EAD

EDUCAÇÃO A DISTÂNCIA

UNIDADE III

Infraestrutura Computacional

Profa. Sandra Bozolan

Gerenciamento de processos

- **O que são processos?**

Definição

Processos são unidades de trabalho em execução dentro de um sistema.

Composição

Um sistema consiste em processos do sistema (código do SO) e processos de usuário.

Execução

Podem ser executados concorrentemente através de multiplexação em CPUs.

Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, um processo e como funciona o gerenciamento de processos.

Tipos de processos

Processos do Sistema

- Executam código do sistema operacional. São responsáveis por funções críticas de baixo nível.

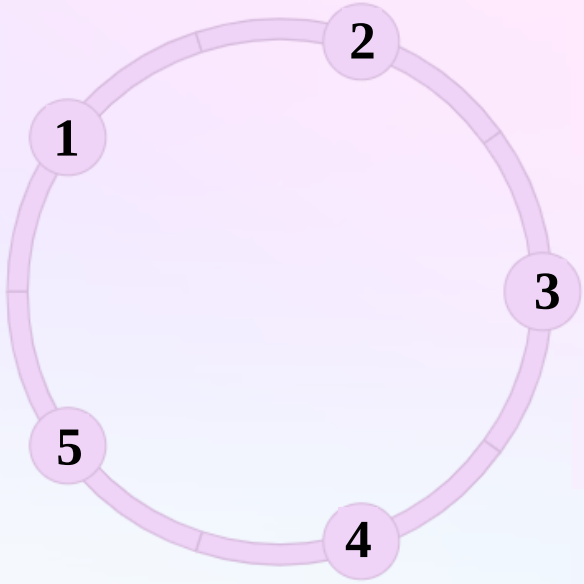
Processos de Usuário

- Executam código de aplicativos dos usuários. Têm menor prioridade e privilégios limitados.

Estados de um processo

Novo

Processo foi criado, mas ainda não foi admitido pelo escalonador.



Pronto

Processo está pronto para executar, aguardando CPU.

Executando

Instruções estão sendo executadas pelo processador.

Bloqueado

Processo aguarda algum evento ou recurso para continuar.

Terminado

Processo concluiu sua execução ou foi abortado.

Fonte: Flux Fast 1.1, 2025 – Álgebra Booleana.

Principais responsabilidades do SO

1. **Escalonamento:** Executar o scheduling de processos e threads nas CPUs disponíveis.
2. **Gerenciamento de Ciclo de Vida:** Criar e excluir processos de usuário e do sistema operacional.
3. **Controle de Execução:** Suspender e retomar processos conforme necessário.
4. **Comunicação:** Fornecer mecanismos de sincronização e comunicação entre processos.

Fonte: Flux Fast 1.1, – Principais responsabilidades do SO.

Escalonamento de processos

Longo Prazo: Controla quais processos são admitidos para competir pelo processador.

