Processos e tarefas em linguagem C

Prof. Fabio Henrique Silva

Apresentação

Você aprenderá a criar e a gerenciar processos e threads para desenvolver aplicações eficientes e concorrentes. Exploraremos a função fork para criação de processos e utilizaremos a biblioteca Pthreads para implementação e sincronização de threads. O conhecimento teórico auxilia a compreender conceitos, evitar condições de corrida e deadlocks, e garantir a execução correta de operações simultâneas.

Propósito

Preparação

Antes de iniciar seus estudos, faça o downloand do passo a passo de acesso do ambiente Linux Ubuntu com o compilador GCC.

Objetivos

Módulo 1

Criação de processos com fork

Reconhecer as formas de programação paralela com a utilização de processos-filho com a função fork.

Módulo 2

Funções para threads básicos

Comparar a execução de funções para tarefas em paralelo (threads) e a criação de processos em paralelo (forks).

Módulo 3

Sincronização de threads

Identificar as questões de sincronização de threads para evitar a inconsistência de dados nas aplicações com tarefas concorrentes.

Introdução

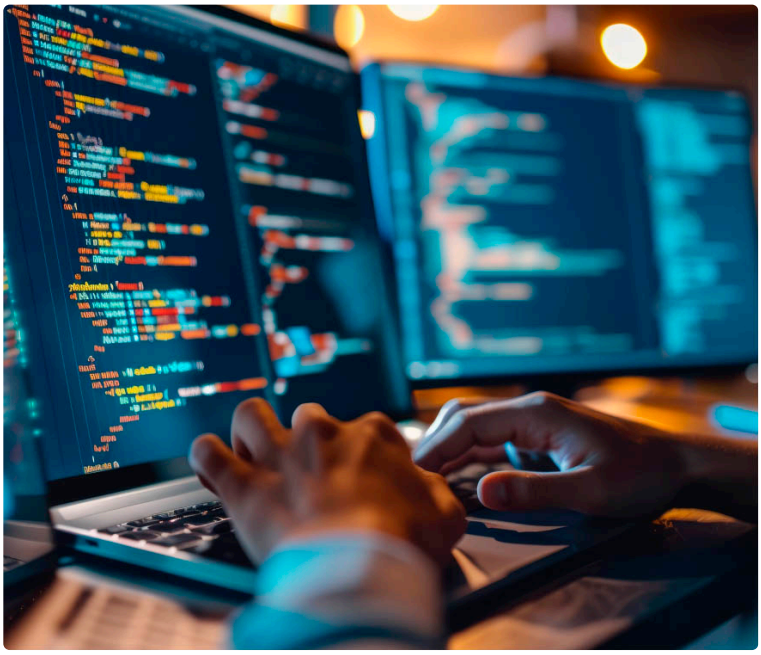
Neste vídeo, você vai conhecer alguns assuntos relacionados com entradas e saídas na linguagem C, tais como uso do fork para criação de processos, execução e sincronização de threads.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Material para download

Clique no botão abaixo para fazer o download do conteúdo completo em formato PDF.

Download material



1 - Criação de processos com fork

Neste módulo, você será capaz de reconhecer as formas de programação paralela com a utilização de processos-filho com a função fork.

O conceito de operação com processos

Um processo é um programa em execução que passa por vários estados, como **pronto, em execução e aguardando**, gerenciados pelo Bloco de Controle de Processo (PCB). Esse controle organiza e executa eficientemente as tarefas no sistema operacional.

Neste vídeo, você vai entender como processos operam de maneira independente e como se comunicam no sistema operacional.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Processo

É como um programa que está sendo executado, seguindo uma ordem sequencial de instruções. Todo programa executado se transforma em processo, realizando suas tarefas.

Quando um programa é carregado na memória para se tornar um processo, ele é segmentado em quatro partes: Pilha, Heap, Dados e Texto. A seguir, você vai conhecer cada um desses segmentos.

Pilha

Armazena dados temporários, como parâmetros de função, endereços de retorno e variáveis locais.

Heap

Memória utilizada dinamicamente durante a execução do processo.

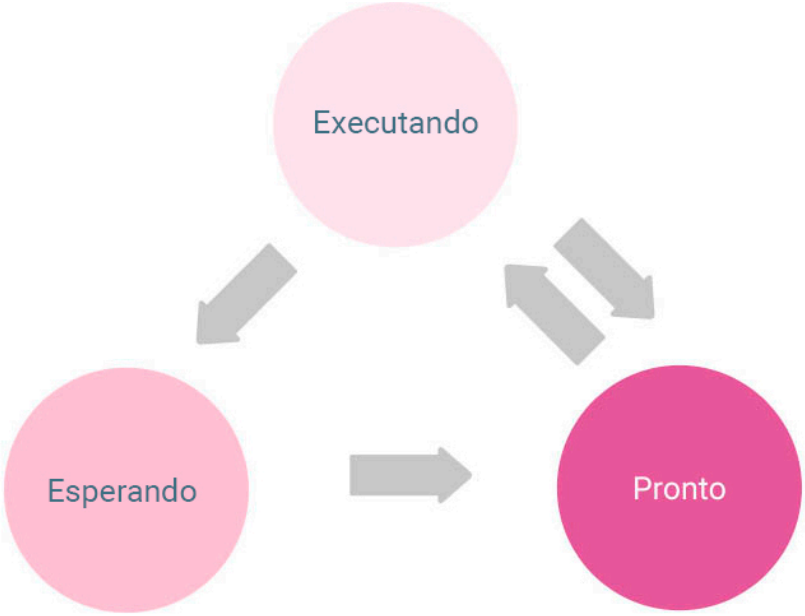
Dados

Contêm variáveis globais e estáticas.

Texto

Representa a instrução atual, incluindo o valor do contador do programa e os registros do processador.

Um **programa** é um conjunto de instruções que realizam uma tarefa específica. Um **processo** é uma execução ativa e dinâmica desse programa.



Estados de um processo.

Durante sua execução, um processo transita por diversos estados, os quais podem diferir, dependendo do sistema operacional. Um processo pode estar em um dos cinco estados listados a seguir. Observe!

Iniciado

Estado inicial, quando o processo é criado.

Pronto

Esperando para ser designado a um processador.

Executando

Quando o processador está executando as instruções do processo.

Esperando/Bloqueado

Aguardando algum recurso, como entrada do usuário ou um arquivo.

Terminado

Quando o processo completa sua execução e aguarda ser removido da memória.

O sistema operacional tem uma estrutura de dados denominada **Bloco de Controle de Processo (PCB)**, na qual estão todas as informações necessárias para gerenciar o processo.

A estrutura do PCB varia de acordo com o sistema operacional e é mantida durante todo o ciclo de vida do processo, sendo excluída quando ele termina.

Cada processo é identificado por um ID de processo (PID). A seguir, você vai conhecer alguns elementos do PCB. Vamos lá!

Estado do processo

Situação atual do processo (pronto, executando, esperando etc.).

Privilégios

Permissões de acesso aos recursos do sistema.

ID do processo

Identificação única do processo.

Ponteiro

Aponta para o processo-pai.

Contador de programa

Aponta para o processo-pai.

Registradores de CPU

Registradores necessários para a execução do processo.

Informações de agendamento

Dados sobre a prioridade e outras informações para o agendamento do processo.

Gerenciamento de memória

Dados sobre tabela de páginas, limites de memória e segmentos.

Informações contábeis

Informações sobre uso de CPU, limites de tempo, ID de execução etc.

Status de E/S

Lista de dispositivos de E/S alocados ao processo.

Atividade 1

Em um sistema operacional, a execução de processos é responsável pelo funcionamento correto e eficiente do sistema. Esses processos podem interagir de várias formas, e a maneira como gerenciam os recursos influencia diretamente o desempenho e a estabilidade do sistema.

Considere um cenário em que um sistema bancário tem múltiplos processos, cada um realizando diferentes operações financeiras, como saques em caixas eletrônicos e compensação de cheques.

Com base no conceito de operação com processos, qual das seguintes alternativas a seguir descreve corretamente o uso de recursos por um

processo no sistema operacional?

A Um processo solicita todos os recursos necessários de uma vez e os libera somente ao finalizar sua execução.

B Um processo pode solicitar um recurso, utilizá-lo e liberá-lo, repetindo essas ações conforme necessário durante sua execução.

C Um processo pode solicitar um recurso e continuar executando mesmo sem obtê-lo, liberando-o quando não for mais necessário.

D Um processo sempre tem acesso garantido a todos os recursos necessários, sem a necessidade de solicitar ou liberar explicitamente.

E Um processo utiliza recursos de forma exclusiva e nunca compartilha recursos com outros processos.

Parabéns! A alternativa B está correta.

Essa abordagem promove eficiência e estabilidade no sistema operacional, pois recursos são distribuídos de forma dinâmica, conforme a demanda dos processos. Permite que os recursos sejam compartilhados entre múltiplos processos, otimizando o uso daqueles disponíveis e evitando a espera desnecessária por recursos ocupados por longos períodos, se não forem liberados prontamente.

A função fork

A chamada de sistema fork cria um novo processo, clonando o atual e abrindo os mesmos descritores de arquivo. O **processo-filho** recebe um valor de retorno diferente de fork, permitindo que ele e o processo-pai continuem a execução a partir do mesmo ponto do programa. Esse

exemplo demonstra como criar e executar um novo processo em sistemas Unix.

Assista ao vídeo para aprender a função fork.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Criando um processo

Ao ser utilizada, a **chamada de sistema fork** cria um novo processo. E, quando fork é chamado, o sistema operacional cria um novo processo, atribuindo uma nova entrada na tabela de processos e clonando as informações do processo atual. Todos os descritores de arquivo abertos no processo-pai serão abertos no processo-filho.

