Engenharia de Computação

Arquitetura de Sistemas Operacionais

Programação concorrente e sincronização de Processos e Threads

Adaptado de
Prof. Anderson Luiz Fernandes Perez
Prof. Márcio Castro

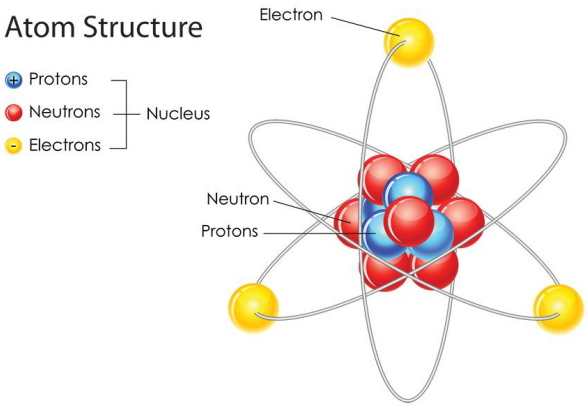
Conteúdo

- Programação concorrente: motivação e definição
- O Problema da Seção Crítica
- Solução de Peterson
- Mecanismos de Sincronização de Processos
- Problemas clássicos

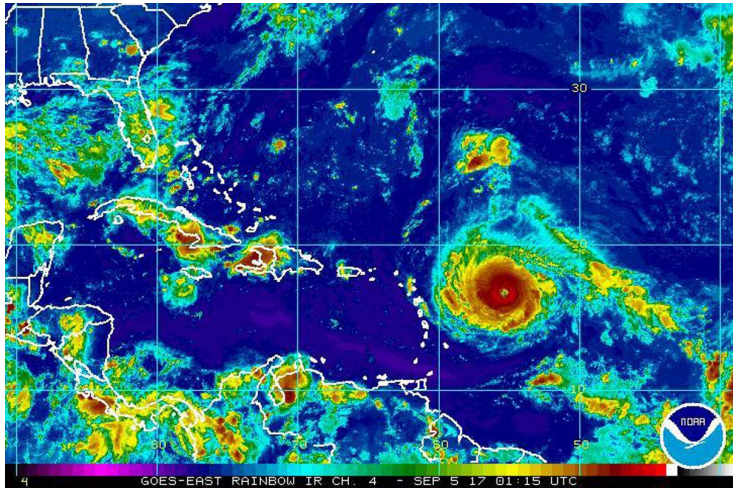
Motivação: maior desempenho computacional

