Programação Paralela em Arquiteturas Multi-Core/Programação em Pthreads

< Programação Paralela em Arquiteturas Multi-Core

Índice

Introdução

Nesse capítulo serão discutidos alguns pontos fundamentais para se entender a programação em Pthreads, passando pelas motivações, por definições e explicações de conceitos básicos e finalmente dando a diretiva inicial para o assunto abordado, ou

seja, a iniciação na programação paralela em Pthreads.

Necessidade de processamento massivo: o que acarreta?

As necessidades de processamento massivo nos dias de hoje são muitas: cálculos matemáticos, empíricos, processamento de enormes quantidades de dados, números, arquivos, simulações de fenômenos naturais, aplicações gráficas cada vez mais requintadas, aplicações médicas, químicas e científicas em geral. Enfim, há uma infinitude de áreas que demandam a resolução de problemas cada vez mais complexos, com o envolvimento de uma enorme quantidade de operações, instruções e dados, e cada vez mais são necessários computadores e formas de processamento mais rápidas do que as atuais para tais aplicações.

Necessidade de paralelismo

Vários fatores explicam a necessidade do processamento paralelo. O principal deles trata da busca por maior desempenho. As diversas áreas nas quais a computação se aplica, sejam científicas, industriais ou militares, requerem cada vez mais poder computacional, em virtude dos algoritmos complexos que são utilizados e do tamanho do conjunto de dados a ser processado.

Além disso, várias aplicações são inerentemente paralelas, e perde-se desempenho pela necessidade de torná-las seqüenciais. O chamado "gargalo de von Neumann", segundo Almasi, tem diminuído a produtividade do programador, daí a necessidade de novas maneiras de organização do processamento computacional.

Contudo, substituir uma filosofia computacional já firmemente estabelecida pelas várias décadas de existência da computação, como é a filosofia de von Neumann, é algo que representa um obstáculo de dimensões muito grandes, e que de certa maneira dificulta a difusão da computação paralela.

Necessidade de compartilhamento fácil de memória

A necessidade de processamento massivo e de paralelismo leva a uma outra questão, relativa ao compartilhamento de memória em sistemas paralelos. Como os diversos processadores utilizam uma mesma memória física, tal tarefa não é trivial e, em geral, limita a escalabilidade do sistema.

Neles, a comunicação é realizada em hardware mediante a troca de mensagens. Isso fica bem claro no nível da aplicação quando o modelo de programação é orientado segundo essa lógica. Porém, embora tal linha de comunicação seja bastante natural, ela pode certamente se mostrar extremamente complicadas em diversas situações, como aquelas nas quais os padrões de interação e compartilhamento de dados são mais complexos e possuem escalas maiores. Além disso, tal lógica exige um maior nível de treinamento por parte dos programadores, o quem nem sempre acontece.

De fato, quando passamos para a dimensão do software paralelo, temos dois paradigmas clássicos de programação paralela nesse sentido: o da **troca de mensagens**, mencionado anteriormente, e o **compartilhamento de memória**.

A programação no paradigma de memória compartilhada é considerada mais simples, pois evita que o programador tenha que se preocupar com a comunicação entre processos, através da troca explícita de mensagens. Para realizar comunicação, um processo apenas escreve dados na memória para serem lidos por todos os outros. Para sincronização, seções críticas podem ser usadas, com a utilização de semáforos ou monitores para garantir exclusão mútua.

No paradigma de troca de mensagens a comunicação é realizada através de primitivas que explicitamente controlam o deslocamento dos dados. Troca de mensagens apresenta várias dificuldades, entre elas controle de fluxo, mensagens perdidas, controle do buffer (*buffering*) e bloqueio (*blocking*). Embora várias soluções tenham sido propostas, programação com troca de mensagens permanece complicada.[1]

Em resumo, os multicomputadores são simples de construir, mas difíceis de programar, enquanto os multiprocessadores são difíceis de construir mais simples de programar.[2] Além disso, com a grande demanda por sistemas paralelos com capacidade de suportar grandes volumes de dados, torna-se evidente a necessidade de maneiras de tornar essa tarefa o mais simples possível para o programador.

O que são threads?

Um thread, algumas vezes chamada um **processo peso leve**, é um fluxo seqüencial de controle dentro de um programa. Basicamente, consiste em uma unidade básica de utilização da CPU, compreendendo um ID, um contador de programa, um conjunto de registradores e uma pilha. Um processo tradicional tem um único thread de controle. Se o processo possui múltiplos threads de controle, ele pode realizar mais do que uma tarefa a cada momento. Essa possibilidade abre portas para um novo modelo de programação.

