Sistemas Operacionais

Módulo 5 – Problemas Clássicos de IPC e Deadlock

Prof. Daniel Sundfeld
daniel.sundfeld@unb.br

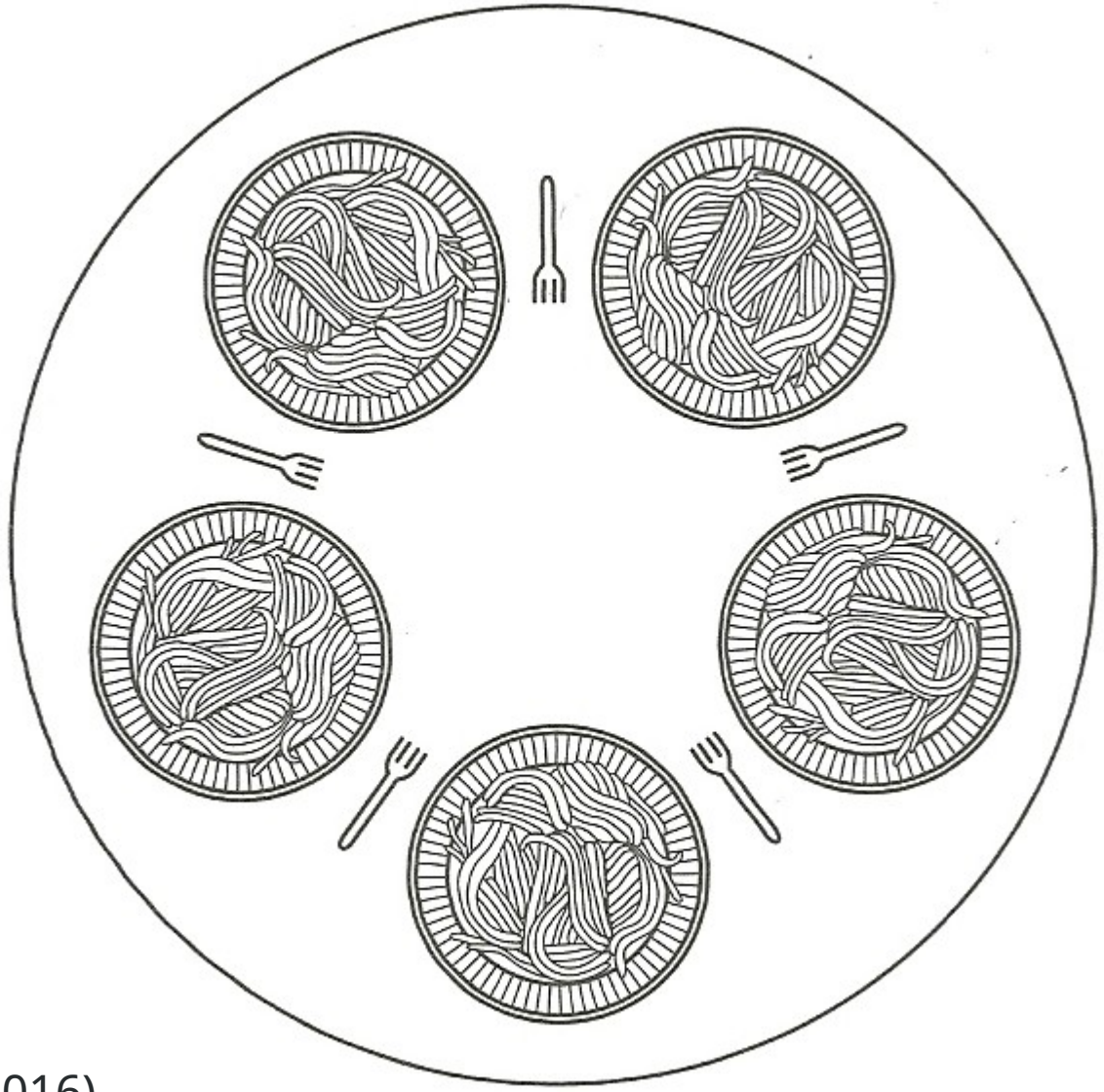
Introdução

- Vimos soluções para comunicação entre threads
- IPC – Inter-Process Communication
- Mecanismos que permite os processos se comunicarem entre si
- Existem diversos problemas na literatura, alguns deles são considerados clássicos
- Vamos estudar alguns dos problemas mais conhecidos

O problema do jantar dos filósofos

- Dijkstra formulou e resolveu o problema do jantar dos filósofos (1965)
- Cinco filósofos estão sentados em torno de uma mesa circular
- Cada filósofo tem um prato de espaguete
- O espaguete é escorregadio e cada filósofo precisa utilizar dois garfos para comer
- Existe um par de pratos entre os garfos

O PROBLEMA DO JANTAR DOS FILÓSOFOS



Fonte: Tanenbaum (2016)

O problema do jantar dos filósofos

- Apesar de anti-higiênico, um filósofo compartilha seus garfos com o filósofo à direita e esquerda
- Se cada filósofo tivesse dois garfos, o problema não existiria
- A vida de um filósofo consiste em alternar períodos de pensar e alimentar
- Depois que um filósofo pensa durante muito tempo, ele tentará se alimentar

O problema do jantar dos filósofos

- Para se alimentar, o filósofo tenta pegar um garfo à sua esquerda e à sua direita, em qualquer ordem
- Se for bem-sucedido em pegar os dois garfos, ele se alimenta
- Problema: você consegue elaborar um problema de forma que todos os filósofos pensem, se alimentem e o programa nunca trava?