A imagem da memória executável também é copiada. Assim que a chamada fork retorna, pai e filho estão executando, no mesmo ponto do programa. A única diferença é que o filho recebe um valor de retorno diferente do fork. O pai obtém o ID do processo-filho recém-criado; o filho recebe um retorno de 0. Se fork retornar -1, o sistema operacional não conseguiu criar o processo.

> Nada é compartilhado após a execução do fork. Mesmo que esteja executando o mesmo código e tenha os mesmos arquivos abertos, o filho mantém seus próprios ponteiros de busca (posições no arquivo) e apresenta sua própria cópia de toda a memória. Se o filho alterar um local de memória, o pai não verá a alteração (e vice-versa).

A seguir, observe um pequeno programa que se clona. O pai imprime uma mensagem informando seu ID de processo e o do filho. Obtém o ID do processo usando a **chamada de sistema getpid** e o ID do processo-filho a partir do retorno do fork. O filho imprime seu ID de processo. O pai e o filho, então, saem.

c

Note que exit leva um parâmetro, que se torna o código de saída do programa. A convenção para sistemas Unix é sair com um código zero em caso de sucesso e diferente de zero em caso de falha, o que auxilia na criação de scripts de programas.

Atividade 2

A função fork é uma chamada de sistema fundamental em sistemas operacionais Unix-like, utilizada para criar novos processos. Quando um processo chama fork, o sistema operacional cria um novo processo-filho, uma cópia quase exata do processo-pai. Esse novo processo herda código, dados e descritores de arquivo do processo-pai, uma funcionalidade que permite executar múltiplas tarefas de forma concorrente.

Considerando a função fork em sistemas operacionais Unix-like, qual das seguintes afirmações a seguir descreve corretamente o comportamento do processo-pai e do processo-filho após a chamada de fork?

A A função fork retorna zero para o processo-pai e um valor positivo (PID do filho) para o processo-filho.

B A função fork retorna o PID do processo-pai para o processo-filho, e o PID do processo-filho para o processo-pai.

C A função fork retorna zero para o processo-filho e um valor positivo (PID do filho) para o processo-pai.

D A função fork retorna zero para ambos, processo-pai e processo-filho.

E A função fork faz com que o processo-pai e o processo-filho compartilhem a mesma pilha de memória.

Parabéns! A alternativa C está correta.

A função fork cria um novo processo-filho ao ser chamada. Esse processo é uma cópia do processo-pai. Para diferenciar, a função fork retorna zero para o processo-filho e o PID do filho para o processo-pai. Esse comportamento permite que ambos os processos saibam seu papel após a chamada de fork: o pai pode identificar o filho por seu PID, enquanto o filho sabe que é um novo processo criado por causa do retorno zero. Isso garante a execução correta e coordenada de tarefas em sistemas Unix-like, em que múltiplos processos podem ser gerenciados de forma eficiente.

Função fork básica: uso em soquetes

Aqui, veremos a comunicação entre processos usando soquetes no modelo cliente-servidor, detalhando os tipos e os passos para estabelecer conexões. Isso inclui um exemplo prático de código em C para um cliente e um servidor simples que utilizam stream sockets e o protocolo TCP.

Neste vídeo, você vai entender como utilizar a função fork para gerenciar múltiplas conexões de soquetes em um servidor.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Modelo cliente-servidor

Geralmente, é usado na comunicação entre processos. O cliente se conecta ao servidor para solicitar informações.

Exemplo

Cliente: alguém fazendo uma chamada telefônica. Servidor: pessoa que atende. O cliente precisa conhecer o endereço do servidor, mas o servidor não precisa saber o endereço do cliente antes de a conexão ser estabelecida. Uma vez conectados, ambos podem enviar e receber informações.

As chamadas de sistema para estabelecer a conexão variam entre cliente e servidor, mas ambos constroem um soquete, uma extremidade de um canal de comunicação entre processos. Cada processo cria seu próprio soquete. A seguir, você vai conhecer os passos para estabelecer um soquete. Vamos lá!

Passos para estabelecer um soquete no lado do cliente:

1. Crie um soquete com socket().
2. Conecte o soquete ao endereço do servidor usando connect().
3. Envie e receba dados com read() e write().

Passos para estabelecer um soquete no lado do servidor:

1. Crie um soquete com socket().
2. Vincule o soquete a um endereço usando bind().
3. Escute conexões com listen().
4. Aceite uma conexão com accept().
5. Envie e receba dados.

Os soquetes podem ser de dois tipos. A seguir, você vai conhecer cada um deles. Vamos lá!

Stream sockets

Tratam a comunicação como um fluxo contínuo de caracteres e usam o protocolo TCP (Transmission Control Protocol), confiável e orientado a fluxo.

Datagram sockets

Precisam ler mensagens inteiras de uma vez e usam o protocolo UDP (User Datagram Protocol), não confiável e orientado a mensagens.

Exemplos de códigos

A seguir, você conhecerá os códigos que ilustram um cliente e um servidor simples que se comunicam usando **stream sockets** no domínio da internet. Podemos determinar isso pelo uso do parâmetro **SOCK_STREAM** na chamada da função socket, tanto no código do servidor quanto no código do cliente.

Código servidor.c

Confira o código!

```c
```

Código cliente.c

Confira o código!

```c
```

Uso do fork() para lidar com cada nova conexão

O código de servidor mostrado anteriormente manipula somente uma conexão e depois encerra. Porém, um servidor deve funcionar indefinidamente, além de lidar com diversas conexões simultâneas, cada uma em seu próprio processo. Isso pode ser feito bifurcando um novo processo para lidar com cada nova conexão.

A seguir, você vai observar a alteração que pode ser feita dentro do arquivo **servidor.c:**, para permitir que o servidor lide com múltiplas conexões simultâneas.

c

Atividade 3

Considerando o uso da função fork em servidores que utilizam sockets para gerenciar múltiplas conexões de clientes, qual das seguintes afirmações a seguir descreve corretamente o comportamento do processo-pai e do processo-filho após a chamada de fork?

A — O processo-pai continua aceitando novas conexões, enquanto o processo-filho trata a conexão específica que foi aceita.

B — O processo-filho assume a responsabilidade de aceitar novas conexões, enquanto o processo-pai se dedica a tratar a conexão atual.

C — Tanto o processo-pai quanto o processo-filho tratam a mesma conexão de forma simultânea.

D — Após a criação do processo-filho, o processo-pai encerra sua execução, e o filho assume o gerenciamento de todas as novas conexões.

E — O processo-filho encerra imediatamente após ser criado, deixando a gestão das conexões totalmente a cargo do processo-pai.

Parabéns! A alternativa A está correta.

Após a chamada de fork em um servidor que utiliza soquetes, o processo-pai continua a executar o loop de aceitação de novas conexões. Isso permite que o servidor continue a ouvir e a aceitar solicitações de novos clientes. Enquanto isso, o processo-filho é responsável por tratar a conexão específica aceita no momento da chamada de fork. Esse comportamento garante que o servidor gerencie múltiplas conexões simultaneamente, de forma eficiente, com o processo-pai delegando o tratamento das conexões individuais aos processos-filhos com um desempenho adequado e dando continuidade ao serviço.

Vantagens e desvantagens do uso de Fork

A chamada de sistema fork em sistemas operacionais possibilita a criação de novos processos para executar tarefas simultaneamente, promovendo a reutilização de código e otimização de memória. Apesar de vantagens como isolamento de processos, existem desvantagens, como sobrecarga de memória e complexidade de comunicação. Por isso, é preciso compreender esses aspectos para desenvolver sistemas eficientes e estáveis.

Neste vídeo, você vai compreender os benefícios e as limitações ao utilizar a função fork para criar processos.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Chamada de sistema fork: vantagens e desvantagens

Poderoso mecanismo para criar processos em sistemas operacionais, a chamada de sistema fork permite a execução simultânea de múltiplas tarefas e a utilização eficiente dos recursos do sistema, replicando o processo-pai.

Vantagens da chamada de sistema fork

A seguir, você conhecerá as vantagens da chamada de sistema fork. Vamos lá!

Execução simultânea de tarefas

A chamada de sistema fork permite a criação de novos processos, o que facilita a execução de múltiplas tarefas simultaneamente. Isso melhora a eficiência do sistema e suas habilidades de multitarefas.

Reutilização de código

O processo-filho herda uma cópia exata do processo-pai, incluindo todo o código, ao usar fork. Isso promove a reutilização de código existente e agiliza a criação de programas complexos.

Otimização de memória

A fork utiliza o método **copy-on-write**, otimizando o uso de memória. Inicialmente, pai e filho compartilham as mesmas páginas de memória física, e só ocorre a cópia quando uma página é alterada, melhorando a eficiência da memória.

Isolamento de processos

Cada processo criado por fork tem sua própria área de memória e conjunto de recursos. Esse isolamento melhora a estabilidade e a segurança do sistema, evitando que os processos interfiram uns nos outros.

Desvantagens da chamada de sistema fork

Agora, você conhecerá as desvantagens da chamada de sistema fork. Vamos lá!

Sobrecarga de memória

Apesar da otimização copy-on-write, há uma sobrecarga de memória inicial, pois o processo-pai é copiado em sua totalidade, incluindo toda a sua memória, o que aumenta o uso da memória.

Duplicação de recursos

O processo-filho duplica todos os descritores de arquivos abertos, conexões de rede e outros recursos do processo-pai.

Isso pode desperdiçar recursos e resultar em ineficiências.

