



INSTITUTO FEDERAL

Norte de Minas Gerais

Campus Januária

Sistemas Distribuídos

- Sincronização -



Sincronização

- Em programação concorrente, **acessos simultâneos a recursos compartilhados** podem levar a uma **condição de corrida** (*race condition problem*).
- Uma condição de corrida ocorre quando **duas ou mais threads competem para acessar e/ou alterar dados compartilhados**.
- Como resultado, o comportamento e valores compartilhados **podem ser imprevisíveis e gerar erros**, dependendo das trocas de contexto das *threads*.



Condição de Corrida

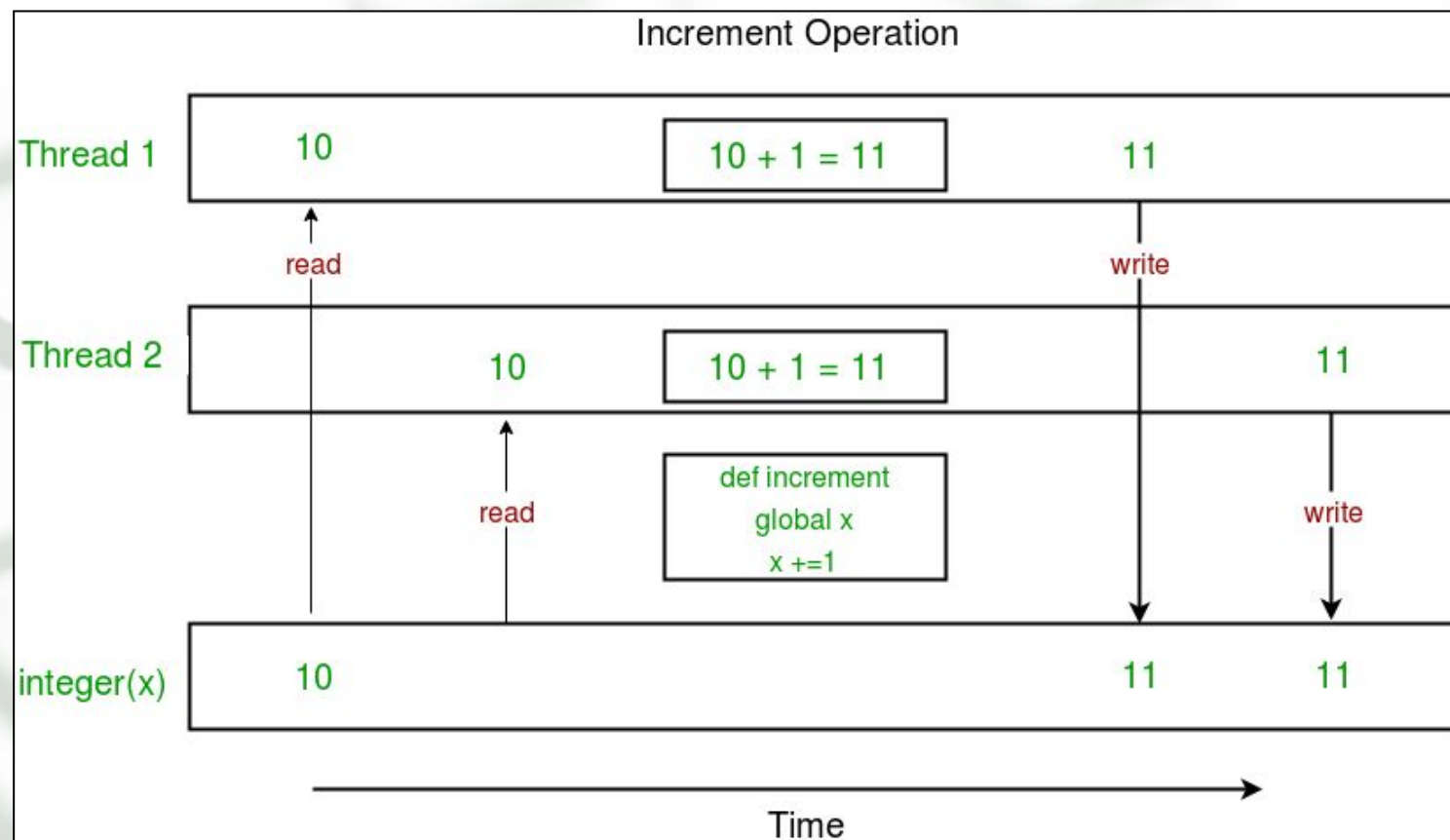
■ Exemplo de Condição de Corrida...

HORA	PESSOA A	PESSOA B
06:00	Olha a geladeira: sem leite	
06:05	Sai para a padaria	
06:10	Chega na padaria	Olha a geladeira: sem leite
06:15	Sai da padaria	Sai para a padaria
06:20	Chega em casa: guarda o leite	Chega na padaria
06:25		Sai da padaria
06:30		Chega em casa: Ops!



Condição de Corrida

■ Exemplo de Condição de Corrida...