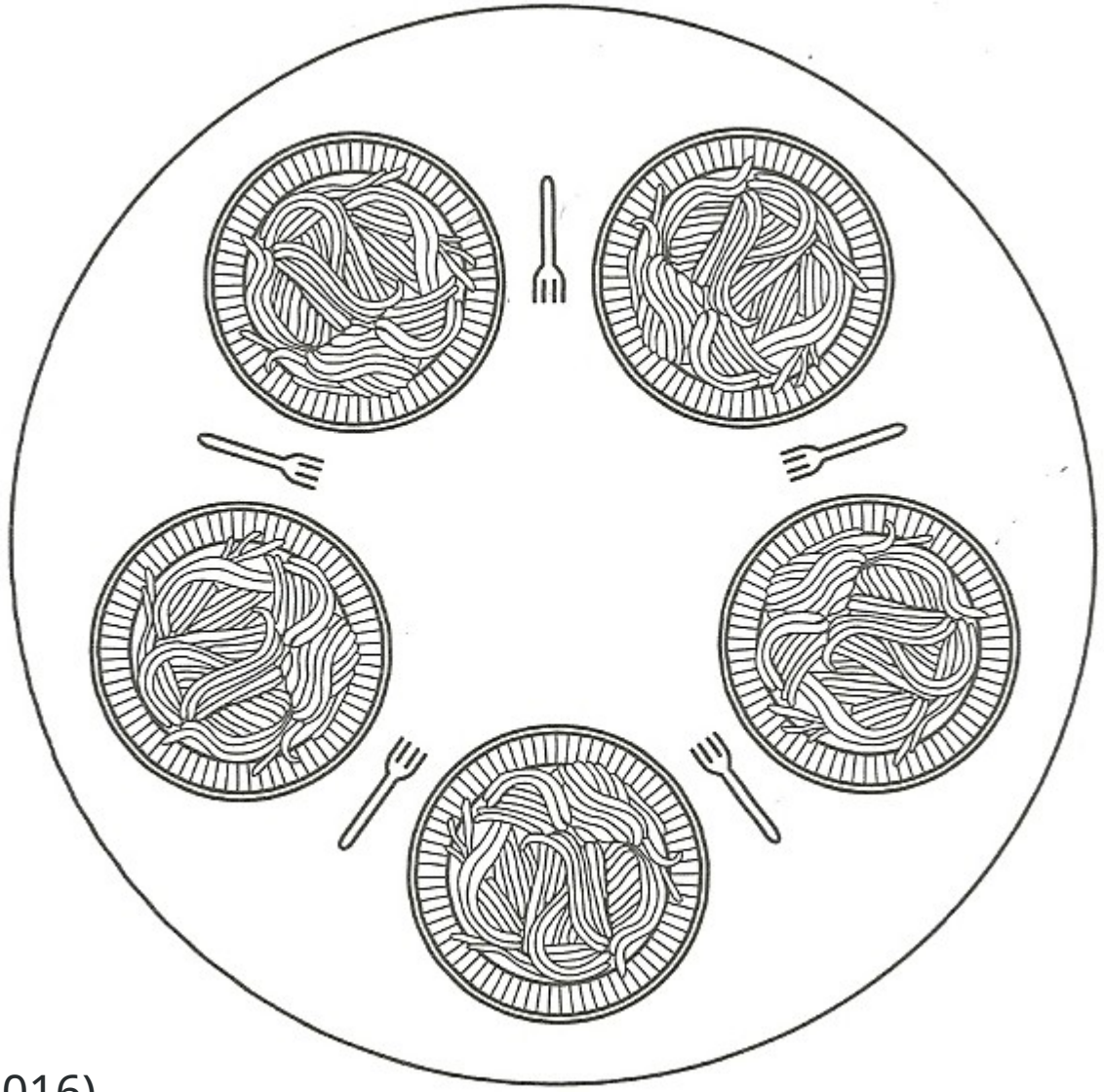
O PROBLEMA DO JANTAR DOS FILÓSOFOS

- Primeira solução:

```
void philosopher(int i)
{
    while (TRUE)
    {
        int left = i;
        int right = (i+1) % N;

        think();
        take_fork(left);
        take_fork(right);
        eat();
        put_fork(left);
        put_fork(right);
    }
}
```

O PROBLEMA DO JANTAR DOS FILÓSOFOS



Fonte: Tanenbaum (2016)

O problema do jantar dos filósofos

- Problema: se o todos os filósofos resolverem pegar o garfo ao mesmo tempo
- E ao obter um garfo, o escalonador troca de contexto para outro filósofo
- Neste caso, todos os filósofos obterão um garfo
- E ficarão bloqueados tentando obter o outro
- Esse problema é conhecido como **deadlock** (ou impasse)

O problema do jantar dos filósofos

- Imagina que nós tentamos uma nova versão do algoritmo, de forma que um filósofo largue o garfo esquerdo, se não conseguiu obter o direito

O problema do jantar dos filósofos

- Imagina que nós tentamos uma nova versão do algoritmo, de forma que um filósofo largue o garfo esquerdo, se não conseguiu obter o direito
- Neste caso, o problema é minimizado
- Porém ele não é corrigido
- Ainda é possível obter um escalonamento problemático

O PROBLEMA DO JANTAR DOS FILÓSOFOS

- Se o escalonador sempre trocar de contexto ao obter ou liberar um garfo:
- F1 obtém garfo, F2, ..., F5 obtém garfo.
  F1 libera garfo, F2, ..., F5 libera garfo
  F1 obtém garfo, F2, ..., F5 obtém garfo
  (...)
- Neste caso, os filósofos estão processando, porém eles fracassam em realizar qualquer progresso
- Esse problema é chamado **starvation** (inanição)

O problema do jantar dos filósofos

- Uma terceira tentativa seria:
- Se o filósofo fracassar em obter o outro garfo, ele espera um tempo aleatório até tentar obter novamente

O problema do jantar dos filósofos

- Uma terceira tentativa seria:
- Se o filósofo fracassar em obter o outro garfo, ele espera um tempo aleatório até tentar obter novamente
- Do ponto de vista prático, essa solução funciona muito bem
- Pacotes que trafegam em cabo de rede do padrão Ethernet, funcionam dessa forma

O problema do jantar dos filósofos

- Do ponto de vista teórico, essa solução não é completa: existe um escalonamento que causa problemas, portanto a solução não pode ser considerada como correta
- Nós estamos interessados numa solução 100% correta, e não uma aproximadamente 99,99%

