Threads

Ricardo de la Rocha Ladeira

Introdução

A aula passada abordou o tema *Comunicação entre Processos*, e apresentou principalmente os mecanismos de *pipe* e *sockets*. Como visto, por exemplo, nos códigos estudados que utilizavam com *sockets*, assumimos que os processos possuíam apenas uma *thread* de controle, o que, pela implementação fornecida, não permitia concorrência. A proposta deste texto é, partindo da base já obtida pelas aulas passadas, introduzir o conceito de *threads* e começar a trabalhar com concorrência utilizando a linguagem C.

Grande parte do texto é baseada no capítulo 4 (*Threads & Concorrência*) de [1], cuja leitura foi solicitada nos exercícios da aula anterior.

Threads

Uma *thread* é uma unidade básica de utilização da CPU. Ela contém um ID de *thread*, um contador de programa (*program counter* – PC), um conjunto de registradores e uma pilha. Ele compartilha com outras *threads* pertencentes ao mesmo processo suas seções de código, de dados e outros recursos do sistema operacional, como arquivos abertos e sinais. Um processo tradicional tem uma única *thread* de controle. Se um processo tiver várias *threads* de controle, ele poderá executar mais de uma tarefa por vez. A Figura 1 ilustra a diferença entre um processo tradicional de única e um processo com múltiplas *threads*.

Sistemas operacionais modernos fornecem recursos que permitem que um processo contenha várias *threads* de controle. Além disso, a maioria dos aplicativos modernos também são *multithreaded*. Identificar oportunidades de paralelismo por meio do uso de *threads* está se tornando cada vez mais importante para sistemas *multicore* modernos que fornecem várias CPUs.

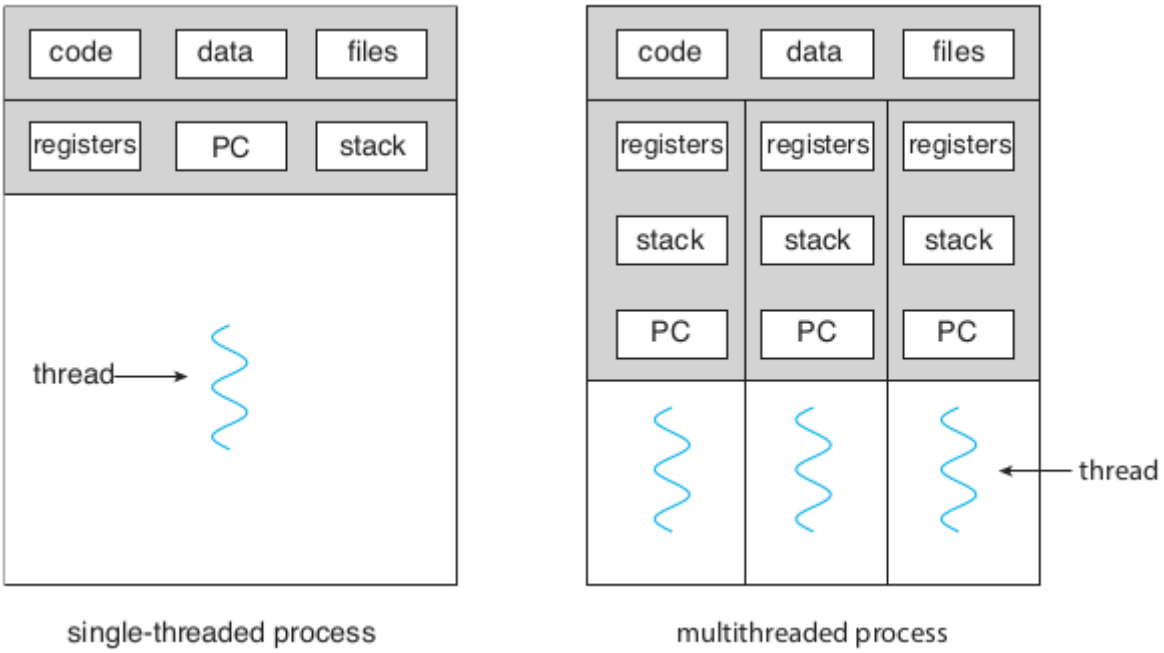

Figura 1. Thread-single-multi.png [1]. Disponível em https://drive.google.com/file/d/1QbjO3Oi5UNC-NtqG3OT6lrgwlB7VKB_R/view?usp=sharing.

Exemplos de aplicações *multithreaded*:

- Um aplicativo que cria figurinhas (*stickers*) de WhatsApp a partir de uma coleção de imagens pode usar *threads* distintas para gerar as figurinhas de cada imagem separada.
- Um navegador web pode ter uma *thread* exibindo imagens ou texto enquanto outra *thread* recupera dados da rede.
- Um processador de texto pode ter uma *thread* para exibir gráficos, outra *thread* para responder às teclas digitadas pelo usuário e uma terceira *thread* para executar a verificação ortográfica e gramatical em segundo plano.
- Um servidor de bate-papo precisará de uma *thread* individual para cada cliente que se conecta a ele.
- *Kernel* Linux, que cria *threads* para gerência de memória, gerência de dispositivos etc.

Em determinadas situações, um único aplicativo pode ser necessário para executar várias tarefas semelhantes. Por exemplo, um servidor web aceita solicitações de clientes para páginas da web, imagens, sons e assim por diante. Um servidor web pode ter vários clientes acessando-o simultaneamente. Se esse servidor fosse executado como um processo tradicional de uma *thread*, ele seria capaz de atender apenas um cliente por vez, e

um cliente poderia ter que esperar muito tempo para que sua solicitação fosse atendida. Situação semelhante ocorreu no código de exemplo de *sockets*, abordado na aula passada.

Uma solução seria criar um processo para cada cliente que realiza uma requisição, mas isso significa mais demora e mais uso de recursos. Na forma atualmente empregada, quando uma solicitação é feita, em vez de criar outro processo, o servidor cria uma nova *thread* para atender à solicitação e retoma a escuta de solicitações adicionais. Isso é ilustrado na Figura 2.

