

# QXD0013 - Sistemas Operacionais

## Comunicação entre Processos II

Thiago Werlley Bandeira da Silva<sup>1</sup>

<sup>1</sup>Universidade Federal do Ceará, Brazil

09/11/2021

# Exclusão Mútua - Mutexes

---



# Exclusão Mútua - Mutexes

---

- Quando a capacidade do semáforo de fazer contagem não é necessária, uma versão simplificada às vezes é usada **chamada de mutex**



# Exclusão Mútua - Mutexes

---

- Quando a capacidade do semáforo de fazer contagem não é necessária, uma versão simplificada às vezes é usada **chamada de mutex**
- Abreviação de *Mutual Exclusion*



# Exclusão Mútua - Mutexes

---

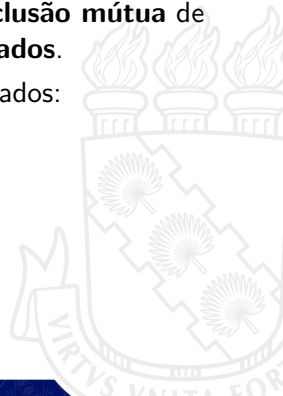
- Quando a capacidade do semáforo de fazer contagem não é necessária, uma versão simplificada às vezes é usada **chamada de mutex**
- Abreviação de *Mutual Exclusion*
- **Mutexes** são bons somente para gerenciar a **exclusão mútua** de algum **recurso** ou trecho de **código compartilhados**.



# Exclusão Mútua - Mutexes

---

- Quando a capacidade do semáforo de fazer contagem não é necessária, uma versão simplificada às vezes é usada **chamada de mutex**
- Abreviação de *Mutual Exclusion*
- **Mutexes** são bons somente para gerenciar a **exclusão mútua** de algum **recurso** ou trecho de **código compartilhados**.
- **Mutex** é uma variável compartilhada de dois estados: **destravado/travado**



# Exclusão Mútua - Mutexes

---

- Quando a capacidade do semáforo de fazer contagem não é necessária, uma versão simplificada às vezes é usada **chamada de mutex**
- Abreviação de *Mutual Exclusion*
- **Mutexes** são bons somente para gerenciar a **exclusão mútua** de algum **recurso** ou trecho de **código compartilhados**.
- **Mutex** é uma variável compartilhada de dois estados: **destravado/travado**
- **Duas rotinas** são usadas com **mutexes** (thread ou processo):  
*mutex\_lock*(destravado)/*mutex\_unlock*(travado)

# Exclusão Mútua - Mutexes

---

- Quando a capacidade do semáforo de fazer contagem não é necessária, uma versão simplificada às vezes é usada **chamada de mutex**
- Abreviação de *Mutual Exclusion*
- **Mutexes** são bons somente para gerenciar a **exclusão mútua** de algum **recurso** ou trecho de **código compartilhados**.
- **Mutex** é uma variável compartilhada de dois estados: **destravado/travado**
- **Duas rotinas** são usadas com **mutexes** (thread ou processo): *mutex\_lock*(destravado)/*mutex\_unlock*(travado)
- Mutexes são **muito simples**, eles podem ser facilmente **implementados no espaço do usuário**, desde que uma **instrução TSL ou XCHG** esteja disponível.



# Exclusão Mútua - Mutexes

---



# Exclusão Mútua - Mutexes

---

- Segue testando a trava repetidamente (**espera ocupada**):



# Exclusão Mútua - Mutexes

---

- Segue testando a trava repetidamente (**espera ocupada**):
  - Processos: escalonador realiza alternância



# Exclusão Mútua - Mutexes

---

- Segue testando a trava repetidamente (**espera ocupada**):
  - Processos: escalonador realiza alternância
  - Threads: não há alternância



# Exclusão Mútua - Mutexes

---

- Segue testando a trava repetidamente (**espera ocupada**):
  - Processos: escalonador realiza alternância
  - Threads: não há alternância
- Thread precisa bloquear-se



# Exclusão Mútua - Mutexes

- Segue testando a trava repetidamente (**espera ocupada**):
  - Processos: escalonador realiza alternância
  - Threads: não há alternância
- Thread precisa bloquear-se
  - *thread\_yield*: dentro do espaço de usuário

mutex\_lock:

TSL REGISTER,MUTEX

CMP REGISTER,#0

JZE ok

CALL thread\_yield

JMP mutex\_lock

ok:

RET

| copy mutex to register and set mutex to 1

| was mutex zero?

| if it was zero, mutex was unlocked, so return

| mutex is busy; schedule another thread

| try again

| return to caller; critical region entered

mutex\_unlock:

MOVE MUTEX,#0

RET

| store a 0 in mutex

| return to caller

# Exclusão Mútua - Mutexes

- Segue testando a trava repetidamente (**espera ocupada**):
  - Processos: escalonador realiza alternância
  - Threads: não há alternância
- Thread precisa bloquear-se
  - *thread\_yield*: dentro do espaço de usuário

mutex\_lock:

TSL REGISTER,MUTEX

CMP REGISTER,#0

JZE ok

CALL thread\_yield

JMP mutex\_lock

ok:

RET

| copy mutex to register and set mutex to 1

| was mutex zero?

| if it was zero, mutex was unlocked, so return

| mutex is busy; schedule another thread

| try again

| return to caller; critical region entered

Tentativa acesso desimpedido

mutex\_unlock:

MOVE MUTEX,#0

RET

| store a 0 in mutex

| return to caller

MUTEX = 0

REGISTER = \*

# Exclusão Mútua - Mutexes

- Segue testando a trava repetidamente (**espera ocupada**):
  - Processos: escalonador realiza alternância
  - Threads: não há alternância
- Thread precisa bloquear-se
  - *thread\_yield*: dentro do espaço de usuário

mutex\_lock:

→ TSL REGISTER,MUTEX

CMP REGISTER,#0

JZE ok

CALL thread\_yield

JMP mutex\_lock

ok: RET

copy mutex to register and set mutex to 1

was mutex zero?

if it was zero, mutex was unlocked, so return

mutex is busy; schedule another thread

try again

return to caller; critical region entered

Tentativa acesso desimpedido

mutex\_unlock:

MOVE MUTEX,#0

RET

store a 0 in mutex

return to caller

MUTEX = 1

REGISTER = 0



# Exclusão Mútua - Mutexes

- Segue testando a trava repetidamente (**espera ocupada**):
  - Processos: escalonador realiza alternância
  - Threads: não há alternância
- Thread precisa bloquear-se
  - *thread\_yield*: dentro do espaço de usuário

mutex\_lock:

```

    TSL REGISTER,MUTEX
    → CMP REGISTER,#0
    JZE ok
    CALL thread_yield
    JMP mutex_lock
ok:  RET
    
```

```

| copy mutex to register and set mutex to 1
| was mutex zero?
| if it was zero, mutex was unlocked, so return
| mutex is busy; schedule another thread
| try again
| return to caller; critical region entered
    
```

Tentativa acesso desimpedido

mutex\_unlock:

```

    MOVE MUTEX,#0
    RET
    
```

```

| store a 0 in mutex
| return to caller
    
```

MUTEX = 1  
REGISTER = 0

# Exclusão Mútua - Mutexes

- Segue testando a trava repetidamente (**espera ocupada**):
  - Processos: escalonador realiza alternância
  - Threads: não há alternância
- Thread precisa bloquear-se
  - *thread\_yield*: dentro do espaço de usuário

mutex\_lock:

