

# Introdução ao Processamento Paralelo e Distribuído



Videoaula

## Notas dos slides

### APRESENTAÇÃO

O presente conjunto de slides pertence à coleção produzida para a disciplina *Introdução ao Processamento Paralelo e Distribuído* ofertada aos cursos de bacharelado em Ciência da Computação e em Engenharia da Computação pelo Centro de Desenvolvimento Tecnológico da Universidade Federal de Pelotas.

Os slides disponibilizados complementam as videoaulas produzidas e tratam de pontos específicos da disciplina. Embora tenham sido produzidos para ser assistidos de forma independente, a sequência informada reflete o encadeamento dos assuntos no desenvolvimento do conteúdo programático previsto para a disciplina.



2

# Programação Multithread

Programação com Pthreads

## Notas da videoaula

### DESCRIÇÃO

Esta videoaula sumariza os elementos principais de programação com Pthreads.

### OBJETIVOS

Apresentar os conceitos fundamentais de programação com Pthreads. Capacitar o aluno a desenvolver programas simples e lhe dar autonomia para identificar e explorar recursos de programação para desenvolvimento de programas mais complexos.

4

“

**Cat: Where are you going?**

**Alice: Which way should I go?**

**Cat: That depends on where you are going.**

**Alice: I don't know.**

**Cat: Then it doesn't matter which way you go.**

Lewis Carroll, Alice in Wonderland

5

## POSIX Threads

Definido no standard POSIX.1c, define uma API para programação multithread. Oferecida na forma de uma biblioteca para programas C, define um conjunto de cerca de 100 primitivas para gerência de threads e sincronização entre threads. A API também define um conjunto de *tipos de dados opacos*.

**Pthreads**

**P**ortable  
**O**perating  
**S**ystem  
**I**nterface  
**B**ased on  
**U**niX

IEEE standard  
1995

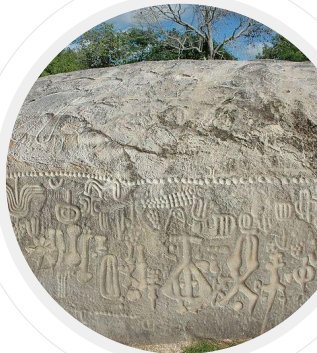
6

## Pthreads

Não está verdadeiramente associada à linguagem C, simplesmente é uma biblioteca explorável em C. Também não foi concebida para programação em larga escala, mas sim para desenvolvimento habilitar a exploração da execução concorrente de programas.

Realidade da época:

- As arquiteturas não ofereciam paralelismo multiprocessado de larga escala;
- Abstrações de alto nível na programação (OO) eram uma novidade;
- Desempenho, acima de tudo.



7

## Tipo de Dado Opaco

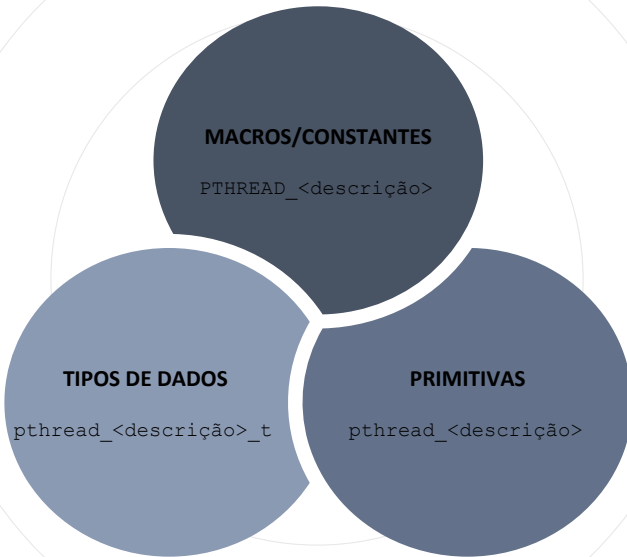
É um tipo de dado definido de forma associada a um conjunto de primitivas concebidas para manipulá-lo. Embute o conceito de ocultação de dados, uma vez que as instâncias dos tipos de dados opacos não devem ser manipuladas diretamente, apenas via as primitivas disponibilizadas.