Threads são diferentes de processos nos seguintes pontos[3]: Um processo é criado pelo sistema operacional como um conjunto de recursos físicos e lógicos para executar um programa. Um processo inclui:

- Memória de heap, estática e de código;
- Registradores para lidar com a execução do código;
- Uma pilha;
- Informação do ambiente, incluindo um diretório de trabalho e descritores de arquivos;
- IDs de processos, de grupos e de usuários;
- Ferramentas de comunicação entre processos e bibliotecas compartilhadas.

Uma thread é o estado de execução de uma instância do programa, chamada algumas vezes de fluxo independente de controle. A thread é uma entidade escalonável. Ela tem propriedades que permitem que ela execute independentemente:

- Registradores para lidar com a execução do código;
- Uma pilha;
- Propriedades de escalonamento (como prioridade);
- Seu próprio conjunto de sinais;
- Algumas informações específicas de threads.

Definições

Introduzem-se aqui termos e conceitos utilizados na programação multi-threaded.

- **Escalonamento**: normalmente feito pelo sistema operacional, ele determina quais threads executam em determinado momento.
- **Sincronização**: quando um programa pára em um ponto, esperando que determinadas threads terminem seu trabalho, diz-se que houve a sincronização entre as threads.
- **Granularidade**: é o tamanho do trabalho atribuído a cada thread antes que elas se sincronizem. Um programa que seja dividido em partes pequenas entre as threads tem granularidade fina; um programa que faz com que suas threads tenham muito trabalho antes de se sincronizarem tem granularidade grossa.
- **Zona crítica**: uma parte do código que tem comportamento indeterminado caso seja executada por mais de uma thread ao mesmo tempo é chamada de zona crítica.

Um exemplo é quando duas threads tentam realizar, ao mesmo tempo, 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.

- **Condição de corrida**: quando duas threads modificam uma variável ao mesmo tempo, ocorre uma condição de corrida; só que nessa corrida, a thread que modificar o valor por último é a que tem seu resultado armazenado. Ocorre em situações como a descrita no exemplo de zona crítica.

- **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. Quando isso ocorre, apenas uma das threads que estavam esperando consegue continuar, enquanto as outras esperam mais, e assim por diante.

- **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.

- **Deadlocks**: ocorrem 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.

Benefícios

Seguem as quatro categorias principais dos benefícios da programação com *multithreads*:

- **Capacidade de Resposta:** A estruturação de uma aplicação intereativa em *multithreads* pode permitir a um programa continuar executando mesmo se parte dele estiver bloqueada ou realizando uma operação prolongada, aumentando desse modo a capacidade de resposta para o usuário. Por exemplo, um navegador de web *multithread* poderá permitir uma interação de usuário em *thread* enquanto uma imagem estará sendo armazenada em outro *thread*.

- **Compartilhamento de Recursos:** Os *threads* compartilham, por padrão, a memória e os recursos do processo ao qual pertencem. A vantagem do compartilhamento de código é permitir que uma aplicação tenha diversos *threads* de atividade, todos dentro do mesmo espaço de endereçamento.

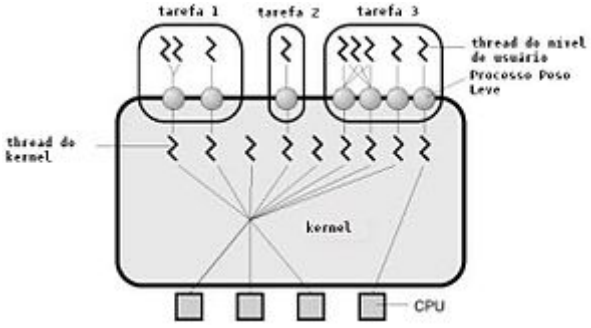
- **Economia:** A alocação de memória e recursos para a criação de processos é custosa. Por outro lado, como os *threads* compartilham recursos do processo ao qual pertencem, é mais econômico criar *threads* e comutar seus contextos. Em geral, consome muito mais tempo criar e gerenciar processos do que *threads*.

- **Utilização de Arquiteturas de Multiprocessadores:** As vantagens da criação de *multithreads* podem ser bastante aumentadas em uma arquitetura de multiprocessadores, onde cada *thread* pode ser executado em paralelo em um processador diferente. Um processo com um único *thread* pode executar somente em uma CPU, não importa quantas estejam disponíveis. A execução de *multithreads* em máquinas com múltiplas CPUs aumenta a concorrência.