O problema do jantar dos filósofos

- A solução usando semáforos binários:
- Inicialmente usamos um semáforo binário para proteger uma seção crítica
- Antes de começar a pegar garfos, um filósofo realiza down em mutex
- Após substituir os garfos, ele realiza up em um mutex

O problema do jantar dos filósofos

- Do ponto de vista teórico, essa solução está correta
- Do ponto de vista prático, essa solução possui um problema de desempenho: vários filósofos pensam em paralelo, mas apenas um filósofo pode comer simultaneamente
- Sabemos que é possível que mais de um filósofo coma ao mesmo tempo

O problema do jantar dos filósofos

- A solução completa, usando semáforos:
- Utilizar duas funções, uma para pegar e outra para liberar os garfos
- As funções são protegidas por um semáforo binário (mutex)
- Existe um array com um semáforo por filósofo
- O filósofo aumenta seu semáforo, se obter os dois garfos, ou bloqueia nele, aguardando que um vizinho libere o garfo necessário

O PROBLEMA DO JANTAR DOS FILÓSOFOS

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

int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(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]);
    }
}
```

O PROBLEMA DO JANTAR DOS FILÓSOFOS

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

int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(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);
}
```

Se os recursos estão disponíveis, existe um up nesse semáforo

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

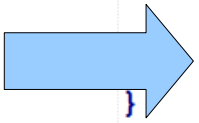
O PROBLEMA DO JANTAR DOS FILÓSOFOS

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

int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
    while (TRUE)
    {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

Se os garfos estão indisponíveis, o processo se bloqueia neste ponto



```
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]);
    }
}
```

O PROBLEMA DO JANTAR DOS FILÓSOFOS

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

int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
    while (TRUE)
    {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

Ao liberar os garfos, o processo libera os processos que estão aguardando

```
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]);
    }
}
```

O problema do jantar dos filósofos

- Soluções completas para o jantar dos filósofos:
  - Permitir que apenas quatro filósofos sentem-se à mesa;
  - Permitir que um filósofo pegue o garfo da esquerda apenas se o garfo da direita estiver disponível (se existe uma operação atômica para pegar e verificar);
  - Permitir que um filósofo ímpar pegue primeiro o seu garfo da esquerda e depois o da direita, enquanto um filósofo par pegue o seu garfo da direita e depois o da esquerda.

Deadlock

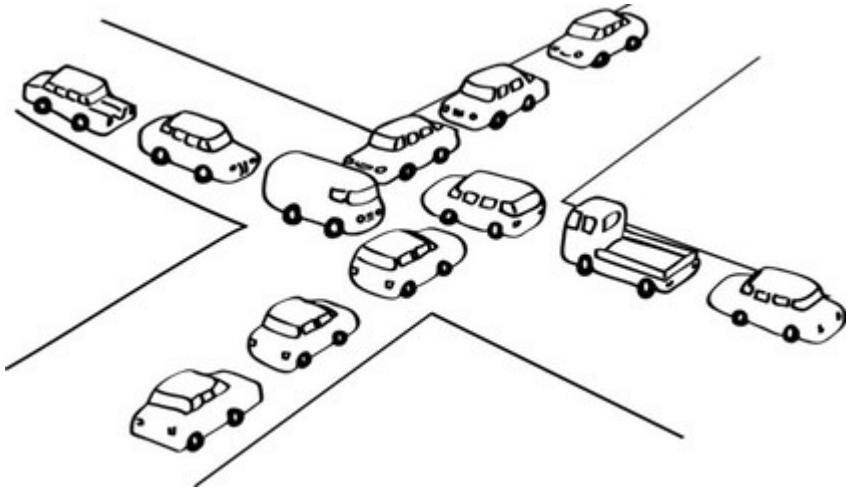
- Conforme vimos no problema dos filósofos, ao implementar mecanismos de proteção à seção crítica, podemos criar outros problemas:
  - **Deadlock**: quando os programas/threads aguardam um recurso que não é liberado
  - **Starvation**: quando os mecanismos de sincronização não permitem que o programa avance

Deadlock

- Deadlocks podem acontecer em diversas situações
- Os sistemas computacionais estão cheio de recursos que devem ser compartilhados e protegidos
- Dois processos acessando a mesma entrada de arquivos, pode causar problema
- É responsabilidade do sistemas operacional não corromper todo o sistema de arquivos

Deadlock

- No entanto, ao garantir a exclusão mútua desses recursos, é possível causar um deadlock (impasse)



https://www.dirceuresende.com/wp-content/uploads/2017/09/deadlock-car.jpg

Deadlock

- É importante frisar que deadlocks não são apenas uma situação hipotética.
- Eles ocorrem realmente na prática

Fonte: https://sites.google.com/site/profferdesiqueiraprogconc/_/rsrc/1461330659214/aulas/7-deadlock/programacao%20concorrente1.png

Deadlock

- Os deadlocks podem acontecer não somente em software, mas também em hardware
- Em software, com mutexes ou outros mecanismos de sincronização de processos
- Em hardware quando, múltiplos programas tentam acessar/usar dispositivos, por exemplo, uma impressora e scanner

Deadlock

- O exemplo mais simples de um deadlock, quando os recursos só podem ser usados por um processo de cada vez, os processos P1 e P2 podem chegar a uma situação de bloqueio eterno, esperando por recursos que nunca são liberados:

```
P1:                 P2:
Obtem(R1);          Obtem(R2);
Obtem(R2);          Obtem(R1);
...                 ...
```

Recursos

- Para analisar melhor a situação, vamos nos referir a recursos, que são objetos que um processo pode adquirir de maneira exclusiva, ou não
- Na obtenção de recursos de maneira exclusiva, apenas um processo pode estar usando ao mesmo tempo (exclusão mútua é implícita)
- Um recurso é qualquer coisa que pode ser adquirida, usada e liberada

Tipos de Recursos

- Preemptíveis:
  - Pode ser retirado do proprietário por uma entidade externa sem causar-lhe prejuízo.
  - Ex: Memória não utilizada
  - Se dois programas, precisam utilizar um espaço de memória e usar a impressora. Imagine a situação que P1 obteve a impressora, P2 obteve a memória e não há mais memória disponível.
  - Aparente impasse, mas neste caso, é possível transferir a memória para P1 de forma que termine sua execução

