



## Interblocagem

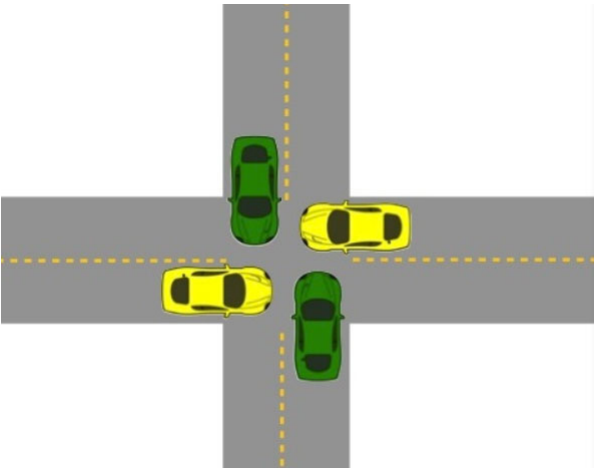
*Deadlock*

Sistemas Operativos – DEI - IST

1



## Interblocagem



Situação em que duas ou mais tarefas são impedidas de continuar e a condição de bloqueio é permanente

Sistemas Operativos – DEI - IST

2



## Situações de Interblocagem

- Podem resultar de erros de programação que já vimos em múltiplas situações
- Estas resolvem-se com programação cuidada
- Veremos mais à frente primitivas de sincronização de nível abstracção mais elevado para tentar eliminar alguns dos erros de programação
- Mas há situações mais complicadas...

```

fechar_leitura()
{
    pthread_mutex_lock(&secCritica);
    if (em_escrita || escritores_espera > 0)
        leitores_espera++;
    pthread_mutex_unlock(&secCritica);
    sem_wait(&leitores); /* leitores_espera--; */
    pthread_mutex_lock(&secCritica);
    if (leitores_espera > 0) {
        nleitores++;
        leitores_espera--;
        sem_post(&leitores);
    }
}
else
    nleitores++;
pthread_mutex_unlock(&secCritica);

```

Sistemas Operativos – DEI - IST

3



## Exemplo básico de interblocagem

**Tarefa A**

`pthread_mutex_lock (mutex1)`

`pthread_mutex_lock (mutex2)`

**Tarefa B**

`pthread_mutex_lock (mutex2)`

`pthread_mutex_lock (mutex1)`

**Tarefa Bloqueada**



Sistemas Operativos – DEI - IST

4



## Solução simples: ordenar as operações de *lock*

- Garantir que os recursos reservados através de *mutexes* ou semáforos são todos adquiridos pela mesma ordem;
- Tarefa A
- Tarefa B

`pthread_mutex_lock (mutex1)`

`pthread_mutex_lock (mutex1)`

Bloqueada

`pthread_mutex_lock (mutex2)`

`pthread_mutex_lock (mutex2)`

Problema é se não é possível estabelecer à priori uma ordem dos recursos para toda uma aplicação

Sistemas Operativos – DEI - IST

5




## Ordenar as operações de *lock*

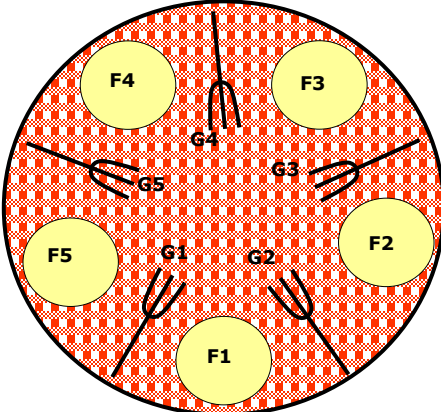
- Difícil, senão mesmo impossível, em projectos complexos
- Mesmo assim, garantir a ordem pode ser difícil como o exemplo seguinte procura ilustrar

Sistemas Operativos – DEI - IST

6



## Jantar dos Filósofos




- Cinco Filósofos estão reunidos para filosofar e jantar spaghetti
- Para comer precisam de dois garfos, mas a mesa apenas tem um garfo por pessoa.

Um dos problemas de sincronização mais conhecidos

Sistemas Operativos – DEI - IST

7



## Jantar dos Filósofos

- Condições:
  - Os filósofos podem estar em um de três estados:
    - Pensar;
    - Ter fome;
    - Comer
  - O lugar de cada filósofo é fixo
  - Um filósofo apenas pode utilizar os garfos imediatamente à sua esquerda e direita.