Suporte dos sistemas operacionais[4]

Threads do Solaris 2

O Solaris 2 é uma versão do UNIX com suporte para threads nos níveis do *kernel* e do usuário, SMP e *scheduling* de tempo real. O Solaris 2 implementa a API Pthread, que será discutida em breve nesse capítulo, além de suportar *threads* de nível de usuário com uma biblioteca contendo APIs para criação e gerenciamento de *threads* (conhecida como *threads UI*). As diferenças entre essas duas bibliotecas são significativas. O Solaris 2 define também um nível intermediário de *threads*. Entre os *threads* de nível de usuário e de *kernel*, existem processos peso leve (LWPs). Cada processo contém pelo menos um LWP. A biblioteca de *threads* multiplexa *threads* de nível de usuário na cadeia de LWPs para o processo, e somente *threads* de nível de usuário conectados no momento a um LWP cumprem a tarefa. Os restantes ou ficam bloqueados ou aguardando por um LWP no qual possam executar.



Threads do Solaris 2

Threads do Windows 2000

O Windows 2000 implementa a API Win32, que é a API primária para a família de sistemas operacionais da Microsoft (Windows 95/98/NT e Windows 2000).

Uma aplicação do Windows executa como um processo separado onde cada processo pode conter um ou mais *threads*. O Windows 2000 utilizava um mapeamento um-para-um, onde cada *thread* de nível de usuário mapeia para um *thread* de *kernel* associado. Entretanto, o Windows também fornece suporte para uma biblioteca de fibra, que oferece a funcionalidade do modelo muitos-para-muitos. Cada *thread* que pertence a um processo pode acessar o espaço de endereçamento virtual do processo.

Os componentes gerais de um *thread* incluem:

- Um ID de *thread* identificando unicamente o *thread*
- Um conjunto de registradores representando o estado do registrador
- Um pilha de usuário utilizada quando o *thread* está executando em modalidade de usuário. De modo semelhante, cada *thread* também tem uma pilha de *kernel* utilizada quando o *thread* está executando em modalidade de *kernel*
- Uma área de armazenamento privada utilizada por várias bibliotecas de *run-time* e bibliotecas de links dinâmicos

O conjunto de registradores, as pilhas e a área de armazenamento privada são conhecidas como o *contexto* do *thread* e são específicos, do ponto de vista da arquitetura, para o hardware no qual o sistema operacional está executando. As estruturas de dados primárias de um *thread* incluem:

- O ETHREAD (bloco de *thread* executivo)
- O KTHREAD (bloco de *thread* do *kernel*)
- O TEB (bloco do ambiente do *thread*)

Os componentes-chave do ETHREAD incluem um ponteiro para o processo ao qual o thread pertence e o endereço da rotina na qual o *thread* inicia o controle. O ETHREAD também contém um ponteiro para o correspondente KTHREAD.

O KTHREAD inclui informação de *scheduling* e sincronização para o *thread*. Além disso, o KTHREAD inclui a pilha do *kernel* (utilizada quando o *kernel* está executando em modalidade de *kernel*) e um ponteiro para o TEB.

O ETHREAD e o KTHREAD existem inteiramente no espaço do *kernel*, o que significa que somente o *kernel* pode acessá-los. O TEB é uma estrutura de dados do espaço do usuário que é acessada quando o *thread* está executando em modalidade de usuário. Entre outros campos, o TEB contém uma pilha de modalidade de usuário e um *array* para dados específicos do *thread* (que o Windows chama de *memória local* do *thread*).

Threads do Linux

O *kernel* do Linux introduziu os *threads* na versão 2.2. O Linux fornece uma chamada de sistema **fork** com a funcionalidade tradicional de duplicação de um processo. O Linux também oferece a chamada de sistema **clone** que é análoga à criação de um *thread*. O **clone** comporta-se como o **fork**, exceto que em vez de criar uma cópia do processo que realizou a chamada, ele cria um processo separado que compartilha o espaço de endereçamento daquele processo. É através deste compartilhamento do espaço de endereçamento do processo pai que uma tarefa clonada comporta-se como um *thread* separado.

O compartilhamento do espaço de endereçamento é permitido por causa da representação de um processo no *kernel* do Linux. Existe uma única estrutura de dados de *kernel* para cada processo no sistema. Entretanto, em vez de armazenar os dados para cada processo nesta estrutura de dados, ele contém ponteiros para outras estruturas de dados onde estes dados são armazenados. Por exemplo, esta estrutura de dados por processo contém ponteiros para outras estruturas de dados que representam a lista de arquivos abertos, informação de manipulação de sinais e memória virtual. Quando **fork** é invocada, um novo processo é criado com uma *cópia* de todas as estruturas de dados associadas do processo pai. Quando a chamada de sistema **clone** é realizada, um novo processo é criado. Entretanto, em vez de copiar todas as estruturas de dados, o novo processo *aponta* para as estruturas de dados do processo pai, desse modo permitindo que o processo filho compartilhe a memória e outros recursos do processo pai. Um

conjunto de *flags* é passado com um parâmetro para a chamada de sistema **clone**. Este conjunto de *flags* é utilizado para indicar quanto do processo pai é para ser compartilhado com o filho. Se nenhum dos *flags* for posicionado, não ocorrerá compartilhamento e ambas as chamadas de sistema mencionadas atuarão igualmente. Se todos os cinco *flags* forem posicionados, o processo filho irá compartilhar todos os recursos com o processo pai. Outras combinações de *flags* permitem vários níveis de compartilhamento entre estes dois extremos.