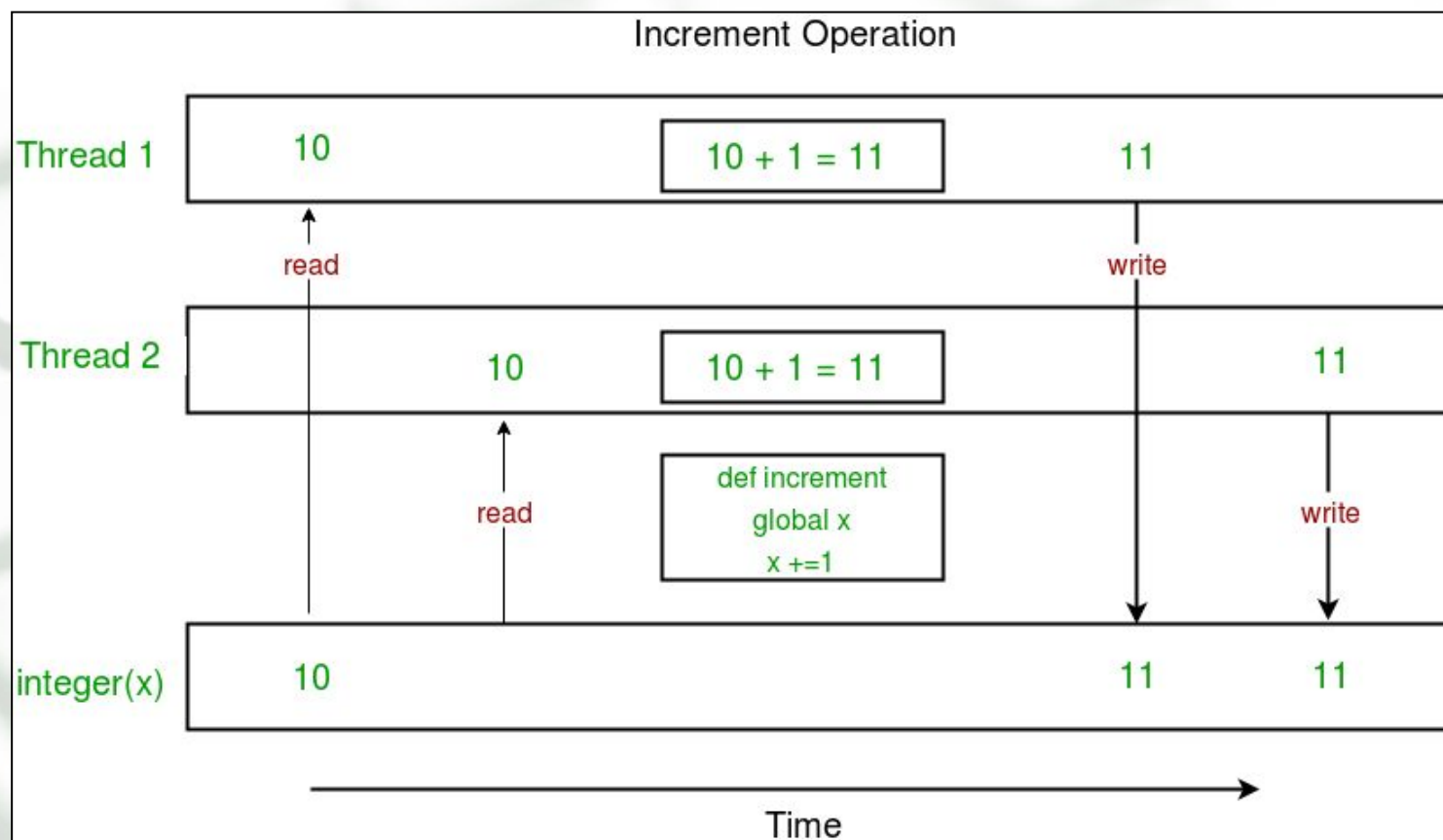




Condição de Corrida

■ Exemplo de Condição de

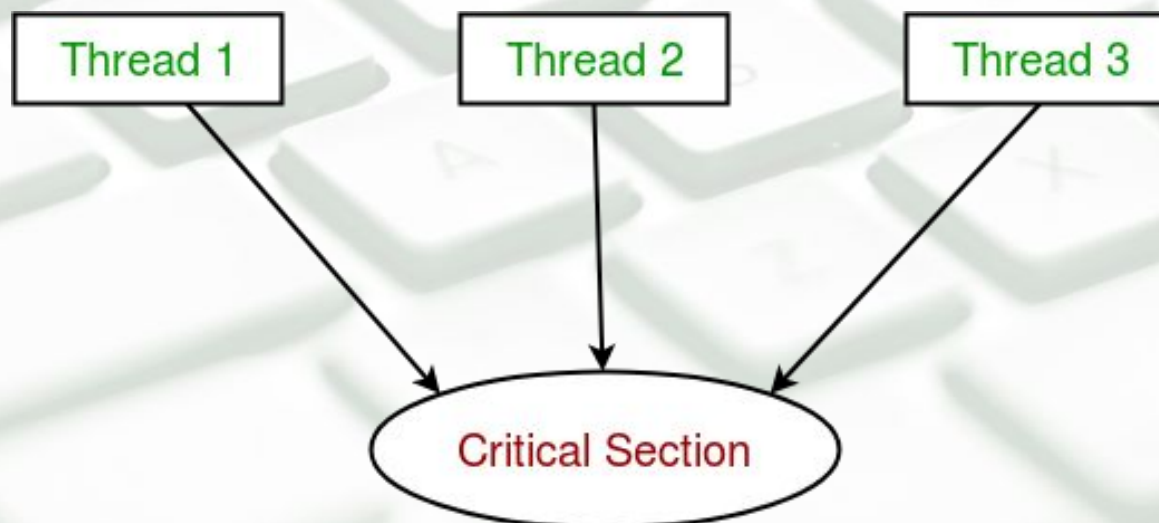
LABORATÓRIO #02.03 O PROBLEMA DA PIZZA!