TSL REGISTER,MUTEX

CMP REGISTER,#0

→ JZE ok

CALL thread\_yield

JMP mutex\_lock

ok: RET

| copy mutex to register and set mutex to 1  
| was mutex zero?

| if it was zero, mutex was unlocked, so return

| mutex is busy; schedule another thread  
| try again

| return to caller; critical region entered

Tentativa acesso desimpedido

mutex\_unlock:

MOVE MUTEX,#0

RET

| store a 0 in mutex  
| return to caller

MUTEX = 1

REGISTER = 0

# Exclusão Mútua - Mutexes

- Segue testando a trava repetidamente (**espera ocupada**):
  - Processos: escalonador realiza alternância
  - Threads: não há alternância
- Thread precisa bloquear-se
  - *thread\_yield*: dentro do espaço de usuário

mutex\_lock:

```
TSL REGISTER,MUTEX
CMP REGISTER,#0
JZE ok
CALL thread_yield
JMP mutex_lock
```

```
| copy mutex to register and set mutex to 1
| was mutex zero?
| if it was zero, mutex was unlocked, so return
| mutex is busy; schedule another thread
| try again
```

ok: → RET

```
| return to caller; critical region entered
```

Tentativa acesso desimpedido

mutex\_unlock:

```
MOVE MUTEX,#0
RET
```

```
| store a 0 in mutex
| return to caller
```

MUTEX = 1  
REGISTER = 0

# Exclusão Mútua - Mutexes

- Segue testando a trava repetidamente (**espera ocupada**):
  - Processos: escalonador realiza alternância
  - Threads: não há alternância
- Thread precisa bloquear-se
  - *thread\_yield*: dentro do espaço de usuário

mutex\_lock:

TSL REGISTER,MUTEX

CMP REGISTER,#0

JZE ok

CALL thread\_yield

JMP mutex\_lock

ok:

RET

| copy mutex to register and set mutex to 1

| was mutex zero?

| if it was zero, mutex was unlocked, so return

| mutex is busy; schedule another thread

| try again

| return to caller; critical region entered

Tentativa acesso impedido

mutex\_unlock:

MOVE MUTEX,#0

RET

| store a 0 in mutex

| return to caller

MUTEX = 1

REGISTER = \*

# Exclusão Mútua - Mutexes

- Segue testando a trava repetidamente (**espera ocupada**):
  - Processos: escalonador realiza alternância
  - Threads: não há alternância
- Thread precisa bloquear-se
  - *thread\_yield*: dentro do espaço de usuário

mutex\_lock:

→ TSL REGISTER,MUTEX

CMP REGISTER,#0

JZE ok

CALL thread\_yield

JMP mutex\_lock

ok: RET

copy mutex to register and set mutex to 1

was mutex zero?

if it was zero, mutex was unlocked, so return

mutex is busy; schedule another thread

try again

return to caller; critical region entered

Tentativa acesso impedido

mutex\_unlock:

MOVE MUTEX,#0

RET

store a 0 in mutex

return to caller

MUTEX = 1

REGISTER = 1

# Exclusão Mútua - Mutexes

- Segue testando a trava repetidamente (**espera ocupada**):
  - Processos: escalonador realiza alternância
  - Threads: não há alternância
- Thread precisa bloquear-se
  - *thread\_yield*: dentro do espaço de usuário

mutex\_lock:

```

    TSL REGISTER,MUTEX
    → CMP REGISTER,#0
    JZE ok
    CALL thread_yield
    JMP mutex_lock
ok:   RET
    
```

```

| copy mutex to register and set mutex to 1
| was mutex zero?
| if it was zero, mutex was unlocked, so return
| mutex is busy; schedule another thread
| try again
| return to caller; critical region entered
    
```

Tentativa acesso impedido

mutex\_unlock:

```

    MOVE MUTEX,#0
    RET
    
```

```

| store a 0 in mutex
| return to caller
    
```

MUTEX = 1  
REGISTER = 1

# Exclusão Mútua - Mutexes

- Segue testando a trava repetidamente (**espera ocupada**):
  - Processos: escalonador realiza alternância
  - Threads: não há alternância
- Thread precisa bloquear-se
  - *thread\_yield*: dentro do espaço de usuário

mutex\_lock:

TSL REGISTER,MUTEX

CMP REGISTER,#0

JZE ok

→ CALL thread\_yield

JMP mutex\_lock

ok: RET

| copy mutex to register and set mutex to 1

| was mutex zero?

| if it was zero, mutex was unlocked, so return

| mutex is busy; schedule another thread

| try again

| return to caller; critical region entered

Tentativa acesso impedido

mutex\_unlock:

MOVE MUTEX,#0

RET

| store a 0 in mutex

| return to caller

MUTEX = 1

REGISTER = 1

# Exclusão Mútua - Mutexes

---





# Exclusão Mútua - Mutexes

---

- Hipótese → com um pacote de threads de espaço de usuário não há um problema com múltiplas threads terem acesso ao mesmo mutex, já que todos os threads operam em um espaço de **endereçamento comum**



## Exclusão Mútua - Mutexes

---

- Hipótese → com um pacote de threads de espaço de usuário não há um problema com múltiplas threads terem acesso ao mesmo mutex, já que todos os threads operam em um espaço de **endereçamento comum**
- Threads: Espaço de usuário



# Exclusão Mútua - Mutexes

---

- Hipótese → com um pacote de threads de espaço de usuário não há um problema com múltiplas threads terem acesso ao mesmo mutex, já que todos os threads operam em um espaço de **endereçamento comum**
- Threads: Espaço de usuário
- Processos:



# Exclusão Mútua - Mutexes

---

- Hipótese → com um pacote de threads de espaço de usuário não há um problema com múltiplas threads terem acesso ao mesmo mutex, já que todos os threads operam em um espaço de **endereçamento comum**
- Threads: Espaço de usuário
- Processos:
  - Centralizado no núcleo → Acesso via chamadas do sistema

# Exclusão Mútua - Mutexes

---

- Hipótese → com um pacote de threads de espaço de usuário não há um problema com múltiplas threads terem acesso ao mesmo mutex, já que todos os threads operam em um espaço de **endereçamento comum**
- Threads: Espaço de usuário
- Processos:
  - Centralizado no núcleo → Acesso via chamadas do sistema
  - SO oferece meios de compartilhamento de estruturas de dados

# Exclusão Mútua - Mutexes

---

- Hipótese → com um pacote de threads de espaço de usuário não há um problema com múltiplas threads terem acesso ao mesmo mutex, já que todos os threads operam em um espaço de **endereçamento comum**
- Threads: Espaço de usuário
- Processos:
  - Centralizado no núcleo → Acesso via chamadas do sistema
  - SO oferece meios de compartilhamento de estruturas de dados
- Mecanismo mais eficiente em threads de usuário

# Exclusão Mútua - *Futexes*

---



## Exclusão Mútua - *Futexes*

---

- Com o **paralelismo** cada vez maior, a **sincronização** eficiente e o **travamento** são muito importantes para o **desempenho**.