Bioinformática

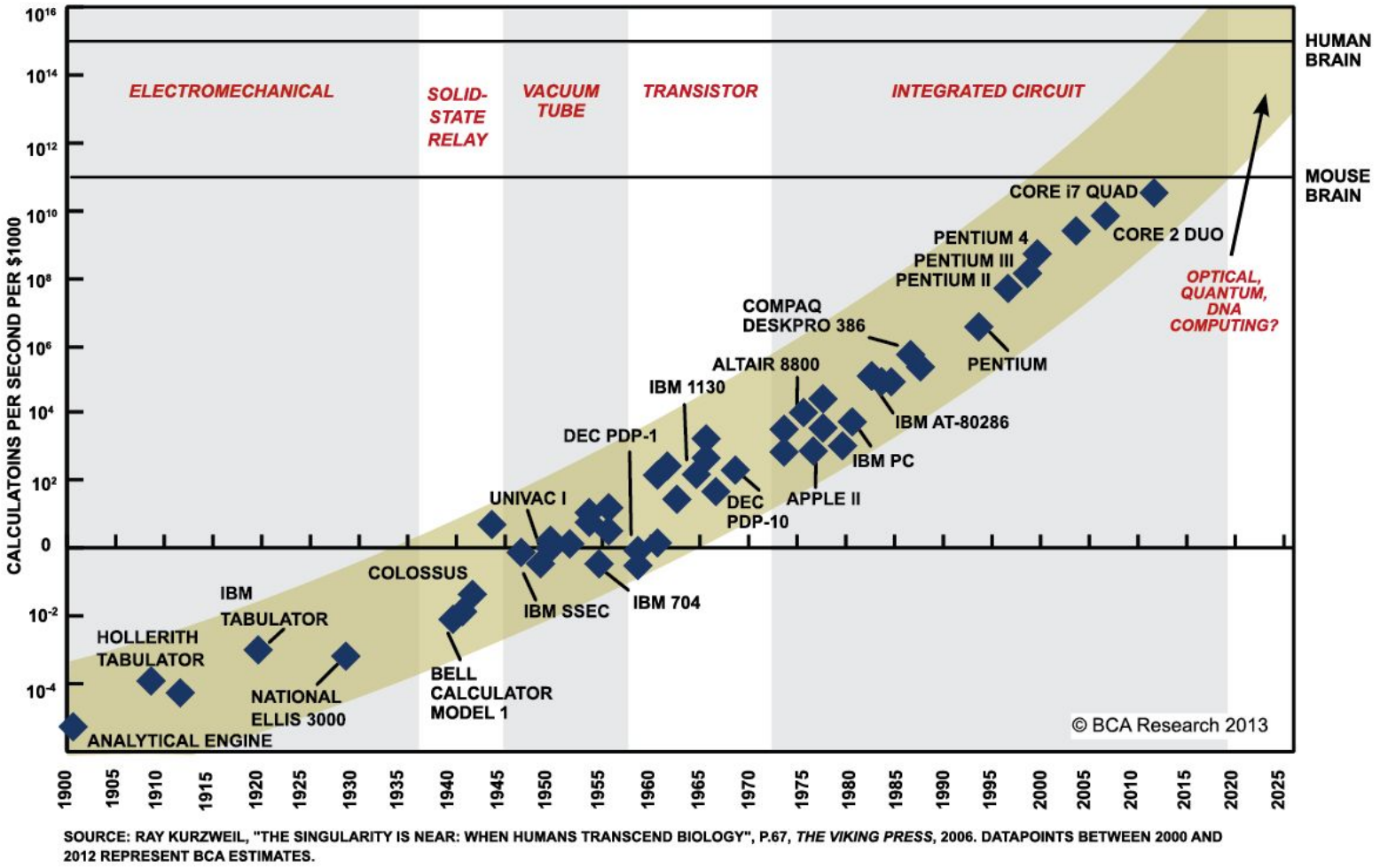
Meteorologia



Física



Motivação: Lei de Moore



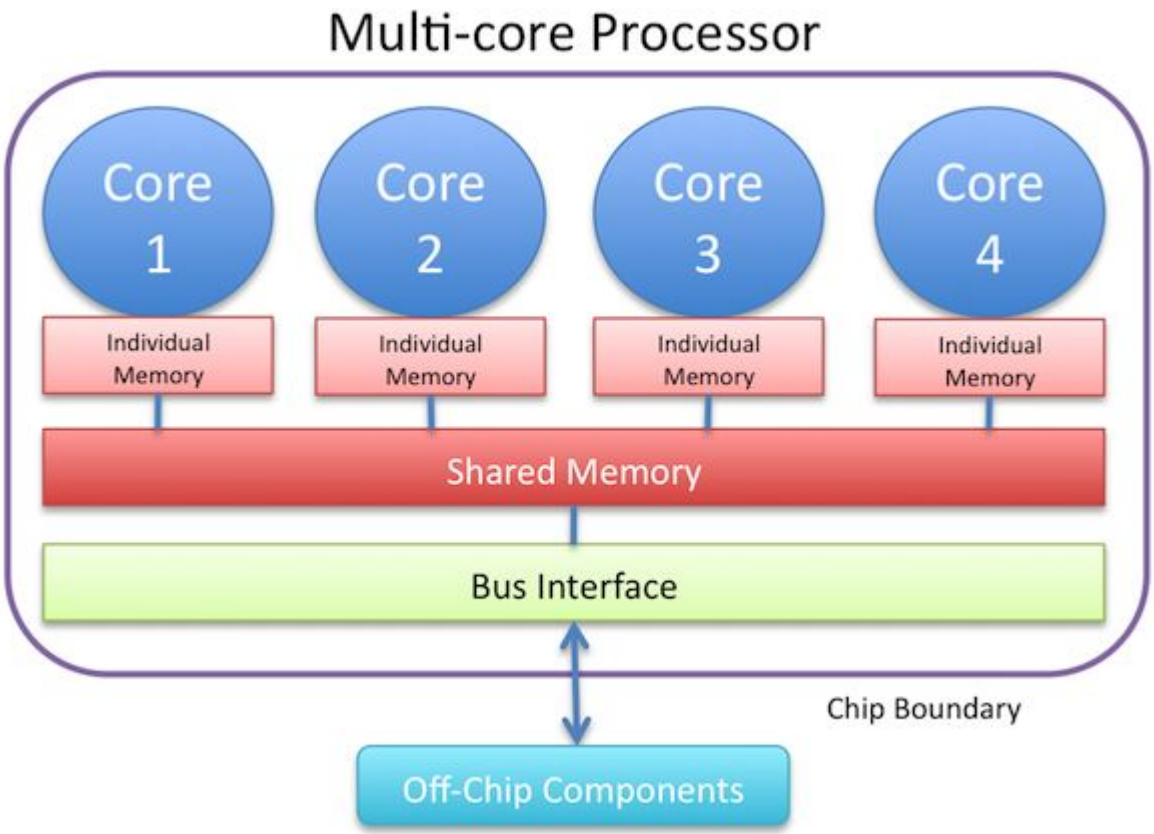
SOURCE: RAY KURZWEIL, "THE SINGULARITY IS NEAR: WHEN HUMANS TRANSCEND BIOLOGY", P.67, *THE VIKING PRESS*, 2006. DATAPOINTS BETWEEN 2000 AND 2012 REPRESENT BCA ESTIMATES.

Motivação: limites

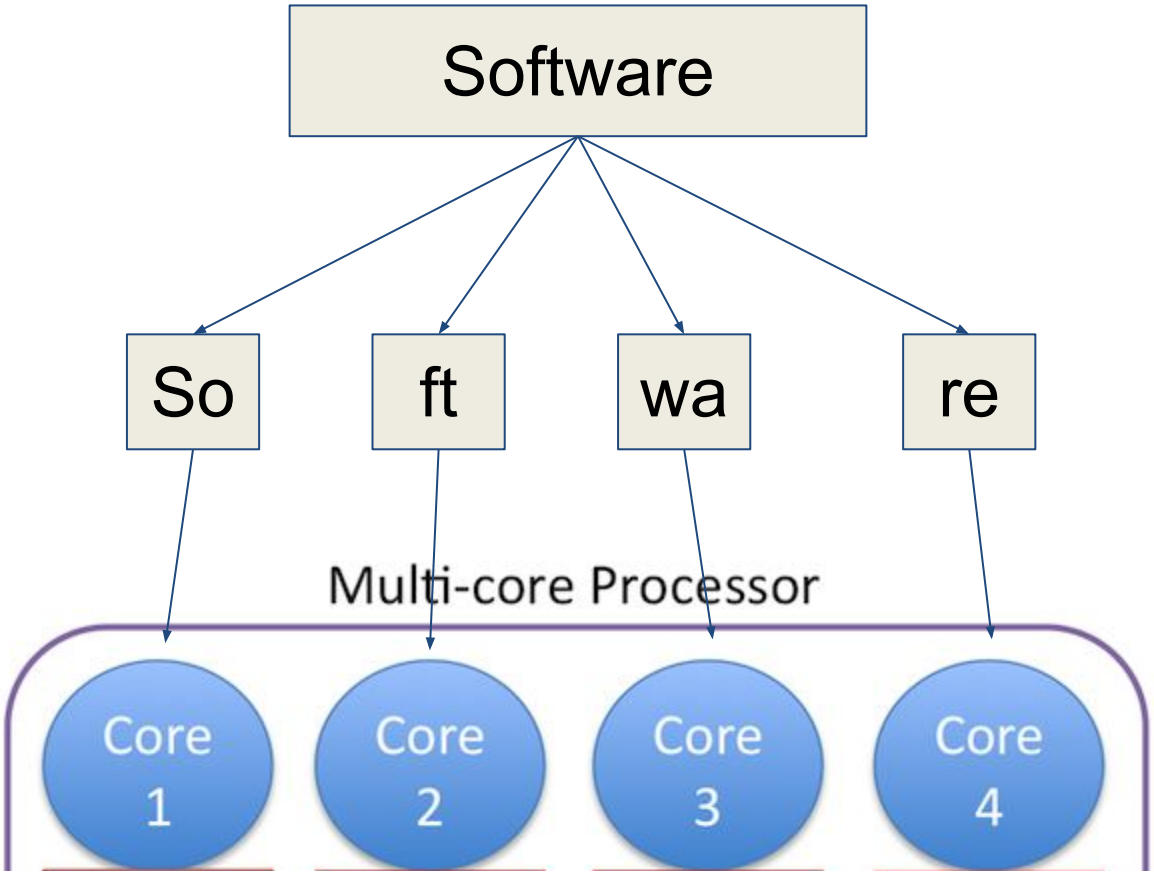
- Banda de comunicação entre CPU e RAM
- Consumo de energia e temperatura máxima

