THREADS

Prof Alberto

Thread

- É uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrentemente.
- O suporte à *thread* é fornecido pelo próprio sistema operacional,
  - no caso da linha de execução ao nível do núcleo (em inglês: *Kernel-Level Thread (KLT)*),
  - ou implementada através de uma biblioteca de uma determinada linguagem, no caso de uma User-Level Thread (ULT).

Pthreads

- cada sistema operacional/hardware implementava a sua própria versão de threads.
- padrão POSIX1003.1-2001 define um *application programming interface* (API) para a escrita de aplicações *multithreaded.*
- POSIX threads ou simplesmente pthreads.

PThreads

- Pthreads são definidos como um conjunto de tipos de dados em C e um conjunto de rotinas.
- Os Pthreads API contêm mais de 60 sub-rotinas.
- Todos os identificadores na livraria começam com pthread_
- A biblioteca pthread.h tem que ser incluída no código fonte.

Observações

- A chamada à função pthread_create() tem quatro argumentos:

  - O primeiro é usado para guardar informação sobre a thread criada.
  - O segundo especifica algumas propriedades da thread a ser criada, utilizamos o valor NULL para significar os valores.
  - O Terceiro é a função que a nova thread vai executar
  - O ultimo é usado para representar argumentos a esta função

Exercício

Escreva o programa do exemplo.

Compile o programa.
(gcc -o criar criar.c -lpthread)

e depois execute-o.

Quantos threads são criadas? Quantas mensagens aparecem na tela?

Terminação de uma Thread

- threads podem executar de forma desunida (*detached*) da thread que as criou ou unidas.
- Desta maneira usando a rotina pthread_join() uma thread pode esperar pela terminação de uma thread específica.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define SIZE 10

int v[SIZE];

void * function (void *arg) {
	int *valor = (int *)(arg);
	int i;
	if (*valor == 1) {
		printf ("Thread 1 executando...\n");
		for (i = 0; i < SIZE / 2; i++) {
			v[i] = 25;
		}
	}
	else {
		printf ("Thread 2 executando...\n");
		for (i = SIZE / 2; i < SIZE; i++) {
			v[i] = 34;
		}
	}
}
```

Continuação...

```
int main () {
        pthread_t t1, t2;
        int a1 = 1;
        int a2 = 2;
        int i;

        pthread_create(&t1, NULL, function, (void *)(&a1));
        pthread_create(&t2, NULL, function, (void *)(&a2));

        pthread_join(t1, NULL);
        pthread_join(t2, NULL);

        for (i = 0; i < SIZE; i++) {
                printf ("%d ", v[i]);
        }
        printf ("\n");

