

Fundamentos sobre Threads

Prof. Dr. Márcio Castro
marcio.castro@ufsc.br



1

Threads

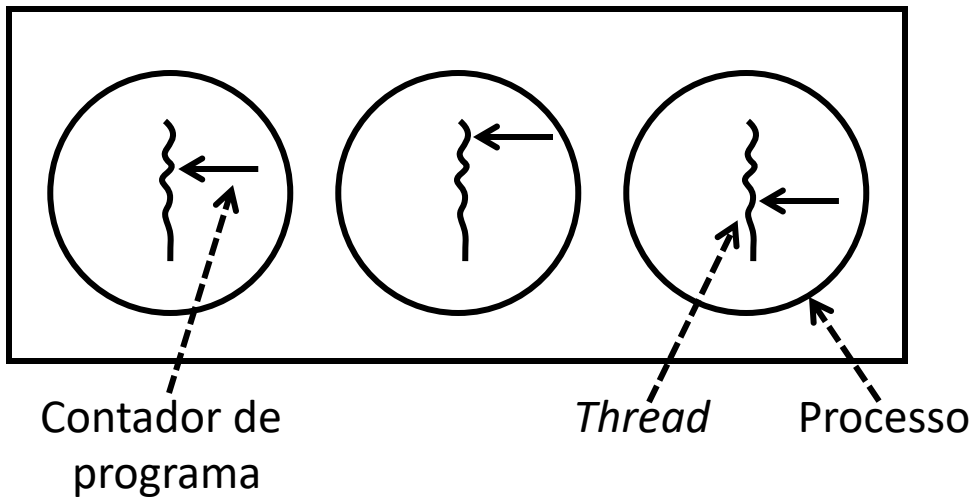
Threads

■ **Threads**

- Permitem que múltiplas linhas de execução **ocorram no mesmo ambiente do processo**, com um certo grau de independência
- **Processos leves:** a criação de uma *thread* é **menos onerosa** do que a criação de um processo pelo SO

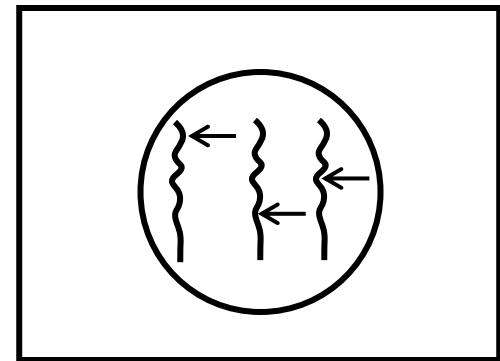
Threads

Sistema Operacional (SO)



Três processos, cada um com uma *thread*

Sistema Operacional (SO)



Um único processo com três *threads*

Threads

- *Threads* distintas em um processo não são tão independentes quanto processos distintos
- *Threads* de um mesmo processo compartilham **compartilham variáveis globais**

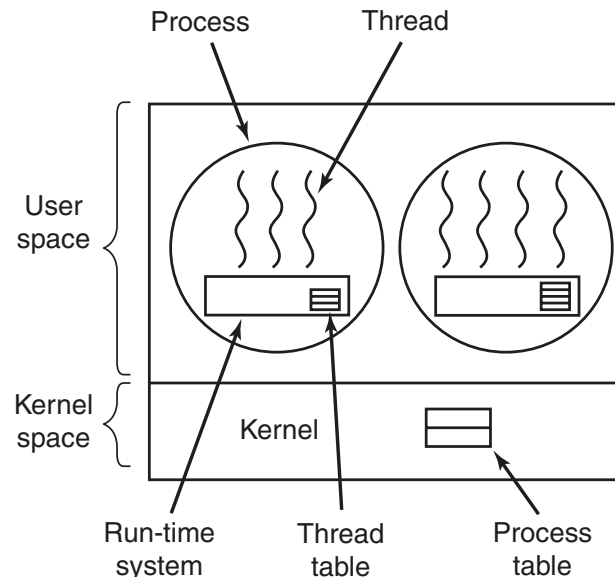
Por processo	Por <i>thread</i>
<ul style="list-style-type: none">- Espaço de endereçamento- Variáveis globais- Arquivos abertos	<ul style="list-style-type: none">- Contador de programa (PC)- Registradores- Pilha- Estado