https://www.technologyreview.com/s/421186/why-cpus-arent-getting-any-faster/

Motivação: CPUs multicores



Motivação: como explorar multicores?



?

Concorrência vs. Paralelismo

- **Concorrência**: é a condição de um sistema no qual múltiplas tarefas estão **logicamente ativas em um determinado momento**

- **Paralelismo**: é a condição de um sistema no qual múltiplas tarefas estão **realmente sendo executadas ao mesmo tempo**

Concorrência vs. Paralelismo

Concorrente, execução não paralela

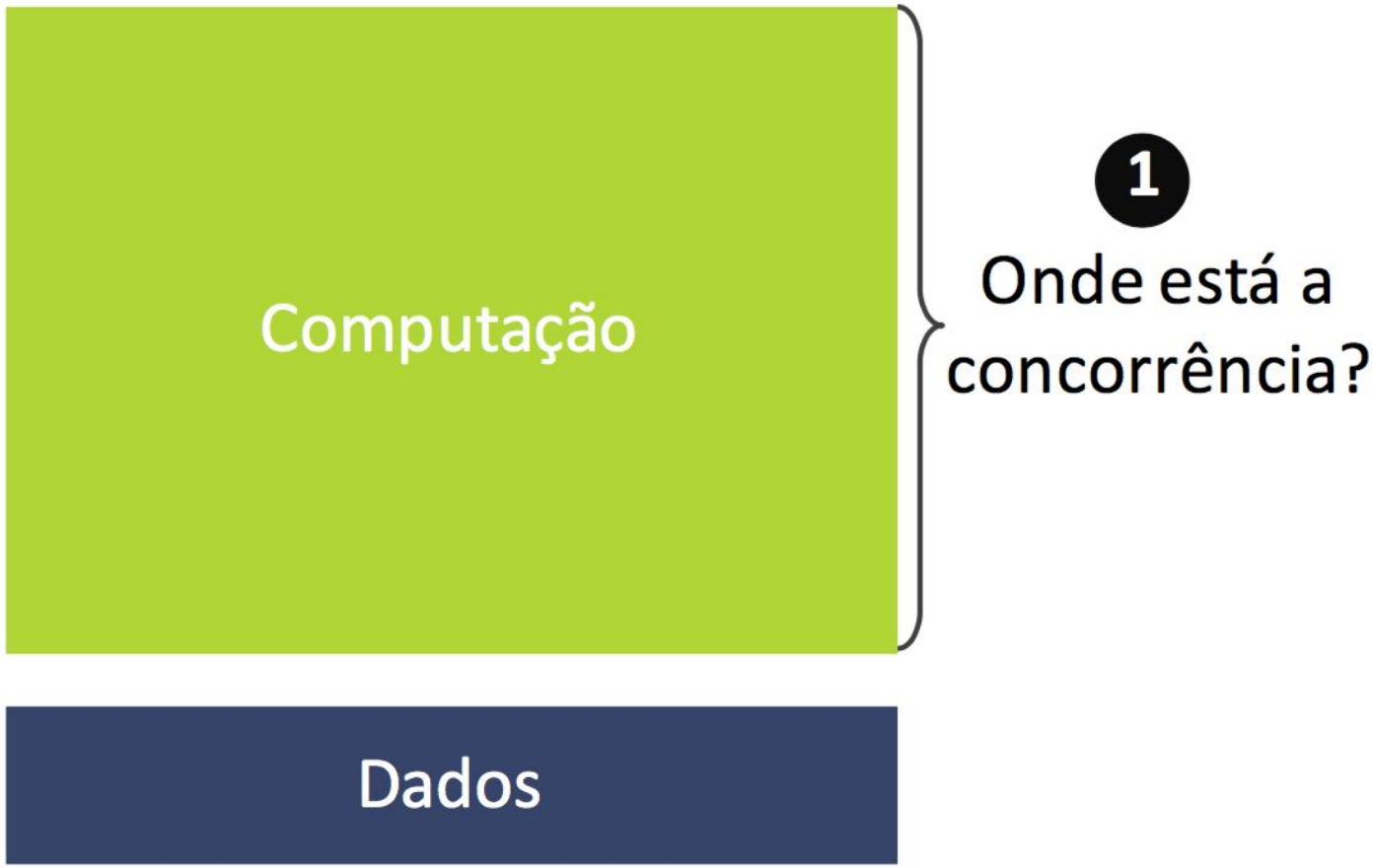
Alternância entre a execução das tarefas
Tarefas não são executadas ao mesmo tempo