## Exclusão Mútua - *Futexes*

---

- Com o **paralelismo** cada vez maior, a **sincronização** eficiente e o **travamento** são muito importantes para o **desempenho**.
- **Travamento** → desperdiçam ciclos de CPU



## Exclusão Mútua - *Futexes*

---

- Com o **paralelismo** cada vez maior, a **sincronização** eficiente e o **travamento** são muito importantes para o **desempenho**.
- **Travamento** → desperdiçam ciclos de CPU
- Chaveamento contínuo para o núcleo → Alto custo



## Exclusão Mútua - *Futexes*

---

- Com o **paralelismo** cada vez maior, a **sincronização** eficiente e o **travamento** são muito importantes para o **desempenho**.
- **Travamento** → desperdiçam ciclos de CPU
- Chaveamento contínuo para o núcleo → Alto custo
- Solução de meio termo para Linux implementando travamento básico



## Exclusão Mútua - *Futexes*

---

- Com o **paralelismo** cada vez maior, a **sincronização** eficiente e o **travamento** são muito importantes para o **desempenho**.
- **Travamento** → desperdiçam ciclos de CPU
- Chaveamento contínuo para o núcleo → Alto custo
- Solução de meio termo para Linux implementando travamento básico
  - mutex rápido de espaço de usuário (*Futex: fast user space mutex*)

## Exclusão Mútua - *Futexes*

---

- Com o **paralelismo** cada vez maior, a **sincronização** eficiente e o **travamento** são muito importantes para o **desempenho**.
- **Travamento** → desperdiçam ciclos de CPU
- Chaveamento contínuo para o núcleo → Alto custo
- Solução de meio termo para Linux implementando travamento básico
  - mutex rápido de espaço de usuário (*Futex: fast user space mutex*)
- **Futex** – (semelhante com mutex), mas **evita adentrar o núcleo**, a não ser que ele realmente tenha de fazê-lo.

## Exclusão Mútua - *Futexes*

---

- Com o **paralelismo** cada vez maior, a **sincronização** eficiente e o **travamento** são muito importantes para o **desempenho**.
- **Travamento** → desperdiçam ciclos de CPU
- Chaveamento contínuo para o núcleo → Alto custo
- Solução de meio termo para Linux implementando travamento básico
  - mutex rápido de espaço de usuário (*Futex: fast user space mutex*)
- **Futex** – (semelhante com mutex), mas **evita adentrar o núcleo**, a não ser que ele realmente tenha de fazê-lo.
- Instrução atômica em modo usuário (TSL)

## Exclusão Mútua - *Futexes*

---

- Com o **paralelismo** cada vez maior, a **sincronização** eficiente e o **travamento** são muito importantes para o **desempenho**.
- **Travamento** → desperdiçam ciclos de CPU
- Chaveamento contínuo para o núcleo → Alto custo
- Solução de meio termo para Linux implementando travamento básico
  - mutex rápido de espaço de usuário (*Futex: fast user space mutex*)
- **Futex** – (semelhante com mutex), mas **evita adentrar o núcleo**, a não ser que ele realmente tenha de fazê-lo.
- Instrução atômica em modo usuário (TSL)
- Chaveamento para o núcleo apenas quando:

## Exclusão Mútua - *Futexes*

---

- Com o **paralelismo** cada vez maior, a **sincronização** eficiente e o **travamento** são muito importantes para o **desempenho**.
- **Travamento** → desperdiçam ciclos de CPU
- Chaveamento contínuo para o núcleo → Alto custo
- Solução de meio termo para Linux implementando travamento básico
  - mutex rápido de espaço de usuário (*Futex: fast user space mutex*)
- **Futex** – (semelhante com mutex), mas **evita adentrar o núcleo**, a não ser que ele realmente tenha de fazê-lo.
- Instrução atômica em modo usuário (TSL)
- Chaveamento para o núcleo apenas quando:
  - thread Bloqueio



## Exclusão Mútua - *Futexes*

---

- Com o **paralelismo** cada vez maior, a **sincronização** eficiente e o **travamento** são muito importantes para o **desempenho**.
- **Travamento** → desperdiçam ciclos de CPU
- Chaveamento contínuo para o núcleo → Alto custo
- Solução de meio termo para Linux implementando travamento básico
  - mutex rápido de espaço de usuário (*Futex: fast user space mutex*)
- **Futex** – (semelhante com mutex), mas **evita adentrar o núcleo**, a não ser que ele realmente tenha de fazê-lo.
- Instrução atômica em modo usuário (TSL)
- Chaveamento para o núcleo apenas quando:
  - thread Bloqueio
  - thread Desbloqueio

## Exclusão Mútua - Mutexes e Pthread

---

Algumas chamadas de Pthreads relacionadas a mutexes.

| Thread call           | Description               |
|-----------------------|---------------------------|
| Pthread_mutex_init    | Create a mutex            |
| Pthread_mutex_destroy | Destroy an existing mutex |
| Pthread_mutex_lock    | Acquire a lock or block   |
| Pthread_mutex_trylock | Acquire a lock or fail    |
| Pthread_mutex_unlock  | Release a lock            |

# Exclusão Mútua - Variáveis de Condição e Pthread

---



# Exclusão Mútua - Variáveis de Condição e Pthread

---

- **Pthreads** proporcionam uma série de funções que podem ser usadas para **sincronizar threads**.



# Exclusão Mútua - Variáveis de Condição e Pthread

---

- **Pthreads** proporcionam uma série de funções que podem ser usadas para **sincronizar threads**.
- Mutex → Permitir/bloquear acesso de threads



# Exclusão Mútua - Variáveis de Condição e Pthread

---

- **Pthreads** proporcionam uma série de funções que podem ser usadas para **sincronizar threads**.
- Mutex → Permitir/bloquear acesso de threads
- Variável de condição → Bloqueio condicional



# Exclusão Mútua - Variáveis de Condição e Pthread

---

- **Pthreads** proporcionam uma série de funções que podem ser usadas para **sincronizar threads**.
- Mutex → Permitir/bloquear acesso de threads
- Variável de condição → Bloqueio condicional
  - Condição para bloqueio/despertar



# Exclusão Mútua - Variáveis de Condição e Pthread

---

- **Pthreads** proporcionam uma série de funções que podem ser usadas para **sincronizar threads**.
- Mutex → Permitir/bloquear acesso de threads
- Variável de condição → Bloqueio condicional
  - Condição para bloqueio/despertar
- Cabe ao programador assegurar que os threads as usem corretamente.





# Exclusão Mútua - Mutexes e Pthread

Algumas das chamadas de Pthreads relacionadas com **variáveis de condição**.

| Thread call            | Description                                  |
|------------------------|--|
| Pthread_cond_init      | Create a condition variable                  |
| Pthread_cond_destroy   | Destroy a condition variable                 |
| Pthread_cond_wait      | Block waiting for a signal                   |
| Pthread_cond_signal    | Signal another thread and wake it up         |
| Pthread_cond_broadcast | Signal multiple threads and wake all of them |