Isso permite que programas possam ser portados em diferentes ambientes, usando diferentes implementações da biblioteca Pthreads.



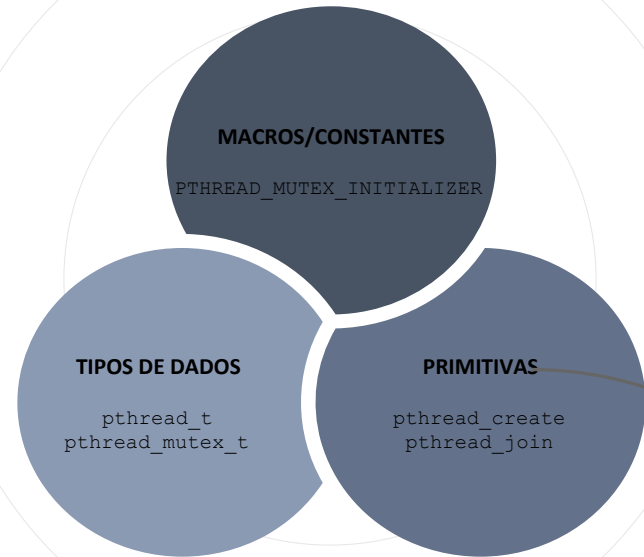
8

## API



9

## API



10

## Unidade de Trabalho: Função

```
void* foo( void* dta ) {  
    <identificadores locais>  
  
    <corpo>  
  
    return <endereço>  
}
```

11

O corpo de uma ut é descrito por uma função C convencional. Como Pthread é implementado como uma biblioteca, não há compatibilidade entre os tipos de dados manipulados pelo programa de aplicação com os manipulados pela biblioteca. O tipo void\* viabiliza a troca de informações.

## Unidade de Trabalho: Função

```
typedef struct { int s, *v; } Input;  
  
void* foo( void* dta ) {  
    Input *in = (Input*) dta;  
    int *soma, i;  
    soma = (int*) malloc(sizeof(int));  
    for(i=0; i<in->s; ++i)  
        *soma += *(in->v+i);  
    return soma;  
}
```

12

### EXEMPLO

Uma função que retorna a soma de todos elementos de um vetor.

## Manipulação de threads

```
int pthread_create( pthread_id *tid, pthread_attr_t *attr,  
void *(*func)(void *), void *dta );
```

```
int pthread_join( pthread_t tid, void ** ret );
```

```
pthread_t pthread_self();
```

CRIAÇÃO  
pthread\_create

SINCRONIZAÇÃO  
pthread\_join

AUTO-IDENTIFICAÇÃO  
pthread\_self

## Manipulação de threads

```
int pthread_create( pthread_id *tid, pthread_attr_t *attr,  
void *(*func)(void *), void *dta );
```

**Exemplo:**

```
pthread_create(&tid, NULL, foo, (void*)par);
```

13

### ATRIBUTOS DE ESCALONAMENTO

pthread\_attr\_t

Como tamanho da pilha,  
prioridade de  
escalonamento,  
joinable/detached

NULL: devem ser  
utilizados o valor default  
de cada atributo


14

## Manipulação de threads

**Visualização:**

```
pthread_create(&tid, NULL, foo, (void*)par);
```

A  ...

B  ...

pthread\_create

Custo da criação do  
thread

Latência de criação

## Manipulação de threads

```
int pthread_join( pthread_id tid, void ** res );
```

**Exemplo:**

```
pthread_join(tid, (void**)&res);
```

15

### RETORNO DE RESULTADO

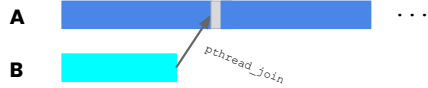
O parâmetro res é um  
parâmetro de saída: é  
passado o endereço de  
memória deste ponteiro  
para que seu conteúdo  
seja atualizado.