Concorrente, execução paralela

Tarefas são executadas ao mesmo tempo

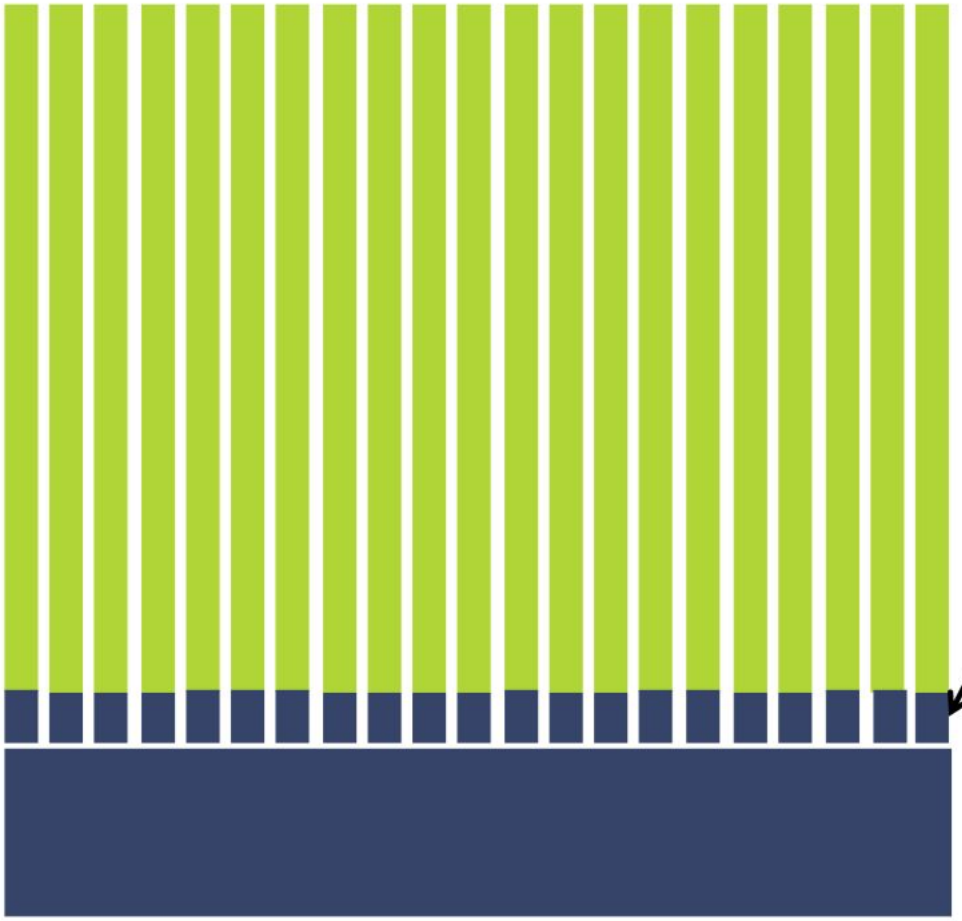
Programação concorrente

Escrevendo um programa paralelo



Programação concorrente

Escrevendo um programa paralelo



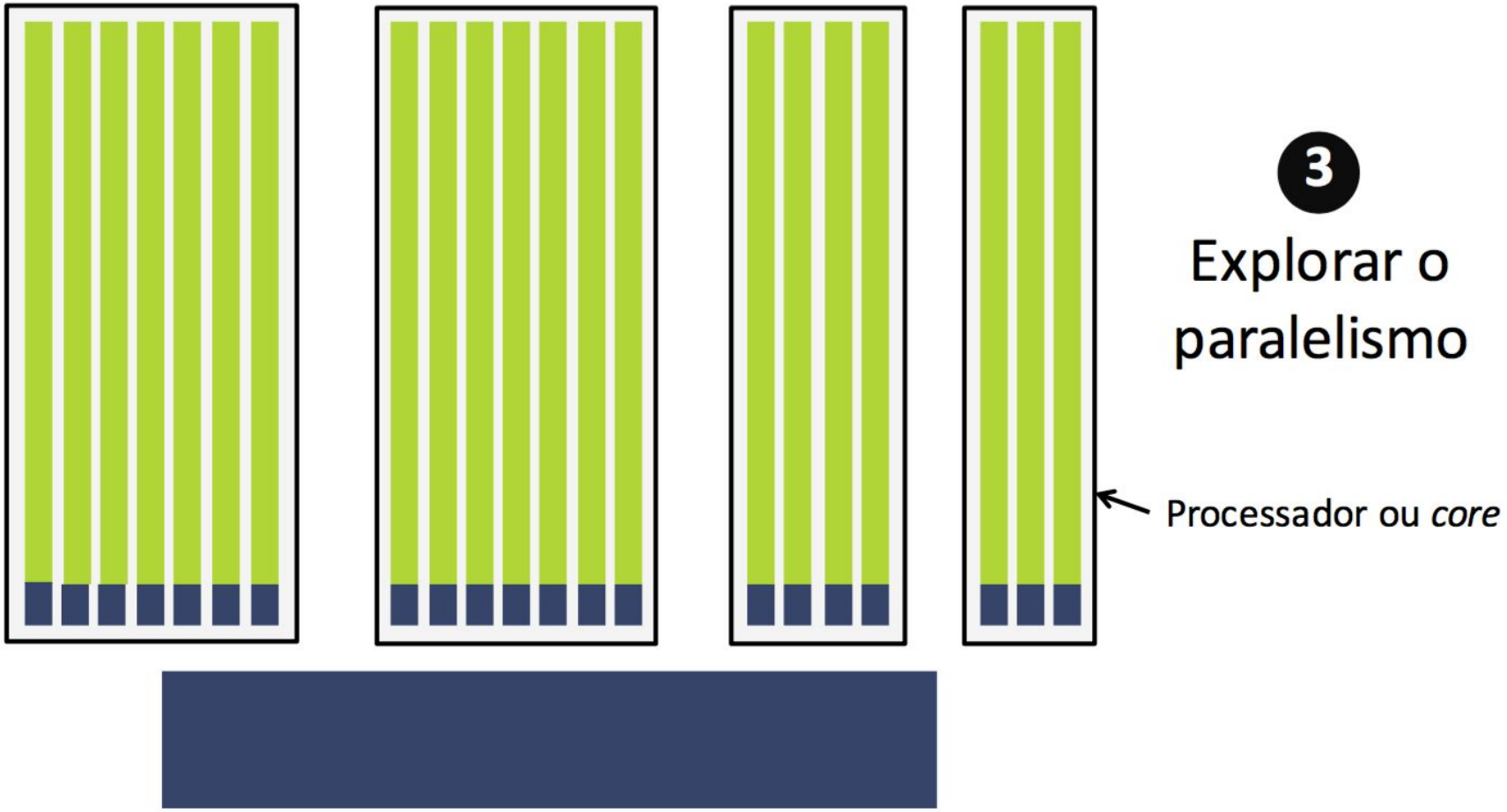
2

Determinar as computações concorrentes

Possível necessidade de utilizar mais dados "locais"