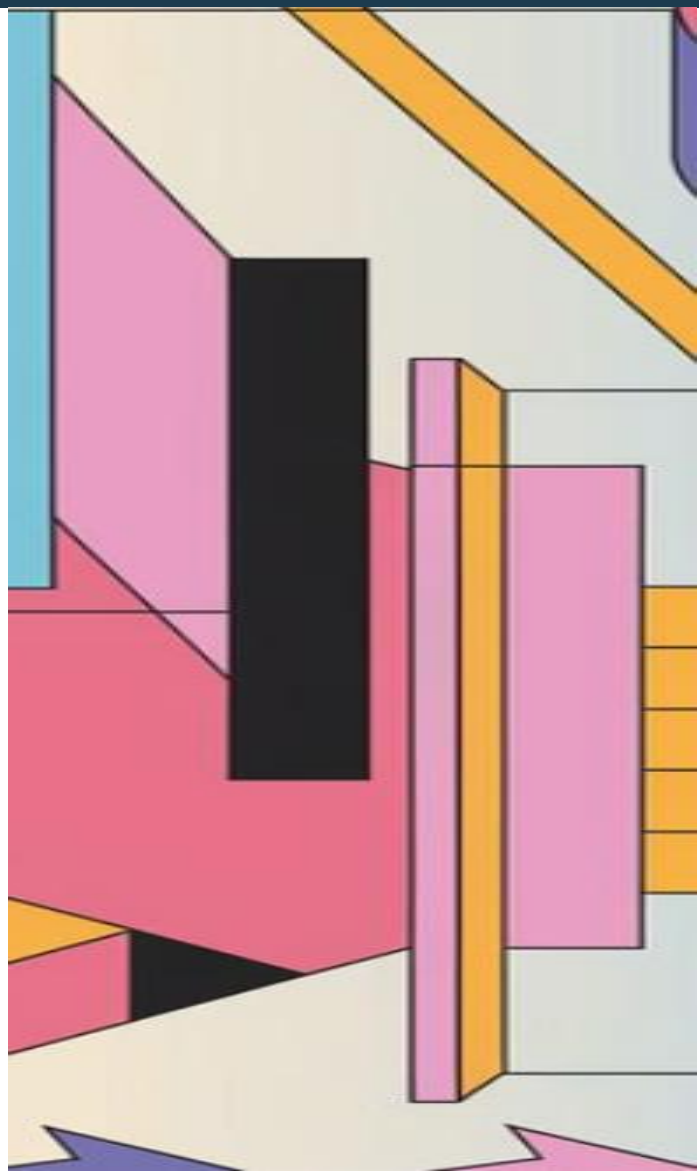
Curto Prazo: Decide qual processo pronto será executado em seguida.

1

2

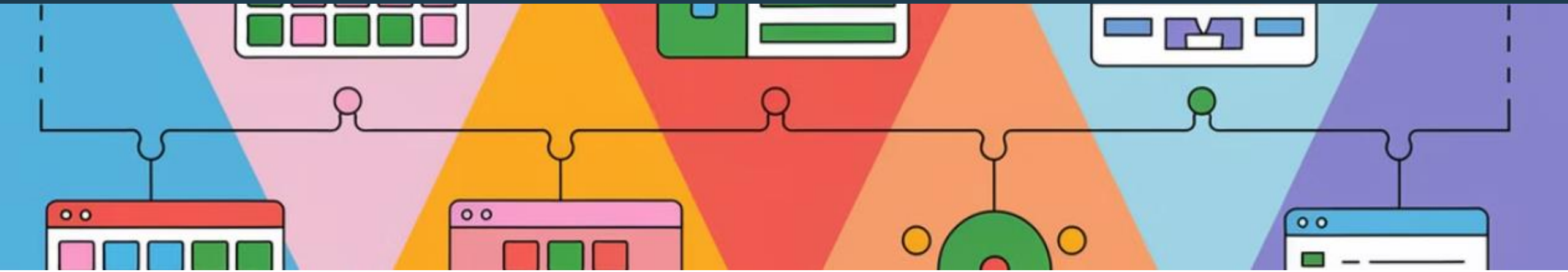
3

Médio Prazo: Gerencia a suspensão e reativação de processos.



Fonte: Flux Fast 1.1, 2025 – Figura abstrata do Escalonamento de Processos.

Algoritmos de escalonamento

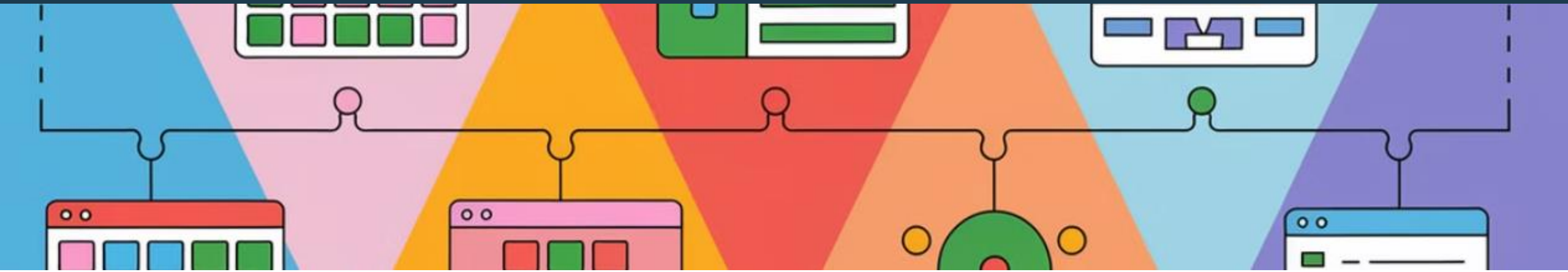


FCFS (First-Come, First-Served): Processos são executados na ordem de chegada. Simples, mas pouco eficiente.