16

## Manipulação de threads

Visualização: Caso em que o thread sincronizado já terminou

```
pthread_join(tid, (void**)res);
```



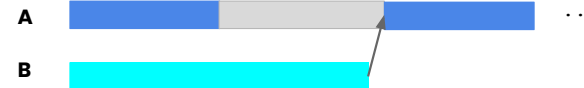
Latência de sincronização

17

## Manipulação de threads

Visualização: Caso em que o thread sincronizado ainda não terminou

```
pthread_join(tid, (void**)res);
```



Espera

Latência de sincronização

18

```
#define TAM 10

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <pthread.h>

typedef struct { int s, *v; } Input;

void* foo( void* dta ) {
    Input *in = (Input*) dta;
    int *soma, i;

    soma = (int*) malloc(sizeof(int));

    for(i=0;i<in->s;++i)
        *soma += *(in->v+i);

    return soma;
}

int main() {
    Input *par;
    pthread_t tid;
    int *res, i;

    par = (Input*) malloc(sizeof(Input));
    par->s = TAM;
    par->v = (int*) malloc(TAM*sizeof(int));

    for(i=0;i<TAM;++i) *(par->v+i) = i;

    if( pthread_create(&tid, NULL, foo, (void*)par) ) {
        printf("Erro na criação.\n");
        exit(1);
    }

    if( pthread_join(tid, (void**)&res) ) {
        printf("Erro na sincronização.\n");
        exit(1);
    }

    printf("Resultado = %d\n", *res);
    free(res);
    free(par->v);
    free(par);

    return 0;
}
```

19

```
#define TAM 10

void* foo( void* dta ) {
    Input *in = (Input*) dta;
    int *soma, i;

    soma = (int*) malloc(sizeof(int));

    return soma;
}

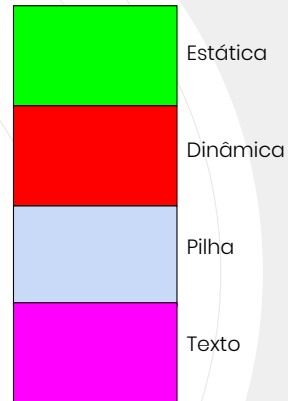
int main() {
    Input *par;
    pthread_t tid;
    int *res, i;

    par = (Input*) malloc(sizeof(Input));
    par->s = TAM;
    par->v = (int*) malloc(TAM*sizeof(int));

    pthread_create(&tid, NULL, foo, (void*)par);
    pthread_join(tid, (void**)&res);

    printf("Resultado = %d\n", *res);
}
```

Código "simplificado", destacando os dados em memória.



20

```
const int TAM = 10;
```

```
void* foo( void* dta ) {
    Input *in = (Input*) dta;
    int *soma, i;
```

```
    soma = (int*) malloc(sizeof(int));

    return soma;
}
```

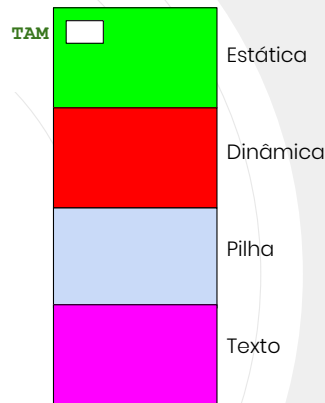
```
int main() {
    Input *par;
    pthread_t tid;
    int *res, i;
```

```
    par = (Input*) malloc(sizeof(Input));
    par->s = TAM;
    par->v = (int*) malloc(TAM*sizeof(int));
```

```
    pthread_create(&tid, NULL, foo, (void*)par);
    pthread_join(tid, (void**)&res);
```

```
    printf("Resultado = %d\n", *res);
}
```

**Uma alteração, apenas para destacar o escopo da memória estática.**



21

```
const int TAM = 10;
```

```
void* foo( void* dta ) {
    Input *in = (Input*) dta;
    int *soma, i;
```

```
    soma = (int*) malloc(sizeof(int));

    return soma;
}
```

