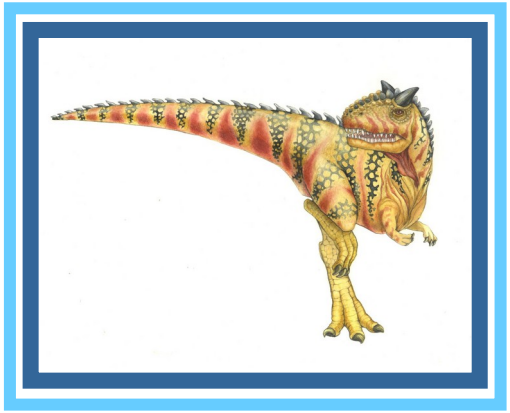
Problemas clássicos de deadlocks e starvation



Deadlock

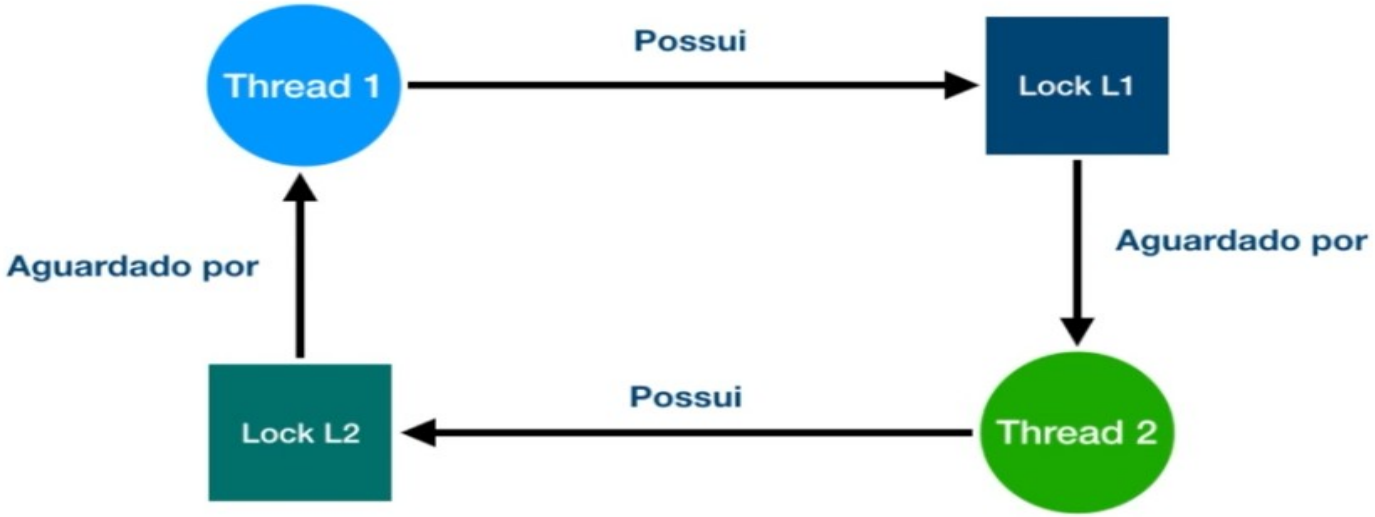
Um exemplo de código com possibilidade de deadlock.

Thread 1

```
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
```

Thread 2

```
pthread_mutex_lock(L2);
pthread_mutex_lock(L1);
```



Starvation x Deadlocks

- Starvation
  - Uma thread não consegue progredir por um período de tempo indefinido

    Mas pode finalizar.
- Deadlock é uma forma de starvation, mas com uma condição mais forte
  - Um grupo de threads espera em ciclo
  - Nenhuma das threads progride

Porque deadlocks ocorrem?

- Sistemas possuem múltiplas dependências de recursos
  - P.ex., no S.O. diversos processos compartilham o disco, a memória, interfaces de rede, etc.
- Rotinas de teste podem não detectar problemas de deadlock
  - Situação pode ser rara

Jantar do Selvagens

Uma tribo de selvagens está jantando ao redor de um grande caldeirão contendo N porções de missionário cozido.
Quando um selvagem quer comer, ele se serve de uma porção no caldeirão, a menos que este esteja vazio. Nesse caso, o selvagem primeiro acorda o cozinheiro da tribo e espera que ele encha o caldeirão de volta, para então se servir novamente. Após encher o caldeirão, o cozinheiro volta a dormir.

Jantar do Selvagens

```
1  task cozinheiro ()
2  {
3     while (1)
4     {
5        encher_caldeirao () ;
6        dormir () ;
7     }
8  }
9
10 task selvagem ()
11 {
12    while (1)
13    {
14       servir () ;
15       comer () ;
16    }
17 }
```

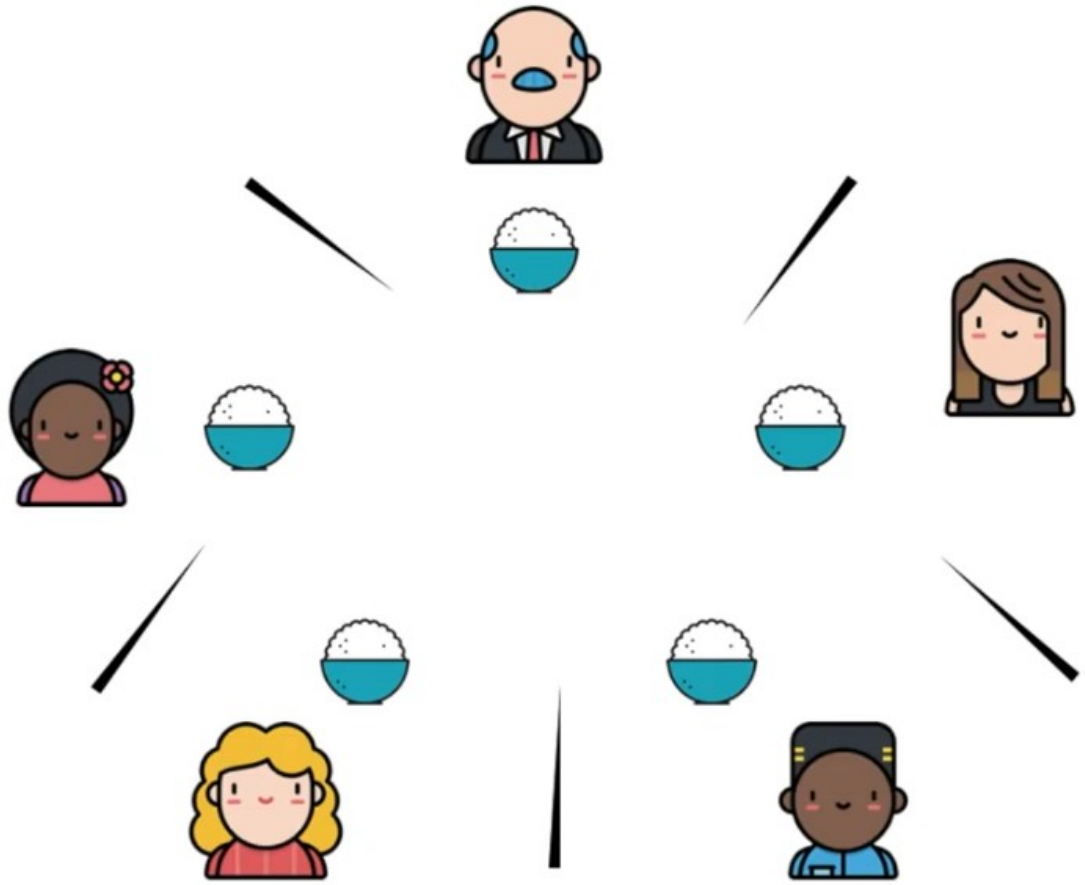
As restrições de sincronização deste problema são as seguintes:

- Selvagens não podem se servir ao mesmo tempo (mas podem comer ao mesmo tempo);
- Selvagens não podem se servir se o caldeirão estiver vazio;
- O cozinheiro só pode encher o caldeirão quando ele estiver vazio.