        exit(0);
}
```

Condições de Corrida e Deadlocks

Quando duas threads tentam realizar, ao mesmo tempo, o acesso a uma região crítica, como por exemplo para a incrementação de uma variável. Pode ser que uma execute apenas depois de a outra acabar de incrementar a variável, como pode ser que as duas comecem a incrementá-la ao mesmo tempo, dando um resultado incorreto.

Esse caso de acesso à zona crítica é chamado de **condição de corrida**.

Quando determinada thread espera por um resultado de outra thread, e vice-versa. O que ocorre é que, como uma está esperando pela outra, nenhuma segue em frente para fornecer o resultado para outra, e o programa fica parado, então ocorre um **deadlock**

Mecanismos para evitar condições de corrida e deadlocks

Fechaduras (locks): utilizadas para garantir o acesso de uma única thread a determinada porção do código. Utilizadas normalmente para proteger zonas críticas. Quando uma thread atinge uma fechadura, ela confere se está trancada. Se não estiver, ele a tranca, e continua a executar o código. Todas as threads que chegarem à fechadura após isso esperarão que a fechadura seja destrancada, que ocorre quando a primeira thread atinge o comando de unlock.

Semáforos: semelhantes a locks. Impõem uma condição para que determinada sessão do código seja acessada. Por exemplo, a operação que realiza i = i + 1 só pode ocorrer se uma determinada variável s for igual a 0. Se isso ocorrer, o acesso é liberado para essa operação. Se não, todas as threads que chegarem ao semáforo esperam que a condição seja atingida.

Exclusão Mútua

Servem para implementar o conceito de fechadura. Como criar, destruir, trancar e destrancar uma fechadura.

Essa função inicializa um mutex, que implementa o paradigma de fechadura.

- pthread_mutex_init(mutex, mutexattr)
    - 'mutex': ponteiro para a estrutura previamente alocada que conterá o mutex.
    - 'mutexattr': ponteiro para a estrutura contendo opções para a criação do mutex. Caso valha NULL, valores padrão serão usados.

Essa função é basicamente a fechadura: caso uma thread chegue a ela e ela esteja destrancada, ela a tranca e continua executando o código após ela. Se uma thread chega a ela e ela está trancada, ela pára sua execução, até que a fechadura seja eventualmente destrancada por outra thread.

- pthread_mutex_lock(mutex)
    - 'mutex': o mutex que será usado como fechadura.

Essa função destranca um mutex.

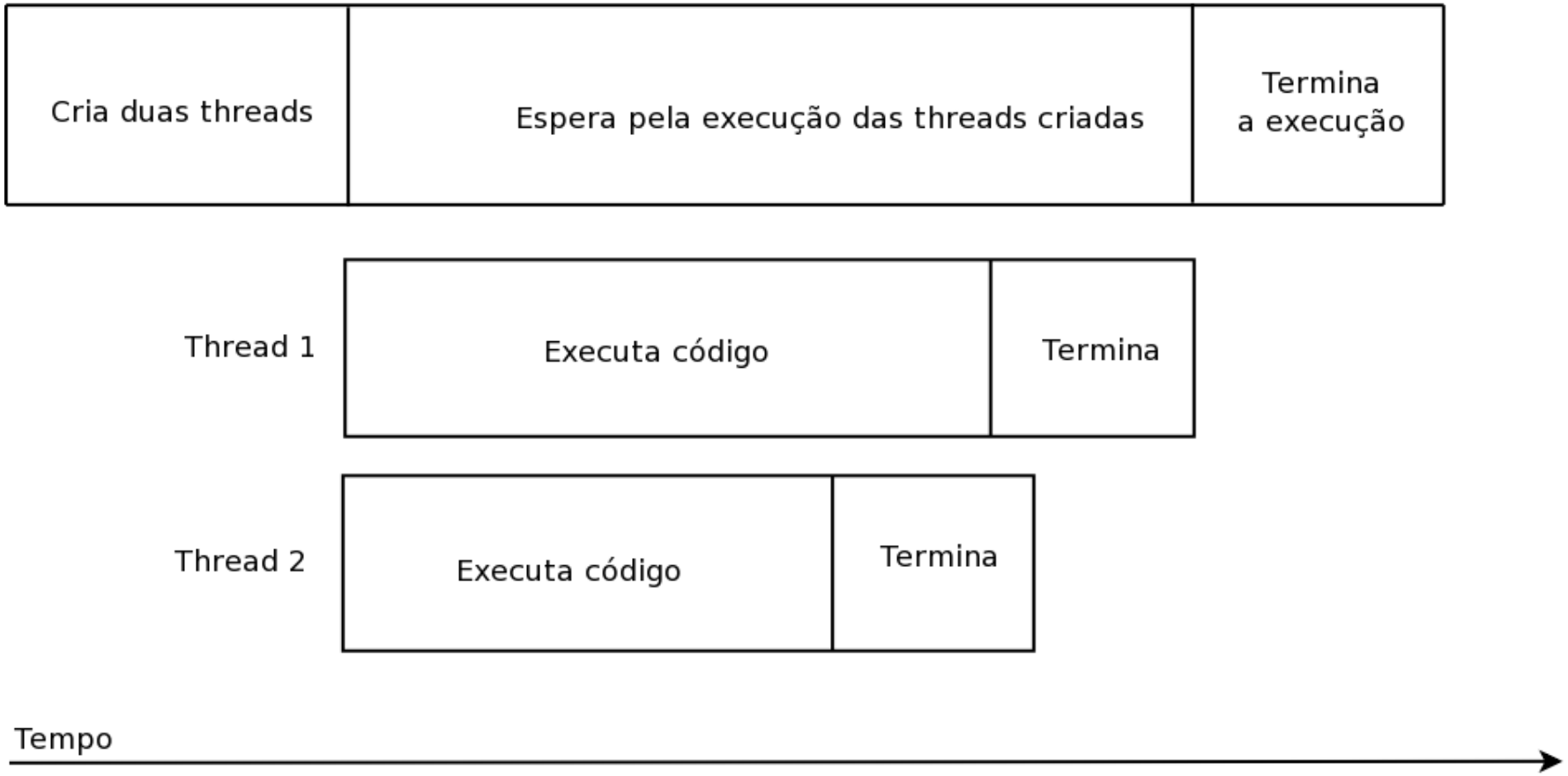
- pthread_mutex_unlock(mutex)
    - 'mutex': fechadura a ser destrancada

Essa função destrói um mutex, desalocando a memória que ele gasta.

- pthread_mutex_destroy(mutex)
    - 'mutex': estrutura que será destruída

Exemplo

O exemplo começa com a thread principal, que cria duas outras threads e espera que elas terminem seu trabalho. Cada uma das threads realiza um trabalho, ao mesmo tempo, e elas terminam o trabalho aproximadamente no mesmo tempo. Depois, elas retornam, e a thread principal termina.

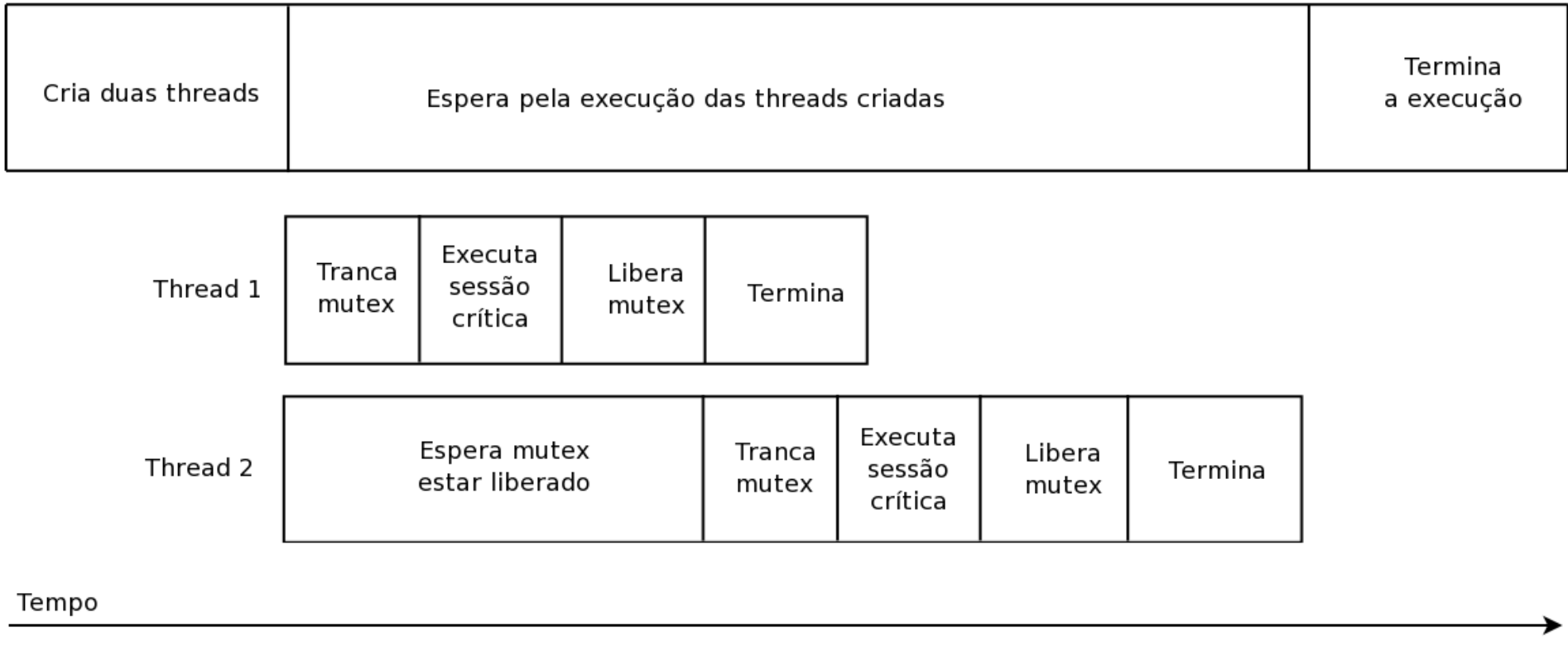