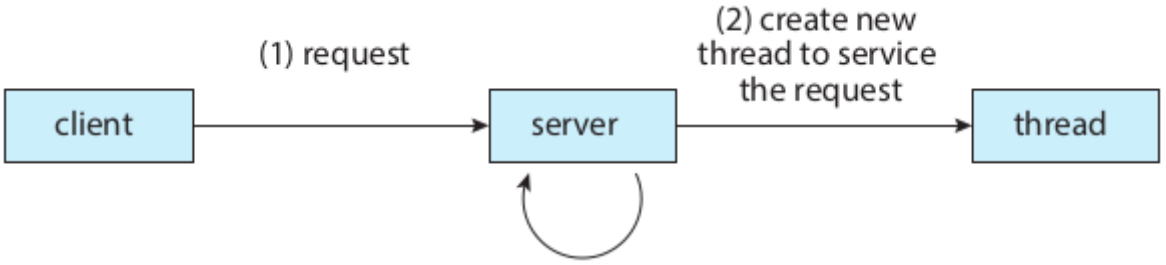

Figura 2. Arquitetura-servidor-multithreaded.png [1]. Disponível em https://drive.google.com/file/d/1pY9BQKM6WRubvQr8tWB5BB8ahO5RmSj5/view?usp=drive_link.

Benefícios da Programação *Multithread*

- **Capacidade de resposta**. *Multithreading* em um aplicativo interativo pode permitir que um programa continue em execução mesmo se parte dele estiver bloqueado ou executando uma operação demorada, aumentando assim a capacidade de resposta ao usuário.
- **Compartilhamento de recursos**. Processos podem compartilhar recursos apenas por meio de técnicas como memória compartilhada e troca de mensagens, o que deve ser explicitamente feito pelo programador. As *threads* compartilham a memória e os recursos do processo ao qual pertencem por padrão. O compartilhamento de código e dados permite que um aplicativo tenha várias *threads* de atividades diferentes dentro do mesmo espaço de endereço.
- **Economia**. Alocar memória e recursos para a criação de processos é caro. Como as *threads* compartilham os recursos do processo ao qual pertencem, é mais econômico criar e alternar *threads* de contexto.
- **Escalabilidade**. Os benefícios do *multithreading* podem ser ainda maiores em uma arquitetura multiprocessador, onde as *threads* podem ser executadas em paralelo em diferentes núcleos de processamento.

Programação *Multicore*

No início da história do projeto de computadores, para obter maior desempenho de computação, os sistemas de CPU única evoluíram para sistemas de várias CPUs. Uma tendência posterior foi colocar vários núcleos de computação em um único chip de processamento, onde cada núcleo aparece como uma CPU separada para o sistema operacional, sistemas que chamamos de *multicore*.

A programação *multithread* fornece um mecanismo para uso mais eficiente desses múltiplos núcleos de computação e simultaneidade aprimorada. Considere um aplicativo com quatro *threads*. Em um sistema com um único núcleo de computação, a simultaneidade significa apenas que a execução dos *threads* será intercalada ao longo do tempo (Figura 3), porque o núcleo de processamento é capaz de executar apenas um *thread* por vez. Já em um sistema com vários núcleos, simultaneidade significa que algumas *threads* podem ser executadas em paralelo, porque o sistema pode atribuir uma *thread* separada para cada núcleo (Figura 4).

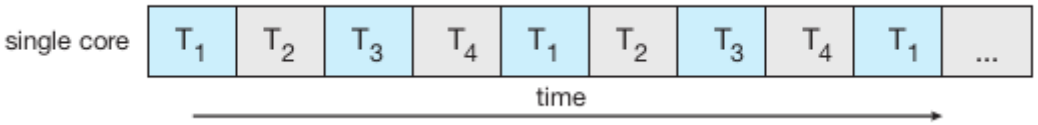

Figura 3. Single-core.png [1]. Disponível em https://drive.google.com/file/d/1-8Gpn16Lck5Tks5xfTEqRDTiIE3nPrXX/view?usp=drive_link.

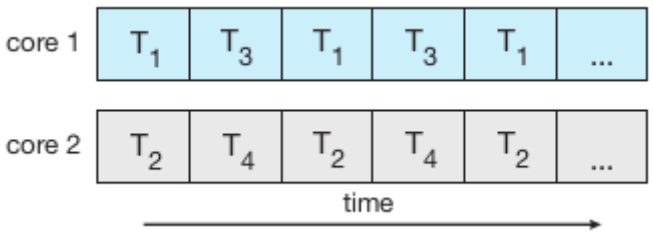

Figura 4. Multicore-system.png [1], disponível em https://drive.google.com/file/d/1SwIxutOHpXzu1rjvar5PG5gWq0NxEEHz/view?usp=drive_link.

Como já discutido na primeira aula, um sistema concorrente suporta mais de uma tarefa, permitindo que todas as tarefas progridam. Em contraste, um sistema paralelo pode executar mais de uma tarefa simultaneamente. Assim, é possível haver concorrência sem paralelismo. Antes do advento das arquiteturas multiprocessador e *multicore*, a maioria dos sistemas de computador tinha apenas um processador, e os escalonadores de CPU foram projetados para fornecer a ilusão de paralelismo ao alternar rapidamente entre os

processos, permitindo assim que cada processo progrida. Esses processos estavam sendo executados simultaneamente, mas não em paralelo.

Desafios

A tendência em direção a sistemas *multicore* continua pressionando os projetistas de sistemas e programadores para fazer melhor uso dos múltiplos núcleos de computação. Os projetistas de sistemas operacionais devem escrever algoritmos de escalonamento que usem múltiplos núcleos de processamento para permitir a execução paralela mostrada na Figura 4. Para os programadores, o desafio é modificar os programas existentes, bem como projetar novos programas *multithread*.

Em geral, cinco áreas apresentam desafios na programação para sistemas *multicore*:

1. **Identificação de tarefas**. Isso envolve o exame de aplicativos para encontrar áreas que podem ser divididas em tarefas separadas e simultâneas.
2. **Equilíbrio**. Ao identificar tarefas que podem ser executadas em paralelo, os programadores também devem garantir que as tarefas executem trabalho igual de valor igual. Em alguns casos, uma determinada tarefa pode não contribuir com tanto valor para o processo geral quanto outras tarefas. Usar um núcleo de execução separado para executar essa tarefa pode não valer o custo.
3. **Divisão de dados**. Assim como os aplicativos são divididos em tarefas separadas, os dados acessados e manipulados pelas tarefas devem ser divididos para serem executados em núcleos separados.
4. **Dependência de dados**. Os dados acessados pelas tarefas devem ser examinados quanto a dependências entre duas ou mais tarefas. Quando uma tarefa depende de dados de outra, os programadores devem garantir que a execução das tarefas seja sincronizada para acomodar a dependência de dados.
5. **Teste e depuração**. Quando um programa está sendo executado em paralelo em vários núcleos, muitos caminhos de execução diferentes são possíveis. Testar e depurar esses programas simultâneos é mais difícil do que testar e depurar aplicativos de *thread* única.