Jantar do Selvagens

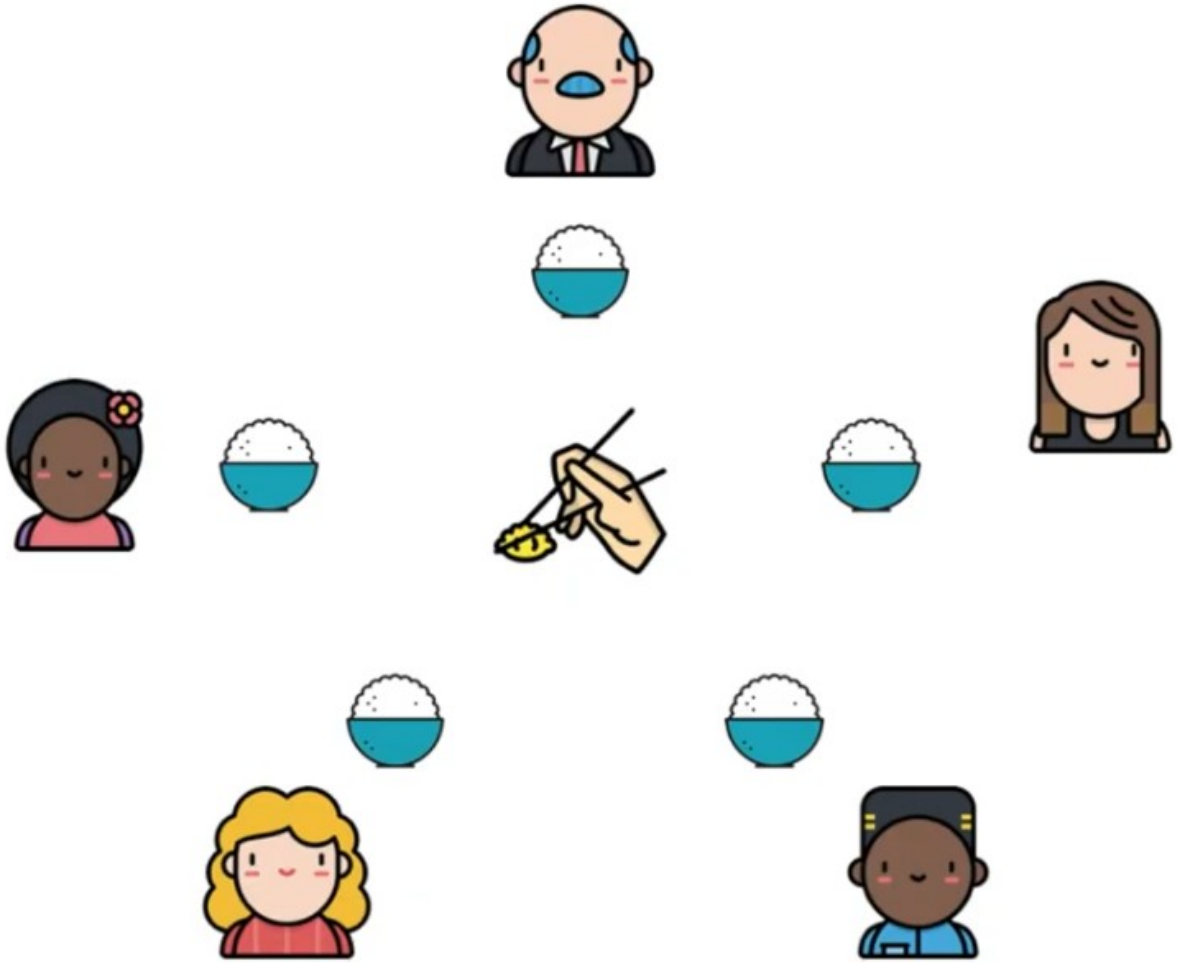
```
1  int porcoes = 0 ;              // porções no caldeirão
2  mutex mc ;                     // controla acesso ao caldeirão
3  semaphore cald_vazio ;         // indica caldeirão vazio (inicia em 0)
4  semaphore cald_cheio ;         // indica caldeirão vazio (inicia em 0)
5
6  task cozinheiro ()
7  {
8     while (1)
9     {
10       down (cald_vazio) ;      // aguarda o caldeirão esvaziar
11       porcoes += M ;           // enche o caldeirão (M porções)
12       up (cald_cheio) ;        // avisa que encheu o caldeirão
13    }
14 }
15
16
17 task selvagem ()
18 {
19    while (1)
20    {
21       lock (mc) ;              // tenta acessar o caldeirão
22       if (porcoes == 0)        // caldeirão vazio?
23       {
24          up (cald_vazio) ;     // avisa que caldeirão esvaziou
25          down (cald_cheio) ;   // espera ficar cheio de novo
26       }
27       porcoes-- ;              // serve uma porção
28       unlock (mc) ;            // libera o caldeirão
29       comer () ;
30    }
31 }
```

Problemas dos filósofos



Os 5 filósofos (processos) comem ou pensam.
Mas tem só 5 hashis.

Problemas dos filósofos



Problemas dos filósofos

Condições necessárias para um deadlock

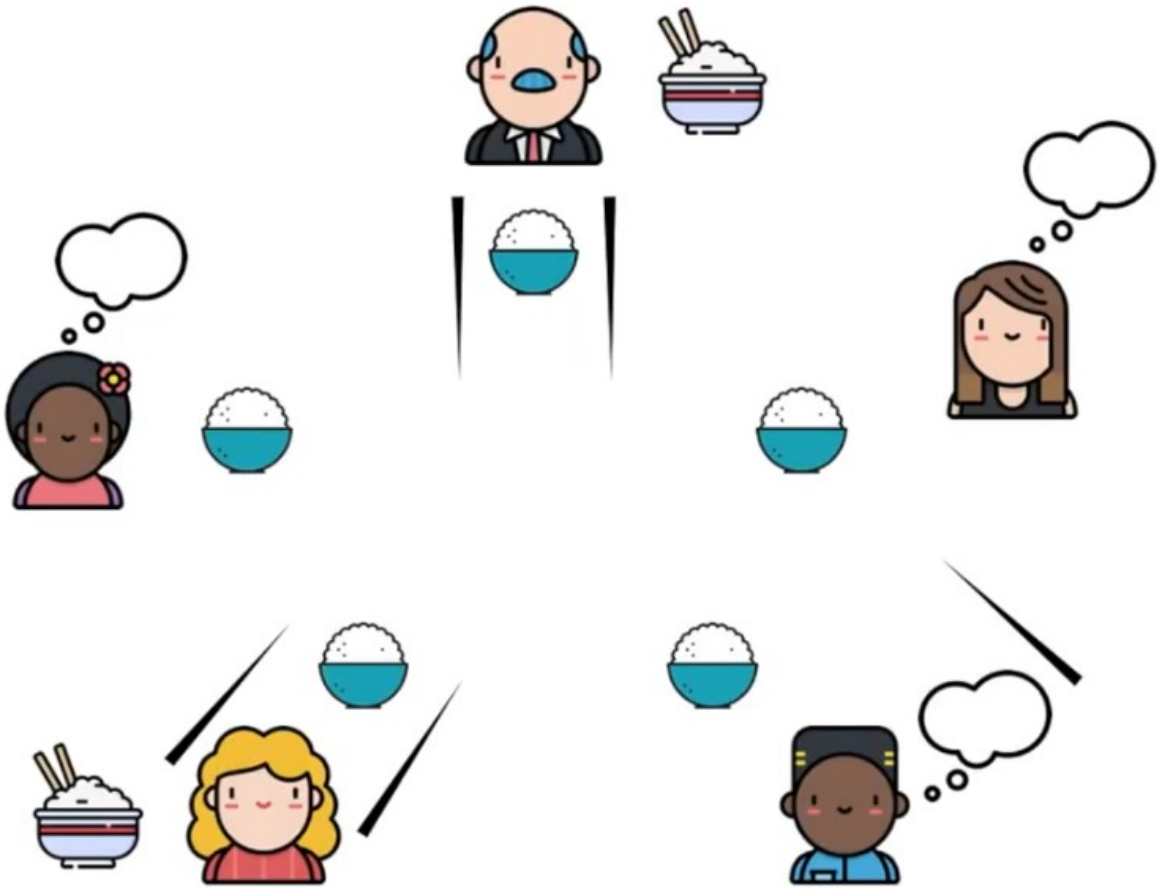
- Exclusão mútua
- Reservar e esperar
- Não preempção
- Espera circular

Problema dos filósofos

Exclusão mútua

- Threads necessitam de um controle exclusivo dos recursos que elas necessitam
- P.ex., uma thread adquire um lock

Problema dos filósofos



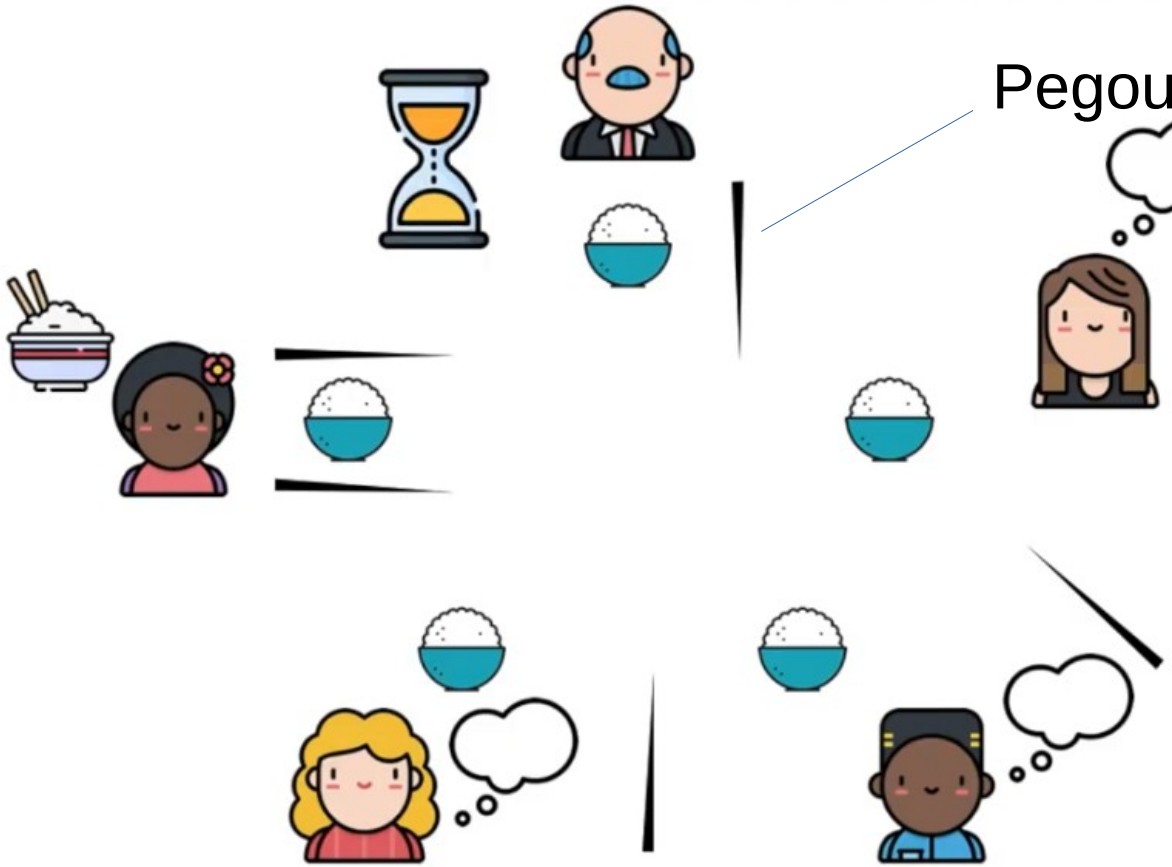
Exclusão mútua

Problema dos filósofos

Reservar e esperar

- Uma thread adquire e toma posse de um recurso enquanto espera por outro recurso
- P.ex., adquire um lock, mas precisa de outros locks para continuar