Complexidade da comunicação

Fork cria processos independentes, que podem precisar se coordenar e se comunicar. Para permitir a transmissão de dados entre processos, métodos como **pipes** ou **memória compartilhada** devem ser implementados, aumentando a complexidade.

Impacto no desempenho do sistema

A bifurcação de um processo duplica a alocação de memória e o gerenciamento de recursos, o que pode afetar o desempenho do sistema, especialmente quando processos são frequentemente iniciados e interrompidos.

Atividade 4

A função fork é uma ferramenta poderosa em sistemas operacionais Unix-like para criar novos processos. Embora ofereça diversas vantagens, como a execução simultânea de múltiplas tarefas, também apresenta algumas desvantagens relacionadas com o uso de recursos e a complexidade de gerenciamento.

Considerando as vantagens e desvantagens do uso da função fork, qual das seguintes afirmações a seguir está correta?

A A função fork melhora a reutilização de código, pois o processo-filho herda uma duplicata exata do processo-pai.

B A função fork não apresenta qualquer sobrecarga de memória, já que o processo-pai e o processo-filho compartilham todas as páginas de memória.

C A função fork simplifica a comunicação entre processos, eliminando a necessidade de mecanismos de sincronização complexos.

D A função fork permite que processos-filhos compartilhem exatamente os mesmos ponteiros de busca de arquivos que o processo-pai.

E A função fork evita completamente a duplicação de recursos, garantindo eficiência máxima no uso de descritores de arquivo e conexões de rede.

Parabéns! A alternativa A está correta.

O processo-filho herda o mesmo código e o mesmo estado do processo-pai no momento da criação, característica vantajosa, porque permite que tarefas complexas sejam divididas entre processos, aproveitando o mesmo código-base, sem necessidade de recriação ou recompilação. Contudo, fork também implica uma duplicação inicial de recursos e pode aumentar a complexidade do gerenciamento de processos e memória.

Desenvolvendo uma aplicação com Fork

A função fork é uma chamada de sistema em sistemas operacionais Unix-like para a criação de novos processos. Permite que um processo (pai) crie um novo processo (filho), uma cópia quase exata de si mesmo. Tal característica é particularmente útil em servidores de rede que precisam lidar com múltiplas conexões de clientes simultaneamente. Devemos compreender e utilizar fork corretamente, ao desenvolver

programas que necessitam de execução concorrente. Vamos mostrar a seguir como criar e gerenciar processos usando fork, garantindo a sincronização correta entre processo-pai e processo-filho.

Neste vídeo, você vai aprender como criar uma aplicação que utiliza fork para multitarefas.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Roteiro de prática

Você é um desenvolvedor de um servidor de rede que precisa atender a múltiplas conexões de clientes simultaneamente. Para isso, deve usar a função fork para criar processos-filhos que lidem com cada conexão de cliente, enquanto o processo-pai continua a aceitar novas conexões. Seu objetivo é implementar um servidor simples que utiliza fork para criar um processo-filho para cada conexão de cliente.

Você implementará um servidor que usa soquetes para aceitar conexões e fork para criar processos-filhos que tratem cada conexão de cliente.

c

A estrutura do código é explicada a seguir:

- **Biblioteca**: inclusão de bibliotecas necessárias para manipulação de soquetes e comunicação em rede.
- **Função error**: exibe mensagens de erro e encerra o programa.
- **A função main no servidor**:
    - Verifica se o número da porta foi fornecido como argumento.
    - Cria um soquete para comunicação.
    - Associa o soquete a um endereço IP e porta específicos.
    - Coloca o soquete em modo de escuta para aceitar conexões.
    - Entra em um loop infinito para aceitar e processar conexões.
    - Cria um novo processo para tratá-la para cada conexão aceita.
- **Processo-filho**:
    - Fecha o soquete original (sockfd).
    - Lê dados do cliente através do novo socket (newsockfd).
    - Exibe a mensagem recebida.
    - Envia uma resposta para o cliente.
    - Fecha o novo soquete e encerra o processo-filho.
- **Processo-pai**:
    - Fecha o novo socket (newsockfd) e retorna ao início do loop para aceitar a próxima conexão.

Faça você mesmo!

Considere o servidor de rede implementado anteriormente utilizando a **função fork()**. Suponha que você queira modificar o servidor para adicionar um recurso de logging em que cada mensagem recebida de um cliente é registrada em um arquivo.

Qual ação deverá ser tomada para garantir que cada processo-filho registre a mensagem recebida de forma correta e sem interferências de outros processos-filhos?

A Abrir o arquivo de log apenas no processo-pai e compartilhar o descritor de arquivo com os processos-filhos.

B Abrir o arquivo de log em cada processo-filho, garantindo que cada processo-filho tenha seu próprio descritor de arquivo.

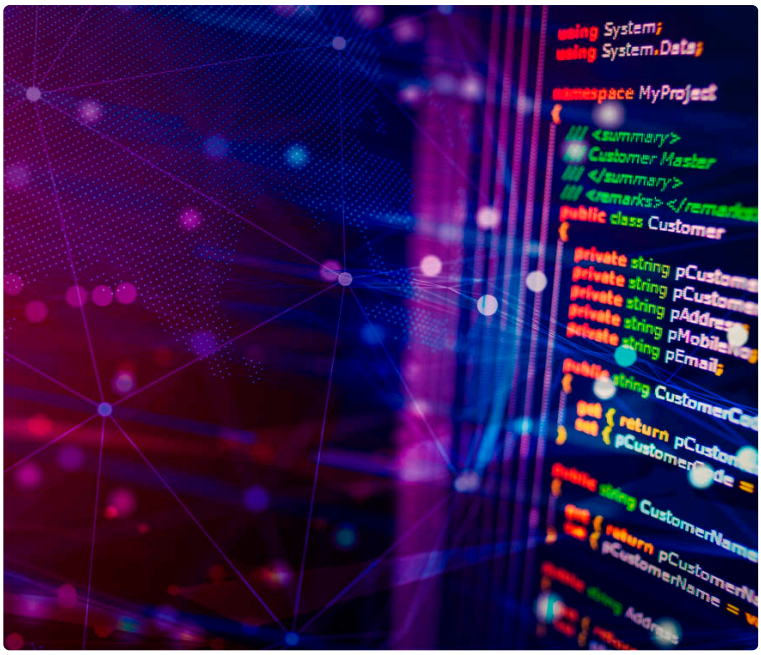
C Usar um mutex para sincronizar o acesso ao arquivo de log entre o processo-pai e o processo-filho.

D Modificar o código para que apenas o processo-pai registre as mensagens recebidas de todos os clientes.

E Usar uma variável global para armazenar as mensagens e registrar todas as mensagens ao final da execução do servidor.

Parabéns! A alternativa B está correta.

Abrir o arquivo de log em cada processo-filho garante que cada processo tenha seu próprio descritor de arquivo, evitando interferências de outros processos-filhos. Isso evita problemas de concorrência e garante que as mensagens sejam registradas correta e independentemente por cada processo-filho.



2 - Funções para threads básicos

Neste módulo, você será capaz de comparar a execução de funções para tarefas em paralelo (threads) e a criação de processos em paralelo (forks).

O conceito de operação com threads

Threads são unidades de execução dentro de um processo, permitindo multitarefas. Enquanto **threads de usuário** são gerenciados por bibliotecas de usuário, **threads do kernel** são controlados pelo sistema operacional. **Multithreading** pode melhorar o desempenho em operações de entrada-saída, mas isso exige uma análise cuidadosa para balancear benefícios e complexidades.

Neste vídeo, você vai ver como threads permitem a execução concorrente dentro de um único processo.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

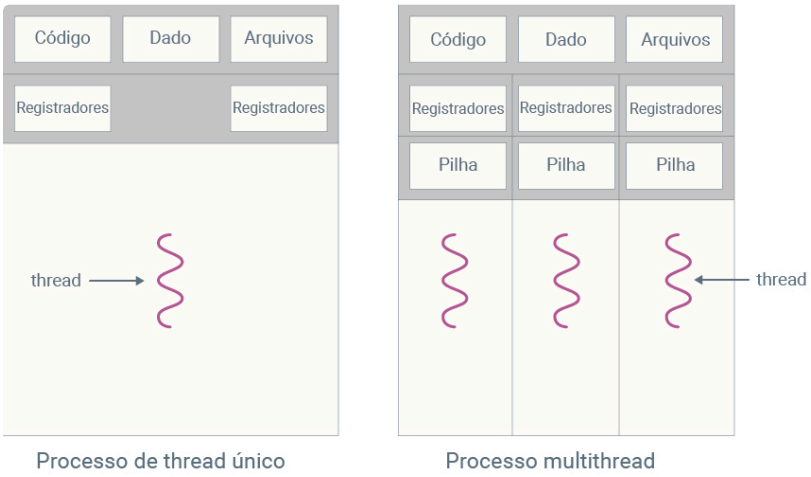
O que são threads?

Ao abrir um programa, um processo é criado. Um thread é uma **unidade de execução** dentro desse processo:

- Um programa com um único thread tem um único thread (geralmente chamado de **thread principal**).

- Um programa com múltiplos threads realiza multitarefas, executando cálculos simultaneamente e acelerando o programa.

A maioria dos sistemas operacionais modernos suporta multiprocessamento e execução de processos em múltiplos threads, conhecidos como **multithreading**. Isso aumenta o desempenho e a escalabilidade, mas também a complexidade do código e a dificuldade de depuração.



Diferença entre processo de thread único e processo multithread.

