T1 – Fundamentos de Processamento Paralelo e Distribuído;

Grupo 4: Brenda Billmann, Filipe Dupont Bandeira e João Pedro Moreira Antunes

O Problema do Jantar dos Filósofos é um problema clássico de sincronização na ciência da computação que envolve vários processos (filósofos) compartilhando um conjunto limitado de recursos (garfos) para realizar uma tarefa (comer). A fim de evitar o bloqueio ou a inanição, é necessário implementar uma solução que garanta que cada filósofo possa acessar os recursos de que precisa para realizar sua tarefa sem interferência de outros filósofos.

Uma solução comum para o Problema do Jantar dos Filósofos utiliza semáforos, um mecanismo de sincronização que pode ser usado para controlar o acesso a recursos compartilhados. Nessa solução, cada garfo é representado por um semáforo, e um filósofo deve adquirir tanto o semáforo do garfo à sua esquerda quanto o semáforo do garfo à sua direita antes de começar a comer. Se um filósofo não puder adquirir ambos os semáforos, ele deve esperar até que fiquem disponíveis. (Código 1)

Usando semáforos para controlar o acesso aos garfos, o Problema dos Filósofos Jantando pode ser resolvido de maneira a evitar bloqueios e inanição. O uso do semáforo mutex garante que apenas um filósofo possa tentar pegar um garfo de cada vez, enquanto o uso dos semáforos dos garfos garante que um filósofo só possa comer se ambos os garfos estiverem disponíveis.

Em resumo, a solução do Problema dos Filósofos Jantando usando semáforos é um exemplo clássico de como mecanismos de sincronização podem ser usados para resolver problemas complexos de sincronização na programação concorrente.

Já na outra solução proposta (código 2), conhecida como "Resource Hierarchy Solution" (Solução de Hierarquia de Recursos), aborda o desafio de permitir que vários filósofos compartilhem garfos para se alimentar de forma ordenada e eficiente. Esta abordagem introduz uma hierarquia parcial entre os recursos (garfos) e estabelece uma convenção rigorosa de solicitar esses recursos em uma ordem específica. Neste contexto, cada filósofo sempre pega primeiro o garfo com o número mais baixo e depois o garfo com o número mais alto, criando um sistema que evita bloqueios, mas que também possui limitações notáveis.

O código também evita deadlocks, mas nem sempre é prática. Por exemplo, se uma thread está com os garfos 3 e 5 determina que precisa do garfo 2, ela deverá liberar 5, depois 3 antes de adquirir 2, e então deverá readquirir 3 e 5 nessa ordem.
Os programas de computador que acessam um grande número de registros de banco de dados não funcionam de forma eficiente se fossem obrigados a liberar todos os registros de numeração mais alta antes de acessar um novo registro, tornando o método impraticável para esse propósito.

Concluindo: cada abordagem possui seus próprios pontos fortes e limitações. A solução com semáforos oferece uma implementação direta e eficaz, evitando bloqueios e inanição. Ela é altamente adaptável e pode ser aplicada a uma variedade de cenários de sincronização, garantindo a justiça no acesso aos recursos compartilhados. Além disso, não exige conhecimento prévio dos recursos necessários.

Por outro lado, a "Resource Hierarchy Solution" (Solução de Hierarquia de Recursos) introduz uma ordem parcial entre os recursos, garantindo que os garfos sejam adquiridos de maneira ordenada e evitando bloqueios. Isso oferece previsibilidade e simplicidade conceitual, tornando-a uma solução fácil de compreender e implementar.

No entanto, essa segunda abordagem também possui desvantagens notáveis. Ela não é tão flexível quanto a solução de semáforos, pois exige que os recursos sejam conhecidos antecipadamente e que sejam adquiridos em uma ordem específica. Além disso, ela não garante a justiça no acesso aos recursos, o que significa que um filósofo mais lento pode ser desfavorecido em relação aos mais rápidos.

Em resumo, a escolha entre essas duas abordagens depende das necessidades específicas do problema e das prioridades de implementação. A solução com semáforos oferece mais flexibilidade e justiça, enquanto a "Resource Hierarchy Solution" (Solução de Hierarquia de Recursos) fornece simplicidade e previsibilidade. A decisão final dependerá do contexto em que o problema dos filósofos está sendo resolvido.

[Código 1] rodando em 20 Segundos
Espera de 1 segundo enquanto come.

```
Philosopher 5 is Eating
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 2 is Hungry
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 5 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 3 is Hungry
Philosopher 1 is Hungry
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 4 is Hungry
Philosopher 2 is Hungry
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking

Filosofo 1 comeu (2) vezes em 20 segundos


Filosofo 2 comeu (2) vezes em 20 segundos


Filosofo 3 comeu (2) vezes em 20 segundos


Filosofo 4 comeu (1) vezes em 20 segundos


Filosofo 5 comeu (2) vezes em 20 segundos
```