Condição de Corrida

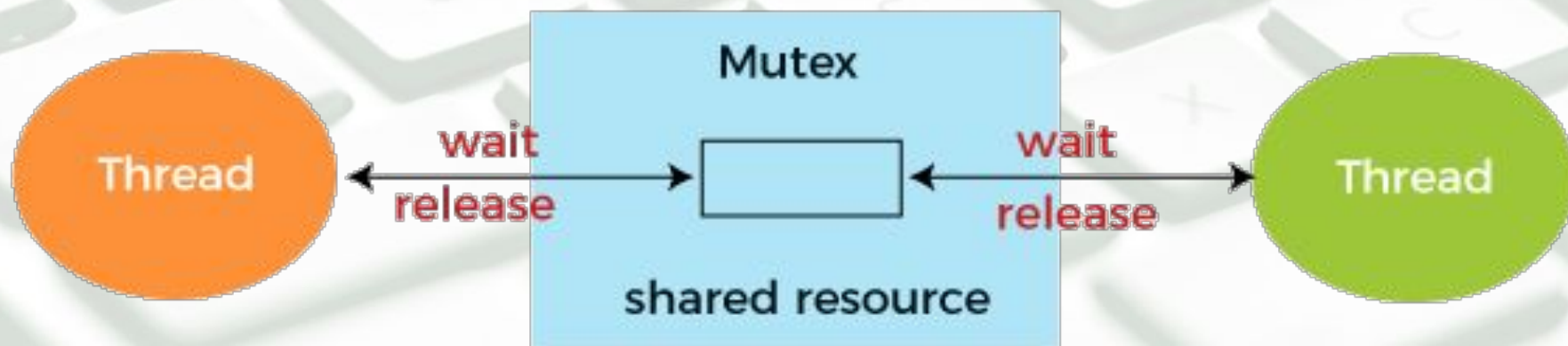
- A solução para o problema envolve dois conceitos...
- **Seção Crítica:** Trecho de código em que apenas uma thread pode executar integralmente por vez.





Exclusão Mútua

- **Exclusão Mútua - MUTEX** são mecanismos que garantem que duas ou mais threads concorrentes **não consigam executar simultaneamente** o trecho de programa sinalizado como **seção crítica**.





Exclusão Mútua

- A **Exclusão Mútua** é uma técnica para garantir acesso controlado à **Seção Crítica** de um problema.

```
do{  
    <<enter critical section>>  
    ...  
    ...  
    ...  
    <<leave critical section>>  
} while(1)
```

SEÇÃO CRÍTICA



Mutex

- Requisitos fundamentais para implementar um **Mutex**:
 - **Exclusão Mútua**: Se T_i adentrou à seção crítica, nenhuma outra thread poderá entrar nela simultaneamente.
 - **Progresso Garantido**: Se nenhuma thread está na seção crítica, qualquer thread pode acessá-la instantaneamente.
 - **Espera Limitada**: A espera para acesso à seção crítica não pode ser infinita.



MUTual EXclusion

■ MUTEX #01

```
ocupado = 0

def enter():
    while(ocupado):
        pass                #espera ocupada
    ocupado=1

def leave():
    ocupado=0
```



MUTual EXclusion

■ MUTEX #01

```
ocupado = 0
```

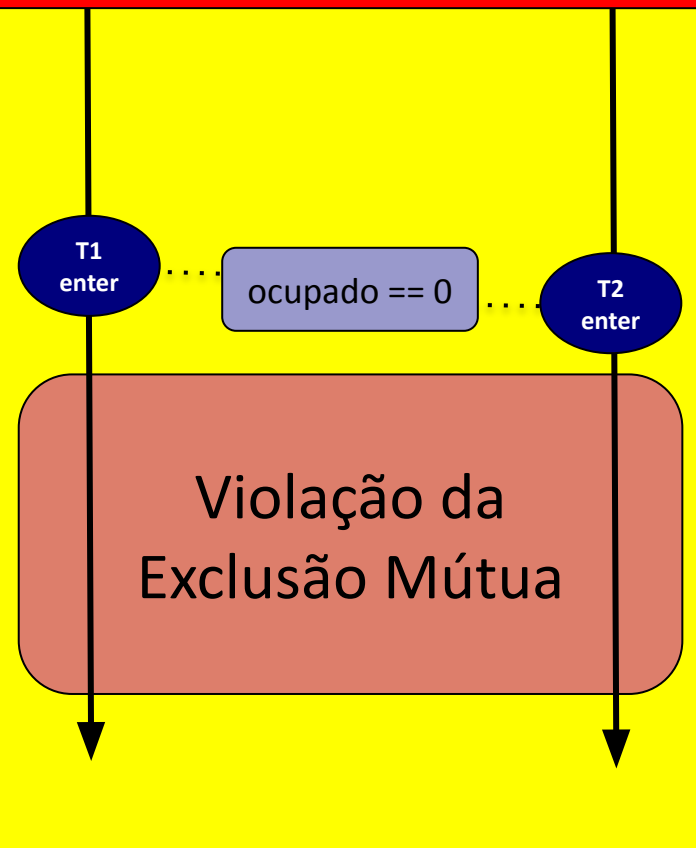
```
def enter():
    while(ocupado):
        pass
    ocupado=1
```

```
def leave():
    ocupado=0
```

Troca de Contexto

#es

FALHA!