SJF (Shortest Job First): Prioriza processos com menor tempo de execução estimado.

Fonte: Flux Fast 1.1, 2025 – Algoritmo de Escalonamento.

Algoritmos de escalonamento



Round Robin: Cada processo recebe um quantum de tempo por vez. Favorece equidade.

Prioridades: Atribui prioridades aos processos. Evita inanição com envelhecimento.

Fonte: Flux Fast 1.1, 2025 – Algoritmo de Escalonamento.

Criação de um processo

Inicialização do Sistema: O SO cria processos iniciais durante a inicialização do sistema.

Requisição de Usuário: Usuário inicia um aplicativo através da interface do sistema.

Chamada de Sistema: Um processo em execução solicita a criação de um novo processo.

Operação em Lote: Inicialização automática de processos programados pelo sistema.

Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, a criação de um processo.

Comunicação entre processos (IPC)

- **Memória Compartilhada:** Processos acessam uma região comum de memória.
- **Troca de Mensagens:** Processos enviam e recebem mensagens via sistema operacional.
- **Pipes e FIFOs:** Canais unidirecionais para comunicação sequencial de dados.
- **Sockets:** Permitem comunicação entre processos em máquinas diferentes.

Sincronização de processos

Semáforos

- Variáveis especiais que controlam acesso a recursos compartilhados.

Monitores

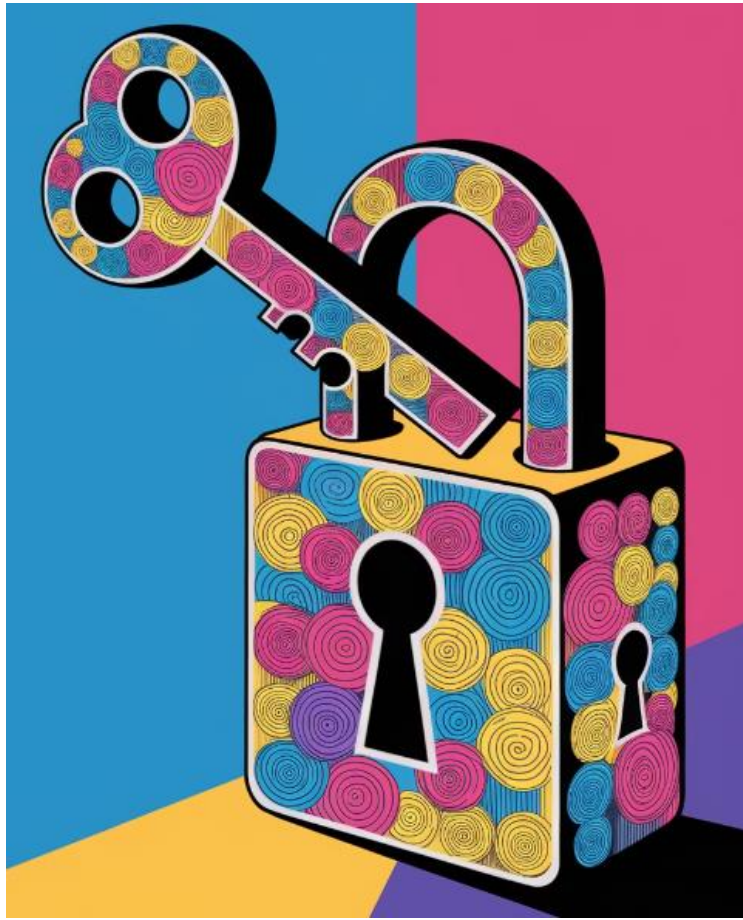
- Estruturas de alto nível que encapsulam dados e procedimentos.