Lei de Amdahl

A Lei de Amdahl é uma fórmula que identifica possíveis ganhos de desempenho ao adicionar núcleos de computação adicionais a um aplicativo que possui componentes seriais (não paralelos) e paralelos. Se *S* for a parte da aplicação que deve ser executada

serialmente em um sistema com *N* núcleos de processamento, a fórmula aparece da seguinte forma:

$$speedup \leq \frac{1}{S + (\frac{1-S}{N})}$$

A Figura 5 ilustra a Lei de Amdahl em vários cenários diferentes.

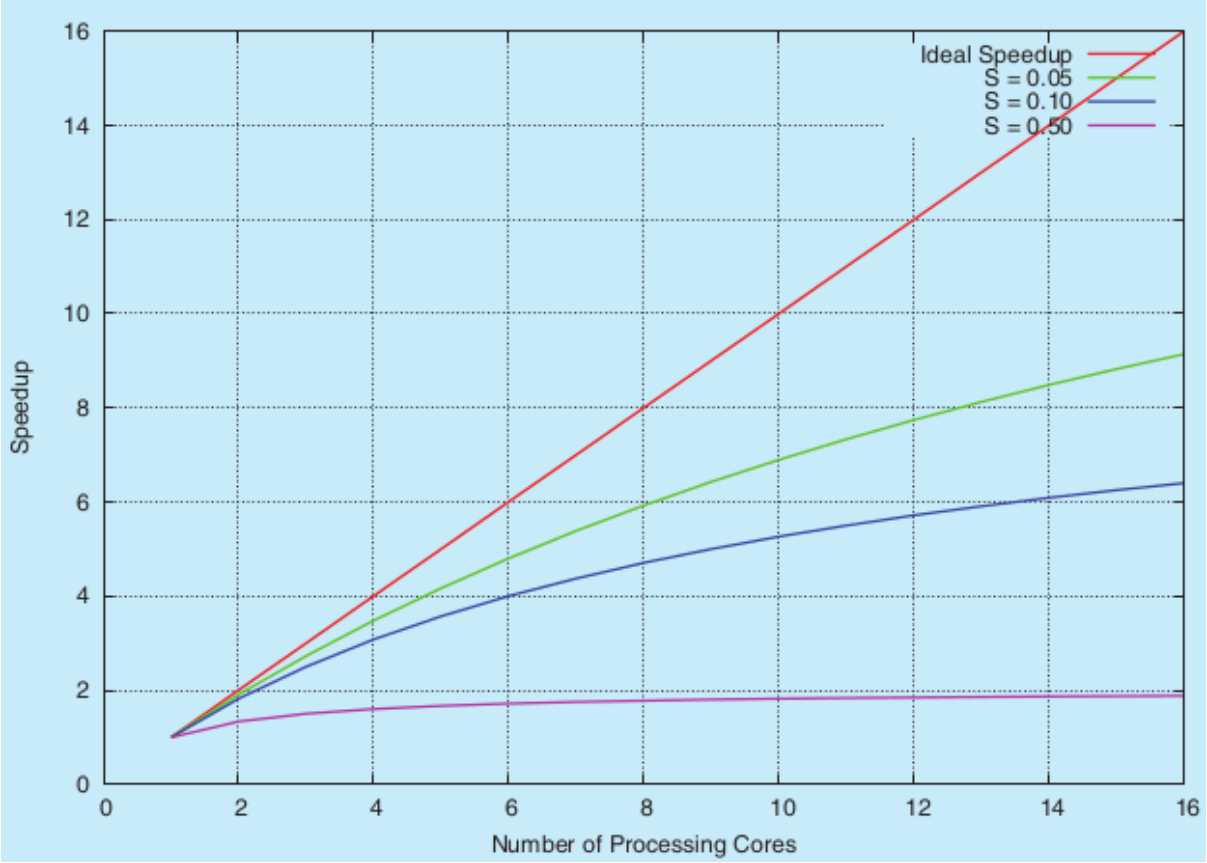

Figura 5. Amdahl-speedup.png [1]. Disponível em https://drive.google.com/file/d/1n_URvuxE81Y3I42QF2EQ_MWxftcYUv_L/view?usp=drive_link.

Exemplo: suponha que tenhamos um aplicativo que é 75% paralelo e 25% serial. Se executarmos este aplicativo em um sistema com dois núcleos de processamento, podemos obter um aumento de velocidade de 1,6 vezes. Se ele tiver quatro núcleos, o *speedup* será de 2,2857 vezes. Os cálculos a seguir mostram como chegar a esses valores:

Sistema com dois núcleos:

$S = \frac{1}{4}$

$N = 2$

$$speedup \leq \frac{1}{(\frac{1}{4} + (\frac{1-\frac{1}{4}}{2}))}$$

$$speedup \leq \frac{1}{(\frac{1}{4} + (\frac{\frac{3}{4}}{2}))}$$

$$speedup \leq \frac{1}{(\frac{1}{4} + \frac{3}{8})}$$

$$speedup \leq \frac{1}{(\frac{5}{8})}$$

$$speedup \leq \frac{8}{5}$$

$$speedup \leq 1,6$$

Sistema com quatro núcleos:

$S = \frac{1}{4}$

$N = 4$

$$speedup \leq \frac{1}{(\frac{1}{4} + (\frac{1-\frac{1}{4}}{4}))}$$

$$speedup \leq \frac{1}{(\frac{1}{4} + (\frac{\frac{3}{4}}{4}))}$$

$$speedup \leq \frac{1}{(\frac{1}{4} + \frac{3}{16})}$$

$$speedup \leq \frac{1}{(\frac{7}{16})}$$

$$speedup \leq \frac{16}{7}$$

$$speedup \leq 2,285714...$$

Conforme *N* se aproxima do infinito, o aumento de velocidade converge para $\frac{1}{S}$. Por exemplo, se 50% de um aplicativo for executado em série, o aumento máximo de velocidade será 2,0 vezes, independentemente do número de núcleos de processamento que adicionarmos. Este é o princípio fundamental por trás da Lei de Amdahl: a porção serial de um aplicativo pode ter um efeito desproporcional no desempenho que ganhamos adicionando núcleos de computação adicionais.

Sabendo que $S = 50\% = \frac{1}{2}$, temos, portanto, que

$$\frac{1}{S} = \frac{1}{(\frac{1}{2})} = \frac{2}{1} = 2$$

Tipos de Paralelismo

Paralelismo de dados: concentrado na distribuição de subconjuntos dos mesmos dados em vários núcleos e na execução da mesma operação em cada núcleo.