Problema dos filósofos



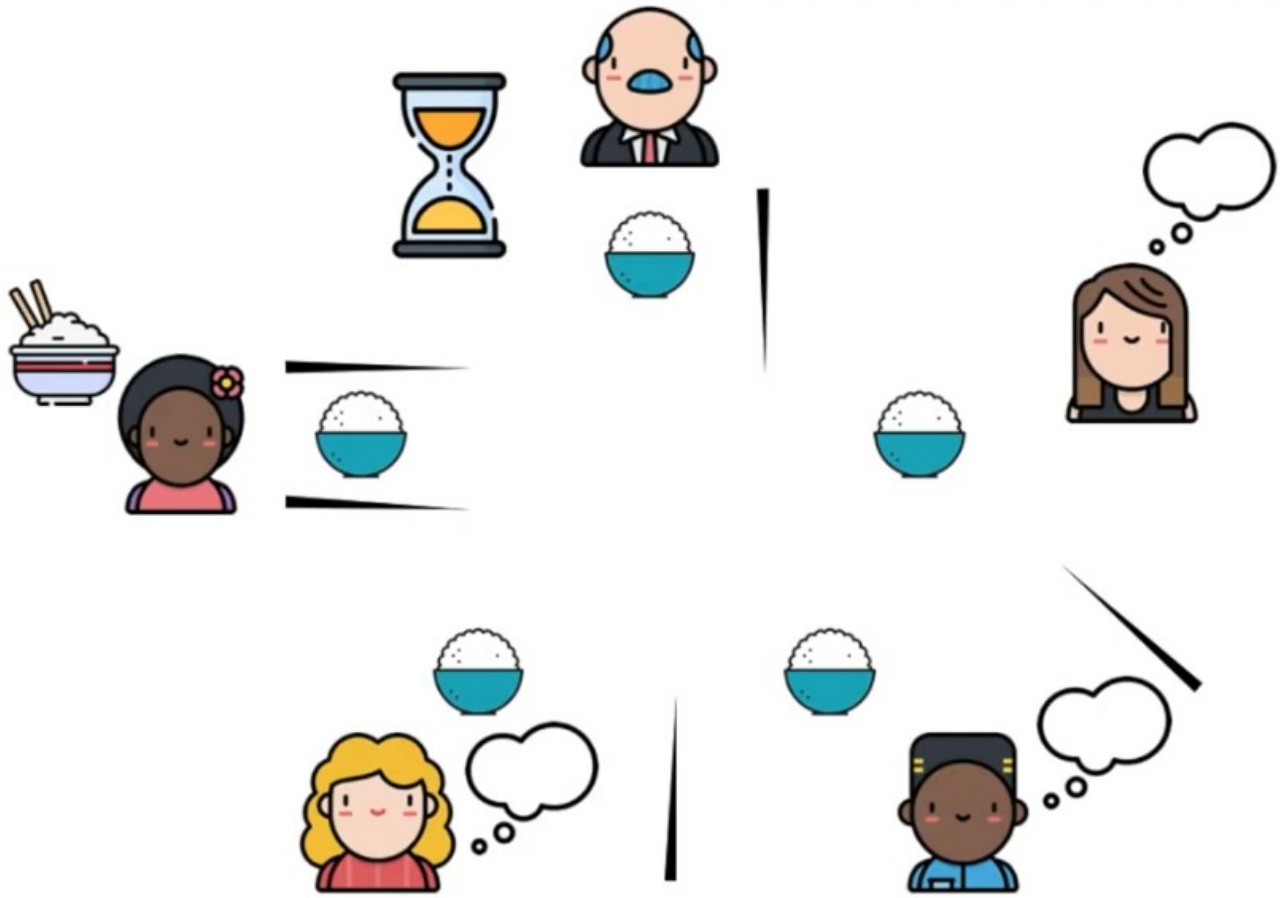
Reservar e esperar

Problema dos filósofos

Não preempção

- Se uma thread adquirir um recurso, o SO não pode retirar dela esse recurso
- SO deve esperar a thread liberá-lo
- P.ex., se uma thread está imprimindo uma folha de impressora, ela não pode ser interrompida

Problema dos filósofos



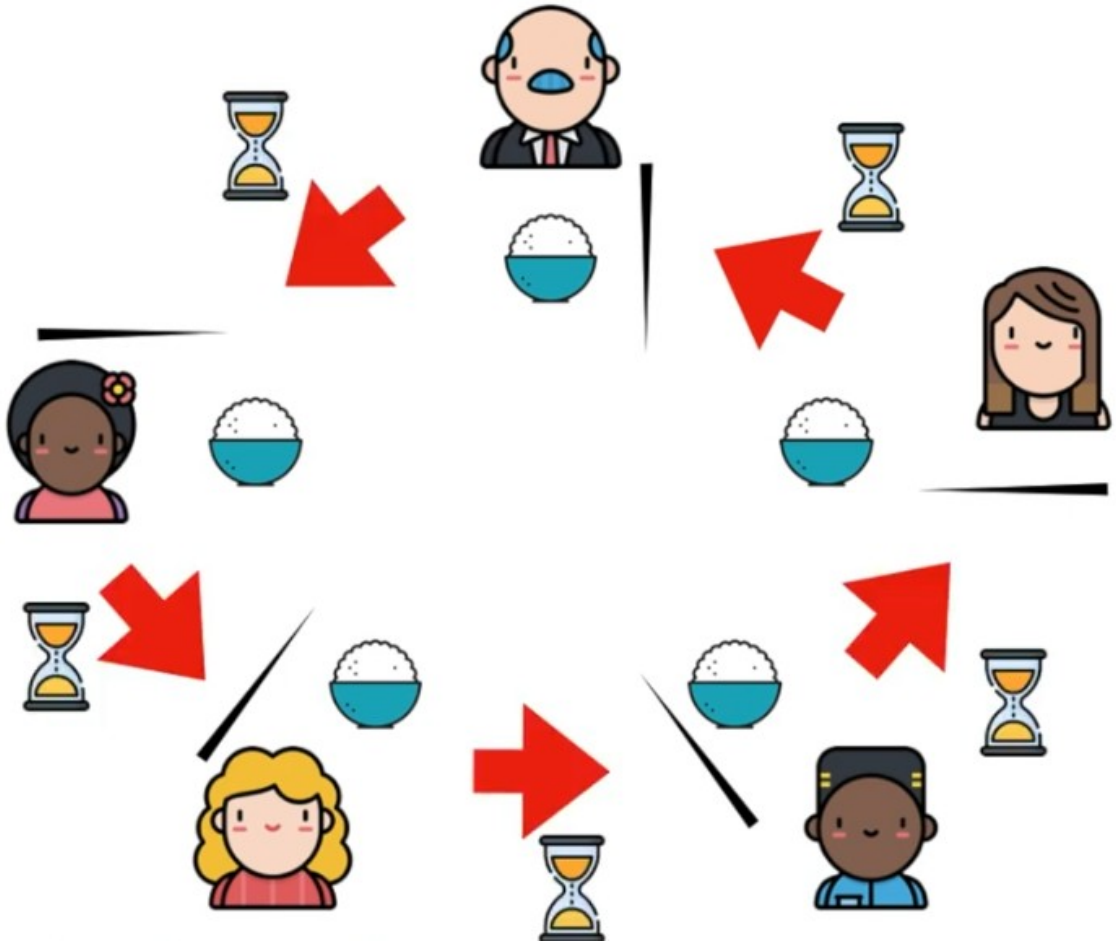
Não preempção

Problemas dos filósofos

Espera circular

- Existe uma cadeia circular de threads
- Cada thread na cadeia possui um ou mais recursos que são necessários pela próxima thread na cadeia

Problemas dos filósofos



Espera circular

Cada filósofo espera que o hashi da direita seja liberado.

Problemas dos filósofos

Como lidar com deadlocks?