Implementação de *threads*

- Implementação em espaço de usuário
- Implementação em espaço de núcleo
- Implementações híbridas

Implementação em espaço de usuário

- O núcleo do SO não fornece uma abstração de *threads*
- Implementação através de **bibliotecas de usuário**
- Funções da API de *threads* são **invocadas em espaço de usuário** (ao invés de chamadas de sistema)
- Núcleo gerencia processos *singlethread*



Implementação em espaço de usuário

■ Vantagens

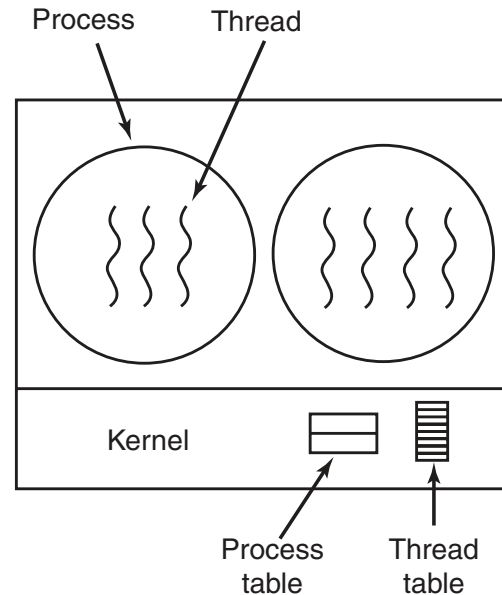
- Pode ser implementado em SOs sem suporte a *threads*
- Chaveamento entre *threads* é mais rápido: não é necessário passar para o modo núcleo
- Cada processo pode ter o seu próprio algoritmo de escalonamento

■ Desvantagem

- Chamadas de sistema realizadas pelas *threads* bloqueiam o processo como um todo

Implementação em espaço de núcleo

- O núcleo do SO fornece uma abstração de *threads*
- Código e estruturas de dados da implementação de *threads* residem no **espaço de núcleo**
- Funções da API de *threads* são invocadas, tipicamente, através de **chamadas de sistema**



Implementação em espaço de núcleo

■ Vantagens

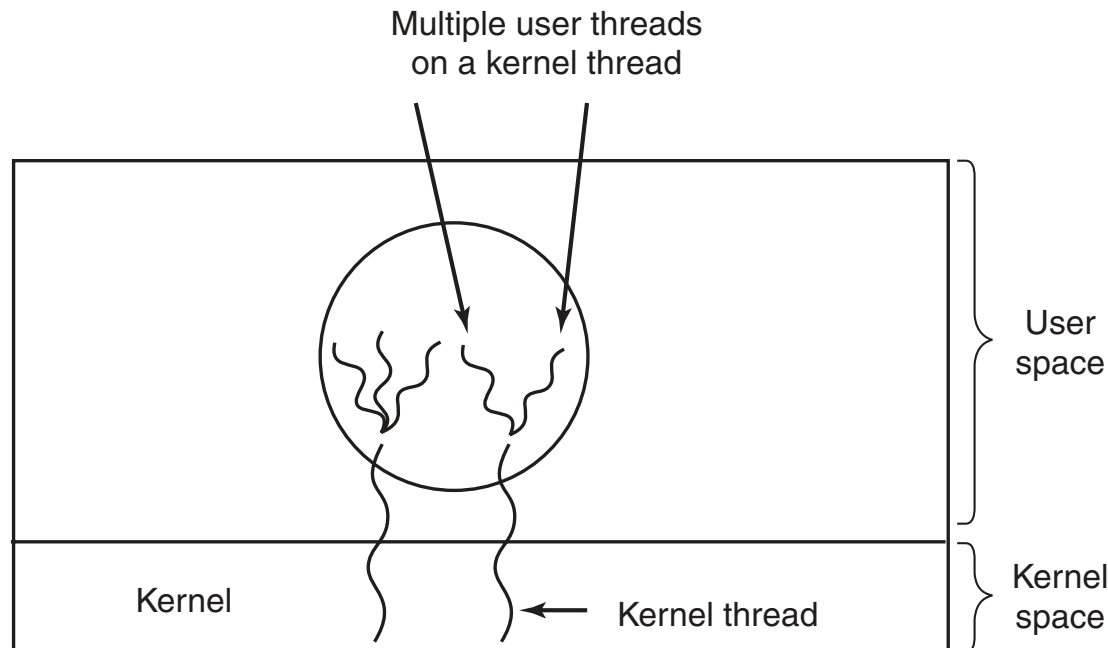
- SO pode escalonar *threads* individualmente
- Melhor utilização dos recursos de *hardware*
- Uma chamada de sistema realizada por uma *thread* não bloqueia as demais *threads* do mesmo processo