Curiosamente, o Linux não faz distinção entre processos e *threads*. De fato, o Linux utiliza em geral o termo *tarefa* - em vez de processo ou *thread* - quando se refere a um fluxo de controle dentro de um programa. À parte o processo clonado, o Linux não suporta geração de *multithreads*, estruturas de dados separadas ou rotinas do *kernel*. Entretanto, várias implementações de Pthreads estão disponíveis para geração de *multithreads* de nível de usuário.

Modelos de programação

Nessa sessão, serão apresentados alguns modelos de programação usando mais de uma thread.[5]

- **Mestre/Escravo**: Uma única thread mestre recebe a entrada ou as requisições e distribui o trabalho entre as diversas threads escravas criadas por ela, determinando o que será feito por cada uma.
- **Dividir para conquistar**: Várias threads individuais realizam trabalhos relacionados independentemente, sem uma thread mestre coordenando o trabalho.
- **Pipelining**: O trabalho é dividido em algumas partes, de forma que quando uma thread termina seu serviço, ela o passa para a próxima thread e recebe o resultado da comutação da thread anterior, seguindo um modelo semelhante ao de uma linha de montagem.

Pthreads

O que é?

Bibliotecas implementando o padrão de POSIX threads são normalmente chamadas de Pthreads. Elas são usadas normalmente em sistemas do tipo UNIX como Linux e Solaris, mas existem implementações para Windows.

Historicamente, vendedores de hardware implementaram suas próprias versões proprietárias de threads. Essas implementações eram muito diferentes umas das outras, fazendo com que fosse difícil criar um programa com threads portável.

Para usar todas as vantagens das capacidades das threads, era necessária uma interface padronizada. Para sistemas UNIX, essa interface foi especificada pelo padrão IEEE POSIX 1003.1c (1995). Implementações que aderem a esse padrão são chamadas de POSIX threads, ou Pthreads.

Basicamente, Pthreads é um padrão que define uma interface para criação e manipulação de threads.

Primitivas básicas

Nessa sessão, serão explicadas algumas das funções mais importantes da API de threads. Será usado o padrão C para explicá-las, mas a interface é basicamente a mesma para as outras linguagens como C++.

Gerenciamento

Serão citadas aqui algumas das funções usadas para se criar, destruir, e sincronizar threads.

- pthread_create(thread, attr, start_routine, arg):

'thread': ponteiro estrutura previamente alocada que conterá os atributos da thread
'attr': estrutura contendo opções de criação para a thread (NULL usa os valores padrão)
'start_routine': função que será executada pela thread
'arg': argumento recebido pela função

Essa função cria uma thread que executa a função por ela especificada.

- pthread_exit(retval)

'retval': valor de retorno da thread

Essa função termina a execução da thread.

- pthread_join(th, thread_return)

'th': thread a ser esperada
'thread_return': valor de retorno da thread.

Essa função faz com que a thread que a chamou espere até que a thread passada como parâmetro retorne.

Exclusão mútua

Serão citadas aqui algumas das funções que implementam o conceito de fechadura, explicado anteriormente. Como criar, destruir, trancar e destrancar uma 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 inicializa um mutex, que implementa o paradigma de fechadura, explicado anteriormente.

- pthread_mutex_lock(mutex)

'mutex': o mutex que será usado como fechadura.

Essa função é basicamente a fechadura explicada anteriormente: 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_unlock(mutex)

'mutex': fechadura a ser destrancada

Essa função destranca um mutex.

- pthread_mutex_destroy(mutex)

'mutex': estrutura que será destruída

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

Variáveis condicionais

Serão citadas aqui algumas das funções de variáveis condicionais, que implementam o conceito de semáforos: threads param em um semáforo, e só avançam quando uma outra thread libere passagem para elas, algo que normalmente é feito sob uma condição.

- pthread_cond_init(cond, cond_attr)

'cond': estrutura a ser inicializada
'attr': opções de criação. Caso NULL seja usado, opções padrão serão usadas

Essa função inicializa a estrutura de condição de threads.