Sistemas Operativos – DEI - IST

8



## Jantar dos Filósofos

```
filosofo(int id){
    while (dining) {
        pensar();
        <adquirir os garfos>
        <comer>
        <libertar os garfos>
    }
}
```

```
void* philospher(void* num)
{
    int* phil = num;

    while (dining) {
        sleep(1); // pensar
        take_fork(*phil);
        sleep(2); // comer
        put_fork(*phil);
    }
}
```

Vamos modelar cada garfo como um recurso unitário representado por um semáforo inicializado a 1

Sistemas Operativos – DEI - IST

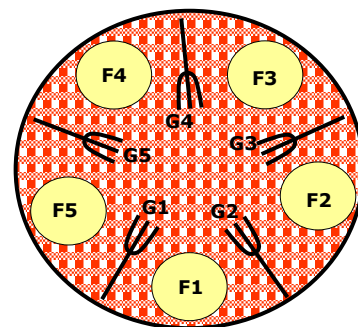
9



## Jantar dos Filósofos programados com Semáforos

```
sem_t sfork[5] = {...};
for (i = 0; i < N; i++) sem_init(&sfork[i], 0, 1);
```

```
void* philospher(void* num)
{
    int* phil = num;
    while (dining) {
        printf("Filosofo %d a pensar\n", *phil+1);
        sleep(0);
        printf("Filosofo %d tem fome \n", *phil+1);
        sem_wait(&sfork[*phil]);
        sem_wait(&sfork[( *phil + 1) % N]);
        printf ("Filosofo %d a comer \n", *phil+1);
        sleep(1);
        sem_post(&sfork[phil]);
        sem_post(&sfork[(phil + 1) % N]);
    }
}
```



Problema?

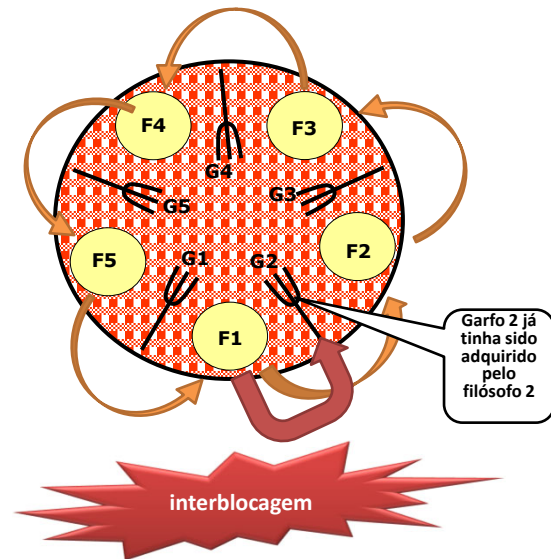
Sistemas Operativos – DEI - IST

10



## Situação de Interblocagem

- A situação de interblocagem pode aparecer numa sequência particular de escalonamento das tarefas:
  - Todos os filósofos pretendem comer ao mesmo tempo
  - F1 adquire o garfo à sua esquerda e perde o processador
  - O ciclo repete-se para os restantes



Sistemas Operativos – DEI - IST

11



## Interblocagem

- Uma situação de interblocagem pode aparecer se as quatro condições seguintes forem **simultaneamente** verdadeiras:
  - Pelo menos um recurso é usado de uma forma não partilhável;
  - Existe pelo menos uma tarefa que detém um recurso e que está à espera de adquirir mais recursos;
  - Os recursos não são “preemptíveis”, ou seja, os recursos apenas são libertados voluntariamente pelas tarefas que os detêm;
  - Existe um padrão de sincronização em que a tarefa  $T_1$  espera por um recurso de  $T_2$  e circularmente  $T_{n-1}$  espera por um recurso de  $T_1$

**É o que sucede no Jantar dos Filósofos**

Sistemas Operativos – DEI - IST

12



## Como resolver a interblocagem no Jantar dos Filósofos?

Sistemas Operativos – DEI - IST

13



## Primeira solução

- Pelo menos um recurso é usado de uma forma não partilhável;
- Existe pelo menos uma tarefa que detém um recurso e que está à espera de adquirir mais recursos;
- Os recursos não são “preemptíveis”, ou seja, os recursos apenas são libertados voluntariamente pelas tarefas que os detêm;
- Existe um padrão de sincronização em que a tarefa  $T_1$  espera por um recurso de  $T_2$  e circularmente  $T_{n-1}$  espera por um recurso de  $T_1$

**Como neste problema as três condições iniciais são verdadeiras  
A solução é evitar que exista um padrão circular !**

Sistemas Operativos – DEI - IST

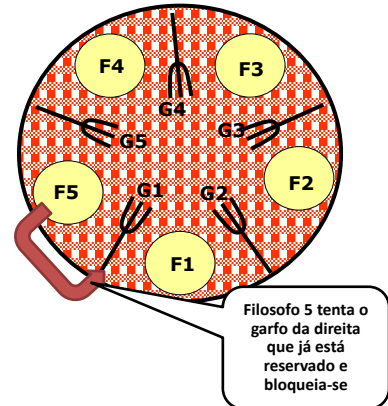
14



## Jantar dos Filósofo: quebrar o ciclo

```
void* philosophers(void* num){
    int* phil = num;
    while (dining) {

        sleep(1); //pensar
        if (*phil !=4) {
            sem_wait(&S[*phil]);
            sem_wait (&S[( *phil +1)%N]);}
        else{
            sem_wait (&S[( *phil +1)%N]);
            sem_wait(&S[*phil]);
        }
        sleep(2); //comer
        sem_post(&S[*phil]);
        sem_post(&S[( *phil +1)%N]);
    }
}
```



