SSC 0640 - Sistemas Operacionais I

Implementação do Jantar dos Filósofos

Prof. Vanderlei Bonato

Carlos Henrique Hannas de Carvalho

n°USP: 11965988

1. Enunciado

Dados os exemplos de variáveis condicionais, de sincronização com semáforos e do protótipo de um monitor, implemente baseado no modelo de monitor o problema do jantar dos filósofos (conforme descrito no livro texto) de maneira que não exista a possibilidade de deadlock e nem de starvation. Cada filósofo deve estar em uma thread e os recursos que usam deverão ser compartilhados (os palitos). Implementar no Linux usando C.

2. Descrição do problema do Jantar dos Filósofos

O problema do Jantar dos Filósofos, proposto por Dijkstra em 1965, é descrito da seguinte forma: cinco filósofos estão ao redor de uma mesa e cada um desses filósofos têm um prato para comer. Entre cada par de pratos, há um hashi - os filósofos só conseguem comer quando possuem dois hashis.

O filósofo possui algumas ações: pensar, sentir fome e comer. Quando ele sente fome, tenta pegar o hashi à esquerda e à direita do seu prato - se bem sucedido, ele come, devolve os hashis à mesa e volta a pensar.

3. Solução

a. Descritiva

A solução do problema deve usar um arranjo (estado), que controle se um filósofo está pensando, sentindo fome ou comendo. O filósofo pode comer apenas se seus vizinhos diretos à esquerda e à direita não estiverem comendo - os vizinhos do filósofo *f* são definidos como *FILO_ESQ* e *FILO_DIR*, na Figura 1. O programa deve ser feito através do modelo de monitores e

semáforos, para que os filósofos sejam bloqueados, se os talheres necessários a ele estiverem ocupados.

No conceito de Sistemas Operacionais, os filósofos são descritos como *threads* e os hashis são *recursos* - "filósofos usarem hashi" é uma analogia à "*threads* usarem *recursos",* para realizar tarefas. Abaixo há a implementação da solução do problema dos filósofos na linguagem C - o arquivo *main.c* também foi disponibilizado junto a este documento.

b. Implementação em C

Figura 1

```c
#include <stdio.h>
#include <pthread.h>

#define N 5 //Define a quantidade de filosofos

//Definicao dos estados de filosofos e mutex
typedef enum { PENSAR, FOME, COMER } Estado;
Estado estados[N];
pthread_mutex_t mutex;
pthread_cond_t cond_filosofo[N];

pthread_mutex_t mutex_garcom;
pthread_cond_t cond_garcom;

//Definicao do filosofo a direita e a esaquerda do filosofo f
#define FILO_ESQ(f) ((f + N - 1) % N)
#define FILO_DIR(f) ((f + 1) % N)
```

A figura 1, acima, trata apenas das definições iniciais para a implementação do problema proposto. Definiu-se $N = 5$, que são o número de filósofos presentes na mesa. Em seguida, nas linhas 7 a 8, define-se os estados dos filósofos (pensar, sentir fome e comer) e os mutexes necessários. Por fim, declarou-se um filósofo *f* e os vizinhos à esquerda e à direita de *f.*

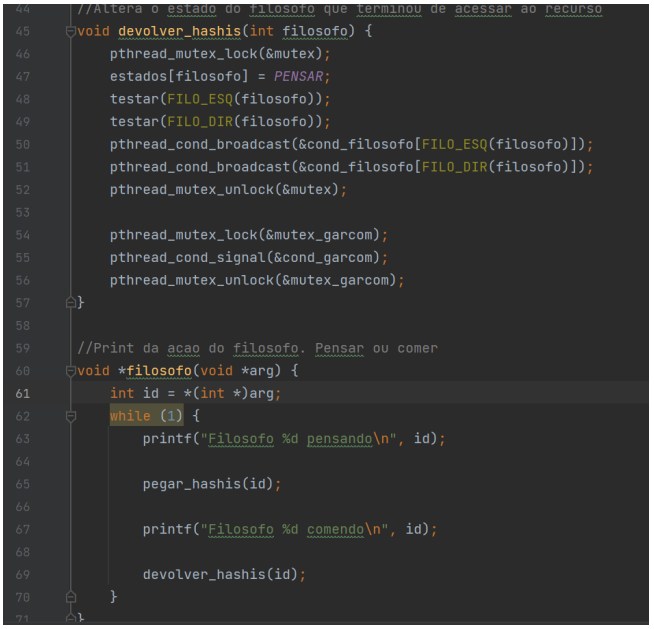
Figura 2

```
19  //Executa o acesso ao recurso
20  void testar(int filosofo) {
21      if (estados[filosofo] == FOME && estados[FILO_ESQ(filosofo)] != COMER && estados[FILO_DIR(filosofo)] != COMER) {
22          estados[filosofo] = COMER;
23          pthread_cond_signal(&cond_filosofo[filosofo]);
24      }
25  }
26
27  //Funcao para filosofo que quer acessar ao recurso
28  void pegar_hashis(int filosofo) {
29      pthread_mutex_lock(&mutex_garcom);
30      while (estados[FILO_ESQ(filosofo)] == COMER || estados[FILO_DIR(filosofo)] == COMER) {
31          pthread_cond_wait(&cond_garcom, &mutex_garcom);
32      }
33      pthread_mutex_unlock(&mutex_garcom);
34
35      pthread_mutex_lock(&mutex);
36      estados[filosofo] = FOME;
37      testar(filosofo);
38      while (estados[filosofo] != COMER) {
39          pthread_cond_wait(&cond_filosofo[filosofo], &mutex);
40      }
41      pthread_mutex_unlock(&mutex);
42  }
```