- pthread_cond_destroy(cond)

'cond': estrutura a ser destruída

Essa função desaloca a memória gasta por uma estrutura de condição inicializada.

- pthread_cond_wait(cond, mutex)

'cond': variável de condição a ser sinalizada
'mutex': mutex a ser destrancado

Essa função faz com que a thread que a chame destrave 'mutex' e espere até 'cond' ser sinalizada para continuar sua execução.

- pthread_cond_signal(cond)

'cond': variável a ser sinalizada

Essa função sinaliza 'cond', fazendo com que uma e apenas uma das threads que estejam paradas esperando por sua sinalização continue sua execução.

- pthread_cond_broadcast(cond)

'cond': variável a ser sinalizada

Semelhante à função anterior, mas faz com que todas as threads que estejam esperando por um sinal continuem sua execução.

Como programar

Essa sessão visa mostrar como programar usando pthreads, usando exemplos na linguagem C.

Como compilar

Uma nota importante é que é necessário linkar a biblioteca de pthreads durante a compilação. Com o gcc, isso é feito da forma "gcc -lpthread programa.c".

Exemplo 1: "Hello World"

O exemplo abaixo faz o seguinte: ele imprime uma mensagem em main, cria uma thread, e espera que ela retorne, e termina. A segunda thread imprime outra mensagem e retorna. Ele mostra como usar as funções pthread_create, pthread_join e pthread_exit.

É importante notar que a função passada para a thread deve retornar void * e receber como parâmetro void *.

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

void *thread(void *vargp);

int main()
{
    pthread_t tid;
    printf("Hello World da thread principal!\n");
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);
    pthread_exit((void *)NULL);
```

```
}

void *thread(void *vargp)
{
    printf("Hello World da thread criada pela thread principal!\n");
    pthread_exit((void *)NULL);
}
```

O exemplo acima começa com uma thread, a principal, que cria uma segunda thread e espera que ela termine. A segunda thread imprime uma mensagem na tela e termina, e depois dela a principal.

Exemplo 2: Passagem de Parâmetros

O exemplo a seguir já é um pouco mais elaborado: ele cria uma thread, usando uma função que necessita de dois parâmetros. Para tal, é necessário criar uma estrutura para acomodar os dois parâmetros:

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

typedef struct
{
    int i;
    int j;
} thread_arg;

void *thread(void *vargp);

int main()
{
    pthread_t tid;
    thread_arg a;

    a.i = 1;
    a.j = 2;

    pthread_create(&tid, NULL, thread, (void *)&a);
    pthread_join(tid, NULL);
    pthread_exit((void *)NULL);
}

void *thread(void *vargp)
{
    // Converte a estrutura recebida
    thread_arg *a = (thread_arg *) vargp;
    int i = a->i;
    int j = a->j;

    printf("Parametros recebidos: %d %d\n", i, j);


    pthread_exit((void *)NULL);
}
```

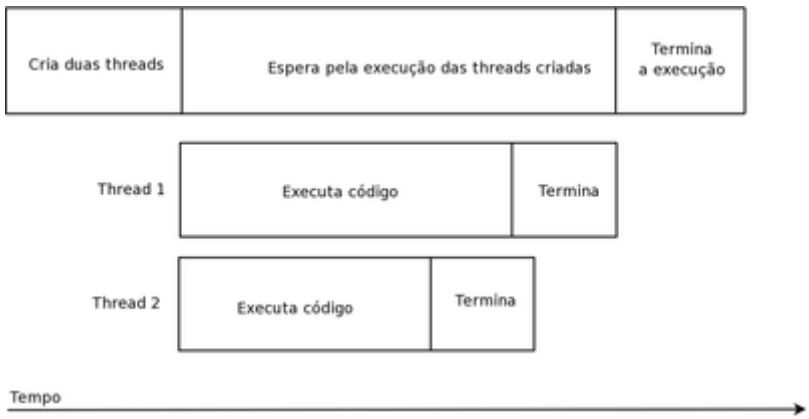
O exemplo acima começa com a thread princpal, que cria uma segunda thread e espera que ela termine. A segunda thread executa, imprime quais os argumentos recebidos e termina, e logo após a thread principal também

Exemplo 3: Execução paralela

O exemplo abaixo mostra duas threads executando ao mesmo tempo, realizando cada uma seu trabalho.