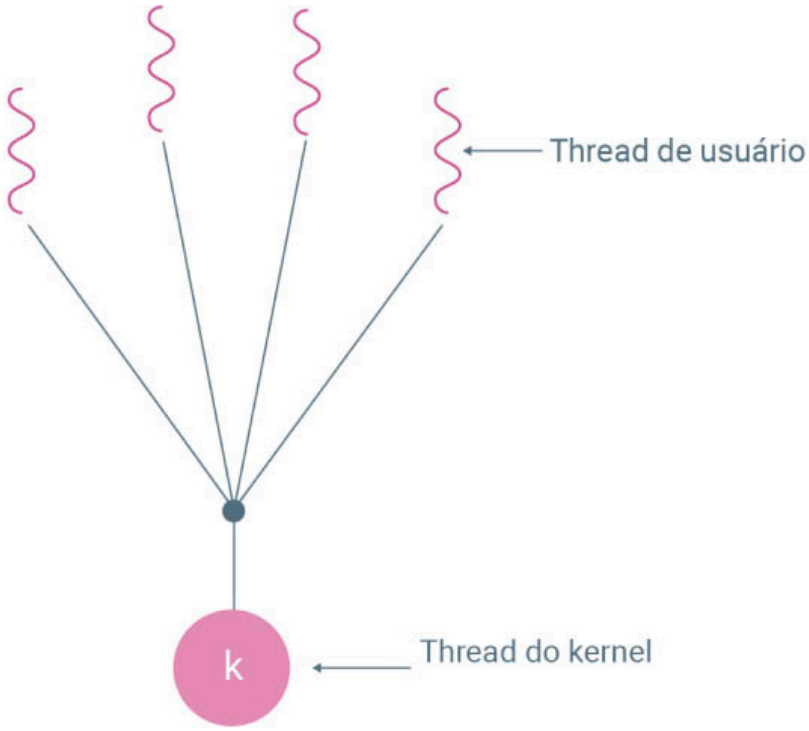
Existem dois tipos de threads:

- **Threads de usuário**: gerenciados por uma biblioteca no nível do usuário, que fornece suporte de tempo de execução e gerenciamento.
- **Threads do kernel**: gerenciados pelo sistema operacional, com o kernel sendo multithread.

Veja a seguir como threads de nível de usuário podem ser mapeados para threads de nível de kernel subjacentes.

Modelo um para muitos

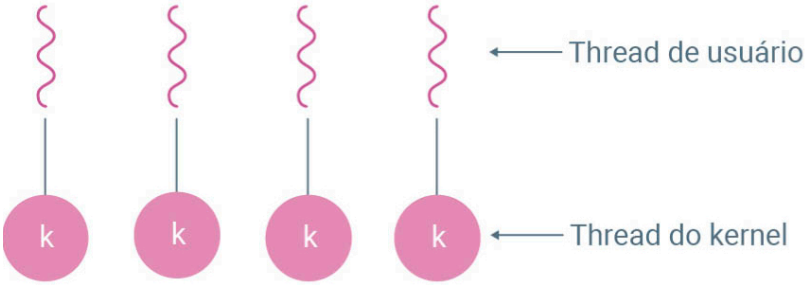
Vários threads de usuário são mapeados para um thread do kernel. A biblioteca de usuário gerencia os threads. Se um thread fizer uma chamada de bloqueio, todo o processo ficará lento.



Representação de modelo um para muitos.

Modelo um para um

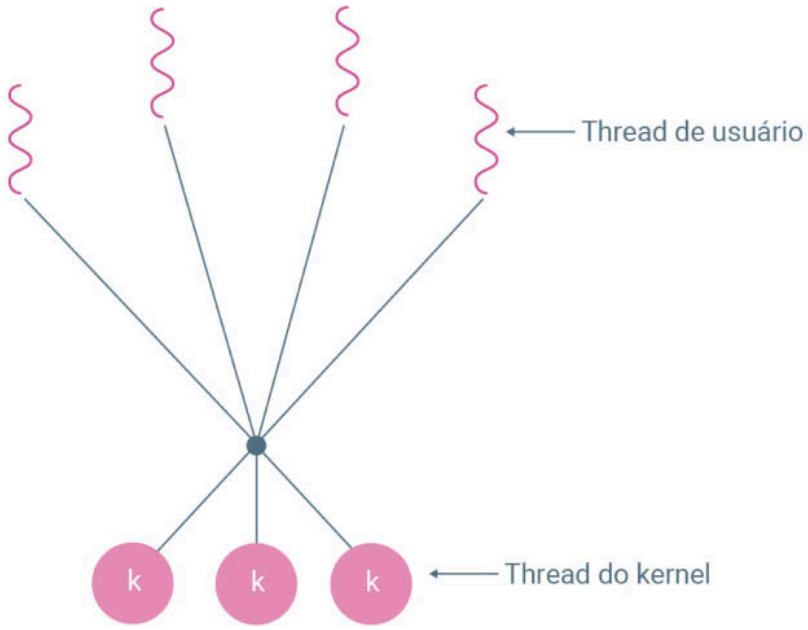
Cada thread de usuário é mapeado diretamente para um thread do kernel. O escalonador do kernel controla os threads. Há um limite no número de threads suportados, para evitar sobrecarga.



Representação de modelo um para um.

Modelo muitos para muitos

Muitos threads de usuário são multiplexados em menos threads do kernel. Supera os problemas dos outros dois modelos, permitindo execução paralela verdadeira sem impor limites ao número total de threads. Porém, é mais difícil de implementar.



Representação de modelo muitos para muitos.

Threads são representados por Thread Control Blocks (TCBs), que contêm alguns elementos. A seguir, você vai conhecer a função de cada um desses elementos. Vamos lá!

Thread ID

Identificador exclusivo, gerado pelo sistema operacional.

Thread states

Estado do thread.

Informações da CPU

Dados necessários sobre o thread, como contagem de programas e conteúdo de registro.

Prioridade do thread

Indica a prioridade do thread em relação a outros.

Ponteiro de processo

Aponta para o processo que iniciou o thread.

Ponteiro(s) de thread

Lista de ponteiros para outros threads criados pelo thread atual.

Vários threads podem acelerar o código usando um servidor web. Como exemplo, criar um novo processo para cada solicitação HTTP seria ineficiente. Criar um novo thread é mais eficiente.

O multithreading é útil em problemas de entrada-saída (IO). Para tarefas que exigem muito poder de computação, múltiplos processos podem ser mais benéficos. Assim, pode acelerar significativamente o código, se usado corretamente.

> Cabe ao desenvolvedor decidir se vale a pena implementar o multithreading, considerando as complexidades e os benefícios.

Atividade 1

Threads são unidades de execução dentro de um processo que permitem a execução simultânea de múltiplas tarefas. Em ambientes de programação concorrente, threads melhoram o desempenho e a eficiência de aplicações, especialmente em sistemas multiprocessadores. O gerenciamento adequado de threads evita problemas como condições de corrida e garante a correta sincronização dos dados compartilhados.

Com base no conceito de operação com threads, qual das seguintes afirmações descreve corretamente como threads funcionam dentro de um processo?

A Threads em um processo têm seu próprio espaço de memória separado e não compartilham dados com outros threads.

B Threads em um processo compartilham o mesmo espaço de memória e podem acessar variáveis globais e heap comuns.

C Cada thread em um processo apresenta sua própria cópia das variáveis globais, evitando a necessidade de sincronização.

D Threads não podem ser usados em sistemas multiprocessadores, pois não suportam execução paralela.

E Threads não podem se comunicar e só interagem por meio do processo-pai.

Parabéns! A alternativa B está correta.

Todos os threads têm acesso às mesmas variáveis e podem modificar os mesmos dados. Essa característica de compartilhamento de memória é vantajosa para a comunicação eficiente entre threads, mas também traz a necessidade de mecanismos de sincronização, como mutexes e semáforos, para evitar condições de corrida e garantir que os dados sejam acessados e modificados de maneira controlada e segura.

A biblioteca Pthreads

Uma das evoluções significativas no uso do C foi a introdução do padrão Posix (Portable Operating System Interface), estabelecido inicialmente em 1988, que atua na uniformização das chamadas de sistema entre diferentes variantes de sistemas operacionais Unix-like.

Isso facilita a portabilidade e a funcionalidade dos programas em múltiplos ambientes. Ao explorar como o Posix estende as capacidades do Ansi-C, ganhamos uma compreensão mais profunda sobre como as interfaces de programação podem influenciar a compatibilidade e a eficiência das aplicações modernas.

Neste vídeo, você vai conhecer a biblioteca Pthreads, uma ferramenta essencial para criar e gerenciar threads em C.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

O que são Pthreads?

No passado, os fornecedores de hardware implementavam suas próprias versões de threads. Isso significa que empresas como IBM, Apple e Intel tinham suas próprias versões proprietárias de threads. Essas implementações diferiam substancialmente umas das outras, tornando difícil para os programadores desenvolver aplicações threaded portáteis.

Para corrigir isso, foi criada uma interface de programação padronizada. Em sistemas Unix, essa interface foi especificada pelo padrão IEEE Posix 1003.1c (1995). As implementações que aderem a esse padrão são chamadas de **threads Posix** ou **Pthreads (POSIX threads)**.

> Pthreads é uma biblioteca C/C++ usada para gerenciar threads com base no padrão Posix.

A maioria dos fornecedores de hardware agora oferece Pthreads além de suas APIs proprietárias. Ao longo dos anos, o padrão Posix continuou a evoluir e a passar por revisões, incluindo as especificações Pthreads.

Os métodos da biblioteca Pthreads podem ser categorizados em quatro grupos. A seguir, você vai conhecer cada um deles. Vamos lá!