- Exemplo: a soma do conteúdo de uma matriz de tamanho *n*.
    - Em um sistema de um núcleo, uma *thread* somaria os elementos *0*, *1*, ... *n−1*.
    - Em um sistema *dual core*, a *thread 0*, rodando no núcleo *0*, pode somar os elementos *0*, *1*, ... *n⁄2−1*, enquanto a *thread 1*, rodando no núcleo *1*, poderia somar os elementos *n⁄2*, *(n/2)+1*... *n−1*.

Paralelismo de tarefas: envolve a distribuição de tarefas (*threads*) em vários núcleos de computação. Cada *thread* está executando uma operação única. Diferentes *threads* podem estar operando nos mesmos dados, ou podem estar operando em dados diferentes.

- Considerando o mesmo exemplo (a soma do conteúdo de uma matriz de tamanho *n*), o paralelismo de tarefas pode envolver duas *threads*, cada um executando uma operação estatística exclusiva na matriz de elementos.

Pode haver aproveitamento das duas estratégias por parte das aplicações.

Modelos *Multithreading*

Os modelos *multithreading* são estratégias de implementação de programação concorrente que permitem a execução simultânea de múltiplas *threads* dentro de um único processo. Essas *threads* compartilham recursos como memória e arquivos, o que pode levar a melhorias de desempenho e eficiência em sistemas modernos com múltiplos núcleos de processamento.

O suporte para *threads* pode ser fornecido no nível do usuário, para *threads* do usuário (*user threads*), ou pelo *kernel*, para *threads* do *kernel* (*kernel threads* ou *threads do sistema*).

Threads do Usuário

- São suportadas acima do *kernel*

- São gerenciadas sem o suporte do *kernel*

Threads do *Kernel*

- São suportadas e gerenciadas diretamente pelo sistema operacional.
- Também são chamadas de *threads* de sistema ou de LWP (*Light Weight Processes* – LWP, ou *processos leves*) [3].

Praticamente todos os sistemas operacionais contemporâneos (Windows, Linux, macOS etc) suportam *threads* de *kernel*.

Modelos de Mapeamento

Deve existir um relacionamento entre as *threads* do usuário e as *threads* do *kernel*. Considerando os dois níveis de *threads*, são possíveis diferentes formas de mapeamento das *threads* do usuário em *threads* do sistema. Os modelos utilizados para estabelecer esse relacionamento são o *muitos-para-um*, *um-para-um*, *muitos-para-muitos* e *dois níveis*, como mostra a Figura 6.

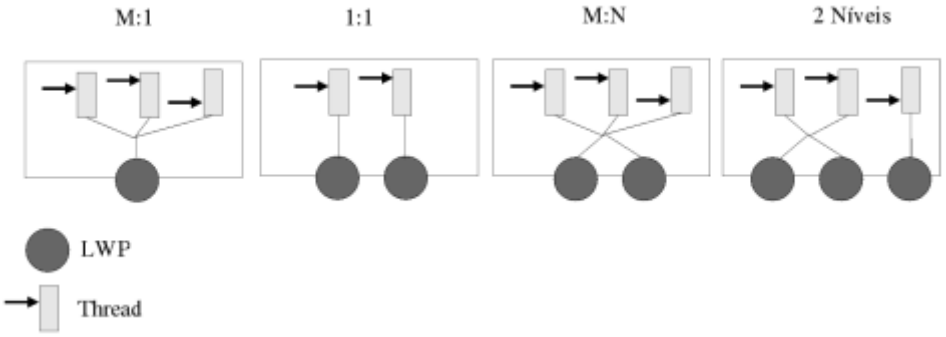

Figura 6. Modelos-de-mapeamento.png [3].

Muitos-para-um, ou *many-to-one* (Mx1): modelo em que muitas *threads* de usuário são mapeadas para uma única *thread* do sistema operacional (*kernel thread*). Isso é gerenciado pela biblioteca de *threads* em nível de usuário, sem a interferência direta do sistema operacional. Embora seja simples de implementar, ele não oferece paralelismo real, pois apenas uma *thread* é executada em um núcleo por vez.

Um-para-um, ou *one-to-one* (1x1): modelo em que cada *thread* de usuário é associada a uma *thread* do sistema operacional individual. Isso permite um verdadeiro paralelismo, pois várias *threads* podem ser executadas em núcleos diferentes simultaneamente. No entanto,

criar muitas *threads* nesse modelo pode sobrecarregar o sistema com a criação e gerenciamento excessivo de *threads* do sistema operacional.

Muitos-para-muitos, ou *many-to-many* (MxN): modelo que combina as vantagens dos dois modelos anteriores. Ele permite que muitas *threads* de usuário sejam associadas a um número menor de *threads* do sistema operacional. Isso evita a sobrecarga de criar muitas *threads* do sistema operacional, mas ainda permite um paralelismo eficiente.

Dois níveis: mapeamento MxN, como o modelo anterior, mas com a possibilidade de especificar que algumas *threads* têm mapeamento 1x1 com LWP [3].

Cada modelo tem suas vantagens e desvantagens, e a escolha depende das necessidades específicas da aplicação e das características do sistema. A implementação de *threads* pode ser feita por meio de bibliotecas de *threads* em nível de usuário (como a `pthreads` em C) ou por meio de suporte direto do sistema operacional (como as *threads* no *kernel*).

Threading Síncrono e Assíncrono

Existem duas estratégias gerais para criar várias *threads*: *assíncrona* e *síncrona*.

- Com ***threading* síncrono**, que ocorre quando a *thread* ancestral cria uma ou mais filhas e, em seguida, deve esperar que todas as suas filhas sejam encerradas antes de continuar. Aqui, as filhas criadas pela *thread* ancestral executam o trabalho simultaneamente, mas a ancestral não pode continuar até que esse trabalho seja concluído. Depois que cada *thread* termina seu trabalho, ela termina e se junta a sua ancestral. Somente depois que todas as filhas tiverem entrado, a *thread* ancestral poderá retomar a execução. Normalmente, o *threading* síncrono envolve compartilhamento significativo de dados entre as *threads*. Por exemplo, a *thread* pai pode combinar os resultados calculados por suas várias filhas.
- Com ***threading* assíncrono**, uma vez que a *thread* ancestral cria uma *thread* filha, a ancestral retoma sua execução, de modo que ambas sejam executadas simultaneamente e independentemente uma da outra. Como as *threads* são independentes, geralmente há pouco compartilhamento de dados entre elas.