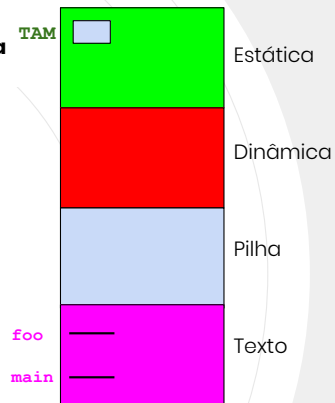
```
int main() {
    Input *par;
    pthread_t tid;
    int *res, i;
```

```
    par = (Input*) malloc(sizeof(Input));
    par->s = TAM;
    par->v = (int*) malloc(TAM*sizeof(int));
```

```
    pthread_create(&tid, NULL, foo, (void*)par);
    pthread_join(tid, (void**)&res);
```

```
    printf("Resultado = %d\n", *res);
}
```

**Identificadores de funções mapeiam endereços na memória de código.**



22

```
const int TAM = 10;
```

```
void* foo( void* dta ) {
    Input *in = (Input*) dta;
    int *soma, i;
```

```
    soma = (int*) malloc(sizeof(int));

    return soma;
}
```

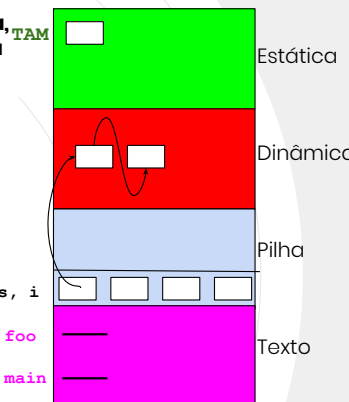
```
int main() {
    Input *par;
    pthread_t tid;
    int *res, i;
```

```
    par = (Input*) malloc(sizeof(Input));
    par->s = TAM;
    par->v = (int*) malloc(TAM*sizeof(int));
```

```
    pthread_create(&tid, NULL, foo, (void*)par);
    pthread_join(tid, (void**)&res);
```

```
    printf("Resultado = %d\n", *res);
}
```

**Ao iniciar a execução do programa, é criado um registro na pilha para a função main, sendo alocadas as variáveis para par, tid, res e i. Variáveis são alocadas na memória dinâmica.**



23

```
const int TAM = 10;
```

```
void* foo( void* dta ) {
    Input *in = (Input*) dta;
    int *soma, i;
```

```
    soma = (int*) malloc(sizeof(int));

    return soma;
}
```

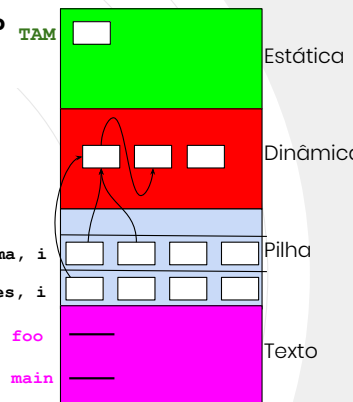
```
int main() {
    Input *par;
    pthread_t tid;
    int *res, i;
```

```
    par = (Input*) malloc(sizeof(Input));
    par->s = TAM;
    par->v = (int*) malloc(TAM*sizeof(int));
```

```
    pthread_create(&tid, NULL, foo, (void*)par);
    pthread_join(tid, (void**)&res);
```

```
    printf("Resultado = %d\n", *res);
}
```

**Ao ser lançado o thread, um novo registro na pilha é criado para a função foo, sendo alocadas as variáveis para dta, in, soma e i.**



24

```
const int TAM = 10;

void* foo( void* dta ) {
    Input *in = (Input*) dta;
    int *soma, i;

    soma = (int*) malloc(sizeof(int));

    return soma;
}

int main() {
    Input *par;
    pthread_t tid;
    int *res, i;

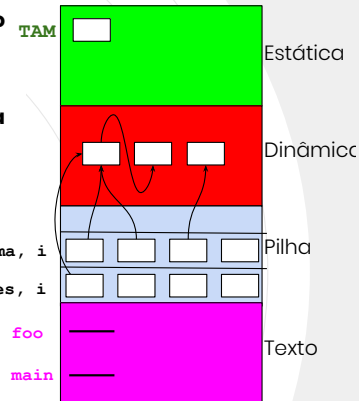
    par = (Input*) malloc(sizeof(Input));
    par->s = TAM;
    par->v = (int*) malloc(TAM*sizeof(int));

    pthread_create(&tid, NULL, foo, (void*)par);
    pthread_join(tid, (void**)&res);

    printf("Resultado = %d\n", *res);
}
```

Ao ser lançado o thread, um novo registro na pilha é criado para a função `foo`, sendo alocadas as variáveis para `dta`, `in`, `soma` e `i`. Durante sua execução, é alocada uma área de memória.