Thread management

Rotinas que funcionam diretamente em threads — criação, desanexação, junção etc. Também incluem funções para

definir/consultar atributos de thread (joinable, agendamento etc.).

Mutexes

Rotinas que tratam de sincronização, chamadas de mutex, abreviatura de **exclusão mútua**. As funções mutex permitem criar, destruir, bloquear e desbloquear mutexes. São complementadas por funções de atributos mutex que definem ou modificam atributos associados a mutexes.

Variáveis de condição

Rotinas que abordam comunicações entre threads que compartilham um mutex, com base nas condições especificadas pelo programador. Esse grupo inclui funções para criar, destruir, esperar e sinalizar com base em valores de variáveis especificados. Funções para definir/consultar atributos de variáveis de condição também estão incluídas.

Sincronização

Rotinas que gerenciam bloqueios e barreiras de leitura/gravação.

Agora, conheça os tipos fornecidos pelo Pthreads.

- `pthread_`: threads próprios e métodos diversos.
- `pthread_t`: identificador de um thread.
- `pthread_mutex_t`: primitiva de sincronização mutex (exclusão mútua).
- `pthread_mutexattr_t`: atributo de criação de mutex.
- `pthread_cond_t`: primitivo de sincronização de variável de condição.
- `pthread_condattr_t`: atributo de criação de variável de condição.
- `pthread_key_t`: chave de armazenamento local do thread.
- `pthread_once_t`: variável de controle de inicialização única.

- `pthread_attr_t`: atributo de criação de thread.

Criando threads

Inicialmente, seu **`programa main()`** compreende um único thread-padrão. Todos os outros threads devem ser criados explicitamente pelo programador. **`pthread_create`** cria um novo thread e o torna executável. Esse método pode ser chamado quantas vezes quiser em qualquer lugar de seu código.

C

Teremos quatro argumentos:

- `thread`: um identificador opaco e exclusivo para o novo thread retornado pela sub-rotina.
- `Attr`: um objeto de atributo opaco que pode ser usado para definir atributos de thread. É possível especificar um objeto de atributos de thread ou Null para os valores-padrão.
- `start_routine`: a função C que o thread executará depois de criado.
- `Arg`: um único argumento que pode ser passado para start_routine. Deve ser passado por referência como um ponteiro convertido do tipo void. Null pode ser usado se nenhum argumento for passado.

Após a conclusão bem-sucedida, pthread_create() retorna o código 0. Caso ocorra erro, será retornado um valor diferente de zero.

O número máximo de threads que podem ser criados por um processo depende da implementação. Os programas que tentam exceder o limite podem falhar ou produzir resultados errados.

> Uma vez criado um thread, ele pode criar outros threads. Não há hierarquia implícita ou dependência entre threads.

A seguir, observe um exemplo básico de um **programa multithread**.

C

Encerrando threads

Existem várias maneiras pelas quais um thread pode ser encerrado.

- O thread retorna normalmente de sua rotina inicial. Seu trabalho está feito.
- O thread faz uma chamada para o método `pthread_exit` — esteja seu trabalho concluído ou não.
- O thread é cancelado por outro thread por meio da rotina `pthread_cancel`.
- Todo o processo é encerrado por uma chamada para `exec()` ou `exit()`.
- Se `main()` terminar primeiro, sem chamar `pthread_exit` explicitamente, todos os outros threads terminarão abruptamente.

Para finalizar, observe o método **pthread_exit**.

C

Atividade 2

A biblioteca Pthreads (POSIX threads) é uma implementação-padrão para criação e gerenciamento de threads em sistemas Unix-like. Pthreads fornece um conjunto de APIs que permitem aos programadores criar e sincronizar threads, proporcionando controle sobre a execução concorrente em aplicações multithread. A biblioteca inclui funções para criação, junção, sincronização de threads e mecanismos como mutexes e variáveis de condição, para garantir a correta coordenação entre threads.

Com relação ao uso da biblioteca Pthreads, qual das afirmações a seguir está correta?

A A função pthread_create bloqueia o thread chamador até que o novo thread criado termine sua execução.

B A função pthread_join permite que um thread espere pela terminação de outro thread específico.

C Mutexes em Pthreads são usados para sinalizar eventos entre threads, similar a variáveis de condição.

D A função pthread_cond_signal é usada para criar um novo thread que sinaliza uma variável de condição.

E Threads criados com pthread_create compartilham completamente seu contexto de execução, incluindo registradores de CPU e pilha.

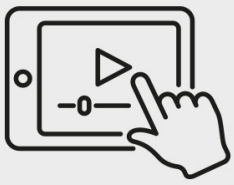
Parabéns! A alternativa B está correta.

A função pthread_join garante a sincronização entre threads, permitindo que um thread termine suas operações antes que outro prossiga. Essa função em programas multithread coordena a finalização de tarefas e evita problemas de concorrência, garantindo que os recursos sejam liberados de forma apropriada e os resultados de threads dependentes sejam corretamente integrados.

Vantagens e desvantagens do uso de threads

O multithreading envolve a execução simultânea de múltiplos threads, com modelos variando entre threads de nível de usuário e threads de nível de kernel. Enquanto threads de usuário oferecem flexibilidade e portabilidade, threads do kernel garantem melhor desempenho e paralelismo. Modelos híbridos combinam vantagens de ambos, embora sejam mais complexos.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Vantagens e desvantagens dos modelos de threading no gerenciamento de processos

Veremos a seguir os threads em nível de usuário, em nível de kernel e os modelos híbridos, bem como vantagens e desvantagens de sua utilização.

Threads em nível de usuário

São gerenciados por uma biblioteca em nível do usuário, sem suporte direto do sistema operacional. A seguir, você vai conhecer suas

vantagens e desvantages. Vamos lá!

Vantagens

- Maior flexibilidade e controle sobre o gerenciamento de threads.
- Melhor portabilidade entre diferentes sistemas operacionais.

Desvantagens

- Desempenho inferior, pois o agendamento de threads é gerenciado pelo próprio aplicativo.
- Paralelismo limitado, pois todos os threads são restritos a um único processador.

Threads em nível de Kernel

São gerenciados diretamente pelo sistema operacional, cada um podendo ser agendado e executado de forma independente. A seguir, você vai conhecer suas vantagens e desvantages. Vamos lá!

Vantagens

- Melhor desempenho em razão do agendamento eficiente pelo sistema operacional.
- Maior paralelismo, permitindo a execução em múltiplos processadores.

Desvantagens

- Menor flexibilidade e controle, com maior dependência do sistema operacional.
- Menor portabilidade entre diferentes sistemas operacionais.

Modelos híbridos

São a combinação de threads em nível de usuário e de kernel. A seguir, você vai conhecer suas vantagens e desvantages. Vamos lá!

Vantagens

- Combina flexibilidade e controle dos threads de usuário com o desempenho dos threads do kernel.
- Melhor escalabilidade, suportando um maior número de threads e processadores.

Desvantagens

- Complexidade maior na implementação e manutenção.
- Requer mais recursos em razão do suporte necessário tanto no nível de usuário quanto no nível de kernel.

Vantagens e desvantagens dos tipos de modelos de threading

A seguir, apresentaremos os tipos de modelos e suas funcionalidades.

Modelo muitos para muitos

Vários threads de usuário são multiplexados para um número igual ou menor de threads do kernel. A seguir, você vai conhecer suas vantagens e desvantages. Vamos lá!

Vantagens

- Se um thread de usuário bloquear, outros threads poderão ser agendados.
- Alta eficiência e melhor uso de recursos.

Desvantagens

- Complexidade na implementação.

Modelo muitos para um

Vários threads de usuário mapeiam para um único thread do kernel. A seguir, você vai conhecer suas vantagens e desvantages. Vamos lá!

Vantagens

- Gerenciamento eficiente em nível do usuário.

Desvantagens

- Se um thread bloquear, todos os outros também bloquearão.
- Sem paralelismo em múltiplos processadores.

Modelo um para um

Cada thread de usuário mapeia diretamente para um thread do kernel. A seguir, você vai conhecer suas vantagens e desvantagens.

Vantagens

- Suporta execução paralela em múltiplos processadores.

Desvantagens

- Criação de threads de usuário requer threads do kernel correspondentes, o que pode limitar o número de threads por causa do overhead.

Vantagens e desvantagens do multithreading no sistema operacional

A seguir, você vai conhecer as vantagens e desvantagens que envolvem o multithreading no sistema operacional. Vamos lá!

Vantagens

- **Minimização do tempo de troca de contexto**: troca de contexto eficiente entre threads.
- **Simultaneidade**: permite a execução de várias instruções ao mesmo tempo em um processo.
- **Criação e troca de contexto econômicas**: troca de threads é eficiente, envolvendo apenas identidades e recursos como contador de programa, registros e ponteiros de pilha.
- **Utilização de arquitetura multiprocessador**: melhor utilização de recursos disponíveis em sistemas multiprocessadores.

Desvantagens

- **Operação sem interrupções**: pode levar a complexidades no gerenciamento de threads.
- **Complexidade do código**: torna o código mais difícil de entender e manter.
- **Custo de gerenciamento de threads**: pode ser excessivo para tarefas simples.
- **Dificuldade na resolução de problemas**: identificação e solução de problemas podem ser um desafio pela complexidade do código.

Atividade 3

Considere um cenário em que uma aplicação realiza operações intensivas de entrada-saída e processamento paralelo de dados, utilizando threads para otimizar o desempenho.

Qual das seguintes alternativas destaca corretamente uma vantagem e uma desvantagem do uso de threads em uma aplicação?