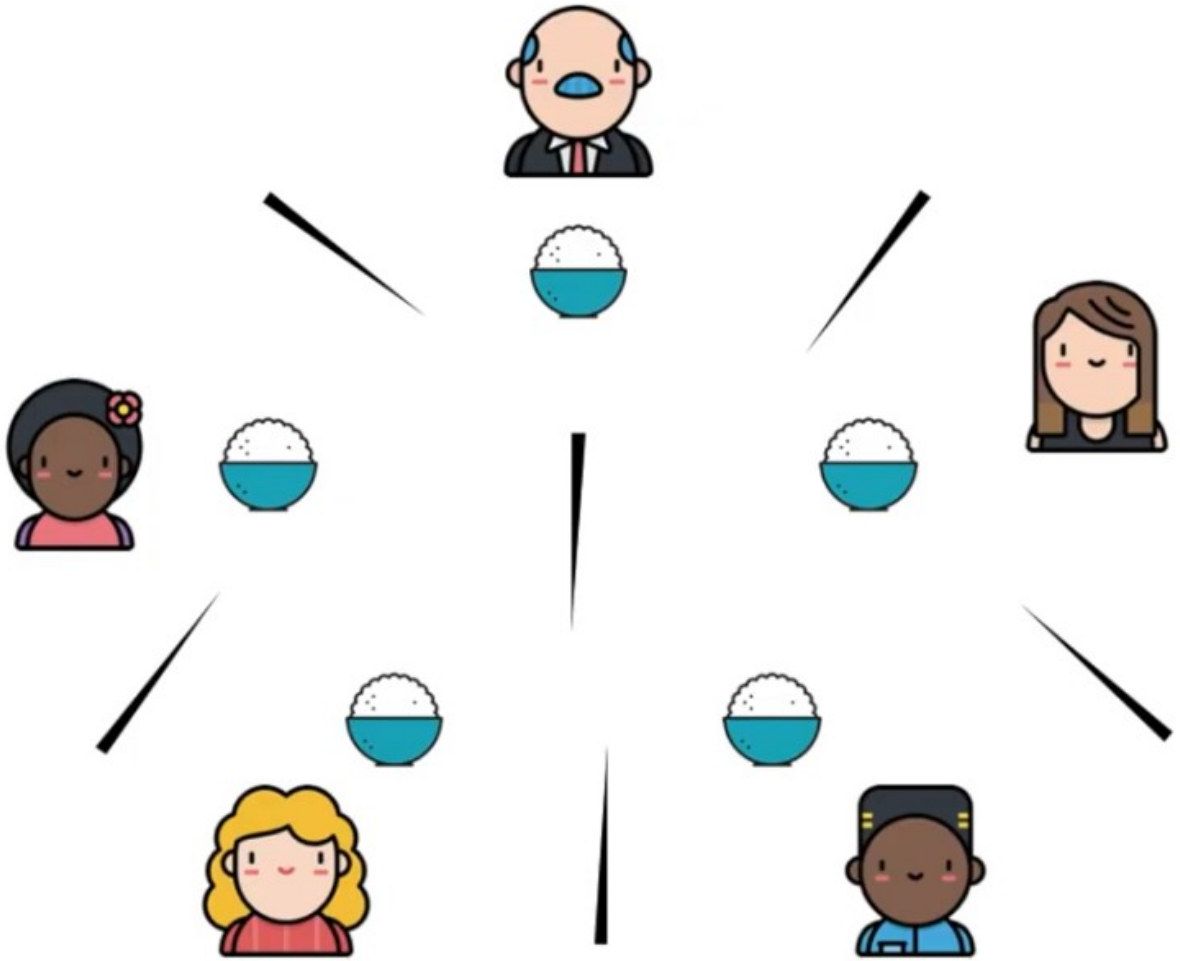
- Prevenir deadlocks
- Evitar deadlocks
- Detectar e recuperar deadlocks

Problemas dos filósofos

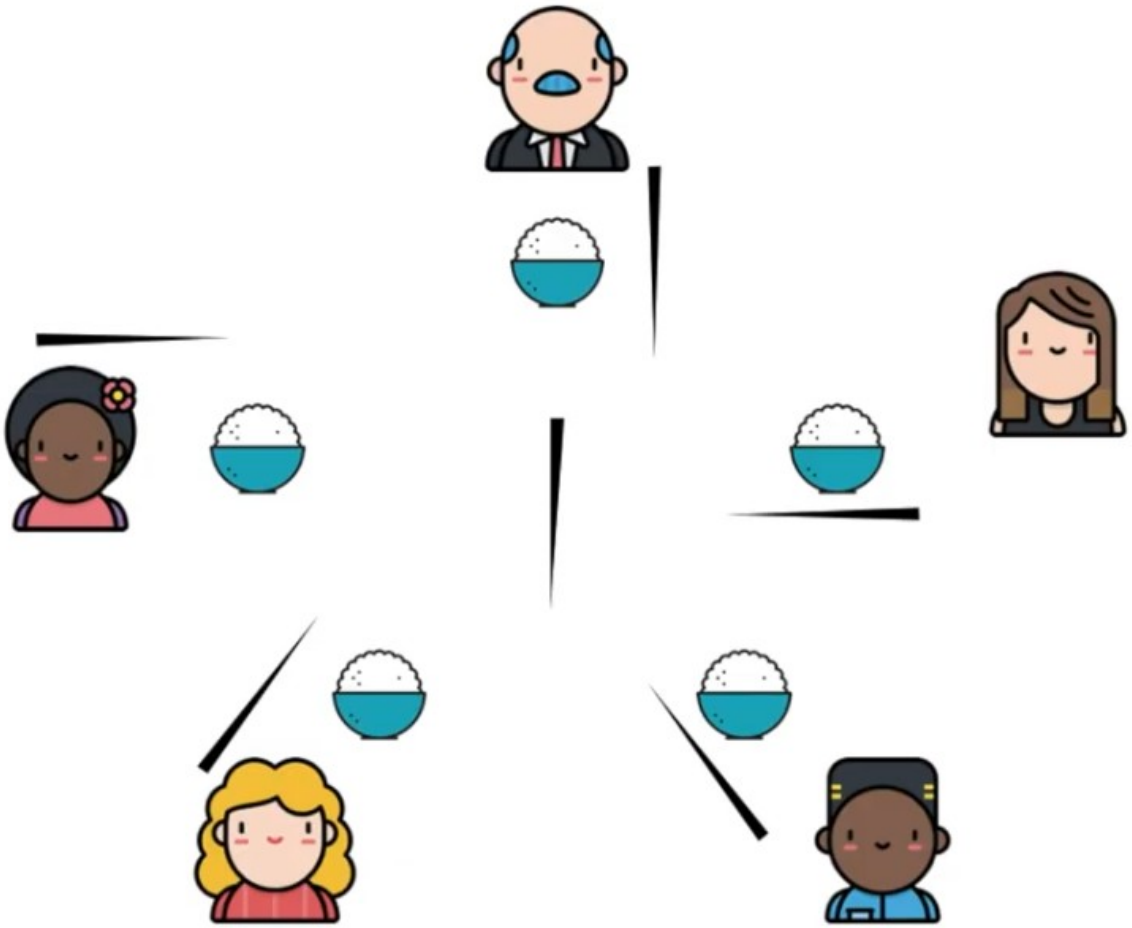
Prevenir deadlocks

- Garantir uma das condições para deadlock não será satisfeita
- Para não satisfazer exclusão mútua
  - Dar mais recursos

Problemas dos filósofos

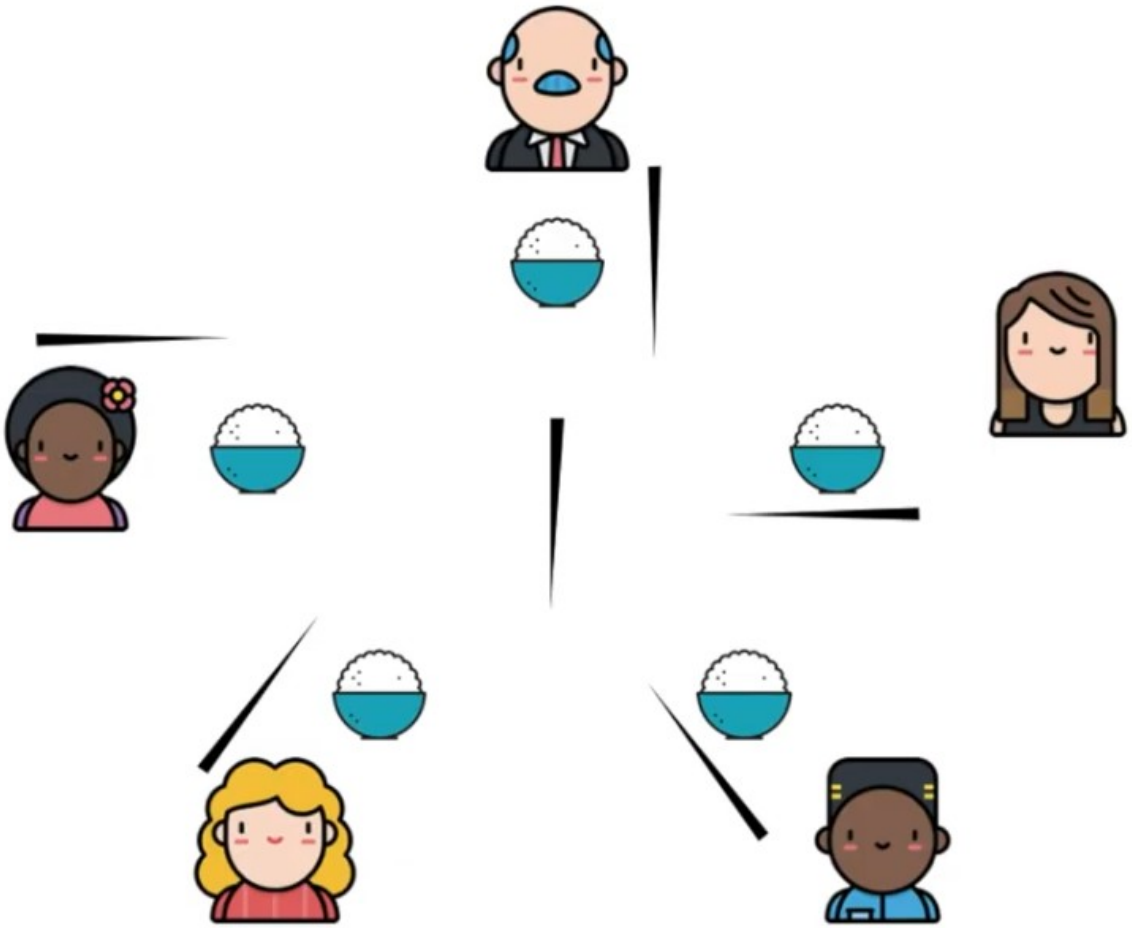


Problemas dos filósofos



Cada filósofo com um hashi e não solta.
Mas tem um hashi (recurso) a mais. Algum filósofo vai pegar e se alimentar.