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

typedef struct
{
    int id;
} thread_arg;

void *thread(void *vargp);
```

```c
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);
}
```

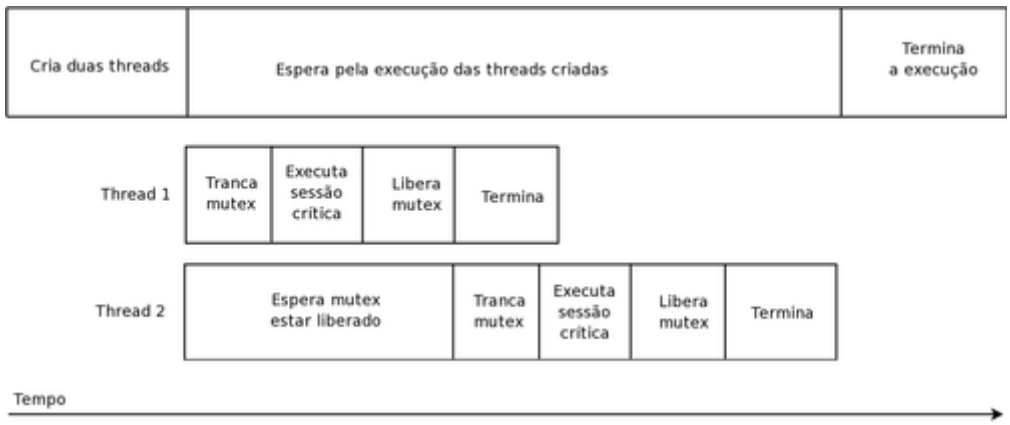


Possível fluxo de execução do exemplo abaixo

O exemplo acima 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.

Exemplo 4: Lock e Unlock (mutex)

Agora, será feita mais uma elaboração: 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.



Possível fluxo de execução do exemplo abaixo

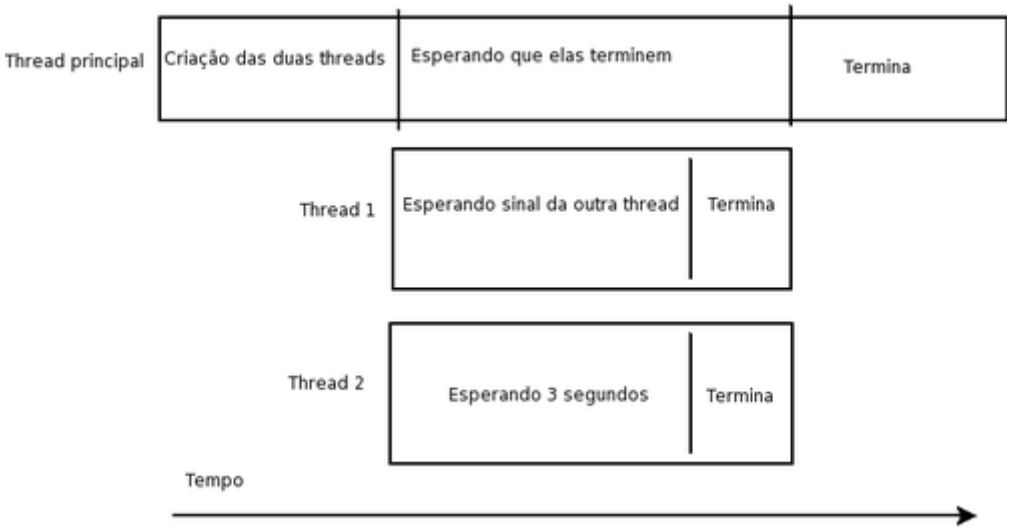
Apesar de o uso de variáveis globais ser desaconselhado, normalmente os mutex são declarados globalmente, pois eles devem ser visíveis a todas as threads.

```
#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);
}
```

O exemplo acima 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.

Exemplo 5: Variáveis Condicionais

O código abaixo mostra um exemplo simples de como funciona o uso de variáveis condicionais. Assim como com os mutex, é comum declarar as variáveis condicionais globalmente, pois elas também devem ser visíveis a todas as threads.



Fluxo de execução de pthreads usando variáveis condicionais

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

typedef struct
{
    int id;
} thread_arg;

void *thread(void
*vargp);

pthread_mutex_t
mutex;
pthread_cond_t cond;

int main()
{
    pthread_t tid[2];
    thread_arg a[2];
    int i = 0;
    int n_threads =
2;

    // Cria o mutex

pthread_mutex_init(&m
utex, NULL);

    // Cria a variavel de condicao
    pthread_cond_init(&cond, 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);

    // Destroi a variavel condicional
    pthread_cond_destroy(&cond);

    pthread_exit((void *)NULL);
}

void *thread(void *vargp)
{
    // Converte a estrutura recebida
    thread_arg *a = (thread_arg *) vargp;

    if(a->id == 0)
    {
        printf("Thread %d: Esperando a outra thread\n", a->id);
        pthread_cond_wait(&cond, &mutex);
        printf("Thread %d: Acabou a espera\n", a->id);
    }
    else
    {
        sleep(3);
        printf("Thread %d: Sinalizando a outra thread\n", a->id);
        pthread_cond_signal(&cond);
    }

    pthread_exit((void *)NULL);
}
```

