8 PROBLEMAS CLÁSSICOS

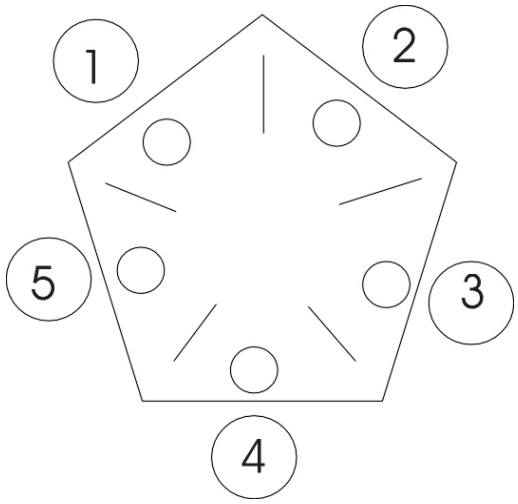
8.1.1 FILÓSOFOS GLUTÕES

Contexto: temos cinco filósofos, cujas únicas atividades são comer macarrão e pensar. Além disso, o macarrão nunca acaba.

Componentes do problema:

- 5 Filósofos
- 5 Pratos com macarrão. Cada prato é destinado a um filósofo
- 5 Garfos para comer macarrão

Problema: para comer o macarrão é preciso manejar dois garfos. Dessa maneira, apenas dois filósofos podem comer simultaneamente, os demais devem ficar "pensando". **Ou seja, este problema modela um determinado número de processos competindo pelo acesso a um número limitado de recursos.** A Figura abaixo ilustra o ambiente do problema.



Uma primeira solução é a execução de 5 processos simultaneamente, onde cada processo representa um filósofo. O código do processo é ilustrado a seguir:

```
#define N        5              // número de filósofos
void philosopher (int i)
{
        while (TRUE)
        {
```

```
            think();
            take_fork(i);
            take_fork((i+1)%N);
            eat();
            put_fork(i);
            put_fork((i+1)%N);
        }
}
```

Algumas considerações:

- As ações *eat* (comer) e *think* (pensar) possuem um tempo de duração aleatório e independente da implementação da solução;
- São 5 processos sendo executados simultaneamente;

Deficiências dessa solução: se todos os filósofos resolverem comer ao mesmo tempo e, cada um deles vai pegar o seu garfo da esquerda. Todos estarão bloqueados, pois ficarão esperando o seu respectivo garfo direito ser liberado. Esta é uma situação de *deadloak*;

Poderia-se sugerir que se um filósofo verificar que o garfo direito estiver ocupado, ele libera o esquerdo. Entretanto, na mesma situação em que todos os filósofos resolvem comer ao mesmo tempo. Acontecerá o seguinte problema:

- Todos pegarão os seus respectivos garfos da esquerda;
- Verificarão que os garfos da direita não estão disponíveis;
- Devolverão os garfos;
- Por coincidência, todos os filósofos vão tentar pegar os garfos novamente;
- Assumindo que isto sempre ocorrerá, os filósofos ficarão em um eterno ciclo de pegar-devolver o garfo, sem comer.

Esta situação é chamada de *starvation*. Mesmo considerando que a possibilidade de ocorrência dessa situação é muito pequena, deve ser observada e prevenida.