Problemas dos filósofos



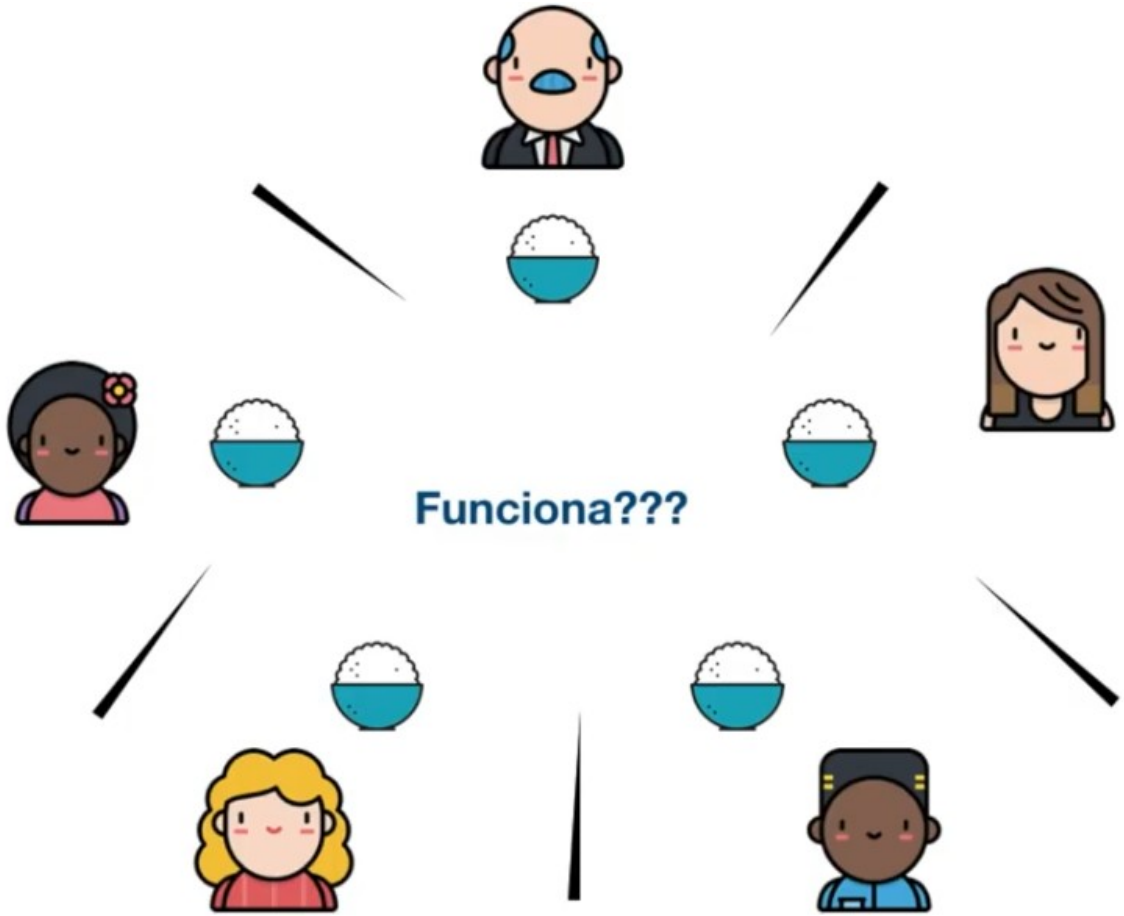
Cada filósofo com um hashi e não solta.
Mas tem um hashi (recurso) a mais. Algum filósofo vai pegar e se alimentar.

Problemas dos filósofos

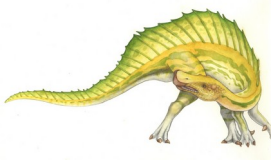
Prevenir deadlocks

- Garantir uma das condições para deadlock não será satisfeita
- Para não satisfazer reservar e esperar
  - Liberar recursos se não conseguir todos os necessários

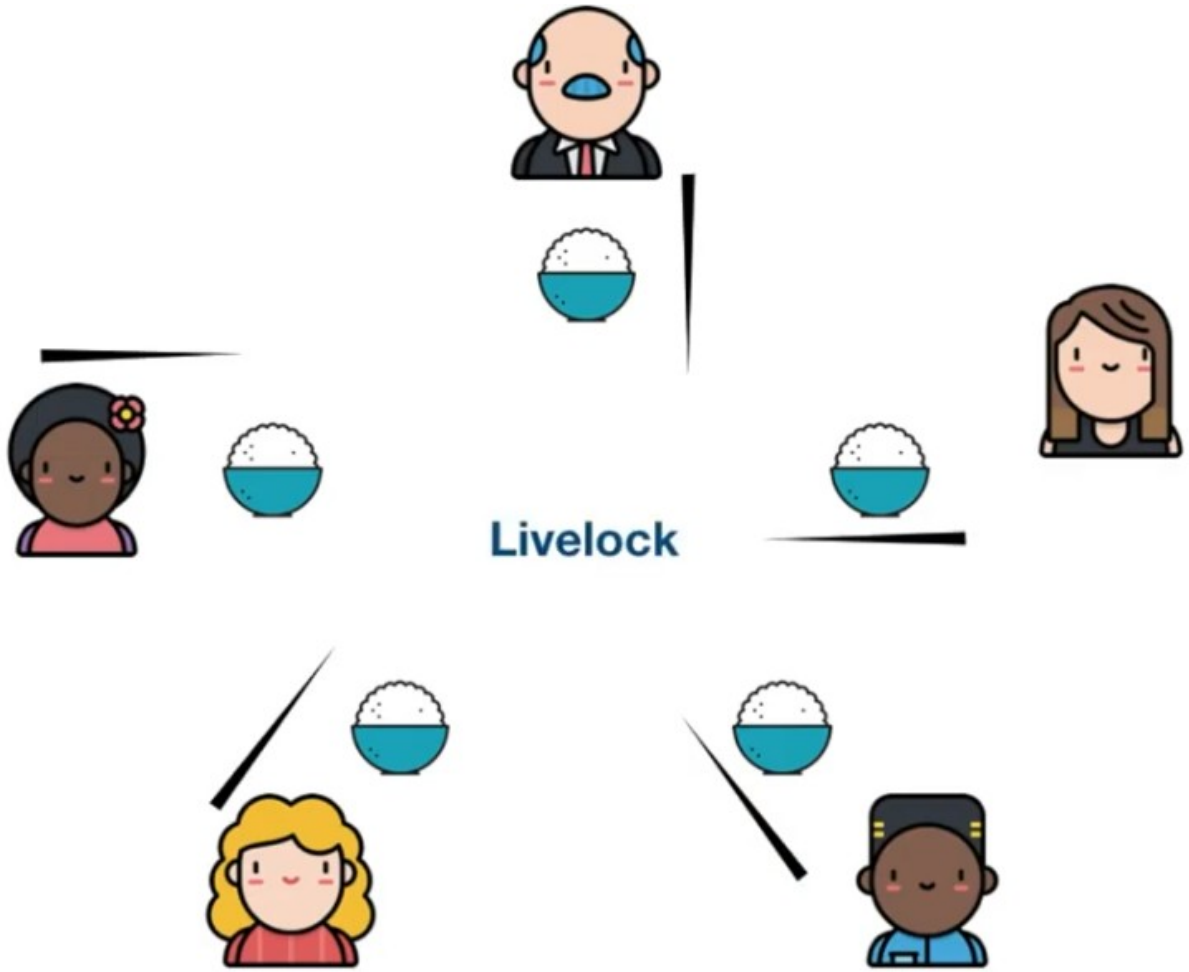
Problemas dos filósofos



Libera o hashi se não conseguir o segundo.
Causa LIVELOCK: cria um loop de solta e pega o hashi.