`foo: dta, in, soma, i`  
`main: par, tid, res, i`



25

```
const int TAM = 10;

void* foo( void* dta ) {
    Input *in = (Input*) dta;
    int *soma, i;

    soma = (int*) malloc(sizeof(int));

    return soma;
}

int main() {
    Input *par;
    pthread_t tid;
    int *res, i;

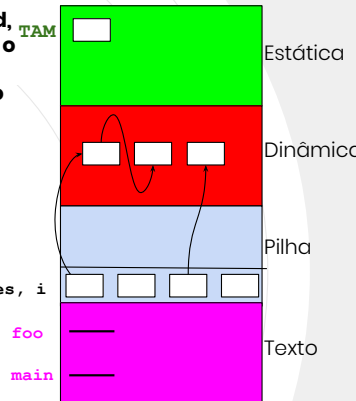
    par = (Input*) malloc(sizeof(Input));
    par->s = TAM;
    par->v = (int*) malloc(TAM*sizeof(int));

    pthread_create(&tid, NULL, foo, (void*)par);
    pthread_join(tid, (void**)&res);

    printf("Resultado = %d\n", *res);
}
```

Ao terminar a execução do thread, seu registro na pilha é destruído e o conteúdo da variável retornada (ou seja, um endereço) é copiado para seu destino.

`main: par, tid, res, i`



26

## Compilação

Pthreads é padrão em qualquer ambiente que siga o padrão POSIX.

Em ambientes Unix, o arquivo de cabeçalho para compilação é `pthread.h`, usualmente encontrada em `/usr/include`. A etapa de link edição deve incluir a biblioteca `libpthread.a`, usualmente encontrada em `/usr/lib`.

Observe: Em geral, o caminho que o compilador utiliza para encontrar os diretórios padrões é pré-configurado.

```
$> gcc prog.c -lpthread
$> █
```

27

## Mecanismos de Sincronização

### SEÇÃO CRÍTICA

Tipo de dado:  
`pthread_mutex_t`

Algumas primitivas:  
`pthread_mutex_init`  
`pthread_mutex_lock`  
`pthread_mutex_unlock`  
`pthread_mutex_trylock`

Pthreads oferece dois recursos\* de sincronização: **mutex**, para seções críticas, e **variável de condição**, para controle de fluxo. Em ambos os casos, estruturas opacas oferecem os tipos de dados utilizados para manipular estes mecanismos de sincronização. As instâncias destes tipos de dados são acessíveis por identificadores sujeitos às mesmas regras de escopo de qualquer identificador em C. Pthreads também define o conjunto de primitivas para manipular estas instâncias.

### CONTROLE DE FLUXO

Tipo de dado:  
`pthread_cond_t`

Algumas primitivas:  
`pthread_cond_init`  
`pthread_cond_broadcast`  
`pthread_cond_signal`  
`pthread_cond_wait`

28

\* Semáforos não são oferecidos no standard Pthread, mas estão presente na especificação POSIX em outra biblioteca de serviços.

# Sincronização: Seção Crítica

## SEÇÃO CRÍTICA

Tipo de dado:  
pthread\_mutex\_t

Algumas primitivas:

pthread\_mutex\_init  
pthread\_mutex\_lock  
pthread\_mutex\_unlock  
pthread\_mutex\_trylock

Um mutex, em determinado momento, pode estar em um de dois estados: *adquirido* ou *livre* (ou fechado/aberto ou bloqueado/desbloqueado). No programa de aplicação, o mutex é associado a algum recurso manipulado, usualmente representado por uma variável compartilhada em memória.

A semântica associada é que, um thread, ao ter posse de um mutex, avança sua execução sobre uma seção crítica, uma vez que nenhum outro thread poderá adquirir o mesmo mutex.

Não existe nenhuma dependência imposta por Pthreads, ou mesmo pela linguagem C, que garanta coerência no uso do mutex com o acesso à variável compartilhada. É responsabilidade do programador garantir esta coerência.