Tipos de Recursos

- Não-preemptível:
  - Um recurso que não pode ser tomado à força.
  - O processo que o possui deve liberá-los de espontaneamente
  - Por exemplo, se um programa começou a escrever um arquivo, ou começar a imprimir uma página, não é possível interromper esse processo sem prejuízos

Recursos

- Em geral, deadlocks ocorrem em recursos não-preemptíveis.
- Em recursos preemptíveis a simples transferência de recursos os resolve

Recursos

- São necessários os seguintes passos para usar um recurso:
  - Solicitar o recurso;
  - Usar o recurso;
  - Liberar o recurso.
- Se o recurso não está disponível quando solicitado, o processo que está solicitando é forçado a esperar

Deadlock

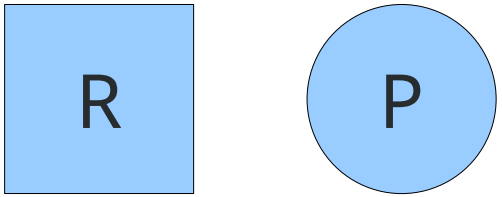
- Definição formal de deadlock:
- “Um conjunto de processos estará em situação de deadlock se cada processo no conjunto estiver esperando por um evento que apenas outro processo no conjunto pode causar”
- Como todos os processos estão esperando, nenhum deles poderá causar qualquer evento que possa despertar outros membros do conjunto, e todos os processos continuam esperando para sempre

Deadlock

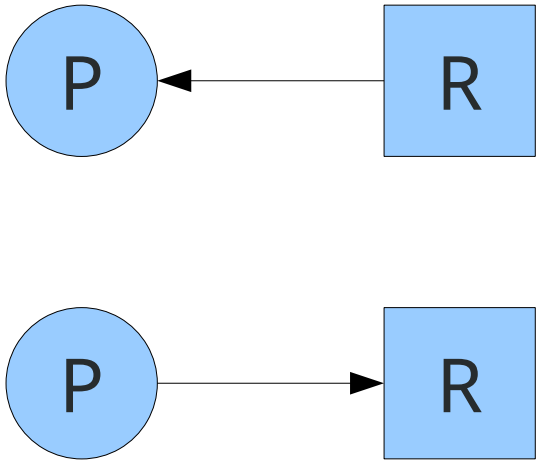
- Na maioria dos casos, o evento que cada processo aguarda é a liberação de um recurso obtido por outro processo
- O número de processos e o número de recursos solicitados devem ser maior que 2

Modelagem de Deadlocks

- Holt (1972) demonstrou como deadlocks podem ser modelados com grafos dirigidos
- Os grafos possuem dois tipos de nós, processos (círculos) e recursos (quadrados)