Problemas dos filósofos

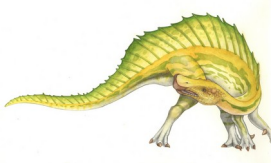


Problemas dos filósofos

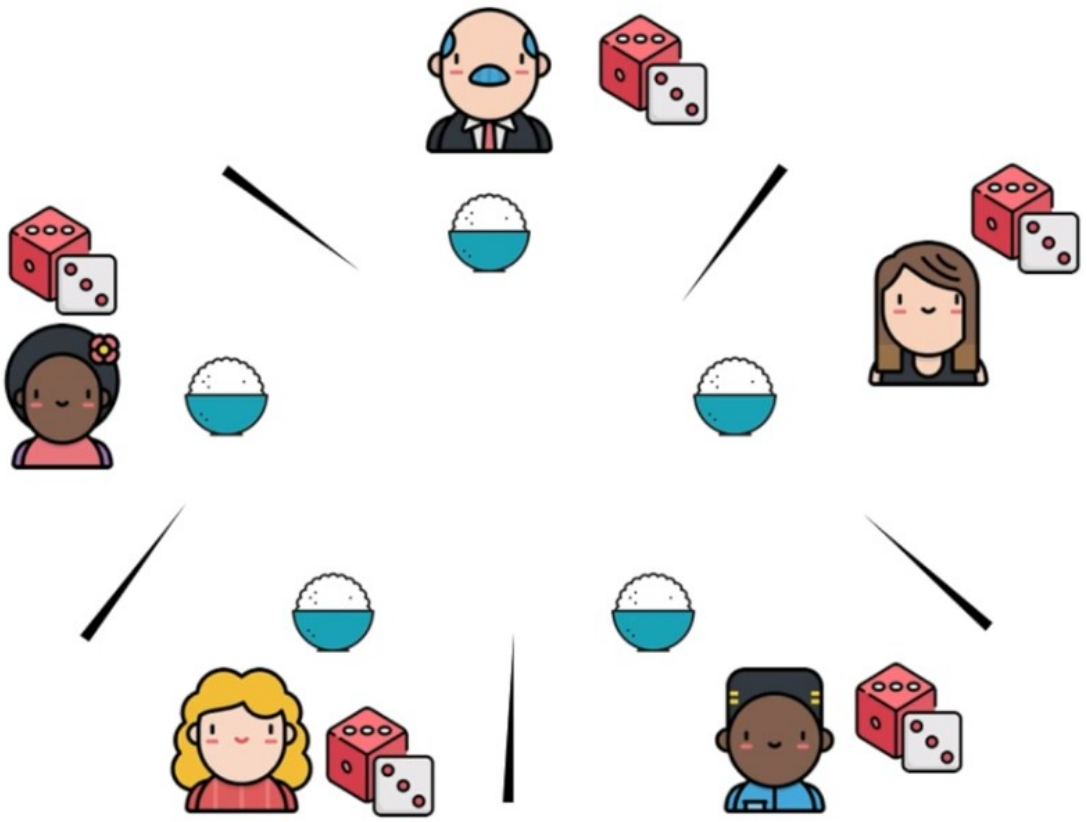
Livelock

- Threads tentam repetidamente realizar uma sequência de utilização de recursos

- Possível solução
  - Esperar um tempo aleatório antes de pegar o hashi
  - Não garante a inexistência, mas reduz sua probabilidade
  - Wi-Fi utiliza mecanismo parecido para compartilhar ponto de acesso

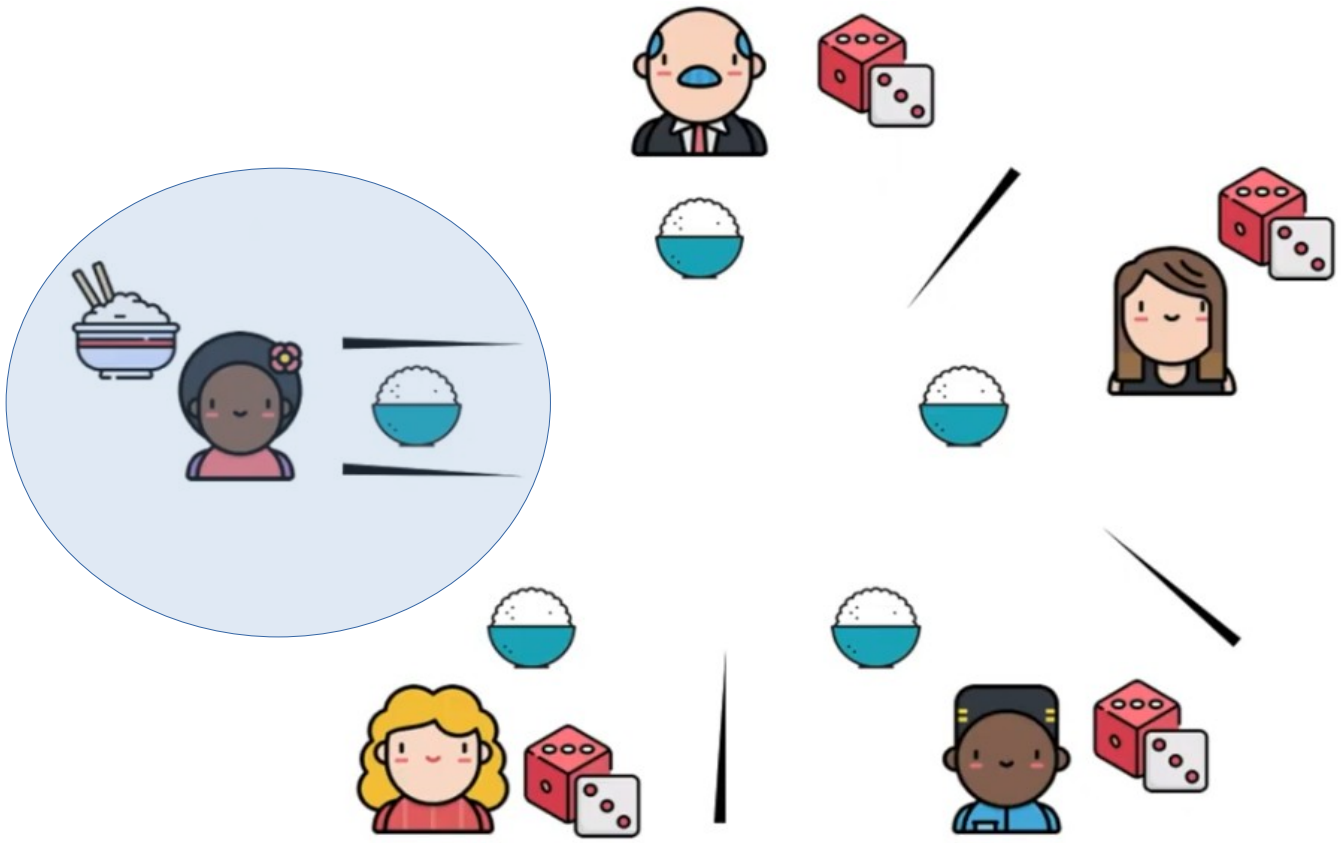


Problemas dos filósofos



Filósofos tentam pegar os hashis, como não conseguem, liberam. Antes de tentar pegar um hashi sorteiam um tempo aleatório.

Problemas dos filósofos



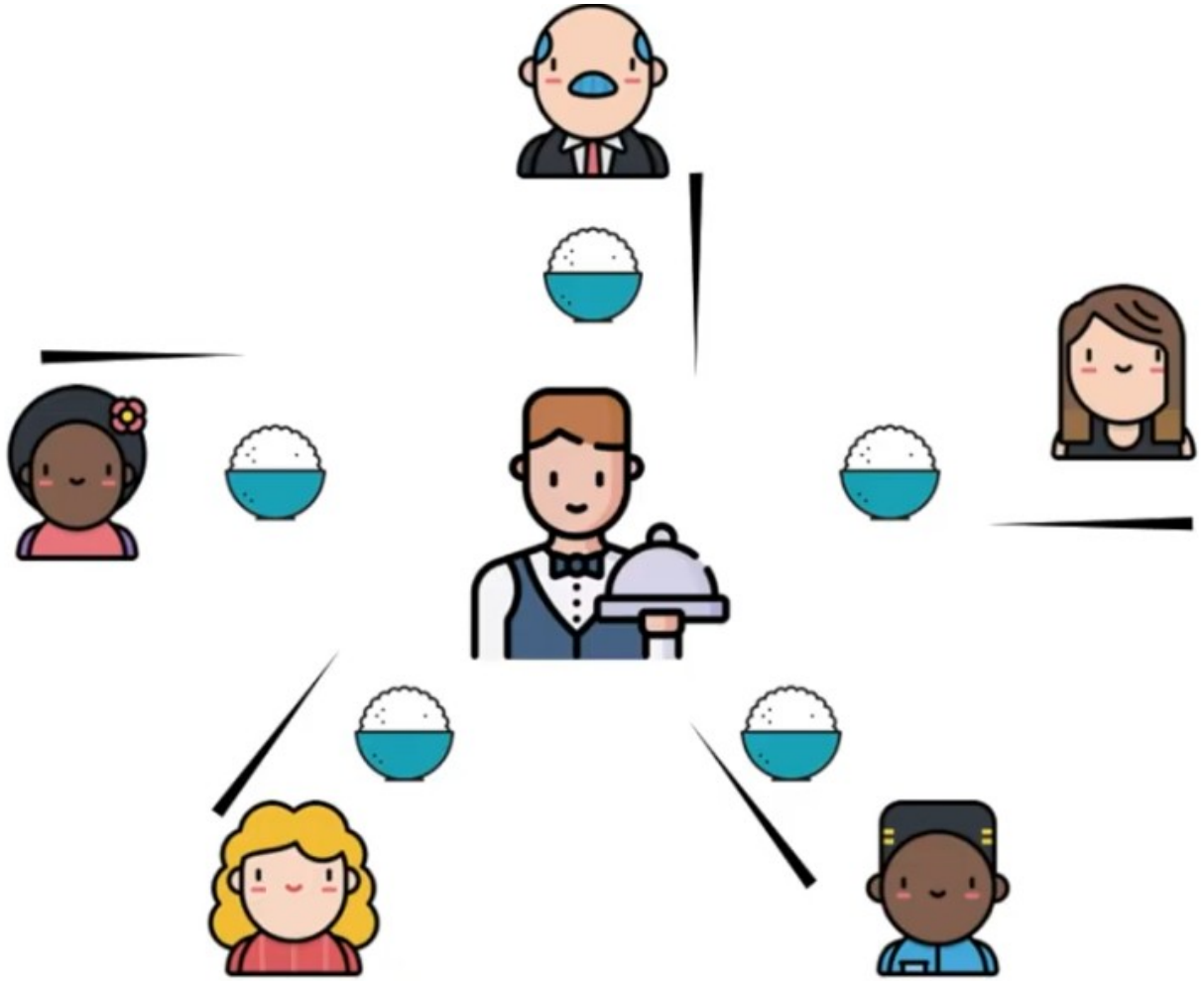
Primeira filósofa sorteada com tempo menor