29

# Sincronização: Seção Crítica

## SEÇÃO CRÍTICA

Tipo de dado:  
pthread\_mutex\_t

Algumas primitivas:

pthread\_mutex\_init  
pthread\_mutex\_lock  
pthread\_mutex\_unlock  
pthread\_mutex\_trylock

## Instanciação e inicialização:

```
pthread_mutex_t m;  
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;  
  
int pthread_mutex_init(pthread_mutex_t *m,  
                        pthread_mutexattr_t *a);
```

O atributo default, indicado por NULL, inicializa o mutex como desbloqueado.

30

# Sincronização: Seção Crítica

## SEÇÃO CRÍTICA

Tipo de dado:  
pthread\_mutex\_t

Algumas primitivas:

pthread\_mutex\_init  
pthread\_mutex\_lock  
pthread\_mutex\_unlock  
pthread\_mutex\_trylock

## Manipulação:

```
int pthread_mutex_lock(pthread_mutex_t *m);  
int pthread_mutex_trylock(pthread_mutex_t *m);  
int pthread_mutex_timedlock(pthread_mutex_t *m,  
                             struct timespec *nanosec);  
int pthread_mutex_unlock(pthread_mutex_t *m);
```

O retorno das funções indica sucesso (zero) ou algum código de falha. O sucesso indica, quando for o caso, que o mutex foi adquirido.

31

# Mutex: Indo Além

## Recurso

Questão tratada: Um mutex está adquirido ao receber uma operação lock. Caso o mesmo thread que o detenha executar um novo lock, este thread entra em uma situação de deadlock.

**No entanto, vários algoritmos possuem uma natureza recursiva.**

## Read/Write

Questão tratada: A aquisição de mutex é realizada indistintamente, não importando se o acesso será realizado em leitura ou escrita.

**No entanto, na maior parte dos casos, a exclusividade é necessária apenas em acessos de escrita.**

Pthreads estende a capacidade de expressão do mutex oferecendo outros mecanismos de manipulação: mutexes recursivos e mutexes read/write.

32



## Mutex: Indo Além

### Recursivo

Mecanismo: Deve ser informado que o mutex é recursivo como um atributo em sua inicialização. O thread deve liberar (executar unlock) no mutex recursivo tantas vezes quantos foram as aquisições (locks) realizados sobre ele.

### Read/Write

Mecanismo: A API oferece um tipo de dado específico (`pthread_rwlock_t`) e um conjunto de primitivas próprio para manipulá-lo (como `pthread_rwlock_lock` e `pthread_rwlock_unlock`).

**Maior poder de expressão**

**Melhor desempenho**

Pthreads **não permite** a aquisição de múltiplos mutex, simultaneamente, independente da ordem em que foram adquiridos.

### CONTROLE DE FLUXO

Tipo de dado:  
`pthread_cond_t`

Algumas primitivas:  
`pthread_cond_init`  
`pthread_cond_broadcast`  
`pthread_cond_signal`  
`pthread_cond_wait`

Variável de condição permite que o algoritmo decida, em função da informação contida em algum dado compartilhado no programa. Deve ser observado que o tipo de dado `pthread_cond_t` não promove a instanciação de uma variável que seja, ela própria, a condição a ser observada. A condição propriamente dita é uma informação mantida pelo programa de aplicação. A instância de `pthread_cond_t` oferece um canal de comunicação entre threads para sinalizar a evolução de valores assumidos pelo dado monitorado.

33

34

## Sincronização: Controle de Fluxo

### CONTROLE DE FLUXO

Tipo de dado:  
`pthread_cond_t`

Algumas primitivas:  
`pthread_cond_init`  
`pthread_cond_broadcast`  
`pthread_cond_wait`

### Instanciação e inicialização:

```
pthread_cond_t c;  
pthread_cond_t c = PTHREAD_COND_INITIALIZER;  
  
int pthread_cond_init( pthread_cond_t *c,  
                      pthread_condattr_t *a );
```

A condição inicial indica que a condição é não satisfeita, não podendo ser possível iniciá-la como satisfeita.