Biblioteca `pthread`

Uma biblioteca de *threads* fornece ao programador uma API para criar e gerenciar *threads*. Há duas maneiras principais de implementar uma biblioteca de *threads*:

1. Inteiramente no espaço do usuário, sem suporte de *kernel*.
2. De nível de *kernel*, suportada diretamente pelo sistema operacional.

Sistemas Unix-like, Linux e macOS geralmente usam Pthreads.

A biblioteca `pthread` fornece várias funções para trabalhar com *threads* em um programa C. Ela é uma implementação da especificação `pthreads`, o padrão POSIX (*Portable Operating System Interface*) que define uma API para criação e sincronização de *threads*.

Pode ser fornecida como uma biblioteca de nível de usuário ou de nível de *kernel*.

A sigla "`pthread`" significa *POSIX Threads*, onde *POSIX* refere-se ao padrão IEEE para interfaces de sistema operacional.

Principais funções da biblioteca `pthread`

- **`pthread_create`**: usada para criar uma nova *thread*. Ela recebe quatro argumentos: um ponteiro para uma variável de tipo `pthread_t` que conterá o identificador da *thread* criada, um ponteiro para um objeto de tipo `pthread_attr_t` que contém atributos de configuração da *thread* (normalmente definido como `NULL` para atributos padrão), uma função que será executada pela *thread* e um ponteiro para argumentos que serão passados para a função da *thread*.
- **`pthread_join`**: usada para esperar até que uma *thread* específica termine sua execução. Ela recebe como argumentos o identificador da *thread* a ser aguardada e um ponteiro para onde o valor de retorno da *thread* será armazenado (se a *thread* retornar um valor). Isso é útil para garantir que a *thread* principal do programa não termine antes das outras *threads*.
- **`pthread_exit`**: termina a execução de uma *thread*. Quando chamada, a *thread* atual é encerrada, e qualquer valor de retorno pode ser passado para a função `pthread_join` para que a *thread* principal possa obter esse valor.
- **`pthread_mutex_init`**: usada para inicializar um objeto de tipo `pthread_mutex_t`, que é usado para criar um mecanismo de exclusão mútua. Os

mutexes são usados para proteger seções críticas de código compartilhadas por várias *threads*, permitindo que apenas uma *thread* por vez acesse essas seções.

- **`pthread_mutex_lock` e `pthread_mutex_unlock`**: Essas funções são usadas para bloquear e desbloquear um *mutex*, respectivamente. Ao chamar `pthread_mutex_lock`, a *thread* aguarda até que o mutex esteja disponível para bloqueio e, quando chamado `pthread_mutex_unlock`, a *thread* libera o *mutex* para que outras *threads* possam acessar a seção crítica protegida por ele.
- **`pthread_cond_init`**: Esta função é usada para inicializar um objeto de tipo `pthread_cond_t`, que representa uma variável de condição. Variáveis de condição são usadas em conjunto com mutexes para permitir que as *threads* esperem por uma condição específica antes de continuar sua execução.
- **`pthread_cond_wait` e `pthread_cond_signal`**: Essas funções são usadas em conjunto com um *mutex* e uma variável de condição. A `pthread_cond_wait` é chamada por uma *thread* que deseja esperar por uma determinada condição, e ela libera o mutex enquanto aguarda. A `pthread_cond_signal` é usada para sinalizar a uma ou mais *threads* que estão esperando por essa condição, e assim elas podem retomar sua execução.

Usando *Threads* em C

Para usar a biblioteca `pthread`, o código precisa obrigatoriamente conter o arquivo de cabeçalho desta biblioteca, `pthread.h` (incluído com `#include <pthread.h>`).

`pthread.h` é um arquivo de cabeçalho em C que fornece a API para trabalhar com *threads* em sistemas operacionais compatíveis com POSIX.

Para compilar um código C utilizando a biblioteca `pthread`, é necessário acrescentar o parâmetro `-lpthread`, que realiza a linkedição (ligação) do código com a biblioteca, como no exemplo abaixo:

```
> gcc -o meuPrograma meuPrograma.c -lpthread
```

Um primeiro exemplo está disponível em `thread-exemplo-1.c`. Note que o código

- cria uma *thread* filha (pois já há uma *thread* de controle implícita em `main()`) e conta os números de 1 até 10.
- usa `pthread_create` para criar a *thread* filha e indica no terceiro parâmetro que `countNumbers` será a função por ela realizada.

- necessita de `pthread_join`, assegurando que a tarefa será concluída. Uma boa forma de entender isso é executar o código original e depois recompilá-lo comentando as linhas 26-31. Observe que sem `pthread_join` a função `countNumbers` não é executada.

O código `thread-exemplo-2.c` calcula, usando *threads*, o somatório de *1* até *n*, sendo esperado no lugar de *n* um número natural, informado pelo usuário na linha de comando (parâmetro `argv`).

- Quando o programa começa, uma única *thread* de controle começa em `main()`.
- Em seguida, `main()` cria uma segunda *thread* que inicia o controle na função `runner()`.
- Ambas as *threads* compartilham a soma global de dados.
- A instrução `pthread_t tid` declara o identificador para a *thread* a ser criada.
- Cada *thread* possui um conjunto de atributos, incluindo tamanho da pilha e informações de agendamento.
- A declaração `pthread_attr_t attr` representa os atributos para a *thread*.
- Os atributos são definidos na chamada de função `pthread_attr_init(&attr)`.
- Uma *thread* separada é criada com a chamada de função `pthread_create()`.
- Além de passar o identificador de *thread* e os atributos para a *thread*, passamos o nome da função onde a nova *thread* iniciará a execução: a função `runner()`.
- Por último, passamos o parâmetro inteiro (`n`) fornecido na linha de comando, `argv[1]`.
- Neste ponto, o programa tem duas *threads*:
    1. a *thread* inicial (ou pai) em `main()`; e
    2. a *thread* de soma (ou filha) executando a operação de soma na função `runner()`.
- Esse programa segue a estratégia de criação/junção de *threads*, segundo a qual, depois de criar a *thread* de soma, a *thread* pai aguardará que ela termine chamando a função `pthread_join()`.
- A *thread* de soma terminará quando chamar a função `pthread_exit()`.
- Depois que a *thread* de soma tiver retornado, a *thread* pai produzirá o valor da soma de dados compartilhados.

Os exemplos mostrados utilizam a abordagem de *threading* síncrono ou assíncrono?