# Exclusão Mútua - Variáveis de Condição e Pthread

---

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;
```

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0;
```

```
/* how many numbers to produce */
```

```
/* buffer used between producer and consumer */
```

```
int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;

void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0; /* buffer used between producer and consumer */

void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;

void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Cria identificadores de threads

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;

void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Inicializa mutex

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;

void *consumer(void *ptr) /* consume data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Inicializa variável de condição



# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;

void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Inicializa variável de condição

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;

void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Cria thread do consumidor

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;

void *consumer(void *ptr) /* consume data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Cria thread do produtor

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0; /* buffer used between producer and consumer */

void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Espera saída do produtor

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0; /* buffer used between producer and consumer */

void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Espera saída do consumidor

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;

void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Destrói variável de Condição

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;

void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Destrói variável de Condição

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;

void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Destrói mutex



# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;

void *consumer(void *ptr) /* consume data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Cons. acessa região crítica

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;

void *consumer(void *ptr) /* consume data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Cons. libera mutex/espera cond

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;

void *consumer(void *ptr) /* consume data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Prod. acessa região crítica

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;

void *consumer(void *ptr) /* consume data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Prod. não entra no laço

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0; /* buffer used between producer and consumer */

void *consumer(void *ptr) /* consume data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Prod. gera novo item

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;

void *consumer(void *ptr) /* consume data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Prod. desperta Cons.

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0; /* buffer used between producer and consumer */

void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Cons. retira item do buffer

# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0; /* buffer used between producer and consumer */

void *consumer(void *ptr) /* consume data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Cons. acorda Prod.



# Exclusão Mútua - Variáveis de Condição e Pthread

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0;

void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *producer(void *ptr) /* produce data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Cons. sai da região crítica

# Monitores

---

**Com semáforos e mutexes a comunicação entre processos parece fácil, certo?**



# Monitores

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

*/\* number of slots in the buffer \*/*  
*/\* semaphores are a special kind of int \*/*  
*/\* controls access to critical region \*/*  
*/\* counts empty buffer slots \*/*  
*/\* counts full buffer slots \*/*

*/\* TRUE is the constant 1 \*/*  
*/\* generate something to put in buffer \*/*  
*/\* decrement empty count \*/*  
*/\* enter critical region \*/*  
*/\* put new item in buffer \*/*  
*/\* leave critical region \*/*  
*/\* increment count of full slots \*/*

*/\* infinite loop \*/*  
*/\* decrement full count \*/*  
*/\* enter critical region \*/*  
*/\* take item from buffer \*/*  
*/\* leave critical region \*/*  
*/\* increment count of empty slots \*/*  
*/\* do something with the item \*/*



# Monitores

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

*/\* number of slots in the buffer \*/*  
*/\* semaphores are a special kind of int \*/*  
*/\* controls access to critical region \*/*  
*/\* counts empty buffer slots \*/*  
*/\* counts full buffer slots \*/*

*/\* TRUE is the constant 1 \*/*  
*/\* generate something to put in buffer \*/*  
*/\* decrement empty count \*/*  
*/\* enter critical region \*/*  
*/\* put new item in buffer \*/*  
*/\* leave critical region \*/*  
*/\* increment count of full slots \*/*

*/\* infinite loop \*/*  
*/\* decrement full count \*/*  
*/\* enter critical region \*/*  
*/\* take item from buffer \*/*  
*/\* leave critical region \*/*  
*/\* increment count of empty slots \*/*  
*/\* do something with the item \*/*



# Monitores

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&mutex);
        down(&empty);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

*/\* number of slots in the buffer \*/*  
*/\* semaphores are a special kind of int \*/*  
*/\* controls access to critical region \*/*  
*/\* counts empty buffer slots \*/*  
*/\* counts full buffer slots \*/*

*/\* TRUE is the constant 1 \*/*  
*/\* generate something to put in buffer \*/*  
*/\* enter critical region \*/*  
*/\* decrement empty count \*/*  
*/\* put new item in buffer \*/*  
*/\* leave critical region \*/*  
*/\* increment count of full slots \*/*

*/\* infinite loop \*/*  
*/\* decrement full count \*/*  
*/\* enter critical region \*/*  
*/\* take item from buffer \*/*  
*/\* leave critical region \*/*  
*/\* increment count of empty slots \*/*  
*/\* do something with the item \*/*



# Monitores

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&mutex);
        down(&empty);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

/\* number of slots in the buffer \*/  
 /\* semaphores are a special kind of int \*/  
 /\* controls access to critical region \*/  
 /\* counts empty buffer slots \*/  
 /\* counts full buffer slots \*/

/\* TRUE is the constant 1 \*/  
 /\* generate something to put in buffer \*/  
 /\* enter critical region \*/  
 /\* decrement empty count \*/  
 /\* put new item in buffer \*/  
 /\* leave critical region \*/  
 /\* increment count of full slots \*/

MUTEX = 1  
EMPTY = 0

/\* infinite loop \*/  
 /\* decrement full count \*/  
 /\* enter critical region \*/  
 /\* take item from buffer \*/  
 /\* leave critical region \*/  
 /\* increment count of empty slots \*/  
 /\* do something with the item \*/



# Monitores

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&mutex);
        down(&empty);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

/\* number of slots in the buffer \*/  
 /\* semaphores are a special kind of int \*/  
 /\* controls access to critical region \*/  
 /\* counts empty buffer slots \*/  
 /\* counts full buffer slots \*/

/\* TRUE is the constant 1 \*/  
 /\* generate something to put in buffer \*/  
 /\* enter critical region \*/  
 /\* decrement empty count \*/  
 /\* put new item in buffer \*/  
 /\* leave critical region \*/  
 /\* increment count of full slots \*/

MUTEX = 1  
EMPTY = 0

/\* infinite loop \*/  
 /\* decrement full count \*/  
 /\* enter critical region \*/  
 /\* take item from buffer \*/  
 /\* leave critical region \*/  
 /\* increment count of empty slots \*/  
 /\* do something with the item \*/



# Monitores

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&mutex);
        down(&empty);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

/\* number of slots in the buffer \*/  
 /\* semaphores are a special kind of int \*/  
 /\* controls access to critical region \*/  
 /\* counts empty buffer slots \*/  
 /\* counts full buffer slots \*/

/\* TRUE is the constant 1 \*/  
 /\* generate something to put in buffer \*/  
 /\* enter critical region \*/  
 /\* decrement empty count \*/  
 /\* put new item in buffer \*/  
 /\* leave critical region \*/  
 /\* increment count of full slots \*/

MUTEX = 0  
EMPTY = 0

/\* infinite loop \*/  
 /\* decrement full count \*/  
 /\* enter critical region \*/  
 /\* take item from buffer \*/  
 /\* leave critical region \*/  
 /\* increment count of empty slots \*/  
 /\* do something with the item \*/





# Monitores

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&mutex);
        down(&empty);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */
```

```
/* TRUE is the constant 1 */
/* generate something to put in buffer */
/* enter critical region */
/* decrement empty count */
/* put new item in buffer */
/* leave critical region */
/* increment count of full slots */
```

MUTEX = 0  
EMPTY = 0

```
void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

```
/* infinite loop */
/* decrement full count */
/* enter critical region */
/* take item from buffer */
/* leave critical region */
/* increment count of empty slots */
/* do something with the item */
```



# Monitores

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&mutex);
        down(&empty);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */
```

```
/* TRUE is the constant 1 */
/* generate something to put in buffer */
/* enter critical region */
/* decrement empty count */
/* put new item in buffer */
/* leave critical region */
/* increment count of full slots */
```

MUTEX = 0  
EMPTY = 0

```
void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