Na linha 21 há um teste *if* que verifica se um filósofo está com fome e se seus vizinhos à esquerda e à direita não estão comendo. Se as condições do teste forem satisfeitas, o filósofo pode pegar os hashi e comer. Caso contrário, as linhas 22 e 23 não serão executadas.

A função *pegar_hashi* é responsável por controlar o acesso dos filósofos (*threads*) aos hashis (*recursos*). Inicialmente há um bloqueio de *mutex* para que apenas um filósofo, por vez, execute o código. Em seguida, caso os filósofos à direita e/ou esquerda estejam comendo, deve-se esperar que eles terminem para o filósofo, que está com fome, possa pegar o hashi - para isso teve que desbloquear o *mutex*. Por fim, bloqueia-se o *mutex* novamente, para controlar o acesso do filósofo, e a função *teste* é solicitada, para que após o filósofo pegar o hashi, poder de fato comer.

Figura 3



```
//Altera o estado do filosofo que terminou de acessar ao recurso
void devolver_hashis(int filosofo) {
    pthread_mutex_lock(&mutex);
    estados[filosofo] = PENSAR;
    testar(FILO_ESQ(filosofo));
    testar(FILO_DIR(filosofo));
    pthread_cond_broadcast(&cond_filosofo[FILO_ESQ(filosofo)]);
    pthread_cond_broadcast(&cond_filosofo[FILO_DIR(filosofo)]);
    pthread_mutex_unlock(&mutex);

    pthread_mutex_lock(&mutex_garcom);
    pthread_cond_signal(&cond_garcom);
    pthread_mutex_unlock(&mutex_garcom);
}

//Print da acao do filosofo. Pensar ou comer
void *filosofo(void *arg) {
    int id = *(int *)arg;
    while (1) {
        printf("Filosofo %d pensando\n", id);

        pegar_hashis(id);

        printf("Filosofo %d comendo\n", id);

        devolver_hashis(id);
    }
}
```

Após o filósofo comer, deve-se devolver o talher à mesa para que outros filósofos também possam comer - a função responsável por isso é *devolver_hashi*. Inicia-se a função com o bloqueio do *mutex* e atribui ao filósofo, que acabou de comer, o valor *PENSAR* (conforme deve ser a solução do problema). Em seguida há um teste para verificar se o filósofo à esquerda pode começar a comer e depois há o mesmo teste para o vizinho à direita.

A função **filosofo* é responsável apenas por fazer o print de que um filósofo está pensando e, depois de chamar a função *pegar_hashi*, fazer um print de que o filósofo está comendo. Após comer, há uma chamada para *devolver_hashi* e realizar a ação conforme já foi descrito anteriormente.

Figura 4

```
int main() {
    pthread_t filosofos[N];
    int ids[N];

    pthread_mutex_init(&mutex, NULL);
    pthread_mutex_init(&mutex_garcom, NULL);
    pthread_cond_init(&cond_garcom, NULL);
    for (int i = 0; i < N; i++) {
        pthread_cond_init(&cond_filosofo[i], NULL);
    }

    //Associacao de uma thread a um filosofo
    for (int i = 0; i < N; i++) {
        ids[i] = i;
        pthread_create(&filosofos[i], NULL, filosofo, &ids[i]);
    }

    //Sincronizacao entre trheads
    for (int i = 0; i < N; i++) {
        pthread_join(filosofos[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    for (int i = 0; i < N; i++) {
        pthread_cond_destroy(&cond_filosofo[i]);
    }

```

A figura 4 é a função *main*. Nela há as associações entre *threads* e filósofos, bem como a sincronização entre as *threads*.

c. Estratégia e problemática

O código em C apresenta mecanismos para a prevenção de *deadlocks*. O *deadlock* ocorre quando um filósofo possui um hashi e aguarda pelo outro hashi - isso resulta em um impasse. A prevenção dessa situação é fruto da estratégia de “prevenção de solicitação circular”.

Em relação aos monitores, utiliza-se o mutex. O mutex *mutex*, no código, é usado para proteger o acesso às variáveis *estado*. O mutex *mutex_garcom* é usado a fim de controlar o acesso ao *garcom*.

Por fim, sobre a situação de starvation encontrou-se um problema. No código implementado, um filósofo só obtém os hashis quando os seus filósofos vizinhos não estão comendo. Entretanto, não houve garantia de que todos tenham oportunidades de comer. Há a possibilidade de um filósofo possuir maior prioridade para comer em relação ao outro - resulta em alguns filósofos com menor prioridade.