Universidade Federal de Ouro Preto

Prova 01 - BCC264 - SISTEMAS OPERACIONAIS

Prof. Dr. Carlos Frederico M. C. Cavalcanti

Enya Luísa Gomes dos Santos

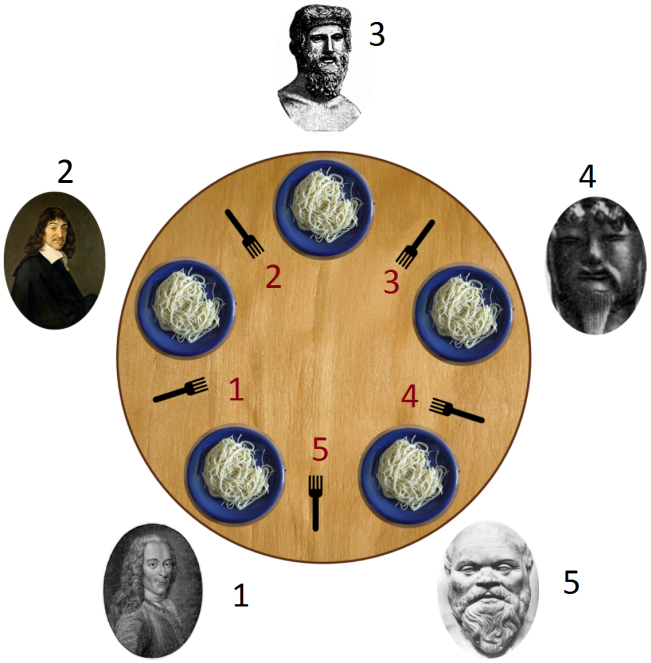
19.2.4201

OURO PRETO - 2021

O Jantar do Filósofo

O problema

A definição do problema é que cinco filósofos estão sentados a uma mesa redonda, com um prato de spaghetti na frente de cada um, e um garfo é colocado entre cada um deles.



FONTE: https://dicasdedevelop.wordpress.com/2015/06/17/c/

Nesse jantar, os filósofos apenas comem quando estão com fome e pensam, alternando entre as duas atividades, entretanto só podem comer quando ambos os

garfos (à sua direita e à sua esquerda) estiverem livre, e um garfo só pode ser utilizado por um filósofo por vez.

Em suma, um filósofo pensa, fica com fome e, se possível, pega os dois garfos ao seu lado, e dessa forma come, coloca os garfos na mesa, e volta a pensar. Se o filósofo 1 tem fome e pega os garfos 1 e 5, os filósofos 2 e 5 não podem comer até que 1 termine de comer e coloque os garfos sobre a mesa.

O problema em C

Para ser possível simular esse problema computacionalmente serão utilizadas as threads e os semáforos, onde cada filósofo é uma thread, e cada garfo é um semáforo.

Concorrência

Programação concorrente evita que dois processos tenham acesso a algum recurso compartilhado ao mesmo tempo. No caso do algoritmo do jantar dos filósofos, a concorrência ocorre para evitar que dois filósofos utilizem o mesmo garfo ao mesmo tempo.

Down e Up

Operação *wait (down())*

Descrementa o valor do semáforo. Se o semáforo está com valor zero, o processo é posto em espera. Para receber um sinal, um processo executa o *wait* e bloqueia se o semáforo impedir a continuação da sua execução.

Operação *signal (up())*

Se o semáforo estiver com o valor zero e existir algum processo em espera, um processo será acordado. Caso contrário, o valor do semáforo é incrementado. Para enviar um sinal, um processo executa um *signal*.

Exemplificação

Onde s é um semáforo

down(s):

```
if (s == 0)
      bloqueia_processo();
else
      s--;
```

up(s):

```
 if (s == 0 && existe processo bloqueado)
      acorda_processo();
else
      s++;
```

Os métodos de down() e up() são utilizados no programa em C para permitir que um filósofo pegue os dois garfos ao seu lado e coma, se possível.

Princípio da exclusão mútua

Em resumo é que só um processo é executado de cada vez.

“O princípio da exclusão mútua ou *mutex*, é uma técnica usada em programação concorrente para evitar que dois processos ou *threads* tenham acesso simultaneamente a um recurso partilhado. [...] um meio simples para exclusão mútua é a utilização do tal semáforo binário, isto é, que só pode assumir dois valores distintos, 0 e 1.”

E no problema do jantar do filósofo se faz necessário apenas um filósofo, de acordo com a regra, não usem do mesmo garfo, assim como na concorrência.

O código em c

O código segue a ideia do exemplo disposto no livro do Tanenbaum capítulo 2, páginas (99 - 100), com algumas alterações para funcionar na linguagem C e visualizar os resultados.

```
#include<stdio.h>
#include<stdlib.h>
#include<semaphore.h>
#include<pthread.h>
#include<unistd.h>
#include <signal.h>

#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT (i+4)%N
#define RIGHT (i+1)%N

#define think() sleep(1)
#define eat() sleep(1)

#define up sem_post
#define down sem_wait

void *philosopher(void *num);
void take_forks(int);
void put_forks(int);
void test(int);

typedef sem_t semaphore;
semaphore mutex;
semaphore S[N]; //inicializacao do semáforo
int state[N];
int nPhilosopher[N]={ 0,1,2,3,4 }; /* armazena os "filosofos" */

void *philosopher(void *num)
{
   while(1)
```

```
   {
      int *i = num;
      think();
      take_forks(*i);
      eat();
      put_forks(*i);
   }
}

void take_forks(int i)
{
   down(&mutex);
   state[i] = HUNGRY;
   printf("Filosofo %d tem fome.\n", i+1);
   test(i);
   up(&mutex);
   down(&S[i]);
   sleep(1);
}

void put_forks(int i)
{
   down(&mutex);
   state[i]=THINKING;
   printf("Filosofo %d deixou os garfos %d e %d.\n", i+1, LEFT+1,
i+1);
   printf("Filosofo %d pensando.\n", i+1);
   test(LEFT);
   test(RIGHT);
   up(&mutex);
}

void test(int i)
{
   if(state[i]==HUNGRY && state[LEFT] != EATING && state[RIGHT] !=
EATING)
   {
      state[i]=EATING;
      eat();
      printf("Filosofo %d agarrou os garfos %d e %d.\n", i+1,
LEFT+1, i+1);
      printf("Filosofo %d comendo.\n", i+1);
```

```
      up(&S[i]);
   }
}

int main() {
   int i;
   pthread_t thread_id[N]; /* identificadores das threads */
   sem_init(&mutex,0,1);

   for(i=0;i<N;i++)
      sem_init(&S[i],0,0);

   for(i=0;i<N;i++)
   {
      pthread_create(&thread_id[i], NULL, philosopher, (void
*)&nPhilosopher[i]);
      //criar as threads
      printf("Filosofo %d esta a pensar.\n",i+1);
   }

   for(i=0;i<N;i++)
   pthread_join(thread_id[i],NULL); /* Esperara a junção das
threads */
   return(0);
}
```