```
/* infinite loop */
/* decrement full count */
/* enter critical region */
/* take item from buffer */
/* leave critical region */
/* increment count of empty slots */
/* do something with the item */
```



# Monitores

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&mutex);
        down(&empty);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

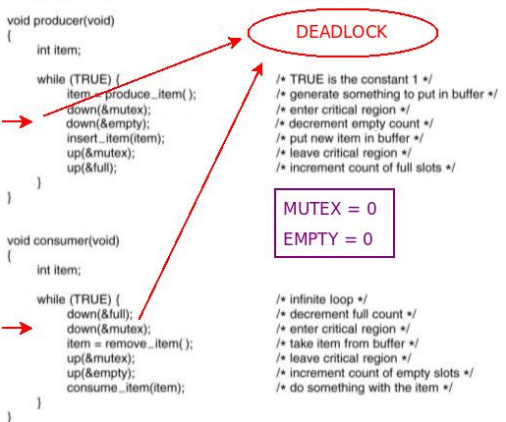
/\* number of slots in the buffer \*/  
 /\* semaphores are a special kind of int \*/  
 /\* controls access to critical region \*/  
 /\* counts empty buffer slots \*/  
 /\* counts full buffer slots \*/

/\* TRUE is the constant 1 \*/  
 /\* generate something to put in buffer \*/  
 /\* enter critical region \*/  
 /\* decrement empty count \*/  
 /\* put new item in buffer \*/  
 /\* leave critical region \*/  
 /\* increment count of full slots \*/

/\* infinite loop \*/  
 /\* decrement full count \*/  
 /\* enter critical region \*/  
 /\* take item from buffer \*/  
 /\* leave critical region \*/  
 /\* increment count of empty slots \*/  
 /\* do something with the item \*/

**DEADLOCK**

MUTEX = 0  
 EMPTY = 0




# Monitores

---



# Monitores

---

- Complexidade no uso de semáforos: Similar à programação de baixo nível



# Monitores

---

- Complexidade no uso de semáforos: Similar à programação de baixo nível
- Para facilitar a escrita de programas corretos, **Brinch Hansen (1973) / Hoare (1974)** → propuseram uma primitiva de sincronização de nível mais alto chamada **Monitor**



# Monitores

---

- Complexidade no uso de semáforos: Similar à programação de baixo nível
- Para facilitar a escrita de programas corretos, **Brinch Hansen (1973) / Hoare (1974)** → propuseram uma primitiva de sincronização de nível mais alto chamada **Monitor**
- **Monitor:**



# Monitores

---

- Complexidade no uso de semáforos: Similar à programação de baixo nível
- Para facilitar a escrita de programas corretos, **Brinch Hansen (1973) / Hoare (1974)** → propuseram uma primitiva de sincronização de nível mais alto chamada **Monitor**
- **Monitor:**
  - Unidade básica de sincronização de alto nível





# Monitores

---

- Complexidade no uso de semáforos: Similar à programação de baixo nível
- Para facilitar a escrita de programas corretos, **Brinch Hansen (1973) / Hoare (1974)** → propuseram uma primitiva de sincronização de nível mais alto chamada **Monitor**
- **Monitor:**
  - Unidade básica de sincronização de alto nível
  - Coleção de rotinas, variáveis e estruturas de dados

# Monitores

---

- Complexidade no uso de semáforos: Similar à programação de baixo nível
- Para facilitar a escrita de programas corretos, **Brinch Hansen (1973) / Hoare (1974)** → propuseram uma primitiva de sincronização de nível mais alto chamada **Monitor**
- **Monitor:**
  - Unidade básica de sincronização de alto nível
  - Coleção de rotinas, variáveis e estruturas de dados
  - Processos têm acesso apenas às rotinas

# Monitores

---

- Complexidade no uso de semáforos: Similar à programação de baixo nível
- Para facilitar a escrita de programas corretos, **Brinch Hansen (1973) / Hoare (1974)** → propuseram uma primitiva de sincronização de nível mais alto chamada **Monitor**
- **Monitor:**
  - Unidade básica de sincronização de alto nível
  - Coleção de rotinas, variáveis e estruturas de dados
  - Processos têm acesso apenas às rotinas
  - Propriedade básica: apenas um processo ativo em um monitor

# Monitores

---

- Complexidade no uso de semáforos: Similar à programação de baixo nível
- Para facilitar a escrita de programas corretos, **Brinch Hansen (1973) / Hoare (1974)** → propuseram uma primitiva de sincronização de nível mais alto chamada **Monitor**
- **Monitor:**
  - Unidade básica de sincronização de alto nível
  - Coleção de rotinas, variáveis e estruturas de dados
  - Processos têm acesso apenas às rotinas
  - Propriedade básica: apenas um processo ativo em um monitor
- **Monitores** são uma **construção da linguagem de programação**

# Monitores

---

- Complexidade no uso de semáforos: Similar à programação de baixo nível
- Para facilitar a escrita de programas corretos, **Brinch Hansen (1973) / Hoare (1974)** → propuseram uma primitiva de sincronização de nível mais alto chamada **Monitor**
- **Monitor:**
  - Unidade básica de sincronização de alto nível
  - Coleção de rotinas, variáveis e estruturas de dados
  - Processos têm acesso apenas às rotinas
  - Propriedade básica: apenas um processo ativo em um monitor
- **Monitores** são uma **construção da linguagem de programação**
- Compilador implementa exclusão mútua (mutex)

# Monitores

---

- Complexidade no uso de semáforos: Similar à programação de baixo nível
- Para facilitar a escrita de programas corretos, **Brinch Hansen (1973) / Hoare (1974)** → propuseram uma primitiva de sincronização de nível mais alto chamada **Monitor**
- **Monitor:**
  - Unidade básica de sincronização de alto nível
  - Coleção de rotinas, variáveis e estruturas de dados
  - Processos têm acesso apenas às rotinas
  - Propriedade básica: apenas um processo ativo em um monitor
- **Monitores** são uma **construção da linguagem de programação**
- Compilador implementa exclusão mútua (mutex)
- Variáveis condicionais → Bloquear processos quando necessário

# Monitores

---

- Complexidade no uso de semáforos: Similar à programação de baixo nível
- Para facilitar a escrita de programas corretos, **Brinch Hansen (1973) / Hoare (1974)** → propuseram uma primitiva de sincronização de nível mais alto chamada **Monitor**
- **Monitor:**
  - Unidade básica de sincronização de alto nível
  - Coleção de rotinas, variáveis e estruturas de dados
  - Processos têm acesso apenas às rotinas
  - Propriedade básica: apenas um processo ativo em um monitor
- **Monitores** são uma **construção da linguagem de programação**
- Compilador implementa exclusão mútua (mutex)
- Variáveis condicionais → Bloquear processos quando necessário
  - *wait/signal*

# Monitores

---





# Monitores

---

- Para evitar ter dois processos ativos no monitor ao mesmo tempo, precisamos de uma regra dizendo o que acontece depois de um *signal*.



# Monitores

---

- Para evitar ter dois processos ativos no monitor ao mesmo tempo, precisamos de uma regra dizendo o que acontece depois de um *signal*.
- O que ocorre quando *signal* é chamado?



# Monitores

---

- Para evitar ter dois processos ativos no monitor ao mesmo tempo, precisamos de uma regra dizendo o que acontece depois de um *signal*.
- O que ocorre quando *signal* é chamado?
- Hoare



# Monitores

---

- Para evitar ter dois processos ativos no monitor ao mesmo tempo, precisamos de uma regra dizendo o que acontece depois de um *signal*.
- O que ocorre quando *signal* é chamado?
- Hoare
  - Chaveamento instantâneo – – propôs deixar o processo recentemente desperto executar, suspendendo o outro



# Monitores

---

- Para evitar ter dois processos ativos no monitor ao mesmo tempo, precisamos de uma regra dizendo o que acontece depois de um *signal*.
- O que ocorre quando *signal* é chamado?
- Hoare
  - Chaveamento instantâneo – – propôs deixar o processo recentemente desperto executar, suspendendo o outro
- Brinch Hansen

# Monitores

---

- Para evitar ter dois processos ativos no monitor ao mesmo tempo, precisamos de uma regra dizendo o que acontece depois de um *signal*.
- O que ocorre quando *signal* é chamado?
- Hoare
  - Chaveamento instantâneo – – propôs deixar o processo recentemente desperto executar, suspendendo o outro
- Brinch Hansen
  - Último comando da rotina – propôs uma saída inteligente para o problema, exigindo que um processo realizando um *signal* deva sair do monitor imediatamente

# Monitores

---

- Para evitar ter dois processos ativos no monitor ao mesmo tempo, precisamos de uma regra dizendo o que acontece depois de um *signal*.
- O que ocorre quando *signal* é chamado?
- Hoare
  - Chaveamento instantâneo – propôs deixar o processo recentemente desperto executar, suspendendo o outro
- Brinch Hansen
  - Último comando da rotina – propôs uma saída inteligente para o problema, exigindo que um processo realizando um *signal* deva sair do monitor imediatamente
- Terceira opção

# Monitores

---

- Para evitar ter dois processos ativos no monitor ao mesmo tempo, precisamos de uma regra dizendo o que acontece depois de um *signal*.
- O que ocorre quando *signal* é chamado?
- Hoare
  - Chaveamento instantâneo – – propôs deixar o processo recentemente desperto executar, suspendendo o outro
- Brinch Hansen
  - Último comando da rotina – propôs uma saída inteligente para o problema, exigindo que um processo realizando um *signal* deva sair do monitor imediatamente
- Terceira opção
  - Esperar que processo saia do monitor



# Monitores

Um esqueleto do problema produtor-consumidor com monitores. Somente uma rotina do monitor está ativa por vez. O buffer tem  $N$  vagas.