O exemplo acima começa com a thread principal, que cria duas outras threads, e espera que elas terminem. Depois, a primeira thread pára em um semáforo, e espera que a variável de condição seja sinalizada por outra thread. Isso só ocorre três segundos depois, quando a segunda thread sinaliza a condição e termina. Depois, a primeira thread também termina sua execução, e depois

a thread principal.

Depuração (*Debugging*)

A depuração consiste em um processo metódico para se encontrar e reduzir o número de erros, ou problemas, em um programa de computador ou um componente de hardware, fazendo-o se comportar como o que foi especificado, aumentando a sua qualidade. A depuração tende a ser bem mais complexa quando vários subsistemas estão estreitamente acoplados, pois as modificações em um deles pode afetar outros.

A questão ainda se complica. Depurar aplicações sequenciais já pode ser complicado, em função de erros, variáveis não inicializadas, apontadores e outros. Quando falamos em depuração de aplicações paralelas, o problema ganha múltiplas dimensões, tais como uma grande propensão para condições de corrida, eventos assíncronos e a dificuldade geral de se entender a execução de vários processos ao mesmo tempo. O problema sem dúvidas se torna formidável.

Para facilitar essa árdua tarefa, algumas ferramentas de depuração oferecem abstrações para que o usuário possa controlar cada aspecto do seu processo. São os chamados depuradores. Outras ferramentas ainda oferecem um perfil de execução da aplicação que pode auxiliar o programador. Uma ferramenta pode ser conferida na referência [6]. De um certo modo, ela tenta modularizar a tarefa de depurar os erros de um programa. No entanto, essa biblioteca é um pouco complicada. Técnicas como colocar prints em locais estratégicos do código são mais simples, e possivelmente surtem um resultado tão bom quanto o dessas bibliotecas dependendo da aplicação, tendo sempre em mente que nos casos mais complexos se o programa não for modular e bem estruturado o programador pode se perder nessa difícil tarefa que é a depuaração.

Além disso, o clássico *gdb*, oferece suporte à depuração de threads, mesmo que minimamente. As versões mais recentes têm tido uma preocupação cada vez maior com esse aspecto. Além disso, projetos da comunidade científica, como nas referências [7], [8] tentam incrementar a ferramenta com um suporte maior ao uso de *Pthreads*.

Em suma, diversas ferramentas se encontram disponíveis para facilitar a depuração de programas com múltiplas threads, que ajudam a reduzir a complexidade da construção e manutenção de aplicações paralelas.

Depurando um programa em Pthreads [9]

Para depurar o seu programa, compile-o com -g. Use o ddd, ou outro depurador gráfigo que te dê acesso ao console do gdb.

Comandos úteis:

info threads, mostra quais threads estão rodando e onde cada uma está (a thread com um * é a que vc está):

```
(gdb) info threads
  5 Thread -1234650192 (LWP 4313)  0xb7fcf199 in __lll_mutex_lock_wait () from
/lib/tls/libpthread.so.0
* 4 Thread -1226261584 (LWP 4312)  thread1 (ptr=0x80560d8) at bellmanford.c:42
  3 Thread -1217872976 (LWP 4311)  thread1 (ptr=0x80560c8) at bellmanford.c:49
  2 Thread -1209484368 (LWP 4310)  thread1 (ptr=0x80560b8) at bellmanford.c:42
  1 Thread -1209481536 (LWP 4096)  0xb7fc97c0 in __nptl_create_event () from /lib/tls/libpthread.so.0
(gdb)
```

O número após o LWP é o número do processo. Na pthread antiga, todas as threads apareciam no ps, mas na nova só a principal aparece:

```
[usuario@maquina]$ ps ux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
coutinho   449  0.0  0.1   6344  1048 ?        Ss   18:52   0:00 /usr/lib/WindowMaker/WindowMaker
coutinho  2057  0.0  0.4   8196  4244 ?        Rs   21:19   0:00 xterm -sl 2048
coutinho  2058  0.0  0.3   6192  3576 pts/14   Ss   21:19   0:00 bash
coutinho  4090  0.0  0.8  13236  8516 pts/14   S    21:56   0:01 ddd principal
coutinho  4092  0.0  0.3   5936  4028 pts/16   Ss+  21:56   0:00 gdb -q -fullname principal
coutinho  4096  0.0  0.0  34428   596 pts/16   Tl   21:57   0:00
```

```
/home/speed/coutinho/grad/aeds3_tp2/principal -g entrada
coutinho  4586  0.0  0.0   2248   960 pts/14   R+   22:57   0:00 ps ux
[usuario@maquina]$
```