MUTual EXclusion

■ MUTEX #02

```
turno = 1      #alterna entre turnos,  
                p.ex. thread par e impar  
  
def enter(id):  
    while((turno%2) != id):  
        pass      #espera ocupada  
  
def leave(id):  
    turno = (turno+1)%2
```



MUTual EXclusion

■ MUTEX #02

Exclusão Mútua OK!

```
turno = 1          #alterna entre turnos,  
                   p.ex. thread par e impar  
  
def enter(id):  
    while((turno%2) != id):  
        pass        #espera ocupada  
  
def leave(id):  
    turno = (turno+1)%2
```



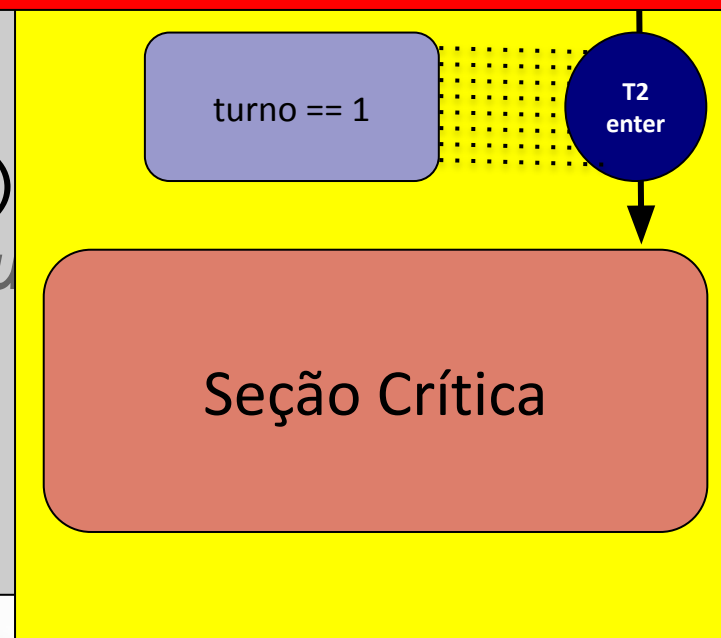

MUTual EXclusion

■ MUTEX #02

Exclusão Mútua OK!

Progresso Garantido?

```
turno = 1  
  
def enter(id):  
    while((turno%2) != id):  
        pass  
  
def leave(id):  
    turno = (turno+1)%2
```





MUTual EXclusion

■ MUTEX #03

```
interesse = [0, 0]  #indicar interesse em  
                    acessar seção crítica
```

```
def enter(id):  
    interesse[id] = 1  
    while(interesse[(id+1)%2])  
        pass        #espera ocupada
```

```
def leave(id):  
    interesse[id] = 0
```



MUTual EXclusion

■ MUTEX #03

Funciona?

```
interesse = [0, 0]  #indicar interesse em  
                    acessar seção crítica
```

```
def enter(id):  
    interesse[id] = 1  
    while(interesse[(id+1)%2])  
        pass        #espera ocupada
```

```
def leave(id):  
    interesse[id] = 0
```



MUTual EXclusion

■ MUTEX #03

Funciona?

Espera Limitada?

```
interesse = [0, 0]
```

```
def enter(id):
    interesse[id] = 1
    while(interesse[(id+1)%2])
        pass        #espera ocupada
```

```
def leave(id):
    interesse[id] = 0
```

Troca de Contexto

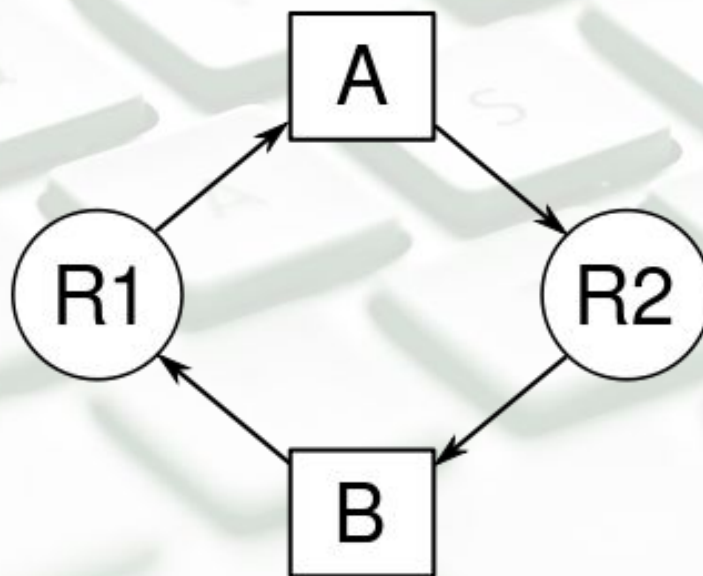
interesse = [1, 1]

Deadlock!



Deadlock

- **Deadlock** refere-se a uma situação em que ocorre um impasse, e duas ou mais *threads* ficam impedidas de continuar suas execuções, ou seja, ficam bloqueadas indefinidamente, **esperando umas pelas outras**.





INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Deadlock





MUTual EXclusion

■ MUTEX #04

```
interesse = [0, 0]           #indicar interesse  
turno = 0                   #indicar turno da vez  
  
def enter(id):  
    interesse[id] = 1  
    turno = id+1  
    while(interesse[(id+1)%2] && (turno%2 != id))  
        pass                #espera ocupada  
  
def leave(id):  
    interesse[id] = 0
```

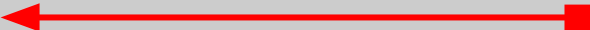


MUTual EXclusion

Funciona?