Modelagem de Deadlocks

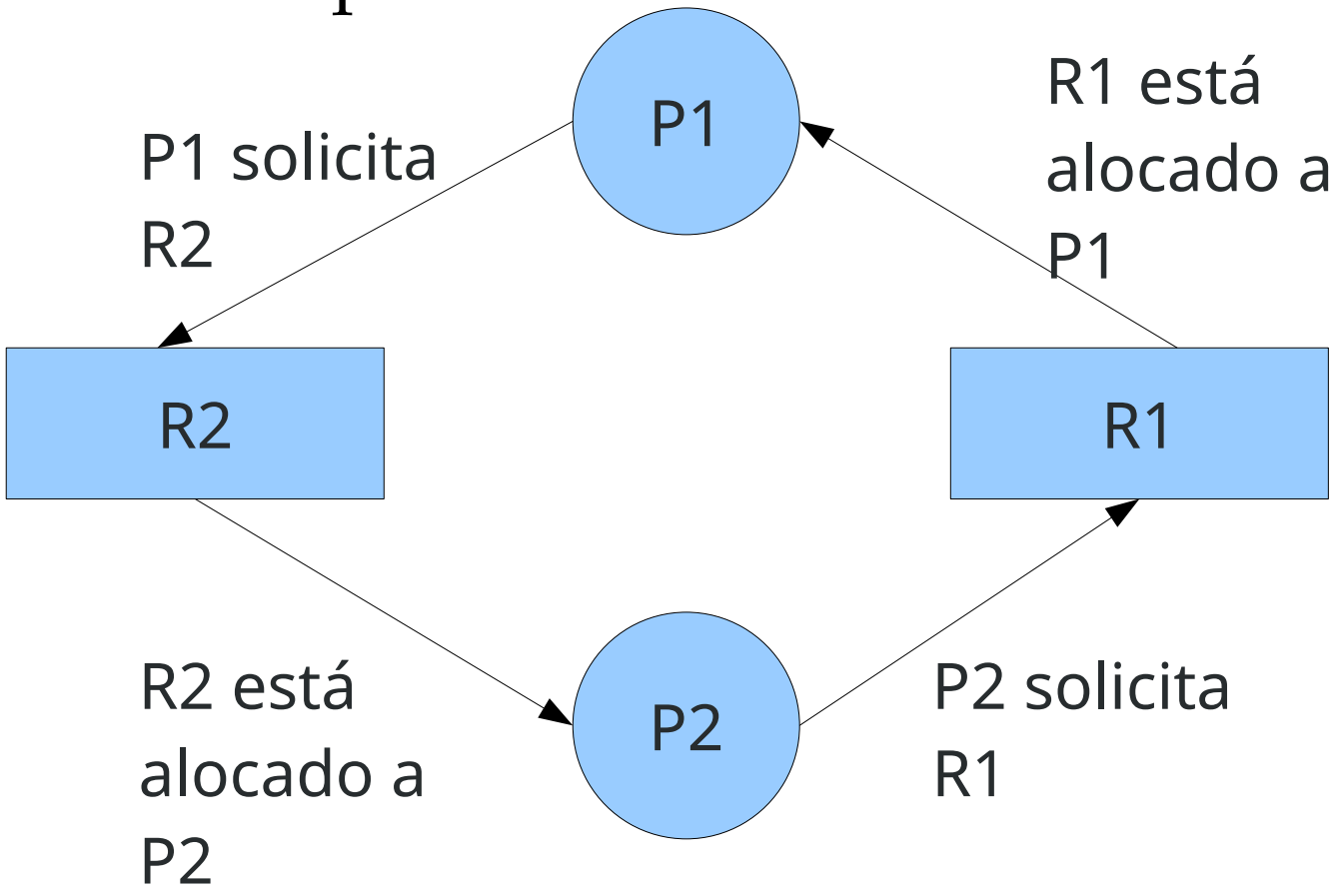


O recurso R está alocado ao processo P

O processo P deseja obter o recurso R e está bloqueado esperando a liberação

Modelagem de Deadlocks

- Graficamente, o deadlock pode ser ilustrado como uma espera circular



Modelagem de Deadlocks

- Em um sistema operacional de computador, três processos estão na seguinte situação:
  - o processo P1 tem a posse do recurso R1.
  - o processo P2 tem a posse do recurso R2.
  - o processo P3 tem a posse do recurso R3.
- O processo P1 solicita o recurso R2, o processo P2 solicita o recurso R3, e o processo P3 solicita o recurso R1. Sobre essa situação, é correto afirmar que:

a) não haverá deadlock, pois o processo P1 não solicitou o recurso R3.

b) tem-se uma condição de deadlock.

c) não haverá deadlock, pois o processo P3 não solicitou o recurso R2

d) só ocorrerá deadlock caso P1 solicite o recurso R3, P2 solicite o recurso R1 e P3 solicite o recurso R2.

e) não haverá deadlock, pois o processo P2 não solicitou o recurso R1.

Fonte: (2018/FCC/DPE-AM/Assistente Técnico de Defensoria - Assistente Técnico de Suporte)

Modelagem de Deadlocks

- Em um sistema operacional de computador, três processos estão na seguinte situação:
  - o processo P1 tem a posse do recurso R1.
  - o processo P2 tem a posse do recurso R2.
  - o processo P3 tem a posse do recurso R3.
- O processo P1 solicita o recurso R2, o processo P2 solicita o recurso R3, e o processo P3 solicita o recurso R1. Sobre essa situação, é correto afirmar que:

a) não haverá deadlock, pois o processo P1 não solicitou o recurso R3.

b) tem-se uma condição de deadlock.

c) não haverá deadlock, pois o processo P3 não solicitou o recurso R2

d) só ocorrerá deadlock caso P1 solicite o recurso R3, P2 solicite o recurso R1 e P3 solicite o recurso R2.

e) não haverá deadlock, pois o processo P2 não solicitou o recurso R1.

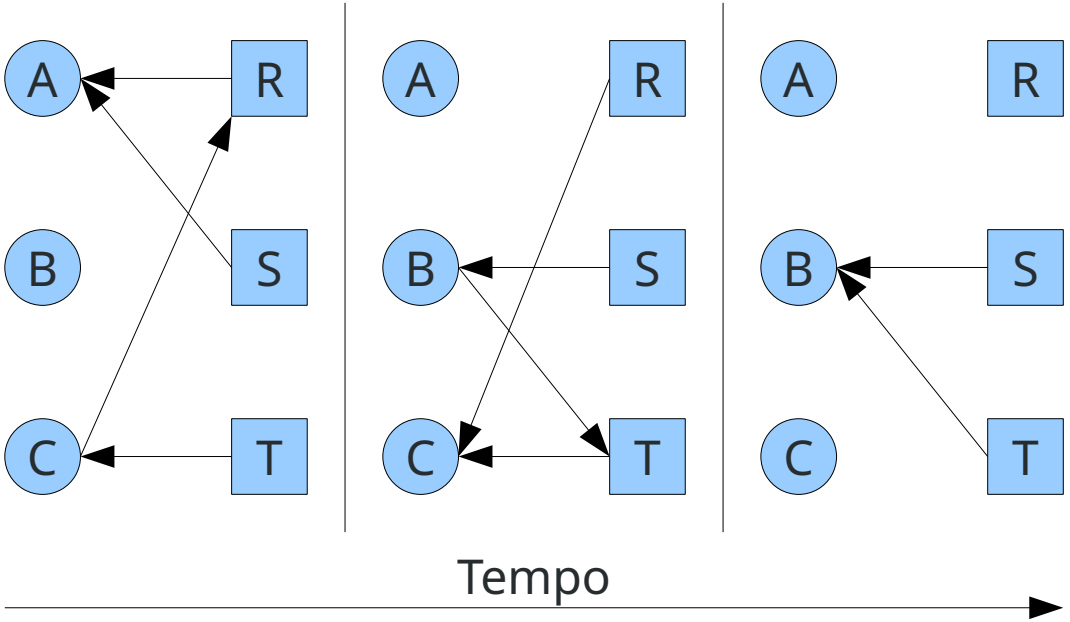
Fonte: (2018/FCC/DPE-AM/Assistente Técnico de Defensoria - Assistente Técnico de Suporte)

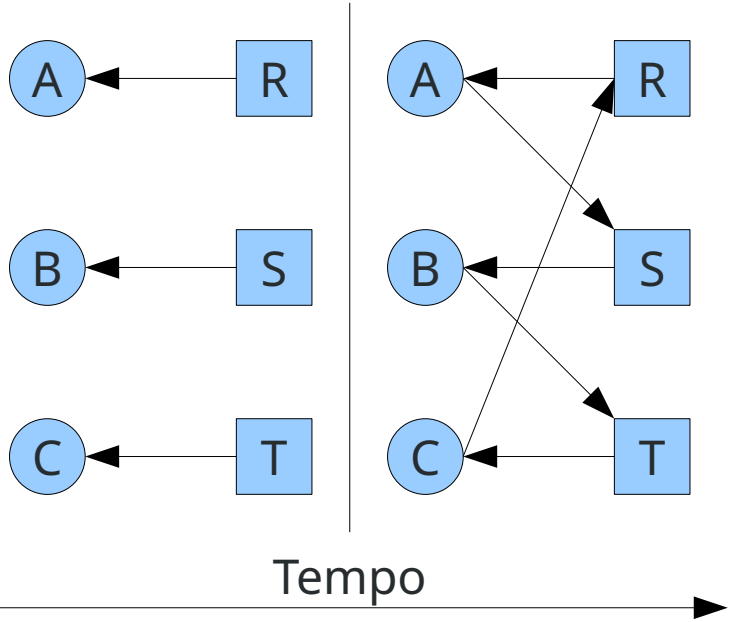
Deadlocks envolvendo 3 processos

Neste caso, temos uma situação de deadlock que ocorre em três processos.