```c
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
typedef struct
{
    int id;
} thread_arg;
void *thread(void *vargp);
int main()
{
    pthread_t tid[2];
    thread_arg a[2];
    int i = 0;
    int n_threads = 2;
    //Cria as threads
    for(i=0; i<n_threads; i++)
    {
        a[i].id = i;
        pthread_create(&(tid[i]), NULL, thread, (void *)&(a[i]));
    }
    // Espera que as threads terminem
    for(i=0; i<n_threads; i++)
    {
        pthread_join(tid[i], NULL);
    }
    pthread_exit((void *)NULL);
}
void *thread(void *vargp)
{
    int i = 0;
    thread_arg *a = (thread_arg *) vargp;
    printf("Comecou a thread %d\n", a->id);
    // Faz um trabalho qualquer
    for(i = 0; i < 1000000; i++);
    printf("Terminou a thread %d\n", a->id);
    pthread_exit((void *)NULL);
}
```

Exemplo

Neste exemplo, duas threads serão criadas, usando a mesma função. No entanto, certa linha dessa função será protegida com o uso de um mutex, já que ela altera o valor de uma variável global (variáveis globais não devem ser usadas, isso é apenas um exemplo!). Essa é uma das técnicas normalmente utilizadas para se proteger zonas críticas do código.

O exemplo começa com a thread principal, que cria outras duas, e espera que elas terminem. Qual das duas threads chegam primeiro ao mutex é indeterminado, mas a que chegar trava o mutex, modifica var, e libera o mutex para que a outra faça o mesmo. Então, ambas terminam, e depois a principal também.