Mutex

- Mecanismo de exclusão mútua para regiões críticas de código.

Variáveis de Condição

- Permitem que processos aguardem até que uma condição seja verdadeira.



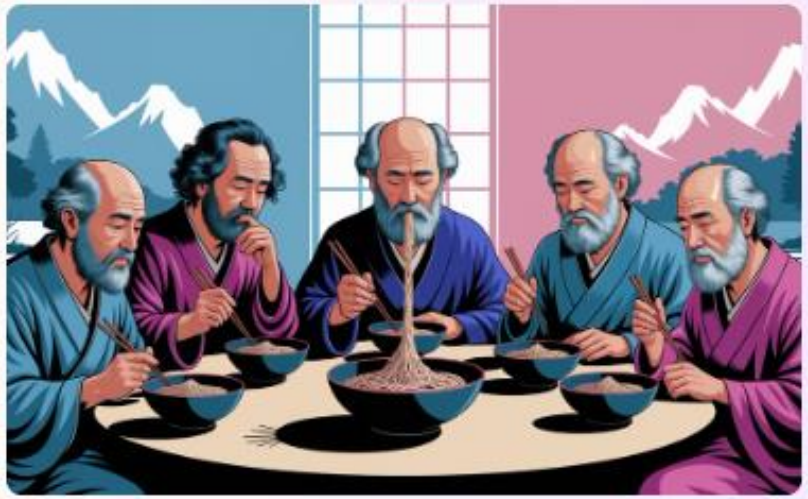
Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, a sincronização de um processo.

Problema da seção crítica

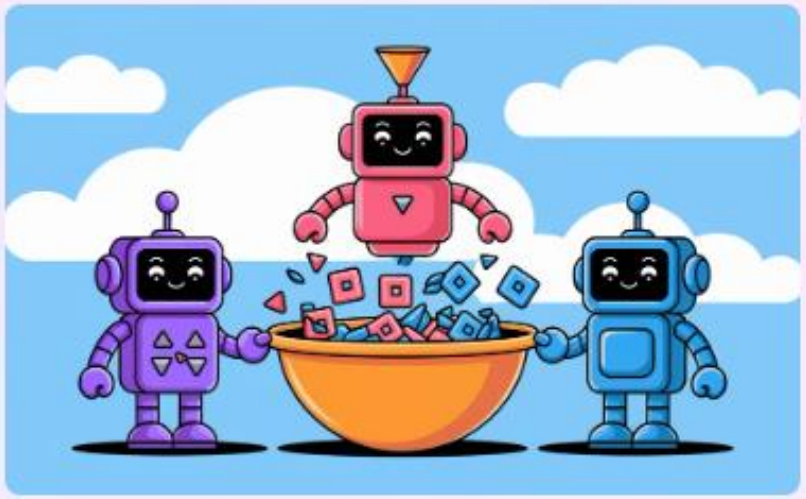
1. **Exclusão Mútua**
- Apenas um processo na seção crítica.
2. **Progresso**
- Seleção sem interferência de processos não interessados.
3. **Espera Limitada**
- Limite de tempo para entrada na seção crítica.

A seção crítica é uma região de código onde processos acessam recursos compartilhados. Uma solução adequada deve satisfazer essas três condições essenciais.

Problemas clássicos de sincronização



Jantar dos Filósofos
Modelo para alocação de recursos e prevenção de deadlocks.



Produtor-Consumidor
Coordenação entre processos que produzem e consomem dados.

Leitores-Escritores
Controle de acesso a dados compartilhados com diferentes privilégios.

Fonte: Flux Fast 1.1, 2025 – As imagens representam, de forma figurativa, os exemplos clássicos de sincronização de processos.

Interatividade

Sobre o processo de escalonamento (scheduling) em sistemas operacionais, analise as afirmações a seguir e assinale a alternativa correta.