Não é obrigatório que todos os processos peçam todos os recursos ao mesmo tempo.

Processo A	Processo B	Processo C
Req(R);	Req(S);	Req(T);
Req(S);	Req(T);	Req(R);
Free(R);	Free(S);	Free(T);
Free(S);	Free(T);	Free(R);





Modelagem de Deadlocks

- No entanto, algumas questões podem cobrar apenas “um processo aguardando outro”
- Neste caso, o grafo a ser modelado envolvendo apenas processos que com seta para outros processos
- Processo P1 aguarda Processo P2

Modelagem de Deadlocks

- Uma engenheira de software precisa verificar a existência e tratar deadlocks no controle de concorrência de um banco de um servidor de arquivos, no seguinte contexto de processos:
  - o processo A está esperando por C e D
  - o processo B está esperando por C
  - o processo C não está em espera
  - o processo D está em espera por C
- Nessas condições, para o servidor funcionar, será necessário:

a) aguardar a execução.

b) finalizar o processo C.

c) finalizar o processo D.

d) finalizar todos os processos.

Fonte: (2018/CS-UFG/SANEAGO - GO/Analista de Sistemas)

Modelagem de Deadlocks

- Uma engenheira de software precisa verificar a existência e tratar deadlocks no controle de concorrência de um banco de um servidor de arquivos, no seguinte contexto de processos:
  - o processo A está esperando por C e D
  - o processo B está esperando por C
  - o processo C não está em espera
  - o processo D está em espera por C
- Nessas condições, para o servidor funcionar, será necessário:

a) aguardar a execução.

b) finalizar o processo C.

c) finalizar o processo D.

d) finalizar todos os processos.

Fonte: (2018/CS-UFG/SANEAGO - GO/Analista de Sistemas)

Deadlocks

- Vimos que é função do sistema operacional gerenciar os recursos computacionais
- O que ele pode fazer nas situações de deadlock?
- Estratégias usadas para tratar deadlock:
  - **Ignorar** o problema;
  - **Detectar** e recuperar o deadlock;
  - **Evitar** o deadlock;
  - **Prevenir** o deadlock.

Ignorar o problema

- "O Algoritmo do Avestruz"
- Enfie a sua cabeça na areia e finja que não há um problema (folclore popular)
- Não vale a pena degradar a performance do sistema para tratar uma situação que ocorre com pouca frequência
- A maioria dos sistemas modernos são desta forma!

Ignorar o problema

- Neste caso, quem trata o deadlock ao ocorrer?
- O programador deve colocar algumas condições
- Se eles ocorrem uma vez a cada cinco anos, mas quedas do sistema decorrentes de falhas no hardware ocorrem uma vez por ano, a maioria dos engenheiros não estaria disposta a pagar um preço alto de desempenho

Detectar e Recuperar o Deadlock

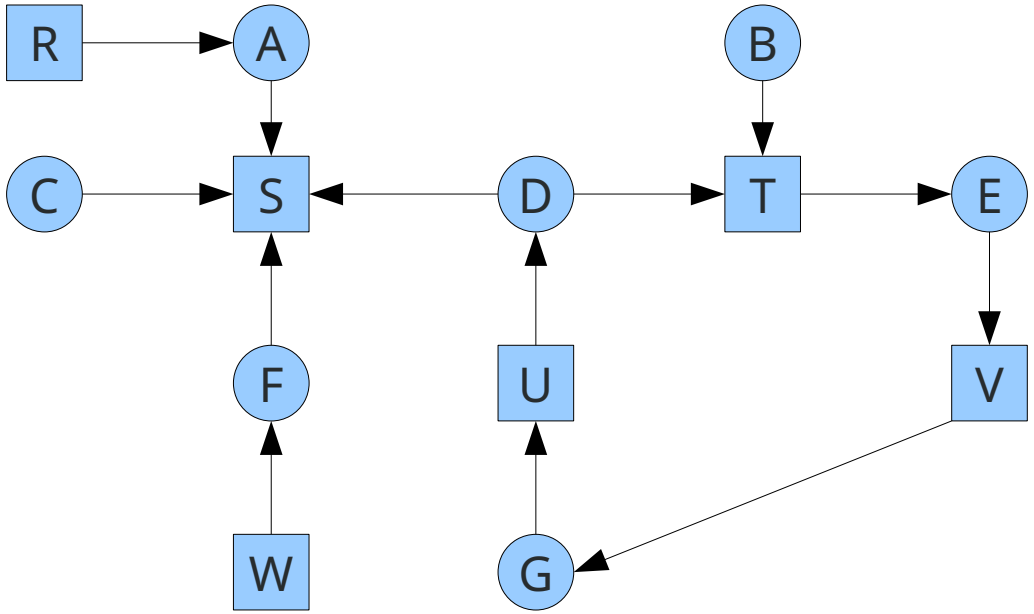
- Como existe a possibilidade de ocorrência do problema, o sistema operacional deve tratá-lo
- Essa técnica é divida em duas fases:
  - Detecção do deadlock;
  - Recuperação do deadlock.