```

procedure producer;
begin
    while true do
        begin
            item = produce_item;
            ProducerConsumer.insert(item)
        end
    end;

procedure consumer;
begin
    while true do
        begin
            item = ProducerConsumer.remove;
            consume_item(item)
        end
    end;

```

```

monitor ProducerConsumer
    condition full, empty;
    integer count;

    procedure insert(item: integer);
    begin
        if count =  $N$  then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;

    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count =  $N - 1$  then signal(full)
    end;

    count := 0;
end monitor;

```

# Monitores

---



# Monitores

---

- Java: exemplo de linguagem com suporte a monitores



# Monitores

---

- Java: exemplo de linguagem com suporte a monitores
- Palavra chave *synchronized* → Exclusão mútua



# Monitores

- Java: exemplo de linguagem com suporte a monitores
- Palavra chave *synchronized* → Exclusão mútua

```
static class our_monitor { // este é o monitor
    private int buffer[] = new int[N];
    private int count = 0, lo = 0, hi = 0; // contadores e índices

    public synchronized void insert(int val) {
        if (count == N) go_to_sleep(); // se o buffer estiver cheio, vá dormir
        buffer[hi] = val; // insere um item no buffer
        hi = (hi + 1) % N; // lugar para colocar o próximo item
        count = count + 1; // mais um item no buffer agora
        if (count == 1) notify(); // se o consumidor estava dormindo, acorde-o
    }

    public synchronized int remove() {
        int val;
        if (count == 0) go_to_sleep(); // se o buffer estiver vazio, vá dormir
        val = buffer[lo]; // busca um item no buffer
        lo = (lo + 1) % N; // lugar de onde buscar o próximo item
        count = count - 1; // um item a menos no buffer
        if (count == N - 1) notify(); // se o produtor estava dormindo, acorde-o
        return val;
    }

    private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {};}
}
```



# Monitores

- Java: exemplo de linguagem com suporte a monitores
- Palavra chave *synchronized* → Exclusão mútua

```
static class our_monitor { // este é o monitor
    private int buffer[] = new int[N];
    private int count = 0, lo = 0, hi = 0; // contadores e índices

    public synchronized void insert(int val) {
        if (count == N) go_to_sleep(); // se o buffer estiver cheio, vá dormir
        buffer[hi] = val; // insere um item no buffer
        hi = (hi + 1) % N; // lugar para colocar o próximo item
        count = count + 1; // mais um item no buffer agora
        if (count == 1) notify(); // se o consumidor estava dormindo, acorde-o
    }

    public synchronized int remove() {
        int val;
        if (count == 0) go_to_sleep(); // se o buffer estiver vazio, vá dormir
        val = buffer[lo]; // busca um item no buffer
        lo = (lo + 1) % N; // lugar de onde buscar o próximo item
        count = count - 1; // um item a menos no buffer
        if (count == N - 1) notify(); // se o produtor estava dormindo, acorde-o
        return val;
    }

    private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {};}
}
```

Diagrama de anotação: O termo "EXCLUSÃO MÚTUA" está circulado em vermelho. Duas setas vermelhas apontam para os métodos `insert` e `remove`, que são ambos precedidos pela palavra-chave `synchronized`, indicando que eles são protegidos por exclusão mútua.

# Monitores

- Java: exemplo de linguagem com suporte a monitores
- Palavra chave *synchronized* → Exclusão mútua

```
static class producer extends Thread {
    public void run() { // o método run contém o código do thread
        int item;
        while (true) { // laço do produtor
            item = produce_item();
            mon.insert(item);
        }
    }
    private int produce_item() { ... } // realmente produz
}