35

## Sincronização: Controle de Fluxo

### CONTROLE DE FLUXO

Tipo de dado:  
`pthread_cond_t`

Algumas primitivas:  
`pthread_cond_init`  
`pthread_cond_broadcast`  
`pthread_cond_signal`  
`pthread_cond_wait`

### Manipulação:

```
int pthread_cond_init( pthread_cond_t *c,  
                     pthread_condattr_t *a );  
  
int pthread_cond_broadcast( pthread_cond_t *c );  
int pthread_cond_signal( pthread_cond_t *c );  
int pthread_cond_wait( pthread_cond_t *c,  
                     pthread_mutex_t *m );
```

As primitivas `broadcast` e `signal` sinalizam que uma condição foi atendida. A primitiva `wait` bloqueia um thread, aguardando uma sinalização sobre o atendimento a uma condição.

36

# Sincronização: Controle de Fluxo

## CONTROLE DE FLUXO

Tipo de dado:  
pthread\_cond\_t

Algumas primitivas:

pthread\_cond\_init  
pthread\_cond\_broadcast  
pthread\_cond\_signal  
pthread\_cond\_wait

Manipulação detalhamento:

```
int pthread_cond_broadcast( pthread_cond_t *c );  
int pthread_cond_signal( pthread_cond_t *c );
```

Diversos threads podem estar bloqueados aguardando uma determinada condição. A primitiva `broadcast` permite desbloquear todos os threads bloqueados, informando-os da situação. A primitiva `signal` sinaliza o atendimento à condição a apenas um dos threads bloqueados.

37

# Sincronização: Controle de Fluxo

## CONTROLE DE FLUXO

Tipo de dado:  
pthread\_cond\_t

Algumas primitivas:

pthread\_cond\_init  
pthread\_cond\_broadcast  
pthread\_cond\_signal  
pthread\_cond\_wait

Manipulação detalhamento:

```
int pthread_cond_wait( pthread_cond_t *c,  
                      pthread_mutex_t *m );
```

Caso um thread observe que a condição esperada no programa não está satisfeita, ele pode suspender sua execução invocando a primitiva `wait`, no aguardo do recebimento de um sinal informando que a condição foi atendida.

38

# Sincronização: Controle de Fluxo

## CONTROLE DE FLUXO

Tipo de dado:  
pthread\_cond\_t

Algumas primitivas:

pthread\_cond\_init  
pthread\_cond\_broadcast  
pthread\_cond\_signal  
pthread\_cond\_wait

Manipulação detalhamento:

```
int pthread_cond_wait( pthread_cond_t *c,  
                      pthread_mutex_t *m );
```

Um mutex deve ser utilizado em conjunto à variável de condição pois a condição é ditada por uma variável do programa, sendo a variável `pthread_cond_t` o canal para comunicar que a condição foi satisfeita. O mutex dita o regime de exclusão mútua no acesso à variável manipulada pelo programa que contém a informação a ser satisfeita e também a própria instância de `pthread_cond_t`. **Importante: ao bloquear-se, o thread deve liberar o mutex, pois outro thread deverá adquiri-lo para manipular a condição do programa.**

39

# Sincronização: Controle de Fluxo

## CONTROLE DE FLUXO

Tipo de dado:  
pthread\_cond\_t

Algumas primitivas:

pthread\_cond\_init  
pthread\_cond\_broadcast  
pthread\_cond\_signal  
pthread\_cond\_wait

Manipulação detalhamento:

```
int pthread_cond_wait( pthread_cond_t *c,  
                      pthread_mutex_t *m );
```

Outro detalhe relevante: diferente do mutex, a variável de condição não possui um **estado** (valor) a ser considerado como satisfeito ou não. Trata-se de um canal para sinalizar que uma condição foi satisfeita. Quando houver uma sinalização, o significado é que a condição foi tornada satisfeita. **No entanto, a semântica da operação não garante que esta condição esteja ainda satisfeita quando o thread for liberado.**

40

# Sincronização: Controle de Fluxo

## CONTROLE DE FLUXO

Tipo de dado:  
pthread\_cond\_t

Algumas primitivas:  
pthread\_cond\_init  
pthread\_cond\_broadcast  
pthread\_cond\_signal  
pthread\_cond\_wait

Onde está a condição?