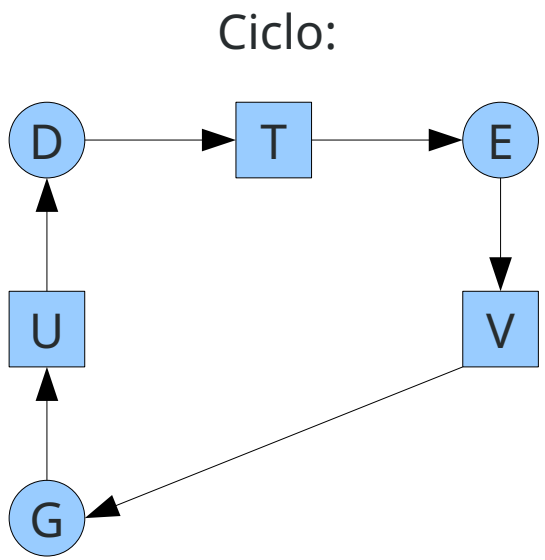
Detecção de Deadlocks

- Aproveitando o modelo de Holt, um grafo é construído com as alocações dos recursos
- Neste grafo, um deadlock é representado por um ciclo. Desta forma, todos os processos que estão nesse ciclo estão em deadlock
- Para a detecção de ciclos, deve-se utilizar um algoritmo para detecção de ciclos em grafos dirigidos: diversos são conhecidos
- Essa estratégia pode ser estendida para usar vetores e detectar múltiplos recursos de cada tipo

Detecção de Deadlocks

- Algoritmos de detecção retornam o ciclo, ou nulo se nenhum ciclo foi encontrado
- Exemplo de um grafo dirigido e um ciclo dele:





Detecção de Deadlocks

- Quando detectar um deadlock?
  - Periodicamente
  - A cada nova solicitação de recurso
  - Quando a utilização de CPU for baixa

Recuperação de Deadlocks

- Na segunda fase, é necessário recuperar de um deadlock:
  - Preempção
  - Rollback
  - Eliminação de processos

Recuperação de Deadlocks

- Preempção
- Em alguns casos, pode ser viável tomar um recursos do seu proprietário à força e devolvê-lo posteriormente
- É muito dependente da natureza do recurso e pode tornar-se uma operação complexa
- Escolher o processo a ser suspenso depende muito de quais processos tem recursos que podem ser devolvidos mais facilmente

Recuperação de Deadlocks

- Recuperação mediante rollbacks (retrocesso)
- Ao saber que a possibilidade de deadlock é grande, pode-se utilizar um arquivo de log que é atualizado periodicamente, que contém uma imagem do processo naquele momento
- Esse ponto de salvaguarda (checkpoint) permite que o processo seja reinicializado a partir daquele momento, em algum instante futuro

Recuperação de Deadlocks

- Recuperação mediante rollbacks (retrocesso)
- Quando um deadlock é detectado, sabe-se quais recursos são necessários
- Para realizar a recuperação, escolhe-se um processo em posse dos recursos e realiza o seu rollback até um momento anterior à aquisição dos recursos
- Desta forma, os recursos podem ser entregues a outros processos e eles não são prejudicados

Recuperação de Deadlocks

- Eliminação de processos
- Provavelmente, a maneira mais bruta de se eliminar um deadlock
- Remove processos que estão em deadlock para quebrar os ciclos
- Com um pouco de sorte (afinal podem existir múltiplos ciclos) é possível que o deadlock se resolva rapidamente

Recuperação de Deadlocks

- Eliminação de processos
- Quando for possível, deve-se eliminar um processo que possa ser reexecutado desde o início sem grandes perdas ou danos
- Alguns processos como atualizadores de banco de dados podem, nem sempre, executar mais uma vez com segurança

Evitando Deadlocks

- Anteriormente, presumimos que a alocação de recursos, eles são obtidos e solicitados todos de uma vez
- Essa não é a realidade na maioria dos sistemas computacionais
- Na detecção de deadlocks estamos preocupados em fazer o sistema decidir para qual processo libera o recurso, baseado em uma decisão correta

Evitando Deadlocks

- Os algoritmos para evitar deadlocks são baseados em estados seguros
- Diz-se que um estado é seguro se existir alguma ordem de escalonamento no qual todos os processos puderem ser executados até sua conclusão mesmo que todos solicitem seu numero máximo de recursos subitamente

Evitando Deadlocks

- Desta forma, o processo (P) possui uma tabela de recursos que obteve (R) e o número máximo de recursos que ele pode obter (M)
- Um exemplo, o estado é seguro, pois existe uma política de escalonamento que permite a execução (Atendendo inicialmente o processo B)

P	R	M
A	3	9
B	2	4
C	2	7

Disponível: 3

P	R	M
A	3	9
B	4	4
C	2	7

Disponível: 1

P	R	M
A	3	9
B	0	-
C	2	7

Disponível: 5

P	R	M
A	3	9
B	0	-
C	7	7

Disponível: 0

P	R	M
A	3	9
B	0	-
C	0	-

Disponível: 7

Evitando Deadlocks

- No entanto, é possível que se atender o processo A, antes do B, pode-se levar a um estado inseguro
- Não há uma sequência que garanta a conclusão

P	R	M
A	3	9
B	2	4
C	2	7

Disponível: 3

P	R	M
A	4	9
B	2	4
C	2	7

Disponível: 2

P	R	M
A	4	9
B	4	4
C	2	7

Disponível: 0

P	R	M
A	4	9
B	-	-
C	2	7

Disponível: 4

Evitando Deadlocks

- Um estado inseguro não é um estado de impasse
- Pode ser que o processo A libere um recurso e o sistema finalize. Mas o sistema não tem como adivinhar o que irá ocorrer
- A diferença é que não pode garantir a finalização

Evitando Deadlocks

- Um algoritmo de escalonamento para evitar deadlocks foi desenvolvido (por Dijkstra, 1965)
- A ideia é que o banqueiro possa garantir recursos para seus clientes
  - Algoritmo baseado em estados seguros: se uma solicitação leva a estados seguros, atende-a. Senão, adia por um tempo.
  - Solicitação segura é quando o banqueiro tem recursos suficientes para atender pelo menos um cliente
- Pode ser generalizado para múltiplos recurso