static class consumer extends Thread {
    public void run() { método run contém o código do thread
        int item;
        while (true) { // laço do consumidor
            item = mon.remove();
            consume_item (item);
        }
    }
    private void consume_item(int item) { ... } // realmente consome
}
```



# Monitores e Semáforos

---





# Monitores e Semáforos

---

- Projetados para resolver exclusão mútua



# Monitores e Semáforos

---

- Projetados para resolver exclusão mútua
- Problema com monitores, e também com semáforos é que



# Monitores e Semáforos

---

- Projetados para resolver exclusão mútua
- Problema com monitores, e também com semáforos é que
  - eles foram projetados para solucionar o problema da exclusão mútua em uma ou mais CPUs



# Monitores e Semáforos

---

- Projetados para resolver exclusão mútua
- Problema com monitores, e também com semáforos é que
  - eles foram projetados para solucionar o problema da exclusão mútua em uma ou mais CPUs
  - Todas com acesso a uma memória comum



# Monitores e Semáforos

---

- Projetados para resolver exclusão mútua
- Problema com monitores, e também com semáforos é que
  - eles foram projetados para solucionar o problema da exclusão mútua em uma ou mais CPUs
  - Todas com acesso a uma memória comum
- Quando movemos para um sistema distribuído



# Monitores e Semáforos

---

- Projetados para resolver exclusão mútua
- Problema com monitores, e também com semáforos é que
  - eles foram projetados para solucionar o problema da exclusão mútua em uma ou mais CPUs
  - Todas com acesso a uma memória comum
- Quando movemos para um sistema distribuído
  - consistindo em múltiplas CPUs



# Monitores e Semáforos

---

- Projetados para resolver exclusão mútua
- Problema com monitores, e também com semáforos é que
  - eles foram projetados para solucionar o problema da exclusão mútua em uma ou mais CPUs
  - Todas com acesso a uma memória comum
- Quando movemos para um sistema distribuído
  - consistindo em múltiplas CPUs
  - cada uma com sua própria memória privada e,



# Monitores e Semáforos

---

- Projetados para resolver exclusão mútua
- Problema com monitores, e também com semáforos é que
  - eles foram projetados para solucionar o problema da exclusão mútua em uma ou mais CPUs
  - Todas com acesso a uma memória comum
- Quando movemos para um sistema distribuído
  - consistindo em múltiplas CPUs
  - cada uma com sua própria memória privada e,
  - conectada por uma rede de área local





# Monitores e Semáforos

---

- Projetados para resolver exclusão mútua
- Problema com monitores, e também com semáforos é que
  - eles foram projetados para solucionar o problema da exclusão mútua em uma ou mais CPUs
  - Todas com acesso a uma memória comum
- Quando movemos para um sistema distribuído
  - consistindo em múltiplas CPUs
  - cada uma com sua própria memória privada e,
  - conectada por uma rede de área local
- Semáforos e monitores são primitivas que se tornam inaplicáveis

# Monitores e Semáforos

---

- Projetados para resolver exclusão mútua
- Problema com monitores, e também com semáforos é que
  - eles foram projetados para solucionar o problema da exclusão mútua em uma ou mais CPUs
  - Todas com acesso a uma memória comum
- Quando movemos para um sistema distribuído
  - consistindo em múltiplas CPUs
  - cada uma com sua própria memória privada e,
  - conectada por uma rede de área local
- Semáforos e monitores são primitivas que se tornam inaplicáveis
- Os **semáforos** são de um nível baixo demais e os **monitores** não são utilizáveis, exceto em algumas poucas linguagens de programação

# Troca de mensagens

---



# Troca de mensagens

---

- Método de comunicação entre processos (**troca de mensagens**)



# Troca de mensagens

---

- Método de comunicação entre processos (**troca de mensagens**)
- Esse método de comunicação entre processos usa duas primitivas *send* / *receive*



# Troca de mensagens

---

- Método de comunicação entre processos (**troca de mensagens**)
- Esse método de comunicação entre processos usa duas primitivas *send* / *receive*
  - Chamadas de sistema como semáforos



# Troca de mensagens

---

- Método de comunicação entre processos (**troca de mensagens**)
- Esse método de comunicação entre processos usa duas primitivas *send* / *receive*
  - Chamadas de sistema como semáforos
  - Não são construções de linguagem como monitores



# Troca de mensagens

---

- Método de comunicação entre processos (**troca de mensagens**)
- Esse método de comunicação entre processos usa duas primitivas *send* / *receive*
  - Chamadas de sistema como semáforos
  - Não são construções de linguagem como monitores
- *send(destino,&msg)*





# Troca de mensagens

---

- Método de comunicação entre processos (**troca de mensagens**)
- Esse método de comunicação entre processos usa duas primitivas *send* / *receive*
  - Chamadas de sistema como semáforos
  - Não são construções de linguagem como monitores
- *send(destino,&msg)*
  - Envia *msg* para *destino*



# Troca de mensagens

---

- Método de comunicação entre processos (**troca de mensagens**)
- Esse método de comunicação entre processos usa duas primitivas *send* / *receive*
  - Chamadas de sistema como semáforos
  - Não são construções de linguagem como monitores
- *send(destino,&msg)*
  - Envia *msg* para *destino*
- *receive(fonte,&msg)*



# Troca de mensagens

---

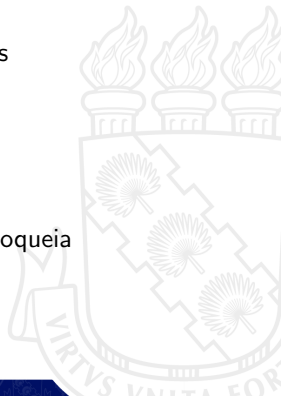
- Método de comunicação entre processos (**troca de mensagens**)
- Esse método de comunicação entre processos usa duas primitivas *send* / *receive*
  - Chamadas de sistema como semáforos
  - Não são construções de linguagem como monitores
- *send(destino,&msg)*
  - Envia *msg* para *destino*
- *receive(fonte,&msg)*
  - Recebe *msg* de *fonte*



# Troca de mensagens

---

- Método de comunicação entre processos (**troca de mensagens**)
- Esse método de comunicação entre processos usa duas primitivas *send* / *receive*
  - Chamadas de sistema como semáforos
  - Não são construções de linguagem como monitores
- *send(destino,&msg)*
  - Envia *msg* para *destino*
- *receive(fonte,&msg)*
  - Recebe *msg* de *fonte*
  - Caso não existam mensagens recebidas receptor bloqueia



# Troca de mensagens

---



# Troca de mensagens

---

- Sistemas de **troca de mensagens** têm **muitos problemas** e **questões de projeto** que não surgem com **semáforos** ou com **monitores**



# Troca de mensagens

---

- Sistemas de **troca de mensagens** têm **muitos problemas** e **questões de projeto** que não surgem com **semáforos** ou com **monitores**
- Mensagens enviadas que não chegam



# Troca de mensagens

---

- Sistemas de **troca de mensagens** têm **muitos problemas** e **questões de projeto** que não surgem com **semáforos** ou com **monitores**
- Mensagens enviadas que não chegam
  - Perda de pacotes na rede





# Troca de mensagens

---

- Sistemas de **troca de mensagens** têm **muitos problemas** e **questões de projeto** que não surgem com **semáforos** ou com **monitores**
- Mensagens enviadas que não chegam
  - Perda de pacotes na rede
  - Confirmação de recebimento (acknowledgment)



# Troca de mensagens

---

- Sistemas de **troca de mensagens** têm **muitos problemas** e **questões de projeto** que não surgem com **semáforos** ou com **monitores**
- Mensagens enviadas que não chegam
  - Perda de pacotes na rede
  - Confirmação de recebimento (acknowledgment)
  - Retransmissão



# Troca de mensagens

---

- Sistemas de **troca de mensagens** têm **muitos problemas** e **questões de projeto** que não surgem com **semáforos** ou com **monitores**
- Mensagens enviadas que não chegam
  - Perda de pacotes na rede
  - Confirmação de recebimento (acknowledgment)
  - Retransmissão
- Confirmação que não chega



# Troca de mensagens

---

- Sistemas de **troca de mensagens** têm **muitos problemas** e **questões de projeto** que não surgem com **semáforos** ou com **monitores**
- Mensagens enviadas que não chegam
  - Perda de pacotes na rede
  - Confirmação de recebimento (acknowledgment)
  - Retransmissão
- Confirmação que não chega
  - Múltiplas recepções da mesma informação



# Troca de mensagens

---