Uma solução mais aceitável é descrita logo abaixo.

```
#define N    5
#define LEFT (i-1)%N // número de vizinhos a esquerda de i
#define RIGHT (i+1)%N // número de vizinhos à direita de i
```

```
#define THINKING  0  //filósofo pensando
#define HUNGRY 1 // filósofo com fome
#define EATING 2 // filósofo comendo

int state[N];
semaphore mutex =1;
semaphore s[N];
void semaphore (int i)
{
        while (TRUE)
        {
                think();
                take_forks(i);
                eat();
                put_forks(i);
        }
}

void take_forks(int i)
{
        down(&mutex);
        state[i]= HUNGRY;
        test(i);
        up(&mutex);
        down(&s[i]);
}

void put_forks(int i)
{
```

```
        down(&mutex);
        state[i]= THINKING;
        test(LEFT);
        test(RIGHT);
        up(&mutex);
}
void test(int i)
{
        if(state[i] = = HUNGRY && state[LEFT] != EATING && state[RIGHT] = EATING)
        {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

Considerações da solução:

- *mutex* é um semáforo binário que controla o acesso a região crítica, no caso o acesso aos garfos;
- Cada filósofo possui seu próprio semáforo para bloqueá-lo, caso não consiga acesso aos recursos;
- Temos cinco processos philosopher executando simultaneamente;
- As funções *take_forks* e *put_forks* são executadas respectivamente quando o filósofo quer pegar os garfos e devolver os garfos;
- A função *test* verifica se os garfos estão livres e cuida para que o filósofo tome os dois garfos de uma só vez.

8.1.2 LEITORES E DESCRITORES

Contexto: Um determinado número de leitores ficam lendo um livro, o qual ainda está em processo de escrita.

Componentes:

- 1 escritor
- Vários leitores

- 1 livro

Problema: **Todos os leitores podem ler o livro simultaneamente. Entretanto, quando o escritor estiver com o livro, ninguém pode ler ele**.

Analogia: **este problema modela o acesso a um banco de dados**. Onde vários usuários podem ler os dados simultaneamente. Entratanto, somente um usuário pode alterar o banco de dados, e enquanto estiver fazendo este processo, nenhum outro usuário pode ter acesso ao banco de dados.

Solução:

```
semaphore mutex = 1;
semaphore db = 1;
int rc = 1;
void reader(void)
{
        while (TRUE)
        {
                down(&mutex);
                rc = rc + 1;
                if (rc = = 1) down(&db);
                up(&mutex);
                read_database();
                down(&mutex);
                rc = rc - 1;
                if (rc = = 0) up(&db);
                up(&mutex);
                use_data_read();
        }
}

void writer(void)
{
```

```
        while (TRUE)
        {
                think_up_data();
                down(&db);
                write_database();
                up(&db);
        }
}
```

Considerações da solução:

- Pode haver vários leitores. Isto significa que se houver N leitores, haverá N processos *reader* ativos e sendo executados simultaneamente;
- primeiro leitor faz um *down*(db), os subsequentes apenas incrementam rc – número de leitores;
- Todo leitor que deixa o acesso a db, deve decrementar o rc. Caso não haja nenhum outro leitor, deve ser liberado o acesso a db.
- Quando houver um leitor acessando o banco de dados, o escritor deve estar bloqueado;
- Só pode haver um escritor escrevendo no banco de dados por vez.

8.1.3 BARBEIRO DORMINHOCO

Contexto: Em uma barbearia, há uma cadeira de atendimento, na qual o cliente é atendido pelo barbeiro. Também existe um determinado número de cadeiras para que clientes que chegam, quando o barbeiro estiver ocupado, sentar e aguadar. Se todas as cadeiras estiverem ocupadas e chegar um outro cliente, o mesmo deve ir embora.

Componentes:

- 1 barbeiro;
- 1 cadeira de barbairo;
- CHAIRS cadeiras para aguardar;
- Vários clientes transitando pela barbearia.

Analogia: **um processo servidor que possui um buffer de atendimento e vários processos clientes que desejam utilizar os serviços do processo servidor;**

Código da solução:

```
#define CHAIRS 5
semaphore customers = 0;
semáfore babers = 0
semaphore mutex = 1;
int waiting = 0;
void Baber(void)
{
        while (TRUE)
        {
                down(customers);
                down(mutex);
                waiting = waiting =1;
                up(babers);
                up(mutex);
                cut_hair();
        }
}

void Customer(void)
{
        down(mutex);
        if( wating <CHAIRS)
        {
                waiting = waiting + 1;
                up(customers);
                up(mutex);
                down(babers);
```

```
            get_haircut();
        }
        else up(mutex);
}
```

Considerações da solução

- Se não houver clientes, o barbeiro é bloqueado;
- *Mutex* controla o acesso às cadeiras do barbeiro;
- barbeiro não está em um loop, pois ele é somente acionado quando requisitado.;