Programação concorrente

Escrevendo um programa paralelo



Vantagens da programação concorrente

- Maior desempenho
  - Aplicações envolvendo grande quantidade de cálculos
  - Aplicações críticas (deadlines)
- Melhor uso dos recursos computacionais
  - Processador e memória
- Robustez
  - Alta capacidade de processamento

Dificuldades da programação concorrente

- Desenvolvimento
- Corretude
- Debug

Ordem em que processos concorrentes executam **NÃO** é determinística

Concorrência

- Processos cooperativos podem afetar outros processos em execução ou ser por eles afetados.
- Processos cooperativos podem:
  - compartilhar um espaço de endereçamento lógico;
  - compartilhar dados através de arquivos ou mensagens.

Concorrência

- Processos produtor e consumidor
  - Um recurso é compartilhado entre dois processos, o produtor que insere dados no buffer e o consumidor que retira dados do buffer.

Concorrência

- Exemplo:
  - Produtor e Consumidor
    - Recurso compartilhado entre dois processos.

Concorrência

- Produtor e Consumidor em C

```
#define BUFFER_SIZE 10

int buffer[BUFFER_SIZE], contador = 0;

// Produtor

int in = 0, out = 0, item = 1;

while (1) {
  // Espera ocupada
  while (contador == BUFFER_SIZE);
  buffer[in] = item++;
  in = (in + 1) % BUFFER_SIZE;
  contador++;
}
```

```
// Consumidor

int itemConsumido, out = 0;

while (1) {
  // Espera ocupada
  while (contador == 0);
  itemConsumido = buffer[out] ;
  out = (out + 1) % BUFFER_SIZE;
  contador--;
}
```

Concorrência

- Processos produtor e consumidor
  - Quais os problemas inerentes ao compartilhamento do recurso (buffer) e da variável contador?

Condição de Corrida!

Outro exemplo

Transação A: adiciona R$ 20 na conta	Transação B: remove R$ 50 da conta
1) tmp = conta.leValor()	1) tmp = conta.leValor()
2) tmp = tmp + 20	2) tmp = tmp – 50
3) conta.escreveValor(tmp)	3) conta.escreveValor(tmp)

Quais os possíveis resultados da execução das transações A e B?

O Problema da Seção Crítica

- Um seção crítica é um segmento de código em que o processo pode estar:
  - alterando variáveis comuns;
  - atualizando uma tabela;
  - gravando um arquivo;
  - ...

O Problema da Seção Crítica

- Quando um processo estiver executando sua seção crítica, NENHUM outro processo deve ter autorização para fazer o mesmo.
- Dois processos NÃO podem estar executando simultaneamente nas suas seções críticas

O Problema da Seção Crítica

- Cada processo deve solicitar permissão para entrar em sua seção crítica (protocolo da seção crítica).

```
do {
   Seção de entrada

   Seção crítica

   Seção de saída
   Seção remanescente

} while (1);
```

O Problema da Seção Crítica

Uma solução para o problema da seção crítica deve satisfazer os quatro requisitos a seguir:

1. Somente um processo por vez acessa a região crítica.
2. Ser independente do número e velocidade de CPUs
3. Um processo fora da região crítica não bloqueia outro processo
4. Nenhum processo espera indeterminadamente para acessar a região crítica

O Problema da Seção Crítica

- Para garantir a seção crítica existem três tipos de soluções possíveis que diferem consideravelmente quanto a recursos utilizados, desempenho e na facilidade de utilização:
  1. Soluções algorítmicas (algoritmo de Dekker e Peterson, algoritmo de Lamport);
  2. Soluções de hardware (inibição de interrupções, instruções especiais);
  3. Objetos do sistema operacional (semáforos, variáveis condicionais, mutex e monitores).

Solução de Peterson

```
#define FALSE 0
#define TRUE 1
#define N 2       //somente dois

int turn;              //variáveis GLOBAIS
int interested[N];

void enter_region(int process)      //process é 0 ou 1
{
    int other = 1 - process;        //o número do outro processo
    interested[process] = TRUE;     //mostra que o processo está interessado
    turn = process;
    while (turn == process && interested[other] == TRUE); // espera ocupada


}
void leave_region(int process)
    interested[process] = FALSE;
}
```

Mecanismos de Sincronização de Processos

- Hardware de Sincronização
  - Muitos sistemas de computação modernos fornecem instruções de hardware especiais que nos permitem testar e modificar o conteúdo de uma palavra ou trocar os conteúdos de duas palavras atomicamente.
  - TestAndSet: testa e modifica uma valor de uma variável.
  - Swap: muda o valor de uma variável em uma única instrução.

Mecanismos de Sincronização de Processos: Mutex

- Abstração muito usada para proteger regiões críticas que evita a espera ocupada (busy waiting)
- Mutex: é um tipo abstrato de dados composto de:
  – Um valor lógico
  – Uma fila de threads: threads bloqueadas, não usam CPU
- Uma variável do tipo mutex pode assumir um dos 2 valores: **livre** ou **ocupado**

Mecanismos de Sincronização de Processos: Mutex

- Duas operações são permitidas em um mutex "m"
  - lock(m): solicita acesso à região crítica
  - unlock(m): libera a região crítica

lock(m)

```
Se (m está livre) então
    Marca m como ocupado
Senão
    Bloqueia a thread e a insere no
    fim da fila do mutex m
```

unlock(m)

```
Se (a fila de m está vazia) então
    Marca m como livre
Senão
    Libera a thread do início da fila do
    mutex m
```

Mutex & Pthreads

Seja mutex uma variável *pthread_mutex_t**

- pthread_mutex_init(mutex, attr)
- pthread_mutex_destroy(mutex)
- pthread_mutex_lock(mutex)
- pthread_mutex_unlock(mutex)