Observe agora o código `thread-exemplo-3.c`. Execute-o da mesma forma que o código anterior.

- Neste código, foram criadas duas *threads* filhas.
- As *threads* filhas executam de forma assíncrona.
- Elas são criadas para executar paralelamente e independentemente uma da outra.
- Cada *thread* filha calcula sua própria soma parcial, e a soma total é calculada apenas após ambas as *threads* filhas terem concluído suas operações.
- No entanto, a *thread* pai está esperando pelas *threads* filhas usando a função `pthread_join`, o que significa que a execução da *thread* pai será bloqueada até que as *threads* filhas terminem.
- Isso garante que o programa terá os resultados corretos das *threads* filhas antes de calcular a soma total.

Threading Implícita

O gerenciamento de *threads* dos desenvolvedores de aplicativos pode ser feito por compiladores e bibliotecas de tempo de execução. Essa estratégia é denominada *threading implícita* (*segmentação implícita* ou *encadeamento implícito*), uma tendência cada vez mais popular.

Geralmente exigem que os desenvolvedores de aplicativos identifiquem tarefas — não *threads* — que podem ser executadas em paralelo.

Algumas abordagens de projeto de aplicações utilizando *threading* implícita são: *pool* de *threads*, *fork join*, OpenMP, GCD (*Grand Central Dispatch*) e *Intel Thread Building Blocks*. As três primeiras estão descritas na sequência.

Pool de Threads

- Define um número de *threads*, inicializa elas e as deixa de prontidão.
- Quando uma solicitação é recebida, esta é encaminhada ao *pool* de *threads* e duas situações podem acontecer:

1) Se há *thread* disponível, ela é ativada e a solicitação é processada;
2) Se não há *thread* disponível, a tarefa vai para uma fila até que haja uma *thread* livre.

- Depois que uma *thread* conclui seu serviço, ela retorna ao *pool* e aguarda mais tarefas.

- Evita problemas de falta de limite na criação de *threads*. Criar *threads* ilimitadamente pode esgotar os recursos do sistema (CPU, memória etc).
- Diminui o custo de criação e destruição de *threads* durante a execução do programa.
- Útil em sistemas que não podem suportar um grande número de *threads* simultâneas.
- Funcionam bem para tarefas assíncronas.

Fork Join

- *Fork* = bifurcar; *Join* = juntar
- A estratégia exibida nos exemplos iniciais.
- Com esse método, a *thread* ancestral principal cria (bifurca) uma ou mais *threads* filhas e então espera que as filhas terminem e se juntem a ela, momento em que pode recuperar e combinar seus resultados (Figura 7).
- Esse modelo síncrono geralmente é caracterizado como criação de *thread* explícita, mas também é um excelente candidato para *thread* implícita.
- Na situação implícita, as *threads* não são construídas diretamente durante o estágio de bifurcação; em vez disso, são designadas tarefas paralelas.
- Uma biblioteca gerencia o número de *threads* criadas e é responsável por atribuir tarefas a elas.
- Pode ser visto como uma versão síncrona de pool de *threads* em que uma biblioteca determina o número real de *threads* a serem criadas.

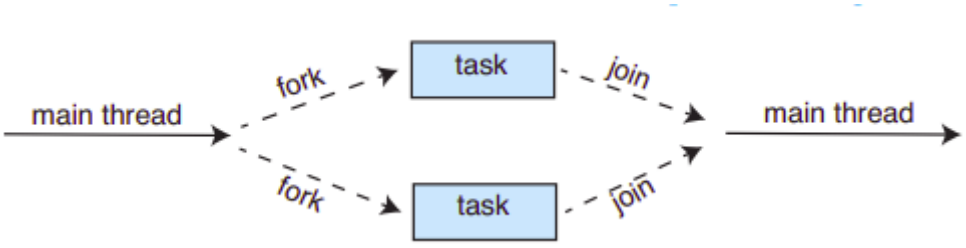

Figura 7. Fork-join.png [1]. Disponível em https://drive.google.com/file/d/1S0KdHppd1hUeRaQrjWi627SSSGqS-xJV/view?usp=drive_link.

OpenMP (*Open Multi-Processing*)

OpenMP é um conjunto de diretivas de compilador, bem como uma API para programas escritos em C, C++ ou FORTRAN que fornece suporte para programação paralela em ambientes de memória compartilhada.

- Identifica regiões paralelas como blocos de código que podem ser executados em paralelo.
- Os desenvolvedores de aplicativos inserem diretivas de compilador em seu código em regiões paralelas, e essas diretivas instruem a biblioteca de tempo de execução OpenMP a executar a região em paralelo.

Exemplos:

- `openMP-exemplo-1.c` (adaptado de [1]); e
- `openMP-exemplo-2.c`.

Para compilar um código C que utilize OpenMP, é necessário utilizar o parâmetro `-fopenmp`:

```
> gcc arquivo.c -fopenmp
```

No primeiro exemplo, quando o OpenMP encontra a diretiva `#pragma omp parallel`, ele cria tantas *threads* quantos núcleos de processamento houver no sistema. Ou seja, num sistema *dual core* são criadas duas *threads*; num *quad core*, quatro; e assim por diante.

Para o processador Intel® Core™ i3-2310M CPU @ 2.10GHz × 4, por exemplo, a mensagem foi exibida quatro vezes, pois ele possui 2 núcleos físicos e suporta a tecnologia HT (*Hyper-Threading*). Isso significa que ele é capaz de executar duas *threads* simultaneamente por núcleo, totalizando quatro *threads*.

Todas as *threads* executam simultaneamente a região paralela. À medida que cada *thread* sai da região paralela, ela é encerrada.

No segundo exemplo, a diretiva contém mais especificadores. A diretiva completa é `#pragma omp parallel for reduction(*: factorial)`

Ela indica ao compilador que o *loop* `for` subsequente pode ser executado em paralelo por várias *threads*.

- **`#pragma omp`**: início de uma diretiva do OpenMP.
- **`parallel`**: indica que o bloco de código seguinte pode ser executado em paralelo por várias *threads*.
- **`for`**: indica que a próxima instrução `for` é candidata a paralelização.

- **`reduction(*: factorial)`**: especifica uma operação de redução que será realizada nas variáveis compartilhadas após o término do bloco paralelo. Neste caso, `factorial` é a variável que será reduzida, e `*` indica a operação realizada (multiplicação).

A operação de redução garante que os resultados das diferentes *threads* sejam combinados de forma correta ao final do bloco paralelo.