- Sistemas de **troca de mensagens** têm  **muitos problemas** e **questões de projeto** que não surgem com **semáforos** ou com **monitores**
- Mensagens enviadas que não chegam
  - Perda de pacotes na rede
  - Confirmação de recebimento (acknowledgment)
  - Retransmissão
- Confirmação que não chega
  - Múltiplas recepções da mesma informação
  - Distinção entre multiplas transmissões



# Troca de mensagens

---

- Sistemas de **troca de mensagens** têm  **muitos problemas** e **questões de projeto** que não surgem com **semáforos** ou com **monitores**
- Mensagens enviadas que não chegam
  - Perda de pacotes na rede
  - Confirmação de recebimento (acknowledgment)
  - Retransmissão
- Confirmação que não chega
  - Múltiplas recepções da mesma informação
  - Distinção entre multiplas transmissões
  - Contador de transmissão



# Troca de mensagens

---

- Sistemas de **troca de mensagens** têm  **muitos problemas** e **questões de projeto** que não surgem com **semáforos** ou com **monitores**
- Mensagens enviadas que não chegam
  - Perda de pacotes na rede
  - Confirmação de recebimento (acknowledgment)
  - Retransmissão
- Confirmação que não chega
  - Múltiplas recepções da mesma informação
  - Distinção entre multiplas transmissões
  - Contador de transmissão
- Ambiguidade entre múltiplos destinos/fontes



# Troca de mensagens

---

- Sistemas de **troca de mensagens** têm  **muitos problemas** e **questões de projeto** que não surgem com **semáforos** ou com **monitores**
- Mensagens enviadas que não chegam
  - Perda de pacotes na rede
  - Confirmação de recebimento (acknowledgment)
  - Retransmissão
- Confirmação que não chega
  - Múltiplas recepções da mesma informação
  - Distinção entre multiplas transmissões
  - Contador de transmissão
- Ambiguidade entre múltiplos destinos/fontes
  - Autenticação





# Troca de mensagens

---

- Sistemas de **troca de mensagens** têm  **muitos problemas** e **questões de projeto** que não surgem com **semáforos** ou com **monitores**
- Mensagens enviadas que não chegam
  - Perda de pacotes na rede
  - Confirmação de recebimento (acknowledgment)
  - Retransmissão
- Confirmação que não chega
  - Múltiplas recepções da mesma informação
  - Distinção entre multiplas transmissões
  - Contador de transmissão
- Ambiguidade entre múltiplos destinos/fontes
  - Autenticação
- Menos eficiente que semáforos/monitores



# Barreiras

---



# Barreiras

---

- **Grupos de processos** – em vez de situações que **envolvem dois processos** do tipo **produtor-consumidor**



# Barreiras

---

- **Grupos de processos** – em vez de situações que **envolvem dois processos** do tipo **produtor-consumidor**
- Algumas aplicações são divididas em fases



# Barreiras

---

- **Grupos de processos** – em vez de situações que **envolvem dois processos** do tipo **produtor-consumidor**
- Algumas aplicações são divididas em fases
  - Passagem apenas quando todos processos prontos



# Barreiras

---

- **Grupos de processos** – em vez de situações que **envolvem dois processos** do tipo **produtor-consumidor**
- Algumas aplicações são divididas em fases
  - Passagem apenas quando todos processos prontos
- Tal comportamento pode ser conseguido colocando uma barreira (**barrier**) no fim de cada fase.



# Barreiras

---

- **Grupos de processos** – em vez de situações que **envolvem dois processos** do tipo **produtor-consumidor**
- Algumas aplicações são divididas em fases
  - Passagem apenas quando todos processos prontos
- Tal comportamento pode ser conseguido colocando uma barreira (**barrier**) no fim de cada fase.
- Quando um **processo** atinge a **barreira**, ele é **bloqueado** até que **todos os processos** tenham atingido a **barreira**.

# Barreiras

---

- **Grupos de processos** – em vez de situações que **envolvem dois processos** do tipo **produtor-consumidor**
- Algumas aplicações são divididas em fases
  - Passagem apenas quando todos processos prontos
- Tal comportamento pode ser conseguido colocando uma barreira (**barrier**) no fim de cada fase.
- Quando um **processo** atinge a **barreira**, ele é **bloqueado** até que **todos os processos** tenham atingido a **barreira**.
- Isso **permite** que grupos de processos **sincronizem**.



# Barreiras

---

- **Grupos de processos** – em vez de situações que **envolvem dois processos** do tipo **produtor-consumidor**
- Algumas aplicações são divididas em fases
  - Passagem apenas quando todos processos prontos
- Tal comportamento pode ser conseguido colocando uma barreira (**barrier**) no fim de cada fase.
- Quando um **processo** atinge a **barreira**, ele é **bloqueado** até que **todos os processos** tenham atingido a **barreira**.
- Isso **permite** que grupos de processos **sincronizem**.
- Exemplo:

# Barreiras

---

- **Grupos de processos** – em vez de situações que **envolvem dois processos** do tipo **produtor-consumidor**
- Algumas aplicações são divididas em fases
  - Passagem apenas quando todos processos prontos
- Tal comportamento pode ser conseguido colocando uma barreira (**barrier**) no fim de cada fase.
- Quando um **processo** atinge a **barreira**, ele é **bloqueado** até que **todos os processos** tenham atingido a **barreira**.
- Isso **permite** que grupos de processos **sincronizem**.
- Exemplo:
  - Matriz → Temperaturas em placa de metal

# Barreiras

---

- **Grupos de processos** – em vez de situações que **envolvem dois processos** do tipo **produtor-consumidor**
- Algumas aplicações são divididas em fases
  - Passagem apenas quando todos processos prontos
- Tal comportamento pode ser conseguido colocando uma barreira (**barrier**) no fim de cada fase.
- Quando um **processo** atinge a **barreira**, ele é **bloqueado** até que **todos os processos** tenham atingido a **barreira**.
- Isso **permite** que grupos de processos **sincronizem**.
- Exemplo:
  - Matriz → Temperaturas em placa de metal
  - Cálculo do efeito de uma fonte de calor pontual

# Barreiras

---

- **Grupos de processos** – em vez de situações que **envolvem dois processos** do tipo **produtor-consumidor**
- Algumas aplicações são divididas em fases
  - Passagem apenas quando todos processos prontos
- Tal comportamento pode ser conseguido colocando uma barreira (**barrier**) no fim de cada fase.
- Quando um **processo** atinge a **barreira**, ele é **bloqueado** até que **todos os processos** tenham atingido a **barreira**.
- Isso **permite** que grupos de processos **sincronizem**.
- Exemplo:
  - Matriz → Temperaturas em placa de metal
  - Cálculo do efeito de uma fonte de calor pontual
  - Leis da termodinâmica → Evolução da matriz a cada instante  $n$

# Barreiras

---

- **Grupos de processos** – em vez de situações que **envolvem dois processos** do tipo **produtor-consumidor**
- Algumas aplicações são divididas em fases
  - Passagem apenas quando todos processos prontos
- Tal comportamento pode ser conseguido colocando uma barreira (**barrier**) no fim de cada fase.
- Quando um **processo** atinge a **barreira**, ele é **bloqueado** até que **todos os processos** tenham atingido a **barreira**.
- Isso **permite** que grupos de processos **sincronizem**.
- Exemplo:
  - Matriz → Temperaturas em placa de metal
  - Cálculo do efeito de uma fonte de calor pontual
  - Leis da termodinâmica → Evolução da matriz a cada instante  $n$
  - Matriz muito grande → Processamento paralelo

# Barreiras

- **Grupos de processos** – em vez de situações que **envolvem dois processos** do tipo **produtor-consumidor**
- Algumas aplicações são divididas em fases
  - Passagem apenas quando todos processos prontos
- Tal comportamento pode ser conseguido colocando uma barreira (**barrier**) no fim de cada fase.
- Quando um **processo** atinge a **barreira**, ele é **bloqueado** até que **todos os processos** tenham atingido a **barreira**.
- Isso **permite** que grupos de processos **sincronizem**.
- Exemplo:
  - Matriz → Temperaturas em placa de metal
  - Cálculo do efeito de uma fonte de calor pontual
  - Leis da termodinâmica → Evolução da matriz a cada instante  $n$
  - Matriz muito grande → Processamento paralelo
  - Início do cálculo em  $n$  depende o resultado em  $n - 1$

# Barreiras

