UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

HEITOR TANOUE DE MELLO
12547260

PROBLEMA DO JANTAR DOS FILÓSOFOS USANDO MONITORES EM C

SÃO CARLOS / SP
2023

1 INTRODUÇÃO

O problema do jantar dos filósofos é um clássico problema de sincronização em sistemas operacionais. Ele descreve uma situação em que um grupo de filósofos está sentado em uma mesa, com um prato de comida e dois garfos ao lado de cada um. Os filósofos passam o tempo pensando e comendo, mas só podem comer se estiverem com dois garfos (um em cada lado da mesa). O problema é garantir que os filósofos não entrem em deadlock (quando todos pegam um garfo e não conseguem pegar o segundo) ou em starvation (quando um filósofo nunca consegue pegar nenhum garfo).

Uma solução para esse problema é usar monitores. Os monitores são estruturas de dados que encapsulam variáveis compartilhadas e funções que operam sobre essas variáveis. Os monitores garantem que apenas um processo por vez possa acessar as variáveis compartilhadas e, assim, evitam condições de corrida e garantem a sincronização entre os processos.

Neste relatório, é descrita a implementação do problema do jantar dos filósofos usando monitores em C. O código foi testado no Linux e está disponível no arquivo *philosophers_dinner.c*.

2 LÓGICA DO CÓDIGO

O programa é composto por uma função *filosofo* que representa o comportamento de um filósofo. O código é composto principalmente por essas funções e estruturas:

1. A estrutura *Monitor* possui um mutex para garantir exclusão mútua durante o acesso aos recursos compartilhados e um array de variáveis de condição *cond_var* para sincronizar as threads dos filósofos.
2. A função *pegarGarfos* é chamada por um filósofo quando ele quer começar a comer. Ela bloqueia o mutex, define o estado do filósofo como "FAMINTO" e chama a função testar para verificar se ele pode comer. Se não for possível comer imediatamente, o filósofo espera na variável de condição associada ao seu índice usando *pthread_cond_wait*. Quando o filósofo é sinalizado por outra thread (por exemplo, quando os garfos estiverem disponíveis), ele sai do bloqueio do mutex e continua a executar.
3. A função *devolverGarfos* é chamada quando o filósofo termina de comer. Ela bloqueia o mutex, atualiza o estado do filósofo para "PENSANDO" e chama a função testar para verificar se seus vizinhos podem começar a comer.
4. A função *testar* verifica se um filósofo faminto pode começar a comer. Ela verifica se o estado do filósofo é "FAMINTO" e se seus vizinhos não estão comendo. Se essas condições forem atendidas, o estado do filósofo é atualizado para "COMENDO" e um sinal é enviado para a variável de condição correspondente, permitindo que um filósofo faminto desperte e tente novamente pegar os garfos.

Cada filósofo é representado por uma thread na função filosofo. O filósofo alterna entre os estados de pensar, ficar faminto, pegar garfos, comer e devolver garfos. Os tempos de pensar e comer são aleatórios. Após cada refeição, o filósofo conta quantas vezes ele comeu.

No main, as threads dos filósofos são criadas e executadas. O programa garante que todas as threads sejam finalizadas antes de encerrar.

Essa abordagem evita a possibilidade de deadlock garantindo que um filósofo só pegue os garfos se seus vizinhos não estiverem comendo. Além disso, evita a

starvation ao permitir que os filósofos esperem usando a função *pthread_cond_wait*, permitindo que outros filósofos tenham a oportunidade de comer.

3 ORIENTAÇÕES PARA EXECUÇÃO

Para executar o código, basta executar o comando *make* no terminal. Isso irá compilar o código e gerar o executável *philosophers_dinner*.

Para executar o programa, basta digitar *make run* no terminal. Isso irá rodar o programa com os parâmetros padrão: 5 filósofos. Para alterar é só mudar o valor do *define* NUM_FILOSOFOS.

Também é possível rodar o comando *make exec* para compilar e rodar o programa instantaneamente. O código-fonte do programa foi inserido abaixo:

1. **Código header** (*philosophers_dinner.h*)

```c
#ifndef DINING_PHILOSOPHERS_H
#define DINING_PHILOSOPHERS_H

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_FILOSOFOS 5
#define ESQUERDA(id) (id)
#define DIREITA(id) ((id + 1) % NUM_FILOSOFOS)

typedef enum
{
    PENSANDO,
    FAMINTO,
    COMENDO
} Estado;

typedef struct
{
    pthread_mutex_t mutex;
    pthread_cond_t cond_var[NUM_FILOSOFOS];
    Estado estados[NUM_FILOSOFOS];
} Monitor;

extern Monitor monitor;
```

```
    void pegarGarfos(int filosofo);
    void devolverGarfos(int filosofo);
    int podeComer(int filosofo);
    void testar(int filosofo);
    void *filosofo(void *arg);

    #endif
```