A OpenMP permite que os desenvolvedores escolham entre vários níveis de paralelismo, podendo definir o número de *threads* manualmente. Ele também permite que os desenvolvedores identifiquem se os dados são compartilhados entre *threads* ou são privados de uma *thread*.

Problemas no Uso de *Threads*

A implementação de programas com múltiplas *threads* traz ainda alguns problemas que requerem alguns cuidados.

Um dos problemas é a semântica da função `fork()` quando o programa é *multithreaded*. Se uma *thread* chama a função `fork()`, o processo criado duplica todas as *threads* ou apenas a *thread* que chamou a função? Alguns sistemas UNIX escolheram ter duas versões de `fork()`, uma que duplica todas as *threads* e outra que duplica apenas a *thread* que invocou a chamada de sistema `fork()` [1].

Outro problema é o cancelamento de uma *thread*. Por exemplo, quando um usuário para o carregamento de uma página web ao clicar no botão “parar” do navegador. Este cancelamento é *assíncrono*, quando uma *thread* para imediatamente, ou *síncrono*, quando ela para de forma controlada. O cancelamento assíncrono libera imediatamente os recursos alocados para a *thread*, mas não todos. Já no cancelamento síncrono (ou *adiado*) a *thread*-alvo verifica um sinalizador para determinar se deve ser cancelada ou não. Isso permite que a *thread* seja cancelada em um ponto em que possa ser cancelada com segurança. Os `pthreads` referem-se a esses pontos como *pontos de cancelamento* (*cancellation points*).

O terceiro problema abordado neste material é o de entrega de sinais. Um sinal é usado nos sistemas Unix-like para informar a um processo a ocorrência de um evento, que pode ser o

redimensionamento da janela, o encerramento do programa teclando CTRL+C, um acesso indevido à memória, entre outros. Se o programa tem múltiplas *threads*, a qual delas o sinal deve ser entregue? A resposta dessa questão sempre dependerá do tipo de sinal, pois alguns precisam ser enviados a *threads* específicas às quais o sinal é aplicável, outros serão entregues a todas as *threads*, outras serão a algumas *threads*, e outras serão entregues a uma *thread* específica destinada ao recebimento de sinais. A função padrão do UNIX para a entrega de um sinal é `kill(pid_t pid, int signal)`. Pthreads POSIX também fornecem a função `pthread_kill(pthread_t tid, int signal)`, que permite que um sinal seja entregue a uma *thread* especificada.

Escalonamento de *Threads*

Nos sistemas operacionais contemporâneos, são as *threads* de *kernel* – e não os processos – que são escalonados. As *threads* de usuário são gerenciadas por uma biblioteca de *threads*, e o *kernel* não as conhece. Para executar em uma CPU, as *threads* no nível do usuário precisam ser mapeadas a uma *thread* no nível do *kernel*, embora esse mapeamento possa ser indireto e utilizar um processo leve (LWP).

Em sistemas que implementam os modelos *muitos para um* e *muitos para muitos*, a biblioteca de *threads* escalona as *threads* no nível do usuário para executarem em um LWP disponível, um esquema conhecido como *escopo de disputa do processo* (*Process Contention Scope* – PCS), pois a disputa pela CPU ocorre entre as *threads* pertencentes ao mesmo processo.

Para decidir qual *thread* do *kernel* será escalonada em uma CPU, o *kernel* utiliza o *escopo de disputa do sistema* (*System-Contention Scope* – SCS). A disputa pela CPU com escalonamento SCS ocorre entre todas as *threads* no sistema. Os sistemas usando o modelo um para um , como Windows e Linux, escalonam as threads usando apenas o SCS.

Tipicamente, o PCS é feito de acordo com a prioridade – o escalonador seleciona a *thread* executável com a maior prioridade para execução. As prioridades da *thread* do usuário são definidas pelo programador, embora algumas bibliotecas de *threads* possam permitir que o programador mude sua prioridade. É importante observar que o PCS normalmente interromperá a *thread* sendo executada em favor de uma *thread* com maior prioridade.

A biblioteca `pthread` fornece funções get e set para obter e definir, respectivamente, a política de escopo de disputa:

- `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`
- `pthread_attr_setscope(pthread_attr_t *attr, int scope)`

O primeiro parâmetro para as duas funções contém um ponteiro para o conjunto de atributos para a thread. O segundo parâmetro para a função `pthread_attr_setscope()` recebe o valor `PTHREAD_SCOPE_SYSTEM` ou `PTHREAD_SCOPE_PROCESS`, indicando como o escopo de disputa deve ser definido. Para `pthread_attr_getscope()`, esse segundo parâmetro contém um ponteiro para um inteiro que é definido como o valor atual do escopo de disputa. Se houver um erro, essas funções retornam valores diferentes de zero.

Em alguns sistemas, somente determinados valores de escopo de disputa são permitidos. Por exemplo, sistemas Linux e Mac OS X permitem apenas `PTHREAD_SCOPE_SYSTEM`. O exemplo de código `threads-escopo.c` (baseado em [1]) obtém o escopo padrão e cria *threads* que serão executadas seguindo a política de escalonamento obtida. Tente alterar a política de escopo de disputa antes de criar as *threads* e verifique se esta alteração é permitida.

Resumo

Nesta aula, introduziu-se o conceito de *threads* e explorou-se a programação concorrente com o uso da linguagem C, seguindo os fundamentos apresentados na aula anterior sobre Comunicação entre Processos e os mecanismos de *pipe* e *sockets*.

As *threads* são unidades básicas de utilização da CPU, compartilhando seções de código, dados e recursos do sistema operacional entre si. Elas permitem a execução simultânea de tarefas em um processo e são essenciais em sistemas multicore. Exemplos de aplicações *multithreaded* são apresentados, como aplicativos de criação de figurinhas, navegadores web e processadores de texto, demonstrando os benefícios da programação *multithread*, como capacidade de resposta, compartilhamento de recursos, economia e escalabilidade.

A programação *multithread* é especialmente relevante em sistemas *multicore*, onde as *threads* podem ser executadas em paralelo em diferentes núcleos de processamento. Modelos de paralelismo, como paralelismo de dados e de tarefas, são discutidos, e a biblioteca `pthread` em C é apresentada como uma ferramenta para trabalhar com *threads*.

A biblioteca fornece funções para criação, gerenciamento e sincronização de *threads*, permitindo a criação de programas *multithread*.

Também abordou-se a Lei de Amdahl, que formula os possíveis ganhos de desempenho na execução de aplicações com blocos sequenciais e potencialmente paralelos ao adicionar núcleos de computação na máquina utilizada.