O filósofo 5 ( $phil==4$ ) tem um programa diferente que quebra o ciclo de alocação. Mas a solução não é genérica

Sistemas Operativos – DEI - IST

15



## Solução genérica

- Outras formas de quebrar o ciclo é limitar o numero de filósofos a comer. Se este for inferior a  $N-1$  o ciclo não se fecha (facilmente implementável com um semáforo inicializado a  $N-1$ )
- Contudo, estas soluções não são genéricas
- Para conseguir ter uma programação genérica temos de usar condições para determinar se o filósofo tem condições de ter os dois garfos ou no caso de apenas conseguir um, liberta-lo para assim permitir que outro possa prosseguir (comer)

Sistemas Operativos – DEI - IST

16





## Solução genérica

- Vamos agora considerar que os semáforos não estão associados ao recurso garfo, mas que cada filósofo tem um semáforo próprio onde espera o evento “pode comer” (semáforo inicializado a 0)
- O evento pode ser despoletado porque um dos seus parceiros à esquerda ou à direita acabou de comer ou porque nenhum deles está a comer

Sistemas Operativos – DEI - IST

17



## Jantar dos Filósofos com Semáforos

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

### Filosofo

```
void* philosopher(void* num)
{
    int* iphil = num;
    state[*iphil] = THINKING;
    while (dining) {
        sleep(rand()%2);
        take_fork(*iphil);
        sleep(rand()%3);
        put_fork(*iphil);
    }
}
```

### Com fome quer os garfos

```
void take_fork(int phnum) {
    pthread_mutex_lock(&trinco);
    // estado com Fome pretende comer
    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);
    // Come se os vizinhos não estão a comer
    test(phnum);
    pthread_mutex_unlock(&trinco);
    // Teste se o evento para comer foi despoletado ou
    bloqueia-se
    sem_wait(&S[phnum]);
}
```

Bloqueia-se à espera que lhe seja assinalado que pode comer

Sistemas Operativos – DEI - IST

18



## Testar a condição que lhe permite comer

```
void test(int phnum)
{
    if (state[phnum] == HUNGRY
        && state[LEFT] != EATING
        && state[RIGHT] != EATING) {

        state[phnum] = EATING; //a comer

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

        sem_post(&S[phnum]);

    }
}
```

Condição que testa se o filósofo à esquerda não está a comer e se o da direita também não (o teste se o filósofo esta *hungry* é redundante vermos a seguir para que serve

Se a condição é verdadeira então pode comer porque os dois garfos estão livres e coloca-se no estado *eating*

Assinala o evento "pode comer" no semáforo do filósofo

Sistemas Operativos – DEI - IST

19



### Test

```
void test(int phnum)
{
    if (state[phnum] == HUNGRY
        && state[LEFT] != EATING
        && state[RIGHT] != EATING) {

        state[phnum] = EATING; //a
        comer

        sem_post(&S[phnum]);

    }
}
```

### Take\_fork

```
void take_fork(int phnum) {
    pthread_mutex_lock(&trincos);

    // estado com Fome pretende comer
    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);
    // Come se os vizinhos não estão a comer
    test(phnum);
    pthread_mutex_unlock(&trincos);
    // Testa se o evento foi despoletado ou bloqueia-se
    sem_wait(&S[phnum]);
}
```

Pode parecer estranho, mas neste caso a tarefa quando vê que as condições para poder comer estão reunidas assinala o semáforo que irá testar em seguida, quando a condição é falsa não faz nada pelo que irá bloquear-se

Sistemas Operativos – DEI - IST

20



## Libertar os garfos

```
void put_fork(int phnum)
{
    pthread_mutex_lock(&trinco);
    state[phnum] = THINKING; //fica a Pensar
    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); //verifica se pode libertar o vizinho à esquerda
    test(RIGHT); //verifica se pode libertar o vizinho à direita
    pthread_mutex_unlock(&trinco);
}
```

Despoleta eventos para os semáforos em que os vizinhos podem estar bloqueados à espera dos garfos. Como os semáforos tem memória só o faz se o vizinho está no estado *hungry* situação em de certeza está bloqueado à espera deste grafo que vai ser libertado (ver a função test)

Sistemas Operativos – DEI - IST

21



## Conclusão

A interblocagem é um problema complexo, mas intrínseco da sincronização

Em primeiro lugar é necessário perceber em que condições pode ocorrer

E que métodos de programação podemos usar para evitar o seu aparecimento

Sistemas Operativos – DEI - IST

22