■ Desvantagem

- Maior custo para chaveamento entre *threads*

Implementação híbrida

- Diversas *threads* de usuário podem ser **multiplexadas** em uma *thread* de núcleo
- SO realiza escalonamento de *threads* de núcleo
- Quando a *thread* de núcleo é multiplexada, somente uma *thread* de usuário pode estar em execução nela



Implementação híbrida

- **Vantagem**

- Maior flexibilidade

- **Desvantagem**

- Maior complexidade de implementação e gerência
 - Mapeamento 1:1 entre *threads* de usuário/núcleo tem maior sobrecusto do que a abordagem pura de *threads* em espaço de núcleo

2 POSIX Threads

POSIX Threads

- **POSIX** é um acrônimo para **Portable Operating System Interface**
 - Normas definidas pela IEEE
- Define uma ***Application Programming Interface (API)*** para compatibilidade de *software* com variantes de Unix e outros SOs
- ***POSIX Threads*** é o padrão POSIX para *threads*

POSIX Threads

▪ Funcionamento geral

- O SO cria um processo quando um programa é iniciado
- Este processo conterá uma única *thread*, denominada *thread principal (main thread)*
- A *thread* principal poderá criar *threads* filhas ou *threads* trabalhadoras (*worker threads*)
- *Threads* trabalhadoras poderão criar outras *threads*

▪ Escalonamento

- O Linux utiliza *threads* em espaço de núcleo
- *Threads* são escalonadas individualmente

POSIX Threads

Criação e execução de uma *thread*:

```
pthread_create(thread, attr, start_routine, arg)
```

Parâmetros:

- **thread**: identificador único retornado pela criação da *thread*
- **attr**: usado para setar atributos da *thread* a ser criada (padrão é NULL)
- **start_routine**: rotina que será executada uma vez que a *thread* foi criada
- **arg**: argumento a ser passado para a *start_routine*

POSIX Threads

Finalizando a execução de uma *thread*:

```
pthread_exit(retval)
```

Parâmetros:

- **retval**: valor a ser retornado para a *thread* pai

POSIX Threads

Sincronização entre *threads* pai/filhas:

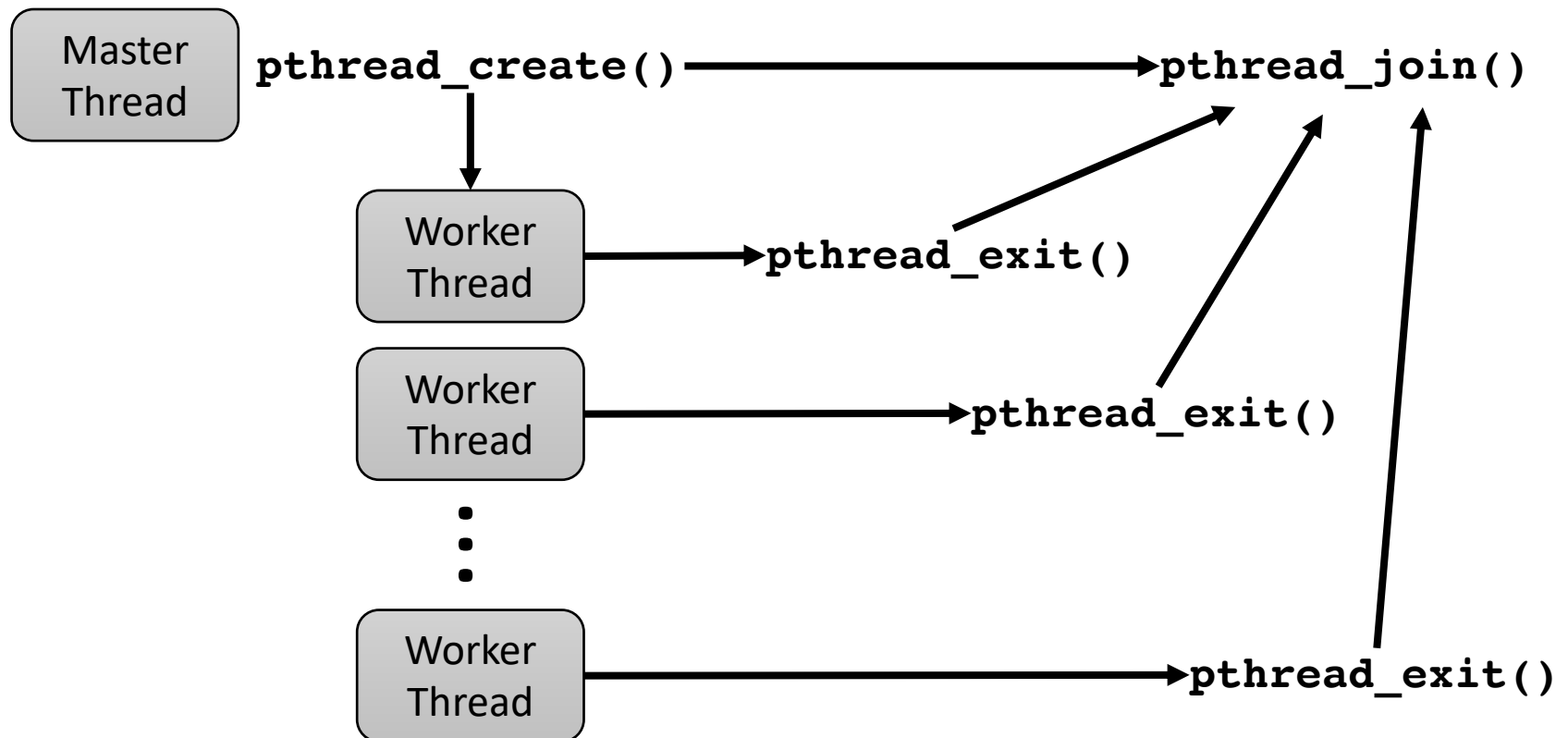
```
pthread_join(thread, retval)
```

Parâmetros:

- **thread**: identificador único retornado por `pthread_create()`
- **retval**: variável que irá armazenar o valor que foi passado para a função `pthread_exit()` (NULL para não armazenar o retorno)

POSIX Threads

Criação e sincronização de *threads*



Hello World!

```
#include <pthread.h>
#include <stdio.h>

void *PrintHello(void *arg) {
    pthread_t tid = pthread_self();
    printf("Thread %u: Hello World!\n", (unsigned int)tid);
    pthread_exit(NULL);
}

int main(int argc, char **argv) {
    pthread_t thread;
    pthread_create(&thread, NULL, PrintHello, NULL);
    pthread_join(thread, NULL);

    return 0;
}
```

3 Compartilhamento de dados

Condição de corrida

- *Threads* podem compartilhar dados na memória
 - Compartilhamento pode causar **condições de corrida**
- Um programa possui uma condição de corrida quando:
 - Duas ou mais *threads* alteram um mesmo conjunto de **dados concorrentemente**
 - O resultado **depende da ordem** na qual os acessos aos dados são feitos

Condição de corrida

- **Exemplo:** contador

Thread 1 (T1)

```
x++;
```

Thread 2 (T2)

```
x++;
```

- **Perguntas**

- Neste caso, temos uma condição de corrida?
- Se inicialmente **x=0**, qual é o valor de **x** após a execução das threads T1 e T2?

Condição de corrida

▪ Exemplo: contador

Thread 1 (T1)

x++:

MV REG1, x

INC REG1

MV x, REG1

Thread 2 (T2)

x++:

MV REG1, x

INC REG1

MV x, REG1

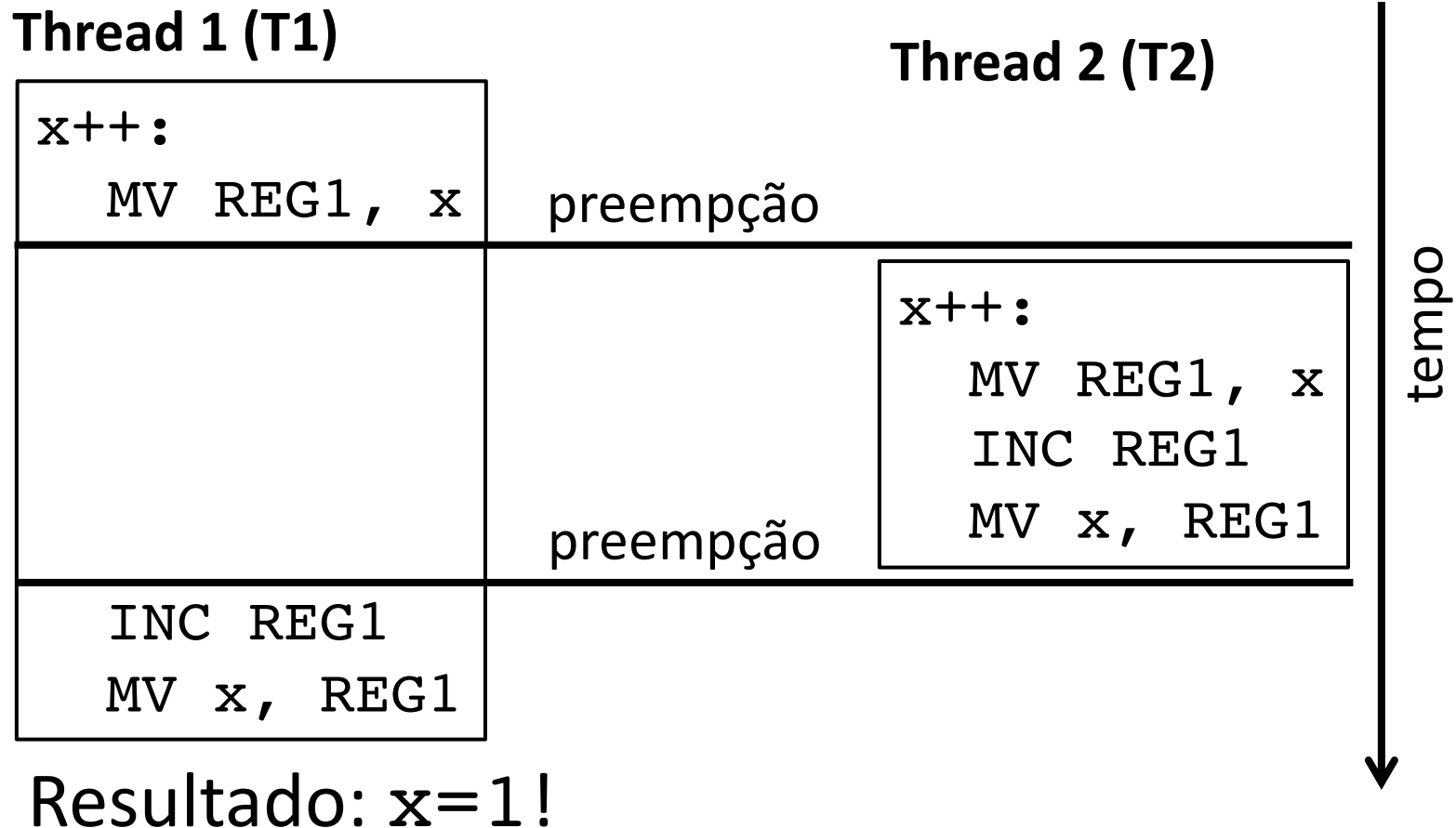
preempção

tempo

Resultado: x=2

Condição de corrida

▪ Exemplo: contador



Sincronização

- Como impedir que mais de uma *thread* leia ou modifique o valor de uma variável compartilhada ao mesmo tempo?
 - Mecanismos de exclusão mútua
- Regiões críticas
 - Partes de um programa concorrente protegidas por mecanismos de exclusão mútua
 - Ausência de proteção acarreta em **condições de corrida**

Sincronização

- **Mecanismos de exclusão mútua**
 - Spin-locks
 - Mutexes
- **Mecanismo de exclusão mútua e sincronização**
 - Semáforos