Problemas dos filósofos

Prevenir deadlocks

- Garantir uma das condições para deadlock não será satisfeita
- Não preempção
  - SO pode retirar o recurso de uma thread

Problemas dos filósofos



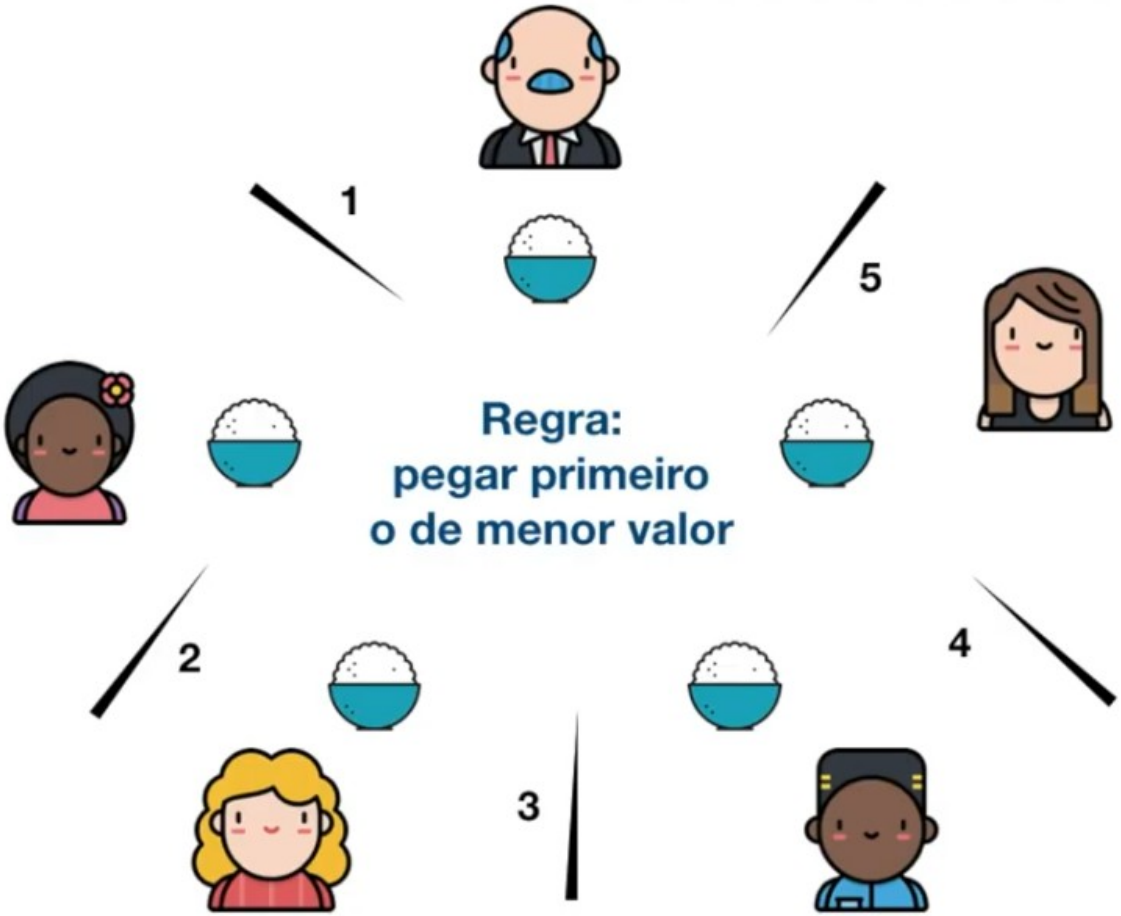
Se ocorreu deadlock o garçom tira o hashi de um filósofo

Problemas dos filósofos

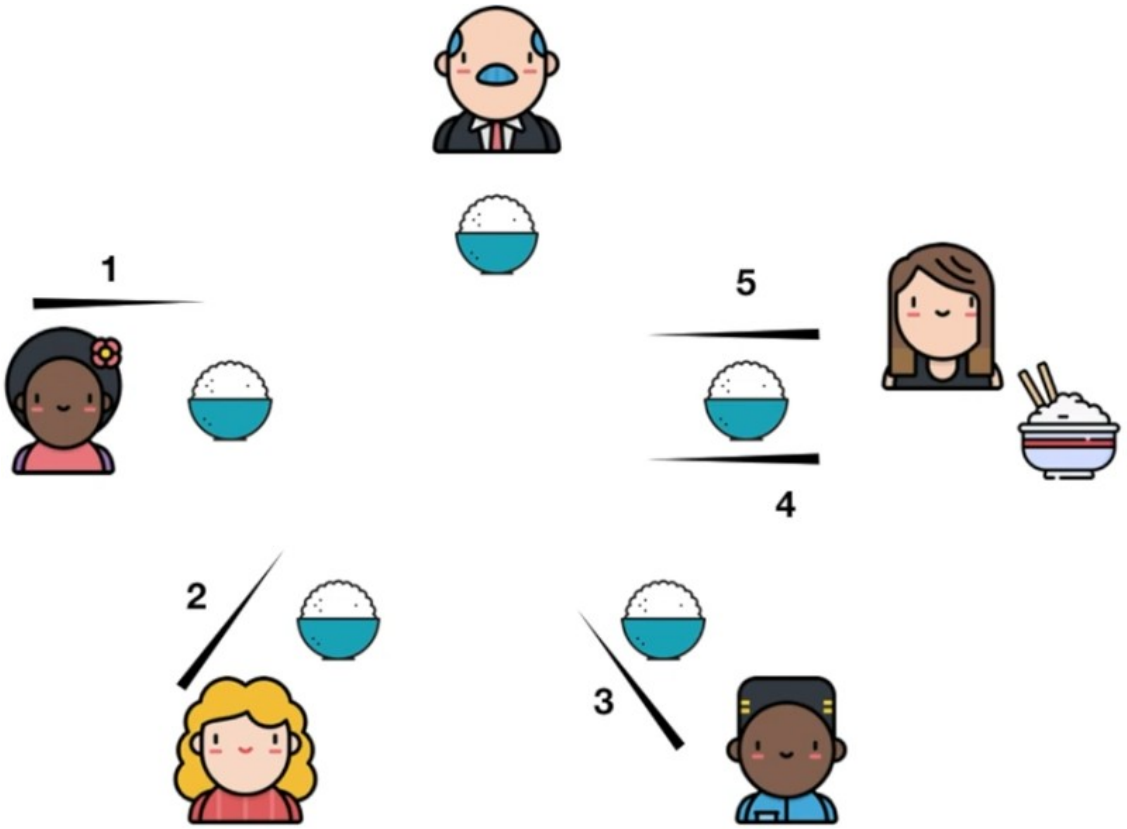
Prevenir deadlocks

- Garantir uma das condições para deadlock não será satisfeita
- Para não satisfazer espera circular
  - Utilização de recursos segue uma certa ordem

Problemas dos filósofos



Problemas dos filósofos



O filóso de cima tenta pegar o hashi 1, mas está ocupado. Logo, ele não pode pegar o 5. Então a filósofa pode pegar os hashis 4 e 5.

Problemas dos filósofos

Como lidar com deadlocks ?

- Prevenir deadlocks

  Responsabilidade do programador.