Os modelos *multithreading*, incluindo muitos-para-um, um-para-um, muitos-para-muitos e dois níveis foram explicados, assim como suas vantagens e desvantagens. O texto também aborda a *threading* implícita, onde o gerenciamento de *threads* é feito por compiladores e bibliotecas de tempo de execução, incluindo abordagens como *Pool* de *Threads*, *Fork Join* e OpenMP.

O texto trouxe ainda algumas dificuldades da programação *multithreading*, tais como o uso de `fork()`, o cancelamento de *threads* e o tratamento de sinais, e conceitos sobre o escalonamento de *threads*, explicando escopos de disputa e o escalonamento com `pthreads`.

Exercícios

1) Suponha que um programa seja 30% não paralelo e calcule o seu *speedup* considerando que ele será executado em um sistema com

a) dois núcleos de processamento.

b) quatro núcleos de processamento.

c) oito núcleos de processamento.

2) (adaptado de POSCOMP 2017, Questão 50) Analise o código a seguir:

```
(...)
void thread (void *ptr) {
     while(1);
}
int main() {
     int i; pthread_t tid[10];
     for (i = 0; i < 10; i++)
          pthread_create (&tid[i], NULL, (void *) thread, NULL);
     getchar();
```

```
}
```

Ao executar esse programa, o processo criado possuirá quantos fluxos de execução (*threads*) no instante em que finalizar o laço `for(;;)`?

a) Um
b) Dois.
c) Nove.
d) Dez.
e) Onze.

3) (UFRRJ 2023 - Analista de TI) Processos e *threads* são unidades de execução de tarefas dentro do sistema operacional. Sobre as características das *threads*, é correto afirmar que

a) o escalonador do sistema operacional conhece e gerencia todas as *threads* de usuário.
b) as trocas de contexto das *threads* de usuário são executadas pelo sistema operacional sem o conhecimento do aplicativo.
c) as trocas de contexto das *threads* de núcleo são executadas por uma biblioteca de aplicativos, sem o conhecimento do sistema operacional.
d) as *threads* de um mesmo processo compartilham o mesmo espaço de endereçamento de memória, arquivos abertos e outros recursos que compõem o contexto global do processo.
e) as *threads* de um mesmo processo compartilham o mesmo contexto de execução, o qual inclui o identificador único da *thread*, o estado da pilha de execução e os valores no apontador de instrução e nos registradores de uso geral.

4) Utilizando `pthread` e criando somente uma *thread* filha, elabore um código que receba uma quantidade aleatória de números inteiros, filtre e exiba apenas os números pares.

5) Para tentar aprimorar seus conhecimentos sobre *threads* e a biblioteca `pthread`, um estudante do IFC tentou criar um código simples que, utilizando quatro *threads*, deveria realizar 100 iterações de incremento a uma variável compartilhada. O código está disponível no arquivo `pthread-exercicio-5-aula-4.c`.

   Porém, ao executar o código cinco vezes, as respostas que ele obteve foram:

   Execução #1:

   ```
   Valor final da variável compartilhada: 400
   ```

Execução #2:

```
Valor final da variável compartilhada: 371
```

Execução #3:

```
Valor final da variável compartilhada: 400
```

Execução #4:

```
Valor final da variável compartilhada: 400
```

Execução #5:

```
Valor final da variável compartilhada: 339
```

5.1) Qual era o valor final esperado para a variável compartilhada?

5.2) Por que há resultados diferentes?

5.3) O que pode ser feito para contornar essa situação?

6) Utilizando OpenMP, crie um código que imprima lado a lado 100 diferentes números primos. Eles não precisam estar em ordem. **Dica**: verifique a diretiva `critical`[1] [2].

7) Leia o capítulo 6 (*Sincronização de processos*) de [1].

Gabarito

1) a) 1,53846154; b) 2,10526316; c) 2,58064516
2) e (a *thread* ancestral e dez filhas)
3) d
4) Uma possível resposta está disponível nos arquivos `geraAleatorios.sh` e `pthread-exercicio-4-aula-4.c`.

   Execute com:

```
> gcc pthread-exercicio-4-aula-4.c -lpthread; ./geraAleatórios.sh 1000 | ./a.out
```

   Ou passo a passo criando o arquivo intermediário:

```
> gcc pthread-exercicio-4-aula-4.c -lpthread
> ./geraAleatórios.sh 1000 > i.input
> ./a.out < i.input
```

5)

[1] Usada para definir uma seção crítica do código, ou seja, um código que deve ser executado por apenas uma *thread* por vez.

5.1) Os resultados estão condizentes com o que foi implementado, pois não há mecanismos de sincronização para a seção crítica do código, que é o incremento da variável compartilhada. Se a seção crítica fosse adequadamente tratada usando um mecanismo de sincronização, o valor final esperado para a variável `variavelCompartilhada` deveria ser igual a `NUM_THREADS * NUM_ITERATIONS`, ou seja, `4 * 100 = 400`. Se o objetivo era, de fato, não sincronizar o acesso à seção crítica, qualquer resultado até 400 poderia ser esperado.

5.2) Porque a seção crítica não foi tratada, o que permite condições de corrida. Neste caso, cada *thread* executaria a sequência de código que incrementa a variável da seguinte forma:

- A *thread* lê o valor atual da variável compartilhada.
- A *thread* incrementa o valor lido.
- A *thread* escreve o novo valor de volta à variável compartilhada.

Se duas ou mais *threads* executarem essas etapas ao mesmo tempo, elas podem ler o mesmo valor original antes de qualquer incremento e, depois, todas incrementarão o valor simultaneamente e escreverão de volta, o que resultará em apenas um incremento efetivo. Isso levará a um resultado incorreto, onde várias operações de incremento não foram consideradas.

5.3) É necessário utilizar algum mecanismo de sincronização. Exemplos: mutex, semáforo, barreira, monitor, sinalizador, variáveis condicionais etc.

6) Resposta no arquivo `openMP-exercicio-aula-4.c`.

7) Livre.

Referências

[1] SILBERSCHATZ, A., GALVIN, P. B., & GAGNE, G. (2018). Operating System Concepts (10ª ed.). Wiley.

[2] OPENMP. OpenMP 5.2 API Syntax Reference Guide. Disponível em: https://www.openmp.org/wp-content/uploads/OpenMPRefCard-5-2-web.pdf. Acesso em: 18 ago. 2023.

[3] PILLA, M. L. Material didático. Processos e Threads. Versão 2010-09-01.