■ MUTEX #04

```
interesse = [0, 0]           #indicar interesse  
turno = 0                   #indicar turno da vez  
  
def enter(id):  
    interesse[id] = 1  
    turno = id+1  
    while(interesse[(id+1)%2] && (turno%2 != id))  
        pass                 #espera ocupada  
  
def leave(id):  
    interesse[id] = 0
```



Aponta Interesse



MUTual EXclusion

Funciona?

■ MUTEX #04

```
interesse = [0, 0]           #indicar interesse
turno = 0                   #indicar turno da vez

def enter(id):
    interesse[id] = 1        ← Aponta Interesse
    turno = id+1            ← Passa a vez para a outra thread
    while(interesse[(id+1)%2] && (turno%2 != id))
        pass                #espera ocupada

def leave(id):
    interesse[id] = 0
```



MUTual EXclusion

■ MUTEX #04

Funciona!!!

```
interesse = [0, 0]  
turno = 0
```

```
def enter(id):  
    interesse[id] = 1  
    turno = id+1
```

```
    while(interesse[(id+1)%2] && (turno%2 != id))  
        pass
```

#espera ocupada

```
def leave(id):  
    interesse[id] = 0
```

Algoritmo de
Peterson (1981)



Sync em Python

■ Técnicas de Sincronização do módulo **threading**:

- **Join()** => Barreira
- **Event()** => Evento
- **Timer()** => Temporizador

**Sem controle de
Seção Crítica**

-
- **Lock()** => Mutex Básico
 - **Semaphore()** => Semáforo
 - **Condition()** => Condição

**Controle de
Seção Crítica**



Sync em Python

■ Técnicas de Sincronização do módulo **threading**:

□ **Join()** => Barreira

□ **Event()** => Evento

□ **Timer()** => Temporizador

Sem controle de
Seção Crítica

□ **Lock()** => Mutex Básico

□ **Semaphore()** => Semáforo

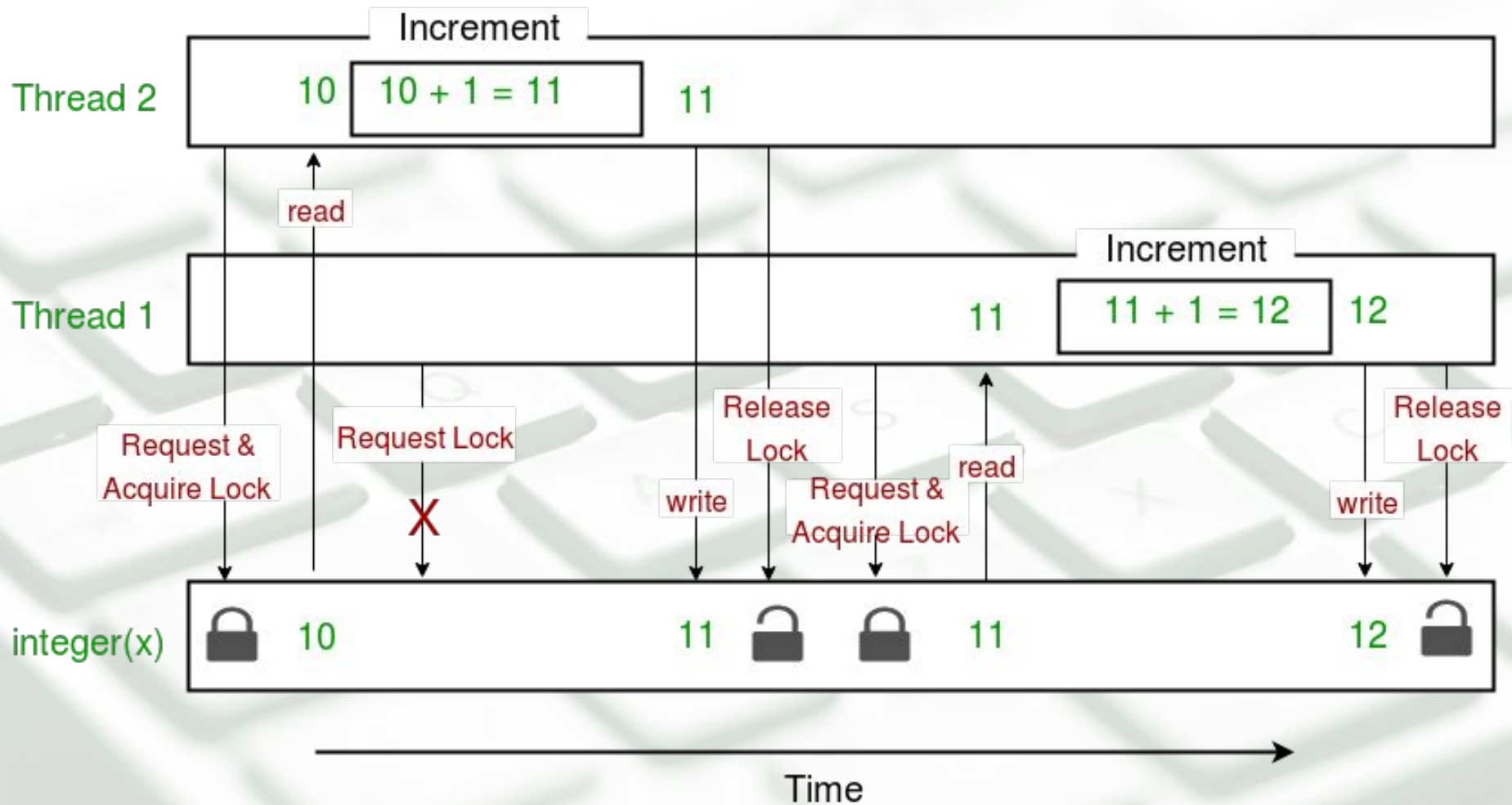
□ **Condition()** => Condição

5 Pontos para explicação e
demonstração de problema real

Seção Crítica

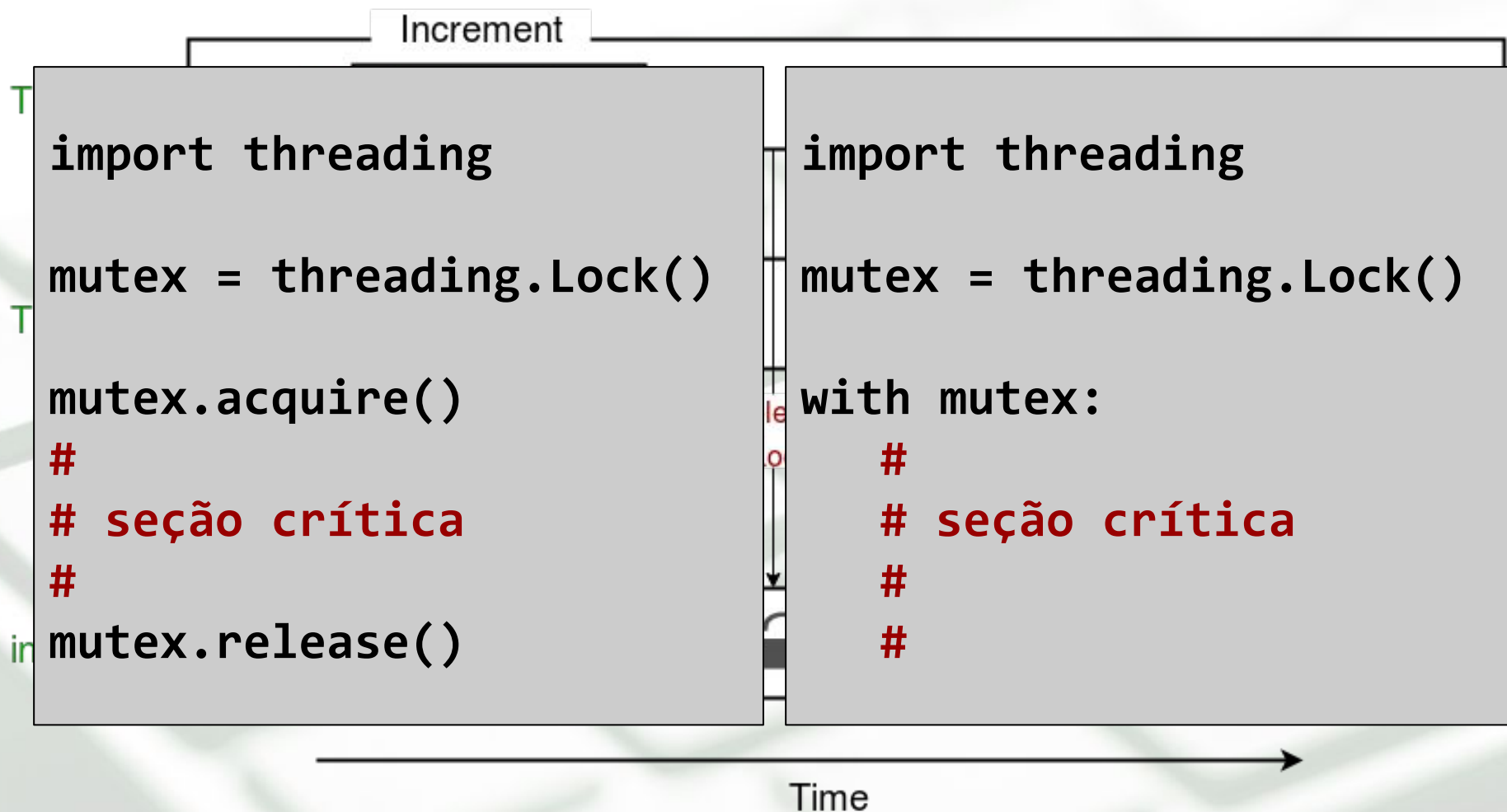


Lock()





Lock()





Laboratório #03.01

■ Problema Clássico de Sincronização

- Imagine que 20 macacos desejam atravessar um desfiladeiro através de uma corda.
- Alguns estão indo para a direção esquerda enquanto outros estão indo para a direita (aleatório).
- Porém, essa corda só suporta o peso de 1 macaco por vez, então, se um macaco já estiver atravessando, todos os outros devem esperar.
- Os macacos não chegam todos de uma vez... Existe um tempo aleatório (use `random.random()`) entre a chegada de um e outro.
- A travessia de um lado ao outro demora “1 segundo”.



INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

■ Problema Clássico



- Os macacos não têm um tempo aleatório de um e outro.
- A travessia de um

Macaco 1 quer ir para esquerda
 1 Macacos esperando travessia
 Macaco 1 atravessando para esquerda
 Macaco 2 quer ir para direita
 Macaco 3 quer ir para esquerda
 Macaco 4 quer ir para direita
 3 Macacos esperando travessia
 Macaco 2 atravessando para direita
 Macaco 5 quer ir para esquerda
 Macaco 6 quer ir para direita
 4 Macacos esperando travessia
 Macaco 3 atravessando para esquerda
 Macaco 7 quer ir para direita
 4 Macacos esperando travessia
 Macaco 4 atravessando para direita
 Macaco 8 quer ir para esquerda
 Macaco 9 quer ir para esquerda
 Macaco 10 quer ir para direita
 6 Macacos esperando travessia
 (...)



Semaphore()

- Semáforo é outra estrutura clássica de sincronização, proposta por Dijkstra.
 - Consiste em *“uma trava associada a um contador”*

Over seinpalen.

Wij beschouwen een aantal onderling "zwak gekoppelde", in zichzelf sequentiële processen. Onder de "zwakke koppeling" versta ik, dat ze op bepaalde punten rekening met elkaar kunnen moeten houden. Als bv. een aantal processen af en toe wel eens van een of andere faciliteit gebruik wil maken, die maar een proces tegelijk kan bedienen, dan betekent dit, dat de processen wel eens even op elkaar kunnen moeten wachten. Als het ene proces informatie verwerkt, dat door een ander proces geleverd moet worden, dan is het ook duidelijk, dat het eerste op het laatste kan moeten wachten. M.a.w. de processen moeten ten opzichte van elkaar in zekere mate gesynchroniseerd kunnen worden.



Semaphore()

- Diversas linguagem implementam versões diferentes...
 - **Semáforos Limitados:** “release” acima de um limite gera uma exceção (*BoundedSemaphore*);
 - **Semáforos Binários:** só assume valores 0 e 1 (mutex convencional);
 - **Semáforos Testáveis:** Verifica se há possibilidade de *Acquire* antes de ser bloqueado;



Semaphore()

- **Semaphore.acquire()**
 - Se contador == 0, aguarda liberação (travado)
 - Se contador > 0, decrementa e adentra à seção crítica

- **Semaphore.release()**
 - Incrementa contador
 - Se há tarefas esperando, apenas 1 é acordada.



Semaphore()

- Semaphore acquire()

```
import threading
```

```
semaforo = threading.Semaphore(4)
```

```
# decrementa semáforo
```

```
semaforo.acquire()
```

```
#
```

```
# seção crítica
```

```
#
```

```
# incrementa semáforo
```

```
semaforo.release()
```

o (travado)

entra à seção crítica

```
# gerenciador contexto
```

```
with semaforo:
```

```
#
```

```
# seção crítica
```

```
#
```

```
#
```




Laboratório #03.02

■ *Lembra do Problema dos Macacos?*

- Agora nossos amigos descobriram uma ponte que suporta o peso de até 3 macacos durante a travessia.
- Entretanto, como essa ponte é muito estreita, os macacos devem atravessar apenas em um sentido por vez.
- Ou seja... Quem estiver indo para a esquerda, não pode encontrar macacos indo para a direita.
- Faça a implementação desta solução, usando Semáforo e Mutex.



■ Lembra do Problema



- Faça a implementação do Problema dos Macacos com o Semáforo e Monitor

Direção Atual: direita

Macaco 1 quer ir para direita

Macaco 1 iniciou travessia para direita

Macaco 2 quer ir para esquerda

Macaco 3 quer ir para direita

Macaco 3 iniciou travessia para direita

Macaco 1 chegou no destino

Macaco 4 quer ir para esquerda

Macaco 3 chegou no destino

Macaco 5 quer ir para esquerda

Macaco 6 quer ir para esquerda

Direção Atual: esquerda

Macaco 5 iniciou travessia para esquerda

Macaco 4 iniciou travessia para esquerda

Macaco 2 iniciou travessia para esquerda

Macaco 7 quer ir para esquerda

Macaco 4 chegou no destino

Macaco 6 iniciou travessia para esquerda

Macaco 5 chegou no destino

Macaco 2 chegou no destino

Macaco 7 iniciou travessia para esquerda



Buffer Limitado

- **Buffer Limitado é um problema clássico em sistemas concorrentes:**
- Considere um *buffer* limitado, de tamanho N.
- Threads **TPs** produzem conteúdo e registram no *buffer*.
- Threads **TCs** consomem (retiram) dados do *buffer*, na ordem em que foram inseridos.
- Se em determinado instante o buffer está cheio:
 - TPs devem ser bloqueados.
- Se em determinado instante o buffer está vazio:
 - TCs devem ser bloqueados.
- TPs e TCs não podem acessar o buffer simultaneamente.