thread 1, vai para a thread número 1 mostrada no info threads

info stack, mostra a pilha da pthread:

```
(gdb) info stack
#0  mybarrier (barreira=0x804aa60, myself=0) at barrier.c:40
#1  0x08048fd0 in thread1 (ptr=0x80560b8) at bellmanford.c:62
#2  0xb7fcaced in start_thread () from /lib/tls/libpthread.so.0
#3  0xb7f5edee in clone () from /lib/tls/libc.so.6
(gdb)
```

frame 1, quando vc está no topo da pilha, vc está no frame #0, vc pode ir para outros frames para ver com quais parâmetros uma função foi chamada, o valor das variáveis locais da função que chamou, etc.

Depurando com Eletric Fence

Para compilar seu programa com electric fence, vc tem que "linkar" ele com a biblioteca -lefence:

gcc ... -D_REENTRANT -lpthread -lefence

A electric fence vai substituir as funções malloc(), realloc() e free(), por versões que colocam uma proteção após a área alocada. Se você tentar acessar um byte após o que foi alocado, vai ocorrer um segmentation fault na hora.

Assim: compile seu programa com electric fence, rode ele no ddd sem colocar breakpoint e ele vai parar no momento exato que seu programa estiver fazendo um acesso inválido. Caso ele pare dentro de uma função da glibc, de info stack e frame X pra ver qual função sua chamou a função que ocasionou o segmentation fault.

Atenção: às vezes o electric fence dá segmentation fault sozinho. Quando tomar um segmentation fault, veja a saida do programa. Se tiver algo como electric fence: internal error, o electric fence tomou segmentation fault sozinho.

O electric fence tem um limite para alocação de memória (cerca de 200MB), se você ultrapassar esse limite vai receber uma mensagem assim:

ElectricFence Exiting: mprotect() failed: Cannot allocate memory

Aplicações

Atualmente, todo tipo de aplicação requer uma versão paralelizada, pois a tendência é o uso de processadores multicore, sendo que em pouco tempo todos os processadores terão mais de um core. O padrão Pthread é uma das opções que permite o uso maximizado desses processadores, sendo que não existe, portanto um tipo específico de aplicação para Pthreads. É claro que em algumas, o seu uso é mais óbvio (como em servidores de web, que recebem múltiplas requisições ao mesmo tempo), mas toda ou quase toda aplicação pode ser paralelizada com o uso de Pthreads.

Prós e contras

Vantagens de se programar utilizando threads:

- Utiliza melhor o potencial dos novos processadores multi-core que estão ficando cada vez mais comuns atualmente.

- O preço da troca de contextos entre threads é menos do que com processos, devido ao fato de as threads serem mais leves.
- Existem diversas aplicações com paralelismo inerente, como os grandes servidores da web, que atendem a múltiplas requisições ao mesmo tempo. Utilizar threads nesses contextos é simples.

Desvantagens de se programar utilizando threads:

- O modelo de programação que utiliza threads é mais complexo do que o modelo seqüencial.
- Converter programas prontos para programas utilizando threads não é uma tarefa trivial, pois todo o programa pode ter de ser reescrito.

Referências Citadas

1. http://www.inf.unisinos.br/~holo/publicacoes/trab.individuais/dsm.pdf
2. TANENBAUM, A. S. Distributed Operating Systems. Prentice Hall, New Jersey, 1995.
3. http://www.llnl.gov/LCdocs/pthreads/index.jsp
4. Silberschatz, A. Galvin, P.B. and Gagne, G., "Fundamentos de Sistemas Operacionais, Sexta Edição."
5. http://www.training.com.br/lpmaia/multithread.pdf
6. http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.genprogc/doc/genprogc/multi-thread_program.htm
7. http://moss.csc.ncsu.edu/~mueller/TDI/
8. http://moss.csc.ncsu.edu/~mueller/TDI/TDI_intro.html
9. http://homepages.dcc.ufmg.br/~coutinho/pthreads/

Referências extras

- Apresentação de Pthreads no SBAC (http://sureco.grude.ufmg.br/moodle16/file.php/16430/programando_multicore.odp)
- Um tutorial de Pthreads (http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html)
- Outro tutorial de Pthreads (http://randu.org/tutorials/threads/)
- Mais um tutorial (http://users.actcom.co.il/~choo/lupg/tutorials/multi-thread/multi-thread.html)