```c
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
typedef struct
{
   int id;
} thread_arg;
void *thread(void *vargp);
pthread_mutex_t mutex;
int var;
int main()
{
   pthread_t tid[2];
   thread_arg a[2];
   int i = 0;
   int n_threads = 2;
   var = 0;
   // Cria o mutex
   pthread_mutex_init(&mutex, NULL);
   //Cria as threads
   for(i=0; i<n_threads; i++)
   {
      a[i].id = i;
      pthread_create(&(tid[i]), NULL, thread, (void *)&(a[i]));
   }
   // Espera que as threads terminem
   for(i=0; i<n_threads; i++)
   {
      pthread_join(tid[i], NULL);
   }
   // Destroi o mutex
   pthread_mutex_destroy(&mutex);
   pthread_exit((void *)NULL);
}
void *thread(void *vargp)
{
   // Converte a estrutura recebida
   thread_arg *a = (thread_arg *) vargp;
   // Como vamos acessar uma variavel global, deve-se protege-la com uma fechadura
   pthread_mutex_lock(&mutex);
   printf("Thread %d: valor de var antes da conta: %d\n", a->id+1, var);
   var = var + a->id + 1;
   printf("Thread %d: valor de var depois da conta: %d\n", a->id+1, var);
   pthread_mutex_unlock(&mutex);
   pthread_exit((void *)NULL);
}
```

Exercício - Trabalho

- Faça um programa para somar os elementos de um vetor usando threads. Cada thread realiza uma soma parcial e acrescenta esta soma em uma variável que no final conterá a soma total. O acesso a essa variável deve estar em uma seção crítica.