Buffer Limitado

- **Buffer Limitado é um problema clássico em sistemas concorrentes.**

- Consiste em:

- Threads

- Threads

- ordem

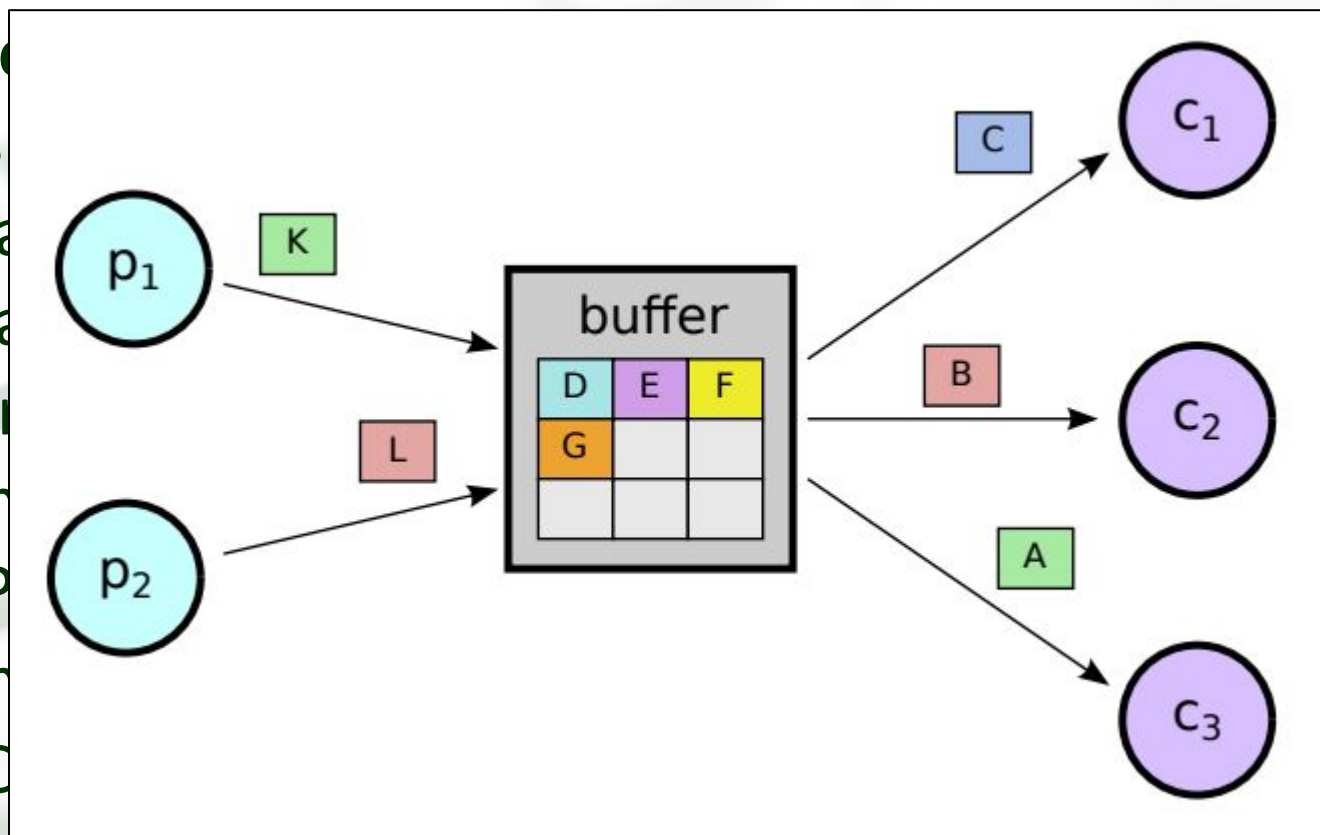
- Se entrar

- TP

- Se entrar

- TC

- TPs e TCs não podem acessar o buffer simultaneamente.



buffer.

r, na



Laboratório #03.03

- Re-implemente o laboratório das *threads* contadoras (**problema da *pizza***), mas agora sob perspectiva do problema do **Buffer Limitado**. Considere para isso...
 - Um buffer limitado de tamanho 10.
 - Duas *threads* produtoras (1 seg/produto cada).
 - Três *threads* consumidoras (3 seg/produto cada).
- O resultado deve ser a impressão da sequência numérica de 1 a 100, respeitando-se o limite imposto pelo buffer.

■ Re-implementando o problema

- Um
- Dua
- Três

■ O resultado numérico pelo bu

```

Consumidor c0 em ação.
Consome: []
Consumidor c1 em ação.
Consome: []
Consumidor c2 em ação.
Consome: []
Produtor p0 Produz: [1]
Consumidor c0 obteve: -----> 1
Produtor p1 Produz: [2]
Consumidor c1 obteve: -----> 2
Produtor p0 Produz: [3]
Produtor p1 Produz: [3, 4]
Consumidor c2 obteve: -----> 3
Produtor p0 Produz: [4, 5]
Produtor p1 Produz: [4, 5, 6]
Consumidor c0 em ação.
Consumidor c1 em ação.
Consome: [4, 5, 6]
Consome: [4, 5, 6]
Consumidor c0 obteve: -----> 4
Consumidor c1 obteve: -----> 5
Produtor p0 Produz: [6, 7]
Produtor p1 Produz: [6, 7, 8]
Consumidor c2 em ação.
Consome: [6, 7, 8]
Consumidor c2 obteve: -----> 6
Produtor p0 Produz: [7, 8, 9]
Produtor p1 Produz: [7, 8, 9, 10]
Produtor p0 Produz: [7, 8, 9, 10, 11]
Produtor p1 Produz: [7, 8, 9, 10, 11, 12]
Consumidor c0 em ação.
Consumidor c1 em ação.
Consome: [7, 8, 9, 10, 11, 12]
Consome: [7, 8, 9, 10, 11, 12]
Consumidor c0 obteve: -----> 7
Consumidor c1 obteve: -----> 8
Produtor p0 Produz: [9, 10, 11, 12, 13]
Produtor p1 Produz: [9, 10, 11, 12, 13, 14]
Produtor p0 Produz: [9, 10, 11, 12, 13, 14, 15]

```

ras

o

...

osto