Exemplo mutex & pthreads

```
pthread_mutex_t mutex; // precisa ser global!

void *func_thread(void *arg) {
        pthread_mutex_lock(&mutex);
        //região crítica
        pthread_mutex_unlock(&mutex);
        return 0;
}

nt main(int argc, char **argv) {
        pthread_mutex_init(&mutex, NULL);
        pthread_create(...);
        pthread_join(...);
        pthread_mutex_destroy(&mutex);
        return 0;
}
```

Mecanismos de Sincronização de Processos

- Semáforos
  - Criado por E. W. Dijkstra (1965)
  - Tipo abstrato de dado composto por um valor inteiro e uma fila de processos
  - Operações permitidas sobre o semáforo:
    - P(s) (testar / holandês - proberen)
      - Espera até que ***s*** seja maior que 0 e então subtrai 1 de ***s***.
    - V(s) (incrementar / holandês - verhogen)
      - Incrementa ***s*** em 1 unidade.

Mecanismos de Sincronização de Processos

- Inicializa o semáforo com o número **n>0**

```
init(n)
{
semaforo S* = calloc(1,sizeof(semaforo);
S->valor = n;
return S;
}
```

Mecanismos de Sincronização de Processos

- Definição de P()

```
P(S)
{
  if (! S->valor > 0) {
    adiciona a lista de espera S->list
    bloqueia;
  }
  S->valor--;
}
```

- Definição de V()

```
V(S)
{
  S->valor++;
  if (S->valor <= 0) {
    remove um processo de S->list
    desbloqueia;
  }
}
```

Mecanismos de Sincronização de Processos

- Variáveis Condicionais
  - São variáveis que não armazenam valores específicos, mas representam condições que podem ser aguardadas por alguns processos.
  - Se um processo está aguardando uma condição, ele é inserido em uma fila de espera até que a condição seja verdadeira.
  - As variáveis condicionais evitam a espera ocupada.
  - A implementação de variáveis condicionais pode ser feita a partir de funções como *wait(c), notify(c)/signal(c), notifyall(c)* ou *broadcast(c)*.

Variáveis cond. & Pthreads

Seja cond do tipo pthread_cond_t*, attr do tipo pthread_condattr_t* e mutex do tipo pthread_mutex_t

- pthread_cond_init(cond, attr)
- pthread_cond_destroy(cond)
- pthread_cond_wait(cond, mutex)
- pthread_cond_signal(cond)
- pthread_cond_broadcast(cond)

Mecanismos de Sincronização de Processos

- Monitores
  - Mecanismo de alto nível para sincronização que engloba em um módulo protegido variáveis e procedimentos compartilhados.
  - O acesso às variáveis de um monitor só pode ser feito através de seus procedimentos.
  - A exclusão mútua é imposta implicitamente pelas chamadas aos procedimentos dos monitores.

Mecanismos de Sincronização de Processos

- Monitores
  - Um monitor consiste de:
    - Um recurso compartilhado, conjunto de variáveis internas ao monitor;
    - Um conjunto de procedimentos que permitem o acesso a essas variáveis;
    - Um mutex ou semáforo para controlar a exclusão mútua. Cada procedimento de acesso ao recurso deve obter o semáforo antes de iniciar e liberar o semáforo ao concluir;
    - Um invariante (condição sobre as variáveis internas do monitor) sobre o estado interno do recurso.

Mecanismos de Sincronização de Processos

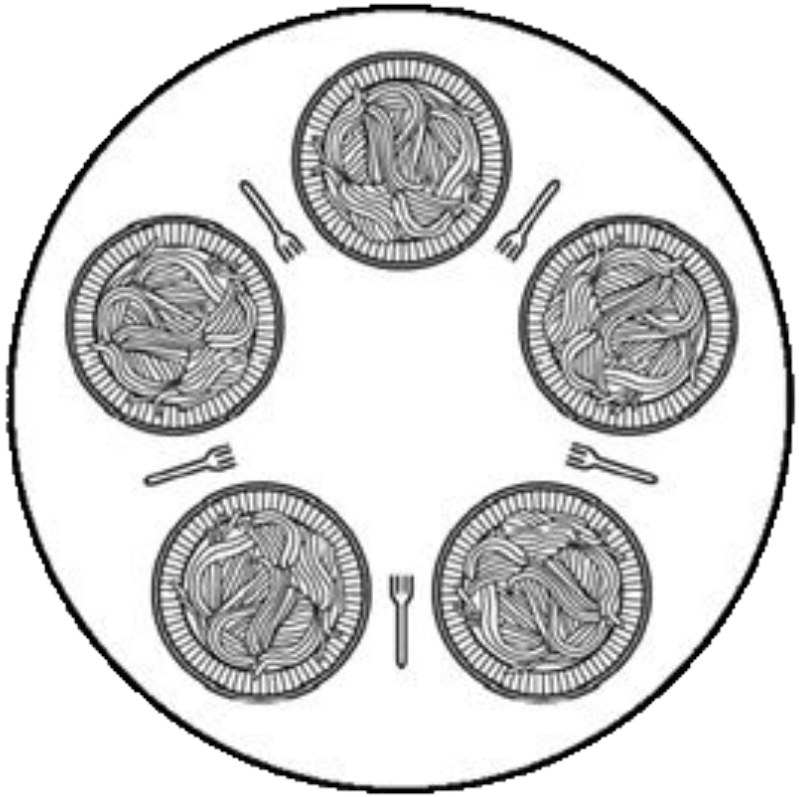
- Monitores
  - Definição de um monitor

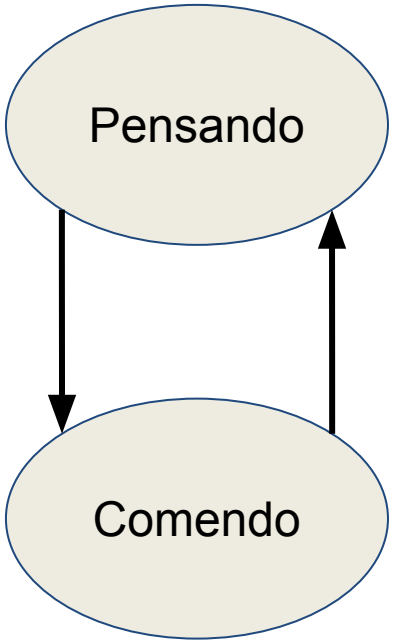
```
monitor nome do monitor {

  // Declaração de variáveis compartilhadas

  procedimento P1() {...}

  procedimento P2() {...}

  procedimento Pn() {...}

  código de inicialização (...) { ... }

}
```

