível removar do buffer, para cada thread, já que o buffer estava vazio.

***Função main()**

-rp agora é inicializado com start -1;
-Um laço foi criado, executando o número de vezes de threads criadas, que faz com que a "thread mãe" aguarde as threads criadas terminarem, para depois terminar, através da função pthread_join();

Perguntas

Perguntas do Documento:

Pergunta 1: Explique por que a vantagem do uso de threads é condicional
R: Porque a troca de contexto é mais simples e rápida
Pergunta 2: Apresente um quadro comparativo com, pelo menos, três aspectos para processos e threads.

	Processo	Thread
Consumo de recursos	Alto	Baixo
Tempo de troca de contexto	Lento	Rápido
Memória compartilhada	Não possui	Compartilhada com outras threads

Pergunta 3: O que é a área de heap?
R: É uma área de alocação dinâmica de variáveis
Pergunta 4: Quais são as funções do dispatcher?
R: 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?
R: Uma memória onde armazena dados e instruções que a CPU pode precisar em breve.

Perguntas do Código:

Pergunta 1: Por que ret não está sendo comparado a nenhum valor?
R: Porque a função myadd() retorna 0 ou 1, e o if compara, se o retorno for verdadeiro (1) ele executa, se for falso (0), ele não executa.
Pergunta 2: Por que não necessita de um cast?

R: Porque já é implícito que o valor “0” é considerado como falso, e qualquer valor diferente de 0 é considerado como verdadeiro (true) para a execução.
Pergunta 3: Para que serve cada um dos argumentos usados em pthread_create()?
R: O primeiro parâmetro é um ponteiro para um thread, o segundo são os atributos desse thread, o terceiro parâmetro é um ponteiro para a função que a thread vai executar, e o quarto parâmetro é um argumento para passar para a função do parâmetro anterior.
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()?
R: A thread “mãe” que criou as outros threads são encerrados e as threads “filhas” continuam executando.
Pergunta 5: Idem à questão anterior, se o término se dá através de um exit()?
R: A thread “mãe” é terminada e as threads “filhas” são interrompidos e finalizados.

Conclusão

Foi possível concluir através deste experimento o funcionamento das threads e como manipulá-las através das funções de ‘pthread_create’ e ‘pthread_exit’, além de aprender a utilizar os pointers dentro do buffer circular onde, foram feitas diversas tentativas no gerenciamento da posição de cada uma usando somente os próprios pointers porém, foi somente possível realizar tal feito usando uma variável a mais no auxílio do buffer.
O experimento do jantar dos filósofos foi interessante o uso da exclusão mútua para o ‘bloqueio’ dos garfos para evitar que dois filósofos usassem o mesmo talher. A necessidade da exclusão mútua foi além de resolver o problema projetado por Dijkstra, mostra uma solução à *race condition* que pode causar um grande problema quando duas threads tentam utilizar a mesma memória sem tal concorrência, acarretando na falha do programa ou sistema em que estiver rodando.