3 Spin-locks

Spin-lock

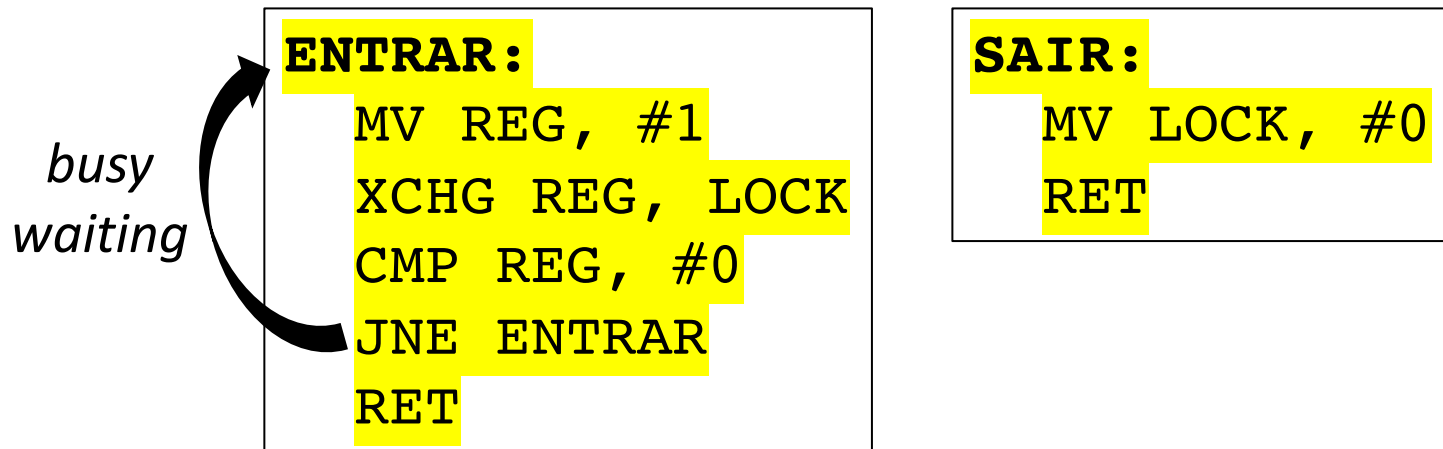
- Mecanismo de sincronização baseado em **espera ocupada** (*busy waiting*)
- Utiliza instrução específica de *hardware*
- **Exemplo:** instrução EXCHANGE (XCHG)

XCHG	REG, MEM:
MV	AUX, [MEM]
MV	[MEM], REG
MV	REG, AUX

} Executado de forma **atômica!**

Spin-lock

- A região crítica é protegida por uma variável em memória (ex. **LOCK**)
- Inicialmente: **LOCK=0**
- Se **LOCK=0** → região crítica está livre
- Se **LOCK=1** → região crítica está ocupada



Spin-lock

■ Problema

- **Espera ocupada:** a *thread* aguarda entrada na região crítica em um *loop*
- **Acarreta em uso desnecessário do processador**

3 Mutex

Mutex

- **Operações básicas sobre um mutex m**
 - `lock(m)`: solicita acesso à região crítica
 - `unlock(m)`: libera a região crítica

`lock(m)`

Se (m está livre) então

 Marca m como ocupado

Senão

 { Bloqueia a thread e a insere
 no fim da fila do mutex m }



A thread permanece no estado
bloqueado (não faz uso da CPU)!

`unlock(m)`

Se (a fila de m está vazia) então

 Marca m como livre

Senão

 Libera a thread do início da
 fila do mutex m

▪ Atomicidade

- As funções **lock()** e **unlock()** precisam ser atômicas
- Implementações de mutex usam **spin-locks** para garantir a **atomicidade** na execução dessas funções

POSIX Threads - mutex

Inicializando um mutex:

```
pthread_mutex_init(mutex, attr)
```

Parâmetros:

- **mutex**: variável do tipo **pthread_mutex_t** a ser inicializada
- **attr**: atributos específicos de configuração do mutex (NULL para configuração padrão)

POSIX Threads - mutex

Destruindo um mutex:

```
pthread_mutex_destroy(mutex)
```

Parâmetros:

- **mutex**: variável do tipo `pthread_mutex_t` a ser destruída

POSIX Threads - mutex

Entrando/saindo de uma região crítica:

```
pthread_mutex_lock(mutex)  
pthread_mutex_unlock(mutex)
```

Parâmetros:

- **mutex**: variável do tipo `pthread_mutex_t` anteriormente inicializada

POSIX Threads - mutex

```
pthread_mutex_t mutex;

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

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

3 Semáforo

Semáforos

■ Operações básicas sobre um semáforo S

- $P(S)$: solicita acesso à região protegida por um semáforo
- $V(S)$: libera o semáforo

$P(S)$

Se (valor de $S > 0$) então

$S = S - 1$

Senão

Bloqueia a *thread* e a
insere no fim da fila
do semáforo S



A *thread* permanece no estado
bloqueado (não faz uso da CPU)!

$V(S)$

Se (fila de S vazia) então

$S = S + 1$

Senão

Libera a *thread* do início da
fila do semáforo S

Semáforos: tipos

- **Binário:** assume somente valores 0 e 1
 - Funciona da mesma forma que um mutex
 - Usado para implementar exclusão mútua
- **Contador:** assume valores $n \geq 0$
 - n indica a quantidade de *threads* que podem “passar” pelo semáforo em um dado instante
 - Usado para a sincronização entre *threads*

Semáforos

▪ Atomicidade

- As funções **P ()** e **V ()** precisam ser atômicas
- Implementações de semáforos usam **spin-locks** para garantir a **atomicidade** na execução dessas funções

POSIX - Semaphore

Inicializando um semáforo:

```
sem_init(&sem, pshared, value)
```

Parâmetros:

- **sem**: variável do tipo **sem_t** a ser inicializada
- **pshared**: *flag* para indicar se o semáforo deve ser compartilhado entre *threads* (=0) ou processos (!=0)
- **value**: valor inicial do semáforo

POSIX - Semaphore

Destruindo um semáforo:

```
sem_destroy(sem)
```

Parâmetros:

- **sem**: variável do tipo **sem_t** a ser destruída

POSIX - Semaphore

Manipulando um semáforo:

```
sem_wait(sem)  
sem_post(sem)
```

Parâmetros:

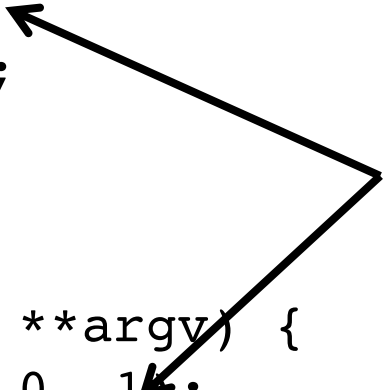
- **sem**: variável do tipo **sem_t** anteriormente inicializada

Exemplo 1 – Semáforo binário

```
#include <semaphore.h>
sem_t semaphore;

void *thread(void *arg) {
    sem_wait(&semaphore);
    // Seção crítica
    sem_post(&semaphore);
    pthread_exit(NULL);
}

int main(int argc, char **argv) {
    sem_init(&semaphore, 0, 1);
    // Criação das threads + joins
    sem_destroy(&semaphore);
    return 0;
}
```



Semáforo binário

Exemplo 2 – Produtor/Consumidor

```
#include <semaphore.h>
#define N 10

sem_t cheio, vazio, lock_prod, lock_cons;
int buffer[N], i = 0, f = 0;

int main(int argc, char **argv) {
    sem_init(&cheio, 0, 0);
    sem_init(&vazio, 0, N);
    sem_init(&lock_prod, 0, 1);
    sem_init(&lock_cons, 0, 1);
    // Criação threads prod e cons + joins
    sem_destroy(&cheio);
    sem_destroy(&vazio);
    sem_destroy(&lock_prod);
    sem_destroy(&lock_cons);
    return 0;
}
```

Exemplo 2 - Produtor/Consumidor

```
void* produtor(void *arg)
{
    while(1)
    {
        sem_wait(&vazio);
        sem_wait(&lock_prod);
        f = (f + 1) % N;
        buffer[f] = produz();
        sem_post(&lock_prod);
        sem_post(&cheio);
    }
    pthread_exit(NULL);
}
```

```
void* consumidor(void *arg)
{
    while (1)
    {
        sem_wait(&cheio);
        sem_wait(&lock_cons);
        i = (i + 1) % N;
        consome(buffer[i]);
        sem_post(&lock_cons);
        sem_post(&vazio);
    }
    pthread_exit(NULL);
}
```


! Obrigado pela atenção!



Dúvidas? Entre em contato:

- marcio.castro@ufsc.br
- www.marciocastro.com



**UNIVERSIDADE FEDERAL
DE SANTA CATARINA**



Distributed Systems Research Lab

www.lapesd.inf.ufsc.br