Mecanismos de Sincronização de Processos

- Monitores
  - O construtor de um monitor assegura que somente um processo de cada vez fique ativo dentro do monitor.
  - O construtor de um monitor é do tipo *condition*.
  - *Condition* é um tipo especial de dado que permite operações do tipo *wait()* e *signal()*.
  - Exemplo: *condition* x, y;

Problemas clássicos: Jantar dos Filósofos

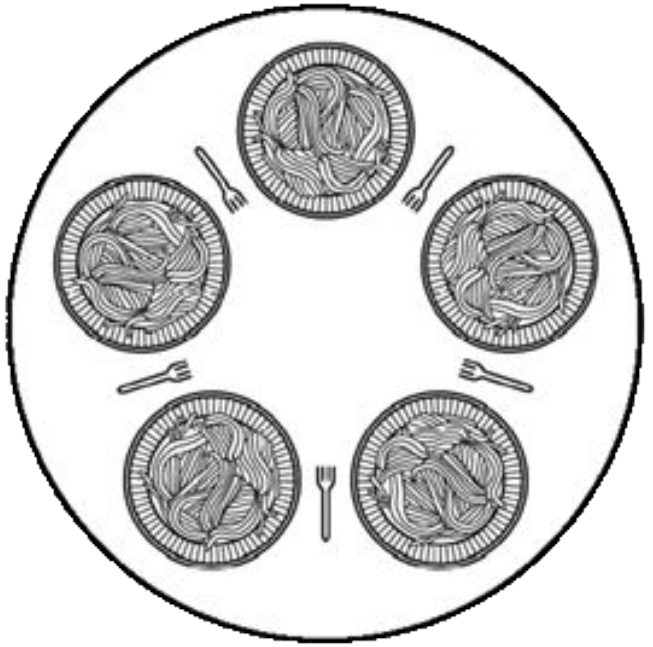




Solução sem bloqueios indefinidos?

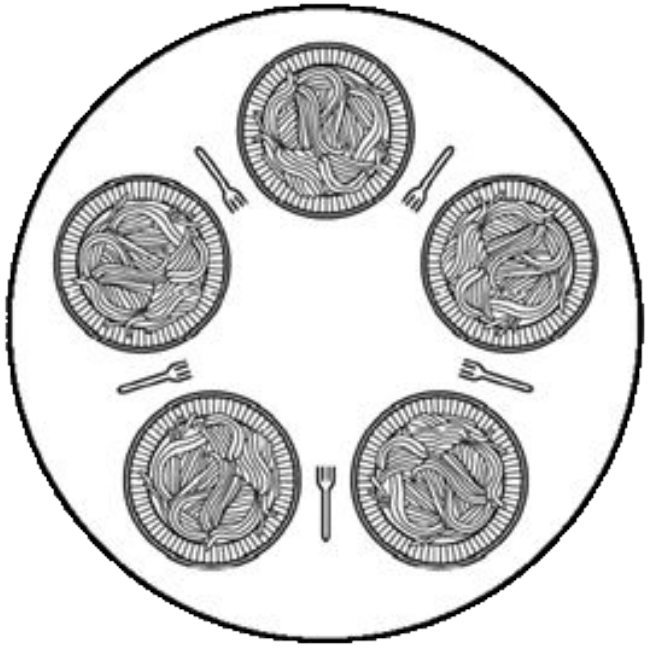
Jantar dos filósofos: solução 1

- Espere até o garfo desejado estar disponível e então pegue-o
- Bloqueio indefinido? Por que?



Jantar dos filósofos: solução 2

1) Pegue o garfo à esquerda
2) Se o garfo à direita está livre, pegue-o. Caso contrário, solte o garfo esquerdo, espere e vá para passo 1)

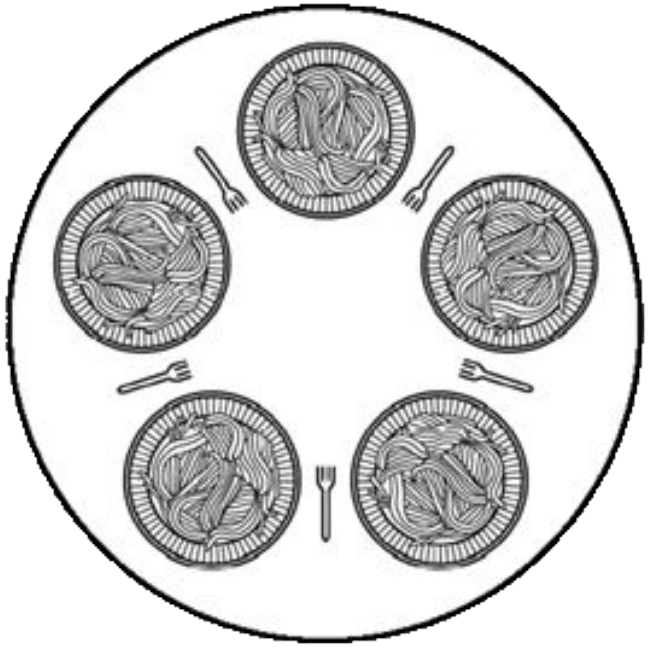


- Bloqueio indefinido?
- Por que?