[Código 2] rodando em 20 segundos
Espera de 1 segundo enquanto come.

```
3 thinks
                                2 eats
                1 is hungry
5 thinks
                                4 eats
                3 is hungry
2 thinks
                                1 eats
                5 is hungry
4 thinks
                                3 eats
                2 is hungry
1 thinks
                                5 eats
                4 is hungry
3 thinks
                                2 eats
                1 is hungry
5 thinks
                                4 eats
                3 is hungry
2 thinks
                                1 eats
                5 is hungry
4 thinks
                                3 eats
                2 is hungry
                4 is hungry
1 thinks
                                2 eats
3 thinks
                                5 eats

Filosofo 1 comeu (7) vezes em 20 segundos


Filosofo 2 comeu (8) vezes em 20 segundos


Filosofo 3 comeu (8) vezes em 20 segundos


Filosofo 4 comeu (8) vezes em 20 segundos


Filosofo 5 comeu (7) vezes em 20 segundos
```

[Código 1 - Uso de semáforos]

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

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

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };
int running = 1;

sem_t mutex;
sem_t S[N];

int quantoComeu[N];


void test(int phnum)
{
    if (state[phnum] == HUNGRY
        && state[LEFT] != EATING
        && state[RIGHT] != EATING) {
        // state that eating
        state[phnum] = EATING;

        sleep(2);

        printf("Philosopher %d takes fork %d and %d\n",
                    phnum + 1, LEFT + 1, phnum + 1);

        printf("Philosopher %d is Eating\n", phnum + 1);
    quantoComeu[phnum]++;

        sem_post(&S[phnum]);
    }
}
```

```
// take up chopsticks
void take_fork(int phnum)
{

    sem_wait(&mutex);

    // state that hungry
    state[phnum] = HUNGRY;

    printf("Philosopher %d is Hungry\n", phnum + 1);

    // eat if neighbours are not eating
    test(phnum);

    sem_post(&mutex);

    // if unable to eat wait to be signalled
    sem_wait(&S[phnum]);

    sleep(1);
}

// put down chopsticks
void put_fork(int phnum)
{

    sem_wait(&mutex);

    // state that thinking
    state[phnum] = THINKING;

    printf("Philosopher %d putting fork %d and %d down\n",
        phnum + 1, LEFT + 1, phnum + 1);
    printf("Philosopher %d is thinking\n", phnum + 1);

    test(LEFT);
    test(RIGHT);

    sem_post(&mutex);
}

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

```
    while (running == 1) {

        int* i = num;

        sleep(1);

        take_fork(*i);

        sleep(0);

        put_fork(*i);
    }
}

int main()
{


    int i;
    pthread_t thread_id[N];

  int n;
  for(n=0; n<N; ++n)
    quantoComeu[n] = 0;

    // initialize the semaphores
    sem_init(&mutex, 0, 1);

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

    for (i = 0; i < N; i++)

        // create philosopher processes
        pthread_create(&thread_id[i], NULL,
                       philosopher, &phil[i]);

    int x = 20;
    sleep(x);
```

```
  for(n=0; n<N; ++n)
    printf("\nFilosofo %d comeu (%d) vezes em %d segundos\n\n", n+1,
quantoComeu[n], x);

}
```

[Código 2 - Resource Hierarchy Solution]

```
#include <iostream>
#include <chrono>
#include <mutex>
#include <thread>
#include <random>
#include <ctime>
#include <stdio.h>

using namespace std;

int quantoComeu[5];

int myrand(int min, int max) {
  static mt19937 rnd(time(nullptr));
  return uniform_int_distribution<>(min,max)(rnd);
}

void philosopher(int ph, mutex& ma, mutex& mb, mutex& mo) {
  for (;;) {  // prevent thread from termination
    int duration = 1000;
    {
      // Block { } limits scope of lock
      lock_guard<mutex> gmo(mo);
      cout<<ph<<" thinks \n";
    }
    this_thread::sleep_for(chrono::milliseconds(duration));
    {
      lock_guard<mutex> gmo(mo);
      cout<<"\t\t"<<ph<<" is hungry\n";
    }
    lock_guard<mutex> gma(ma);
    // sleep_for() Delay before seeking second fork can be added here
but should not be required.
    lock_guard<mutex> gmb(mb);
    //duration = myrand(200, 800);
    {
      lock_guard<mutex> gmo(mo);
      cout<<"\t\t\t\t"<<ph<<" eats \n";
      quantoComeu[ph-1]++;
    }
    this_thread::sleep_for(chrono::milliseconds(duration));
```

```
  }
}

int main() {
  cout<<"dining Philosophers C++11 with Resource hierarchy\n";
  mutex m1, m2, m3, m4, m5;   // 5 forks are 5 mutexes
  mutex mo;           // for proper output
  // 5 philosophers are 5 threads
  thread t1([&] {philosopher(1, m1, m2, mo);});
  thread t2([&] {philosopher(2, m2, m3, mo);});
  thread t3([&] {philosopher(3, m3, m4, mo);});
  thread t4([&] {philosopher(4, m4, m5, mo);});
  thread t5([&] {philosopher(5, m1, m5, mo);});  // Force a resource
hierarchy
  this_thread::sleep_for(chrono::milliseconds(20000));

  for(int n=0; n<5; ++n)
    printf("\nFilosofo %d comeu (%d) vezes em 20 segundos\n\n", n+1,
quantoComeu[n]);
 }
```