Condições para ocorrência de Deadlocks

- Coffman et al. (1971) demonstraram que existem quatro condições para ocorrer um deadlock:
  - **1**. condição de exclusão mútua: cada recurso está associado a um processo ou está disponível
  - **2**. condição de posse e espera: processo atualmente de posse de recursos que foram concedidos antes podem solicitar novos recursos
  - **3**. condição de não-preempção: recursos concedidos antes não podem ser liberados à força
  - **4**. condição de espera circular: deve haver uma lista circular de dois ou mais processos esperando por recursos detido pelo próximo membro da cadeia

Prevenção de Deadlock

- As 4 condições precisam ocorrer para se ter um deadlock. Se uma delas não ocorrer, o deadlock não ocorrerá!
- Para prevenir, podemos:
  - Atacar a condição de exclusão mútua;
  - Atacar a condição de posse e espera;
  - Atacar a condição de não-preempção;
  - Atacar a condição de espera circular.

Prevenção de Deadlock

- Impedir a exclusão mútua
- Se nunca acontecer de um recurso ser alocado exclusivamente para um processo, acaba com o problema do deadlock
- Nenhum outro processo irá esperar pelo recurso
- É possível centralizar o processo de controle do recurso, e fazer os outros processos se comunicarem com ele (troca de mensagens)
- Spoll de impressão, SGBD para registros

Prevenção de Deadlock

- Impedir a exclusão mútua
- Nem todos os recursos podem ser gerenciados assim
- Muitas vezes na prática é inviável
- No entanto a ideia de diminuir ao máximo o número de processos que possui acesso ao recurso pode ser uma boa prática

Prevenção de Deadlock

- Impedir a posse e espera
- Se evitarmos que os processos que possuem recursos, peçam mais recursos, o problema pode ser resolvido
- Uma solução é fazer os processos pedirem todos os recursos que necessitam, antes da execução
- Mas é uma necessidade de conhecimento futuros

Prevenção de Deadlock

- Impedir a posse e espera
- Nem sempre sabe-se se recursos serão necessários
- É um desperdício, ficar muito tempo com o recurso se não for utilizá-lo
- Uma outra tática é fazer que o processo solicita o recurso deve liberar todos os recursos que adquiriu anteriormente, para depois adquirir todos o que precisa. Isto pode ser muito complexo

Prevenção de Deadlock

- Impedir a não preempção
- Garantir uma maneira de sempre retirar um recurso de um processo
- Isso pode causar sérios problemas no funcionamento e pode levar a resultados inesperados

Prevenção de Deadlock

- Impedir a espera circular
- Forçar que um processo tenha apenas um recurso por vez, porém para cenários complicados esta restrição é inviável
- É possível manter uma numeração global de todos os recursos do sistema e fazer com que os processos sigam a ordem de requisição dos recursos

Prevenção de Deadlock

- Impedir a espera circular
- Como a ordem de requisição é a mesma, pode-se impedir ciclos nos grafos de requisição
  - Uma variante desse algoritmo diz que um processo não pode pedir um recurso de menor prioridade do que ele já possui
- Problema: grande quantidade de recursos exige uma quantidade muito grande, que não é facilmente enumerada

Prevenção de Deadlock

- Nos exemplos vistos, os deadlocks ocorreram pois os recursos foram invertidos na ordem de que foram obtidos

P1:	P2:
Obtem(R1);	Obtem(R2);
Obtem(R2);	Obtem(R1);
...	...

- Se ambos processos tivessem a ordem de obter o Recurso R1 e depois o recurso R2, o deadlock não aconteceria

Prevenção de Deadlock

Recurso R;	Processo A	Processo B	Processo C
Recurso S;	Req(R);	Req(S);	Req(T);
Recurso T;	Req(S);	Req(T);	Req(R);
	Free(R);	Free(S);	Free(T);
	Free(S);	Free(T);	Free(R);

- Deadlock de três processos: se os recursos foram declarados na ordem alfabética, R, S, T. Assim, o problema aconteceu pois o processo C pediu os recursos na ordem T e depois R.

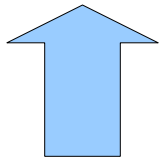
Prevenção de Deadlock

- Deadlock do problema do jantar dos filósofos:

```
void philosopher(int i)
{
    while (TRUE)
    {
        int left = i;
        int right = (i+1) % N;

        think();
        take_fork(left);
        take_fork(right);
        eat();
        put_fork(left);
        put_fork(right);
    }
}
```

Filósofo 0: garfos 0 e 1.
Filósofo 1: garfos 1 e 2.
Filósofo 2: garfos 2 e 3.
Filósofo 3: garfos 3 e 4.
Filósofo 4: garfos 4 e 0.



Problema!

Prevenção de Deadlock

- As técnicas de prevenção de deadlocks são muito importantes para conhecer
- Afinal, elas evitam os deadlocks com baixo custo, enquanto a detecção e prevenção de deadlocks envolvem algoritmos muito caros
- É possível que a existência dessas técnicas popularizaram o algoritmo do avestruz nos SOs modernos...

Conclusão

- Os deadlocks, também chamados de impasses, precisam de quatro condições para ocorrer
- Eles ocorrem quando todos os membros de um conjunto são bloqueados esperando por eventos que somente podem ser causados por membros desse conjunto
- Essa situação faz que todos processos esperem para sempre

Conclusão

- Existem formas para detectar e tratar deadlocks, mas são teoricamente possíveis mas inviável na prática
- A maioria dos sistemas operacionais modernos ignoram e não detectam
- Por isso, é responsabilidade para os programadores de conhecer os mecanismos de prevenção de deadlock e implementá-los

Referências

- Capítulos 2 e 6 – TANENBAUM, A. S. *Sistemas Operacionais Modernos*. 4ª ed. Prentice Hall, 2016. (Capítulos 2 e 3 em versões antigas)
- Capítulo 7 – MACHADO, F. B.; MAIA, L. P. *Arquitetura de Sistemas Operacionais*. 5ª ed. Rio de Janeiro: LTC, 2013.