A condição está na variável do programa. Ou seja, é um dado manipulado no contexto do programa. Testar seu valor, portanto, é parte do algoritmo. Sabendo que um thread deve se bloquear (wait) quando observar que a condição não está satisfeita, ao receber um sinal, uma nova verificação da condição deve ser realizada. O uso de variáveis de condição, portanto, estão associados a laços de repetição de teste de condição.

# Sincronização: Controle de Fluxo

## DADOS COMPARTILHADOS

```
pthread_cond_t c;  
pthread_mutex_t m;  
Buffer *b;  
int nelem = 0;
```

## PRODUTOR

```
while(notFim) {  
    it = Produz();  
    lock(&m);  
    insere(b, it);  
    ++nelem;  
    if(nelem==1)  
        signal(&c);  
    unlock(&m);  
}
```

## CONSUMIDOR

```
while(notFim) {  
    lock(&m);  
    while(nelem==0)  
        wait(&c, &m);  
    retira(b, it);  
    --nelem;  
    unlock(&m);  
}
```

# Sincronização: Controle de Fluxo

## DADOS COMPARTILHADOS

```
pthread_cond_t c;  
pthread_mutex_t m;  
Buffer *b;  
int nelem = 0;
```

A variável que registra a condição

## PRODUTOR

```
while(notFim) {  
    it = Produz();  
    lock(&m);  
    insere(b, it);  
    ++nbelem;  
    if(nbelem == 1)  
        signal(&c);  
    unlock(&m);  
}
```

Verificando se a condição foi atendida

Sinalizando condição atendida

## CONSUMIDOR

```
while(notFim) {  
    lock(&m);  
    while(nelem==0)  
        wait(&c, &m);  
    retira(b, it);  
    --it;  
    unlock(&m);  
}
```

Testa a condição, se atendida, segue o algoritmo. Caso não esteja atendida, libera o mutex e aguarda nova sinaliza

# Sincronização: Controle de Fluxo

## DADOS COMPARTILHADOS

```
pthread_cond_t c;  
pthread_mutex_t m;  
Buffer *b;  
int nelem = 0;
```

Testa a condição, se atendida, segue o algoritmo. Caso não esteja atendida, libera o mutex e aguarda um sinal informando que a condição foi satisfeita. Deve disputar o mutex novamente.

## PRODUTOR

```
while(notFim) {  
    it = Produz();  
    lock(&m);  
    insere(b, it);  
    ++nbelem;  
    if(nbelem == 1)  
        signal(&c);  
    unlock(&m);  
}
```

## CONSUMIDOR

```
while(notFim) {  
    lock(&m);  
    while(nelem==0)  
        wait(&c, &m);  
    retira(b, it);  
    --it;  
    unlock(&m);  
}
```

Necessário pois a condição pode ter sido alterada após ter sido sinalizada e antes de ser obtido o mutex.

## Programando Concorrente

1. Crie um programa que receba como entrada o número de threads a ser criado e o tamanho de um vetor. Inicialmente o vetor deve ser criado e, então, inicializado com valores quaisquer. Este programa deve imprimir, como saída, o somatório de todos os valores do vetor. O somatório deve ser computado de forma concorrente, nas  $n$  threads informadas. Devem ser implementadas duas versões do programa:
  - a. A variável que irá acumular o resultado é deve ser um dado externo aos threads (na memória estática ou no heap).
  - b. Cada thread calcula de forma independente sua soma parcial e retorna seu resultado parcial, sendo acumulado, ao final, os resultados parciais.



45

## Programando Concorrente

2. Crie um programa no modelo produtor/consumidor. Este programa deve receber 3 valores numéricos, informando o número de threads produtores e threads consumidores que devem ser criados e o número de itens a serem produzidos. O programa deve manipular um buffer compartilhado e permitir, com auxílio de variável de condição, garantir coordenação no acesso ao buffer compartilhado.

Atividade de Acompanhamento

Submeta sua solução, mesmo se não estiver funcionando 100%:  
<https://forms.gle/2oLz57CTBVwXy2wD6>



46

Obrigado!

47