2. Código principal (*philosophers_dinner.c*)

```
    /*
        PROBLEMA DO JANTAR DOS FILÓSOFOS
        Heitor Tanoue de Mello - 12547260
    */

    #include "philosophers_dinner.h"

    Monitor monitor;

    void pegarGarfos(int filosofo)
    {
        // bloqueia mutex para entrar na região crítica
        pthread_mutex_lock(&monitor.mutex);
        monitor.estados[filosofo] = FAMINTO;
        testar(filosofo);

        // enquanto não puder comer, espera a condição
        if (monitor.estados[filosofo] != COMENDO)
        {
                        pthread_cond_wait(&monitor.cond_var[filosofo],
&monitor.mutex);
        }

        // desbloqueia mutex para sair da região crítica
        pthread_mutex_unlock(&monitor.mutex);
    }

    void devolverGarfos(int filosofo)
    {
```

```
        pthread_mutex_lock(&monitor.mutex);
        monitor.estados[filosofo] = PENSANDO;

        // testa se os vizinhos podem comer
        testar(ESQUERDA(filosofo));
        testar(DIREITA(filosofo));

        pthread_mutex_unlock(&monitor.mutex);
    }

    int podeComer(int filosofo)
    {
        // para evitar deadlock, o filósofo só pode comer se os seus
vizinhos não estiverem comendo
        return monitor.estados[filosofo] == FAMINTO &&
            monitor.estados[ESQUERDA(filosofo)] != COMENDO &&
            monitor.estados[DIREITA(filosofo)] != COMENDO;
    }

    void testar(int filosofo)
    {
        if (podeComer(filosofo))
        {
            monitor.estados[filosofo] = COMENDO;

            // sinaliza que o filósofo pode comer
            pthread_cond_signal(&monitor.cond_var[filosofo]);
        }
    }

    void *filosofo(void *arg)
    {
        int id = *(int *)arg;
        int qntRefeicoes = 0;

        while (1)
        {
            // Filósofo pensando
            printf("Filósofo %d está pensando...\n", id);

            sleep(rand() % 3); // Tempo aleatório pensando
```

```
            // Filósofo faminto
            printf("Filósofo %d está faminto...\n", id);

            pegarGarfos(id);

            // Filósofo comendo
            printf("Filósofo %d está comendo...\n", id);

            sleep(rand() % 3); // Tempo aleatório comendo

            devolverGarfos(id);

            qntRefeicoes++;
             printf("Filósofo %d terminou de comer pela %dª vez.\n",
id, qntRefeicoes);
        }
    }

    int main()
    {
        pthread_t filosofos[NUM_FILOSOFOS];
        int ids[NUM_FILOSOFOS];

        srand(time(NULL));

        // inicializa mutex e variáveis de condição
        pthread_mutex_init(&monitor.mutex, NULL);
        for (int i = 0; i < NUM_FILOSOFOS; i++)
        {
            pthread_cond_init(&monitor.cond_var[i], NULL);
        }

        // cria as threads
        for (int i = 0; i < NUM_FILOSOFOS; i++)
        {
            ids[i] = i;
            pthread_create(&filosofos[i], NULL, filosofo, &ids[i]);
        }

        // aguarda as threads terminarem
```

```
    for (int i = 0; i < NUM_FILOSOFOS; i++)
    {
        pthread_join(filosofos[i], NULL);
    }

    // destrói mutex e variáveis de condição
    pthread_mutex_destroy(&monitor.mutex);
    for (int i = 0; i < NUM_FILOSOFOS; i++)
    {
        pthread_cond_destroy(&monitor.cond_var[i]);
    }

    return 0;
}
```

4 RESULTADOS E PROBLEMAS ENCONTRADOS

O programa foi testado com diferentes números de filósofos e diferentes tempos de pensamento e de comida.

Esse código é eficiente em termos de evitar deadlock e starvation, pois garante que um filósofo só pode comer se seus vizinhos não estiverem comendo. Além disso, o uso de variáveis de condição permite que os filósofos esperem até que possam pegar os garfos e comer, sem consumir processamento desnecessário, evitando assim a espera ativa. No entanto, a implementação do código pode levar a um alto uso de memória, pois cria uma variável de condição para cada filósofo, o que pode ser problemático se o número de filósofos for muito grande.