- **Evitar deadlocks**

- Detectar e recuperar deadlocks

Problemas dos filósofos

Evitar deadlocks

- Entidade externa (SO) possui conhecimento global do uso de recursos
- Liberação de recursos pela entidade ocorre de forma a evitar deadlocks
- Jantar dos filósofos
  - Garçom permite filósofo pegar hashi se
    - Não é o último hashi
    - É o último, mas alguém está com dois hashi

Problemas dos filósofos

Evitar deadlocks

- P.ex., algoritmo do banqueiro
  - Banker's algorithm - Dijkstra
- Muito difícil de implementar
  - Exige um conhecimento global dos recursos e da necessidade de recursos das threads

Problemas dos filósofos

Detectar e recuperar

- Assumir a ocorrência eventual de deadlocks
- Recuperar quando ocorrer
- P.ex., reiniciar o computador quando ocorrer

  Em sistemas não críticos. Não se faz em um avião
- P.ex., deteção e retrocesso (rollback) em sistemas de bancos de dados

Problemas dos filósofos

CONCLUSÃO

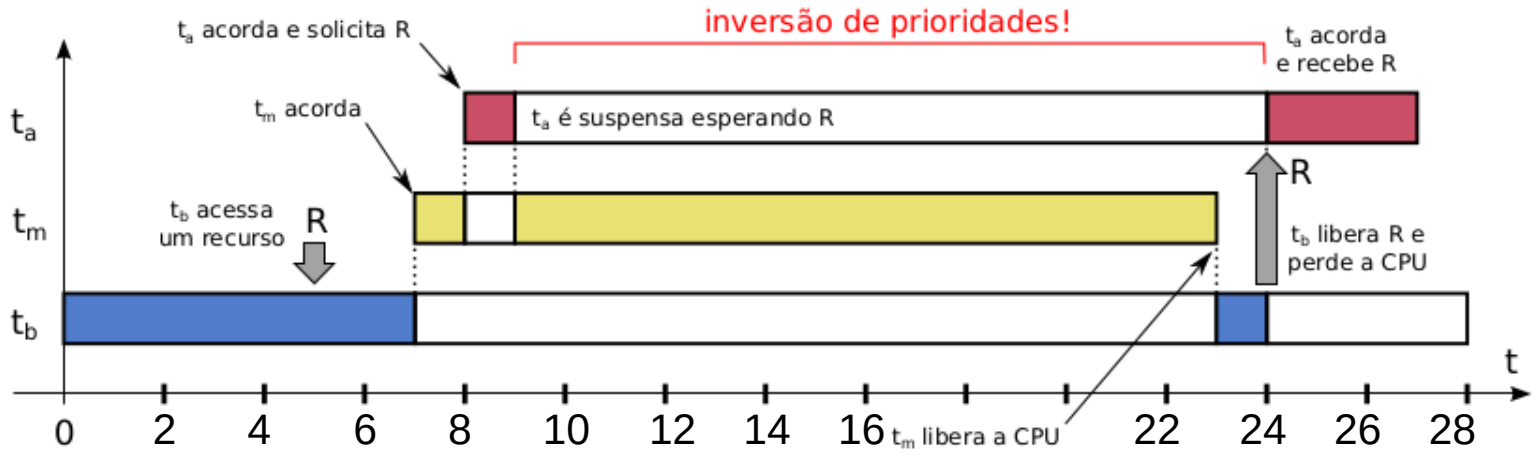
- Quatro situações necessárias
  - Exclusão mútua
  - Reservar e esperar
  - Não preempção
  - Espera circular

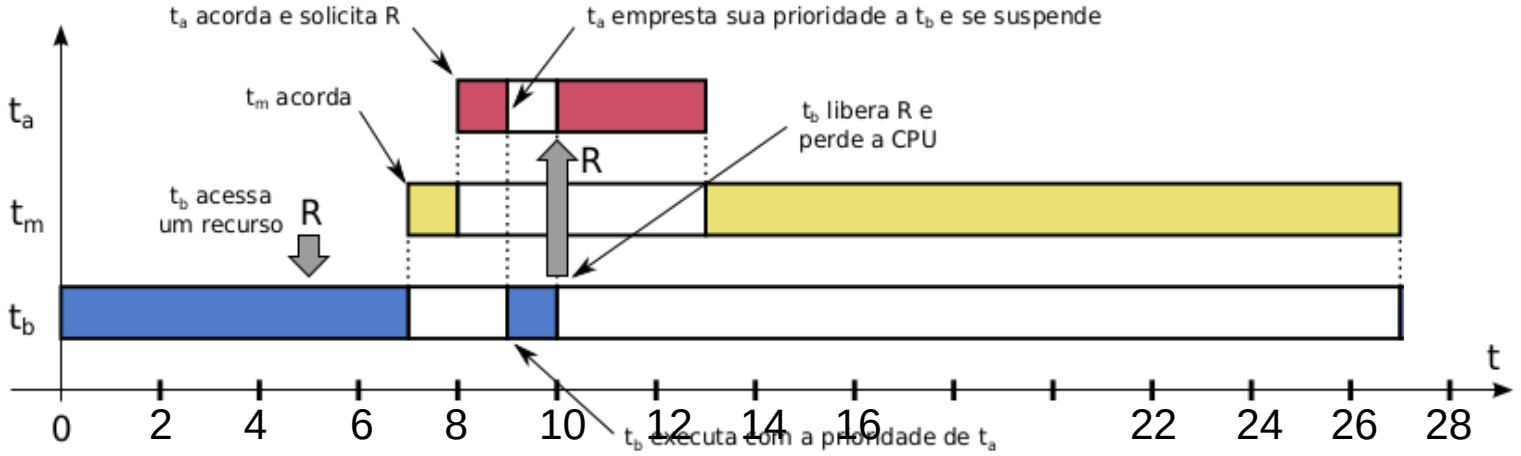
Atacando uma destas podemos resolver deadlock

- Às vezes, a melhor solução é reiniciar o computador

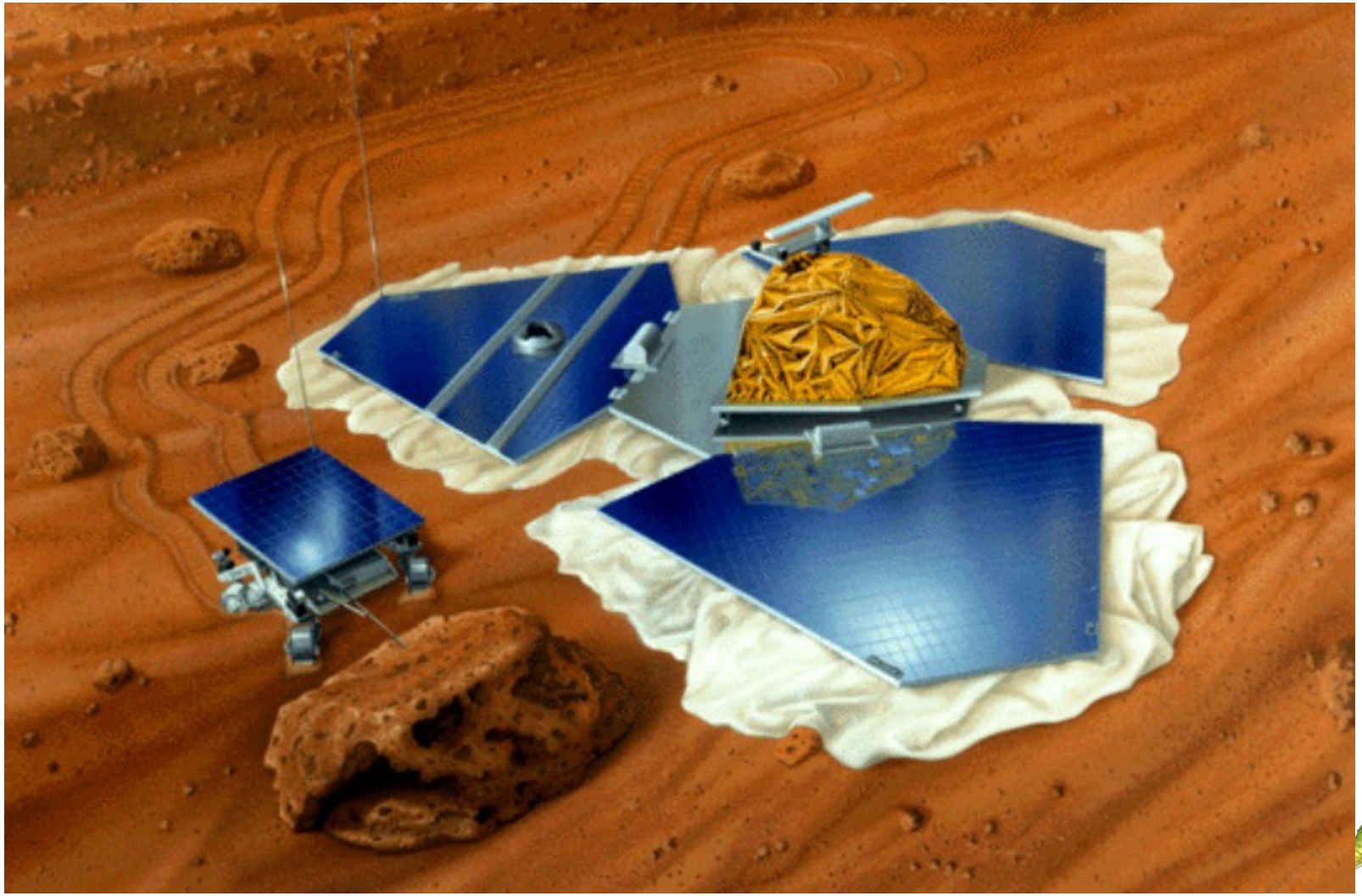
Mars Path Finder

Inversão e herança de prioridades





Mars Path Finder



Mars Path Finder

tarefa	função	prioridade	duração
t_{ger}	gerência da área de transferência	alta	curta
t_{met}	coleta de dados meteorológicos	baixa	curta
t_{com}	comunicação com a Terra	média	longa