A Vantagem: simplificam a depuração do código. Desvantagem: aumentam o tempo de troca de contexto.

B Vantagem: permitem utilizar melhor os sistemas multiprocessadores. Desvantagem: podem introduzir complexidade na sincronização de dados compartilhados.

C Vantagem: eliminam a necessidade de sincronização de dados. Desvantagem: não são eficientes em sistemas multiprocessadores.

D Vantagem: garantem acesso exclusivo a recursos compartilhados. Desvantagem: não permitem a execução simultânea de múltiplas tarefas.

E Vantagem: reduzem a complexidade do código. Desvantagem: consomem mais memória do que processos.

Parabéns! A alternativa B está correta.

Threads permitem executar tarefas paralelamente em múltiplos núcleos, aumentando a eficiência, mas requerem mecanismos de sincronização adequados, para evitar condições de corrida, o que aumenta a complexidade do código.

A criação de threads na prática

Criar threads possibilita que um programa execute múltiplas tarefas de forma concorrente, aumentando a eficiência e o desempenho, especialmente em sistemas multiprocessadores. Usar a biblioteca Pthreads em C facilita a criação e o gerenciamento de threads. Assim,

cada thread executa uma função específica. Criação e gerenciamento de threads tornam possível desenvolver aplicações que realizam tarefas simultâneas de forma eficaz. Vamos aprender a seguir como criar threads, passar argumentos para as funções dos threads e sincronizar a execução usando a função pthread_join.

Neste vídeo, você vai entender como implementar threads em seu código com exemplos práticos.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Roteiro de prática

Você está desenvolvendo uma aplicação que precisa realizar várias operações de cálculo matemático simultaneamente, para melhorar o desempenho. Cada operação será realizada por um thread separado. Seu objetivo é implementar a criação de múltiplos threads, em que cada thread executa uma função que realiza uma operação matemática específica, e garantir que o programa principal aguarde a conclusão de todos os threads antes de finalizar.

Você implementará uma aplicação C que cria múltiplos threads usando a biblioteca Pthreads. Cada thread executará uma função que realiza uma operação matemática, e o programa principal usará `pthread_join` para aguardar a conclusão de todos os threads. Confira a resolução!

c

A estrutura do código é explicada a seguir.

- **Biblioteca**: inclusão das bibliotecas necessárias para manipulação de threads e operações de entrada-saída.
- **Definição de constantes**: indicação do número de threads que serão criados.
- **Função** `perform_operation`: será executada por cada thread.
    - Obtém o ID do thread a partir do argumento.
    - Realiza uma operação matemática baseada no ID do thread.
    - Exibe o resultado da operação.
    - Encerra o thread.
- **Função** `main`:
    - Declara arrays para armazenar os identificadores dos threads e seus IDs.
    - Cria os threads em um loop.
    - Verifica se houve erro na criação de cada thread.
    - Espera que todos os threads completem suas operações usando `pthread_join`.
    - Exibe uma mensagem indicando que todos os threads finalizaram suas operações.

Faça você mesmo!

Considere a aplicação multithreaded descrita anteriormente, que cria e gerencia múltiplos threads para realizar operações matemáticas. Suponha que você deseja adicionar mais uma operação, em que o thread de ID 5 realiza a operação de divisão por 2.

Qual ação deve ser tomada para implementar essa funcionalidade corretamente?

A Modificar a função perform_operation para incluir um caso adicional para o ID 5 e criar um novo thread com

esse ID.

B Modificar a função perform_operation para incluir a operação de divisão em todos os casos existentes.

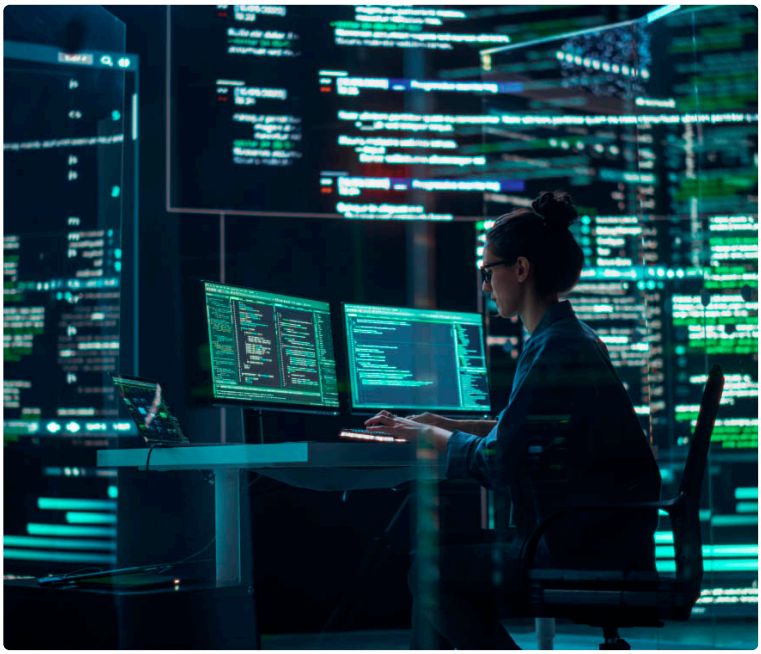
C Adicionar um mutex para garantir que a operação de divisão seja executada sem interferência de outros threads.

D Remover a função perform_operation e implementar todas as operações diretamente na função principal.

E Duplicar o código de criação de threads para incluir a nova operação de divisão.

Parabéns! A alternativa A está correta.

Para adicionar a operação de divisão por 2 ao thread de ID 5, é necessário modificar a função perform_operation, a fim de incluir um caso adicional para o ID 5 que realiza a operação de divisão. Além disso, um novo thread com o ID 5 deve ser criado na função principal para executar essa operação específica. Isso garante que a nova funcionalidade seja implementada de forma organizada e consistente com a estrutura existente.



3 - Sincronização de threads

Ao final deste módulo, você será capaz de identificar as questões de sincronização de threads para evitar a inconsistência de dados nas aplicações com tarefas concorrentes.

Visão geral sobre sincronização de processos

A sincronização de processos coordena a execução de múltiplos processos para garantir acesso controlado a recursos compartilhados, evitando condições de corrida e garantindo a integridade dos dados. Utilizando técnicas como semáforos e a solução de Peterson, a sincronização garante o funcionamento eficiente de sistemas multiprocessos, embora possa aumentar a complexidade e a sobrecarga do sistema.

Neste vídeo, você vai compreender as técnicas de sincronização que garantem a coordenação correta entre processos.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Sincronização de processos

É a coordenação da execução de múltiplos processos em um sistema multiprocesso, para garantir acesso controlado e previsível a recursos compartilhados. O objetivo é evitar condições de corrida e garantir a consistência e a integridade dos dados. Existem dois tipos de processos. A seguir, você vai entender cada um deles. Vamos lá!

Processo independente

Sua execução não afeta outros processos.

Processo cooperativo

Sua execução pode afetar ou ser afetada por outros processos.

Problemas de sincronização

Surgem em processos cooperativos em razão do compartilhamento de recursos. A seguir, acompanhe a descrição desses problemas.

Condição de corrida

Ocorre quando vários processos acessam e manipulam dados simultaneamente, resultando em saídas imprevisíveis. Para evitar condições de corrida, seções críticas, em que o acesso a variáveis compartilhadas deve ser controlado, devem ser tratadas como instruções atômicas. O uso adequado de bloqueios ou variáveis atômicas também pode evitar condições de corrida.

Exemplo

Dois processos, P1 e P2, que compartilham uma variável (compartilhada = 10). Se P1 executa primeiro, incrementa a variável e entra em espera; P2, então, executa e decrementa a variável. Dependendo da ordem de execução, o valor final da variável pode ser 9 ou 11, resultando em dados inconsistentes.

Problema da seção crítica

Segmento de código que pode ser acessado por apenas um processo por vez, uma seção crítica contém variáveis compartilhadas que precisam ser sincronizadas para manter a consistência dos dados. Confira a estrutura de uma seção crítica de um processo específico!

c

Três requisitos devem ser atendidos para resolver o problema da seção crítica. A seguir, você vai conhecer cada um deles. Vamos lá!

Exclusão mútua

Apenas um processo pode estar na seção crítica de cada vez.

Progresso

Se nenhum processo estiver na seção crítica, os processos aguardando devem decidir quem entra na seção crítica sem atraso.

Espera limitada

Deve haver um limite para o tempo de espera de um processo para entrar na seção crítica.

A seguir, observe um exemplo de Implementação de exclusão mútua.

c

Propostas de solução

É possível utilizar dois tipos de solução para resolver o problema da seção crítica. A seguir, você vai conhecer cada um deles.

Solução de Peterson

Solução clássica, baseada em software, para o problema da seção crítica, utilizando duas variáveis compartilhadas. Acompanhe.

Sinalizador booleano[i]

Indica se o processo quer entrar na seção crítica.

Turn

Indica qual processo tem a vez de entrar na seção crítica.

A seguir, observe um exemplo da solução de Peterson.

c

Garante-se, assim, a exclusão mútua, o progresso e a espera limitada, mas envolve espera ocupada e é limitada a dois processos. Não é adequada para arquiteturas de CPU modernas.

Semáforos

Mecanismo de sinalização que usa duas operações atômicas, wait e signal, para sincronização de processos. Existem dois tipos de semáforos. Acompanhe.

Semáforos binários

Podem ser 0 ou 1, funcionando como bloqueios mutex para garantir exclusão mútua.

Semáforos de contagem

Podem ter qualquer valor e controlam o acesso a recursos com limitações no número de acessos simultâneos.