Jantar dos filósofos: solução 3

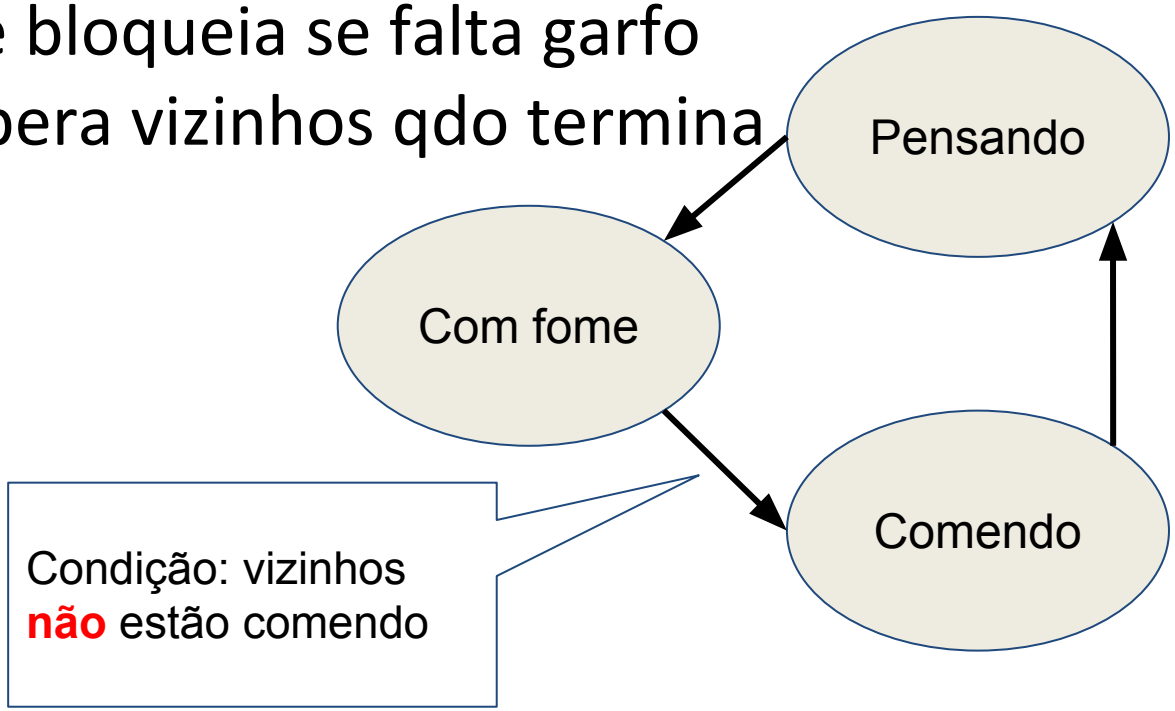
- Um único mutex garante uso exclusivo dos garfos

- Bloqueio indefinido?
- Eficiente?

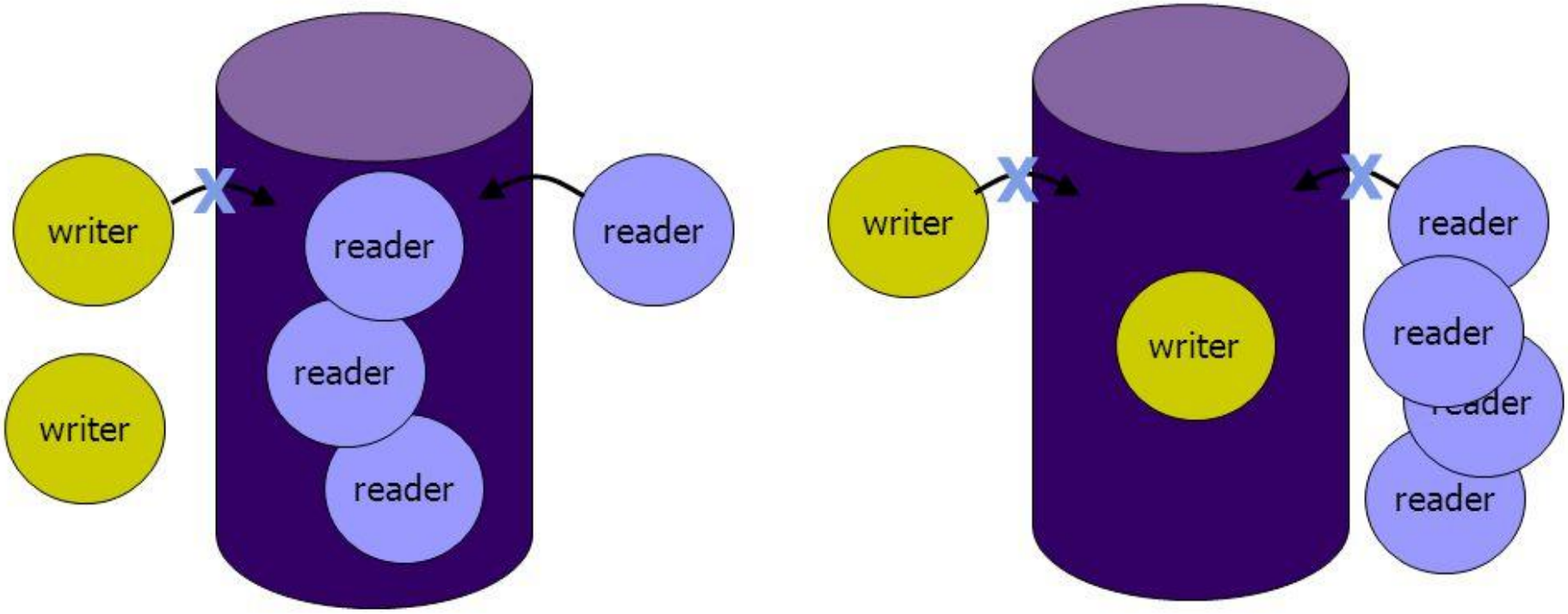


Jantar dos Filósofos: solução 4

- Manter o estado de cada filósofo
- Um mutex para região crítica
- Um mutex por filosofo
  - Filosofo se bloqueia se falta garfo
  - Filosofo libera vizinhos qdo termina



Problemas clássicos: Leitores e escritores



Leitores e escritores: solução

- Um mutex para proteger banco
- Primeiro leitor pega mutex
- Último leitor ativo libera mutex
- Escritor somente escreve qdo pega mutex

Questões

1) Como identificar o último leitor ativo?
2) Escritor pode esperar indeterminadamente?