A seguir, observe um exemplo de exclusão mútua.

c

Vantagens e desvantagens da sincronização de processos

Existem vantagens e desvantagens de operar e sincronizar processos. Agora, você vai entender cada uma delas. Acompanhe!

Vantagens

- Garante consistência e integridade dos dados.
- Evita condições de corrida.

- Apoia o uso eficiente de recursos compartilhados.

Desvantagens

- Adiciona sobrecarga ao sistema.
- Pode levar à degradação do desempenho.
- Aumenta a complexidade do sistema.
- Pode causar impasses, se não for implementada corretamente.

Atividade 1

A sincronização de processos garante, em sistemas operacionais, que múltiplos processos acessem recursos compartilhados de maneira controlada e previsível. Técnicas de sincronização, como semáforos e mutexes, são utilizadas para coordenar a execução de processos e proteger seções críticas do código.

Qual das alternativas a seguir descreve corretamente um aspecto fundamental da sincronização de processos?

A Semáforos são usados exclusivamente para comunicação entre processos, sem proteger seções críticas.

B A sincronização de processos elimina completamente a necessidade de comunicação entre processos.

C Mutexes garantem que múltiplos processos possam acessar simultaneamente uma seção crítica sem conflitos.

D Condições de corrida ocorrem quando dois ou mais processos acessam dados compartilhados de maneira ordenada e previsível.

E A sincronização de processos garante que apenas um processo por vez acesse uma seção crítica, prevenindo condições de corrida.

Parabéns! A alternativa E está correta.

A sincronização previne condições de corrida, que ocorrem quando múltiplos processos tentam acessar e modificar dados compartilhados simultaneamente de maneira desordenada, levando a resultados imprevisíveis. Por meio da sincronização adequada, os sistemas operacionais asseguram a consistência dos dados e a execução correta dos processos, permitindo que os recursos sejam utilizados de forma segura e eficiente.

O conceito de deadlock

Deadlock é uma situação em que processos ficam bloqueados, aguardando recursos uns dos outros. Métodos para lidar com deadlocks incluem prevenção, detecção e recuperação, e ignorância. Cada abordagem tem suas vantagens e desvantagens, e a escolha depende do contexto e dos requisitos do sistema operacional.

Neste vídeo, você vai entender o que é deadlock, suas causas e como evitá-lo em sistemas concorrentes.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Deadlock em sistemas operacionais

Um processo no sistema operacional usa recursos de três maneiras: **solicita um recurso**, **utiliza o recurso** e **libera o recurso**.No deadlock, um conjunto de processos fica bloqueado, pois cada processo está retendo um recurso e aguardando outro, que está sendo retido por outro processo.

Processo aguardando para ser desbloqueado.

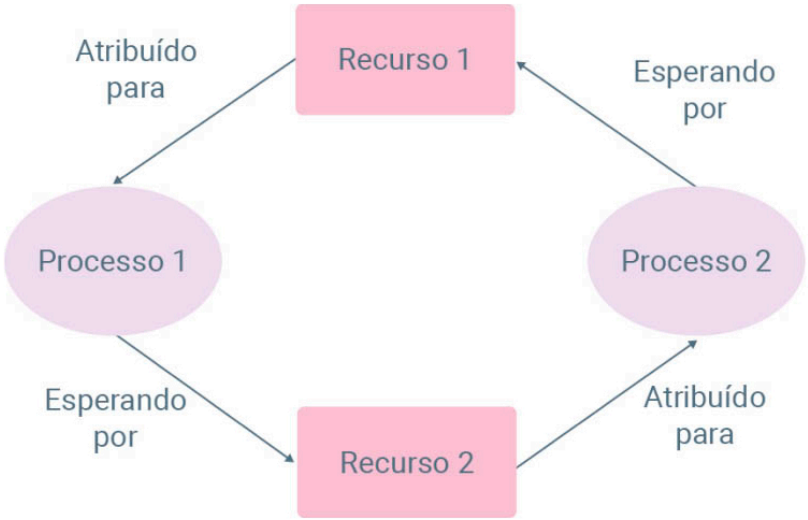
Em um exemplo clássico, temos dois trens em direções opostas no mesmo trilho; nenhum deles pode se mover até que o outro saia do caminho.

Se houver um sistema que tem como recursos duas unidades de disco e dois processos, P0 e P1, que têm, cada um, uma unidade de disco, um precisa do outro. Se ambos bloquearem os recursos, os processos entrarão em deadlock: o processo 1 está mantendo o recurso 1 e aguardando o recurso 2; esse é adquirido pelo processo 2; e esse está aguardando o recurso 1.

Exemplo

P0 executa wait(A) e depois wait(B); P1 executa wait(B) e depois wait(A) (exemplo com semáforos).

Observe um exemplo de deadlock representado na imagem a seguir.



Ocorrência de deadlock.

Note que ambos os processos ficam bloqueados à espera do recurso que o outro tem. Existem algumas condições necessárias para a ocorrência de um deadlock. Observe!

Exclusão mútua

Recursos não são compartilháveis.

Hold and wait

Processos detêm recursos enquanto aguardam outros.

Sem preempção

Recursos não podem ser retirados à força de processos.

Espera circular

Conjunto de processos esperando uns pelos outros de forma circular.

Métodos para lidar com deadlock

A seguir, você vai conhecer as principais abordagens para lidar com deadlocks.

Prevenção de deadlock

Garante que pelo menos uma das quatro condições necessárias para o deadlock não ocorra. Isso pode ser feito de várias maneiras. Observe!

- Eliminar a exclusão mútua.
- Resolver hold and wait.
- Permitir preempção.
- Solução de espera circular.

Detecção e recuperação de deadlock

A seguir, entenda a função de cada uma dessas fases.

Detecção

Verifica se há deadlock no sistema.

Recuperação

Uma vez detectado no sistema, aplica um algoritmo para recuperar do deadlock.

A recuperação do deadlock pode ocorrer a partir de **intervenção manual** ou **recuperação automática do sistema**. Nesse último caso, o sistema pode optar por encerrar processos ou realizar preempção de recursos. A seguir, você vai entender como isso ocorre. Vamos lá!

Encerramento do processo

Você vai entender a seguir as ações no caso de o sistema optar pelo encerramento do processo.

Abortar todos os processos em deadlock

Quebra o ciclo de deadlock, mas pode resultar em perda significativa de dados.

Abortar um processo de cada vez

Seleciona e aborta processos até que o deadlock seja resolvido, mas envolve sobrecarga adicional.

Preempção de recursos

Agora, você vai entender as ações no caso de o sistema optar pela realização de preempção de recursos.

Seleção de vítimas

Escolhe processos para liberar recursos, minimizando o custo da recuperação.

Reversão

Reverte processos para um estado seguro, abortando e reiniciando processos, conforme necessário.

Prevenção da fome (starvation)

Garante que o mesmo processo não seja sempre escolhido como vítima, limitando o número de vezes que um processo pode ser preemptado.

Se os deadlocks forem raros, a abordagem pode ser ignorá-los e reiniciar o sistema quando ocorrerem. Isso é conhecido como **algoritmo de avestruz**, adotado por sistemas como Windows e Unix.

Assim, quando não há deadlock, podemos dizer que há um estado seguro, alcançado se:

- Um processo pode esperar até que um recurso seja liberado por outro processo.
- Todos os recursos solicitados podem ser eventualmente alocados sem causar deadlock.

Atividade 2

Deadlocks são comuns em sistemas de computação em que múltiplos processos competem por recursos limitados. Entender as condições que causam deadlocks e como preveni-los ou resolvê-los mantém a eficiência e a estabilidade do sistema.

Com base no conceito de deadlock, qual das afirmações a seguir descreve corretamente uma condição necessária para que um deadlock ocorra?

A Todos os processos devem ser capazes de compartilhar recursos livremente, sem restrições.

B Os processos podem solicitar e liberar recursos a qualquer momento, sem precisar esperar.

C Cada processo deve manter pelo menos um recurso enquanto aguarda recursos adicionais.

D Recursos podem ser preemptados de um processo e alocados a outro, sem restrições.

E Os processos podem entrar e sair da seção crítica, sem precisar de sincronização.

Parabéns! A alternativa C está correta.

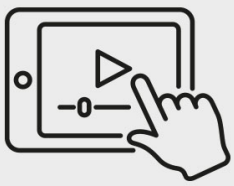
Essa é a condição de **hold and wait**, uma das quatro necessárias para que ocorra um deadlock. Cada processo retém pelo menos um recurso enquanto aguarda recursos adicionais. Isso significa que os processos mantêm os recursos já adquiridos enquanto solicitam novos recursos. Isso pode levar a uma situação em que nenhum processo continua, porque cada um está esperando que outros liberem os recursos necessários. Essa condição, junto com as outras três (exclusão mútua, não preempção e espera circular), forma a base para a ocorrência de deadlocks.

Sincronização de threads com mutex

Pthreads em C no Linux permitem múltiplos fluxos de execução em um processo, compartilhando dados globais e heap. Para sincronização e exclusão mútua, utilizam-se mutexes e variáveis de condição. O exemplo de código demonstra como esses mecanismos são aplicados, a fim de resolver o problema produtor-consumidor, garantindo acesso seguro e eficiente a recursos compartilhados entre threads.

Neste vídeo, você vai compreender como utilizar mutexes para garantir a sincronização adequada entre threads.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Sincronização de threads Posix em C

Threads Posix, conhecidos como Pthreads, fornecem múltiplos fluxos de execução dentro de um processo. Threads compartilham dados globais e heap, necessitando de sincronização, para evitar acessos simultâneos a dados compartilhados, garantindo, assim, a exclusão mútua e a sincronização adequada.

Observe um exemplo que ilustra a necessidade de exclusão mútua.

Exemplo

Considere uma variável saldo_da_conta em um aplicativo bancário. Se dois threads, `thread_verificar_liberação` e `thread_sacar_dinheiro`, acessarem simultaneamente `saldo_da_conta` para deduzir diferentes valores, isso pode resultar em inconsistências.

Para evitar o problema de exclusão mútua, utilizamos mutexes, para garantir que apenas um thread execute a seção crítica de código por vez.

> Em Pthreads, usamos objetos mutex para exclusão mútua. Um mutex é bloqueado no início e desbloqueado no final da seção crítica.

Para **variáveis alocadas dinamicamente**, inicializamos o mutex com `pthread_mutex_init`. As principais chamadas são `pthread_mutex_lock` e `pthread_mutex_unlock`.

As **variáveis de condição** em Pthreads ajudam a gerenciar a sincronização em que múltiplos threads precisam acessar várias instâncias de um recurso. Usamos uma variável de condição, um mutex e um predicado (por exemplo, uma **instância de recurso está disponível**).

> Um thread que libera uma instância de recurso sinaliza a condição, enquanto outro thread que deseja adquirir o recurso espera até que a condição seja sinalizada.

Exemplo de código

A seguir, veja um exemplo simples de programa que demonstra o uso de mutexes e variáveis de condição para resolver o problema produtor-consumidor.

C

Confira as saídas esperadas:

Terminal

Atividade 3

A sincronização de threads evita condições de corrida e garante que múltiplos threads acessem recursos compartilhados de maneira controlada e segura. Um dos principais mecanismos para garantir essa sincronização é o uso de mutexes (exclusão mútua).

Qual das afirmações a seguir descreve corretamente o uso de mutexes para sincronização de threads?

A

Mutexes permitem que múltiplos threads entrem na mesma seção crítica simultaneamente.

B Mutexes garantem que apenas um thread por vez possa acessar uma seção crítica, bloqueando outros threads até que o mutex seja liberado.

C Mutexes são usados apenas para inicializar threads, não para sincronizar a execução de threads.

D Mutexes eliminam a necessidade de qualquer outra forma de sincronização entre threads.

E Mutexes são usados para sinalizar a conclusão de tarefas entre threads, similar a variáveis de condição.

Parabéns! A alternativa B está correta.

Mutexes asseguram exclusão mútua, isto é, apenas um thread por vez pode acessar uma seção crítica do código. Quando adquire um mutex, um thread bloqueia outros threads na mesma seção crítica até que o mutex seja liberado. Esse mecanismo previne condições de corrida, em que múltiplos threads tentam modificar dados compartilhados simultaneamente, resultando em comportamento imprevisível e erros. Assim, mutexes garantem que o acesso aos recursos compartilhados seja feito de maneira controlada e segura.

Demonstração prática da sincronização de threads

A sincronização de threads garante que múltiplos threads acessem recursos compartilhados de maneira segura e ordenada. Utilizar mutexes (exclusão mútua) é uma prática comum para proteger seções críticas do código em que variáveis compartilhadas são modificadas. Condições de corrida podem ocorrer sem a devida sincronização, resultando em comportamentos imprevisíveis e dados corrompidos.

Vamos ver a seguir como utilizar mutexes para sincronizar threads em uma aplicação C, garantindo que apenas um thread por vez execute uma seção crítica do código.

Neste vídeo, você vai aprender a implementar a sincronização de threads em um programa.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Roteiro na prática

Você está desenvolvendo uma aplicação multithreaded que simula operações bancárias. Múltiplos threads representam clientes que realizam saques de uma conta compartilhada. Sem a devida sincronização, dois ou mais threads podem tentar acessar e modificar o saldo da conta ao mesmo tempo, resultando em saldo incorreto. Seu objetivo é implementar a sincronização adequada utilizando mutexes, para evitar condições de corrida e garantir a consistência dos dados.

Você implementará uma aplicação C que utiliza mutexes para proteger a seção crítica na qual o saldo da conta é modificado. Confira a resolução!

c

O código pode ser explicado da seguinte maneira:

- **Biblioteca**: inclui bibliotecas necessárias para manipulação de threads e operações de entrada-saída.
- **Definição das constantes**: indicação do número de threads e do saldo inicial da conta.
- **Inicialização do mutex**: declara e inicializa um mutex para garantir exclusão mútua no acesso ao saldo da conta.
- **Função `withdraw`**: executada por cada thread para realizar um saque.
    - Obtém a quantia a ser sacada a partir do argumento.
    - Bloqueia o mutex, para garantir que apenas um thread acesse o saldo da conta por vez.
    - Verifica se há saldo suficiente para realizar o saque.
    - Atualiza e exibe o novo saldo ou uma mensagem de erro em caso de fundos insuficientes.
    - Desbloqueia o mutex.
- **Função `main`**:
    - Declara um array para armazenar os identificadores dos threads e outro para as quantias sacadas.
    - Cria os threads, passando as quantias correspondentes como argumentos.
    - Espera que todos os threads completem suas operações usando pthread_join.
    - Exibe o saldo final da conta.
    - Destrói o mutex.

Faça você mesmo!

Considere a aplicação multithreaded indicada anteriormente, que utiliza mutexes para sincronizar acessos ao saldo de uma conta bancária. Suponha que você queira modificar o código para permitir que múltiplos threads possam verificar o saldo ao mesmo tempo, mas ainda garantir que apenas um thread possa modificar o saldo por vez.

Qual ação deve ser tomada para implementar essa funcionalidade?

A Remover o mutex, ao verificar o saldo, e adicionar um novo mutex, para proteger apenas as operações de saque.

B Adicionar um novo mutex exclusivo para a operação de verificação de saldo, sem alterar o mutex atual.

C Usar uma variável global sem sincronização para armazenar o saldo, permitindo acesso simultâneo.

D Dividir o mutex atual em dois, um para leitura e outro para escrita, garantindo exclusão mútua durante as operações de saque.

E Duplicar o código da seção crítica para cada thread, garantindo que cada um tenha sua própria versão do saldo.

Parabéns! A alternativa D está correta.

Essa ação permite que múltiplos threads leiam o saldo da conta simultaneamente, sem interferir uns nos outros, e que apenas um thread modifique o saldo por vez. Isso possibilita maior concorrência para operações de leitura, mantendo a consistência durante operações de escrita.

O que você aprendeu neste conteúdo?

- O uso da função fork na criação de novos processos.
- O comportamento do processo-pai e do processo-filho após a chamada de fork.
- A sincronização de threads com mutex.

- A importância da sincronização, para evitar condições de corrida.
- O uso de mutexes, para garantir que apenas um thread acesse uma seção crítica por vez.
- A criação e o gerenciamento de múltiplos threads, utilizando a biblioteca Pthreads.
- O modo de passar argumentos para threads.
- O uso de pthread_join, para sincronizar a conclusão dos threads.
- As condições necessárias para que ocorra um deadlock.
- Os métodos para prevenir, evitar, detectar e recuperar deadlocks.

Explore +

Veja no YouTube o vídeo **Sabor de treinamento**, do instrutor da Linux Foundation John Bonesio, e entenda como o Linux transita do modo usuário para o modo kernel com uma chamada do sistema.

Busque pelo **OS Simulator**, que fornece processos multithreaded e suporta várias simulações de CPU. Funciona com o **CPU Simulator** e gerencia vários processos e memória virtual usando diferentes mecanismos de agendamento de processos.

Procure informações sobre o **Projeto GNU**, cuja filosofia é o software livre, e entenda melhor as quatro liberdades essenciais dos usuários para: executar um programa; estudar e mudar seu código-fonte; redistribuir cópias exatas; e distribuir versões modificadas.

Referências

GEEKSFORGEEKS. **Fork function call**. GeeksforGeeks, 24 set. 2023. Consultado na internet em: 19 maio 2024.

GEEKSFORGEEKS. **Introduction of deadlock in operating system**. Consultado na internet em: 19 maio 2024.

GEEKSFORGEEKS. **Introduction of process synchronization**. Consultado na internet em: 19 maio 2024.

GEEKSFORGEEKS. **Multithreading models in process management**. Consultado na internet em: 19 maio 2024.

GUDABAYEV, T. **Introduction to PThreads**. Tamerlan.dev, 28 fev. 2022. Consultado na internet em: 19 maio 2024.

GUDABAYEV, T. **What are threads?** Tamerlan.dev, 21 fev. 2022. Consultado na internet em: 19 maio 2024.

JOHRI, K. **POSIX threads synchronization in C**. SoftPrayog, 29 fev. 2024. Consultado na internet em: 19 maio 2024.

KRISHNAMOORTHY, M. S. **Sockets tutorial**. Computer Science, 14 set. 1998. Consultado na internet em: 18 maio 2024.

KRZYZANOWSKI, P. **C tutorial**: playing with processes. PK.org, 28 mar. 2019. Consultado na internet em: 18 maio 2024.

OPERATING system: processes. Tutorials Point, s. d. Consultado na internet em: 18 maio 2024.

PROCESS synchronization in C/C++. Tutorials Point, s. d. Consultado na internet em: 18 maio 2024.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Fundamentos de sistemas operacionais**: princípios básicos. New Jersey: John Wiley & Sons, Inc., 2011.

STEVENS, W. R.; FENNER, B.; RUDOFF, A. M. **Programação de rede Unix [BV:MB]**. 3. ed. Porto Alegre: Bookman, 2008.

Relatar problema