I. O escalonamento de processos visa otimizar o uso da CPU, definindo qual processo deve ser executado a cada momento.

II. Em sistemas que utilizam prioridade estática, cada processo sempre mantém a mesma prioridade durante todo o seu tempo de execução.

III. No escalonamento preemptivo, a troca de contexto só ocorre quando o processo termina sua execução.

Está correto o que se afirma em:

a) I, apenas.

b) I e II, apenas.

c) II e III, apenas.

d) I e III, apenas.

e) I, II e III.

Resposta

Sobre o processo de escalonamento (scheduling) em sistemas operacionais, analise as afirmações a seguir e assinale a alternativa correta.

I. O escalonamento de processos visa otimizar o uso da CPU, definindo qual processo deve ser executado a cada momento.

II. Em sistemas que utilizam prioridade estática, cada processo sempre mantém a mesma prioridade durante todo o seu tempo de execução.

III. No escalonamento preemptivo, a troca de contexto só ocorre quando o processo termina sua execução.

Está correto o que se afirma em:

a) I, apenas.

b) I e II, apenas.

c) II e III, apenas.

d) I e III, apenas.

e) I, II e III.

Implementação de sincronização em sistemas operacionais modernos

- A sincronização em sistemas operacionais modernos é fundamental para garantir o funcionamento adequado de processos concorrentes. Este tema abrange desde primitivas básicas de sincronização até boas práticas de programação, essenciais para desenvolvedores e administradores de sistemas. Exploraremos os conceitos fundamentais, mecanismos de implementação e estratégias para evitar problemas comuns, como deadlocks e condições de corrida, fornecendo uma visão abrangente sobre como a sincronização é implementada nos sistemas operacionais atuais.

Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, a implementação de sincronizaçao em sistemas operacionais modernos.

Conceitos fundamentais de concorrência

Execução Simultânea

- A concorrência permite que múltiplos processos e threads sejam executados simultaneamente, otimizando o uso dos recursos do sistema e melhorando significativamente a responsividade.

Desafios Inerentes

- A execução concorrente traz desafios como deadlocks, condições de corrida e starvation, que exigem mecanismos específicos de sincronização para garantir a segurança e a corretude do sistema.

Importância Crescente

- Com o avanço da tecnologia e o aumento da complexidade dos sistemas, a importância da concorrência e da sincronização continua a crescer exponencialmente.

Primitivas de sincronização

Mutexes

- Mecanismos de exclusão mútua que garantem que apenas um processo ou thread possa acessar um recurso compartilhado por vez, evitando conflitos de acesso simultâneo.

Semáforos

- Estruturas que controlam o acesso a recursos limitados, permitindo que um número específico de processos acessem simultaneamente um conjunto de recursos compartilhados.

Variáveis de Condição

- Permitem que threads aguardem até que uma condição específica seja satisfeita, facilitando a coordenação entre múltiplos processos que dependem de eventos específicos.

Boas práticas de programação concorrente

A programação concorrente:

- É um desafio, pois exige que os programadores considerem os aspectos de sincronização, a exclusão mútua e a comunicação entre processos. Para garantir a segurança e a eficiência de programas concorrentes, é importante seguir algumas boas práticas.

Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, quais são as boas práticas de programação concorrente.

Boas práticas de programação concorrente

Minimizar o tamanho da seção crítica:

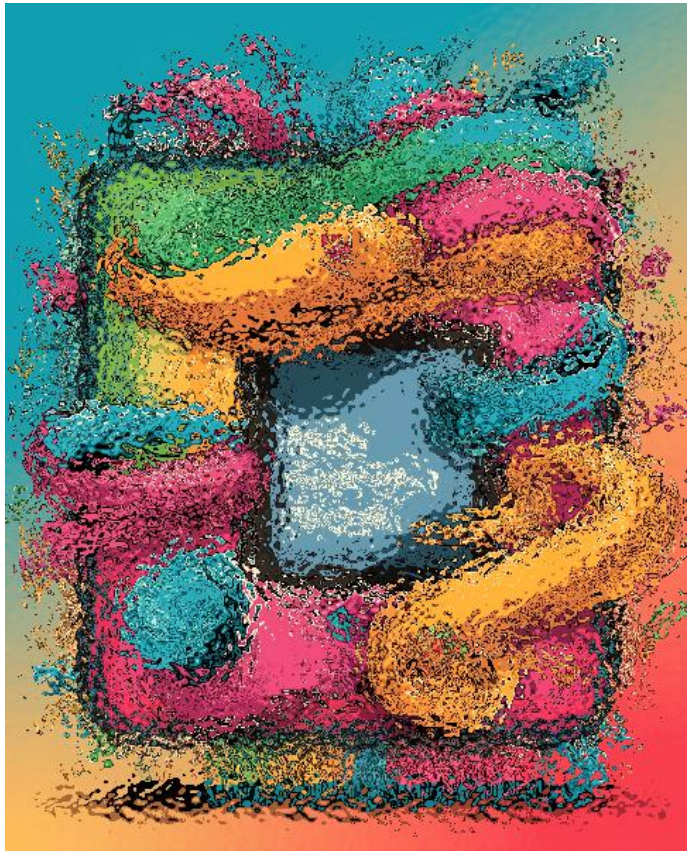
- A seção crítica é a parte do código que acessa recursos compartilhados. É importante minimizar o tamanho da seção crítica para reduzir a quantidade de tempo que um processo bloqueia outros processos.

Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, quais são as boas práticas de programação concorrente.

Boas práticas de programação concorrente

Usar primitivas de sincronização apropriadas:

- Escolher as primitivas de sincronização corretas é fundamental para garantir a segurança e a eficiência do código. Por exemplo, se apenas um processo precisa acessar um recurso compartilhado por vez, um mutex é a melhor escolha. Se vários processos precisam esperar por um evento, uma variável de condição é mais adequada.



Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, quais são as boas práticas de programação concorrente.

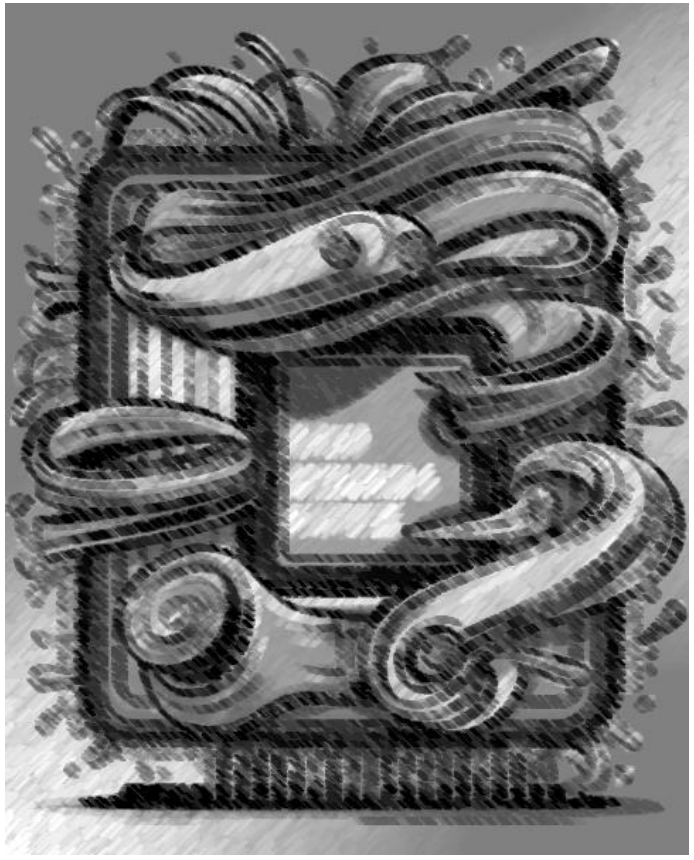
Boas práticas de programação concorrente

- **Evitar condições de corrida:** condições de corrida ocorrem quando o resultado de um processo depende da ordem em que os processos acessam recursos compartilhados. É importante evitar condições de corrida usando primitivas de sincronização para garantir que as operações sejam atômicas.

Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, quais são as boas práticas de programação concorrente.

Boas práticas de programação concorrente

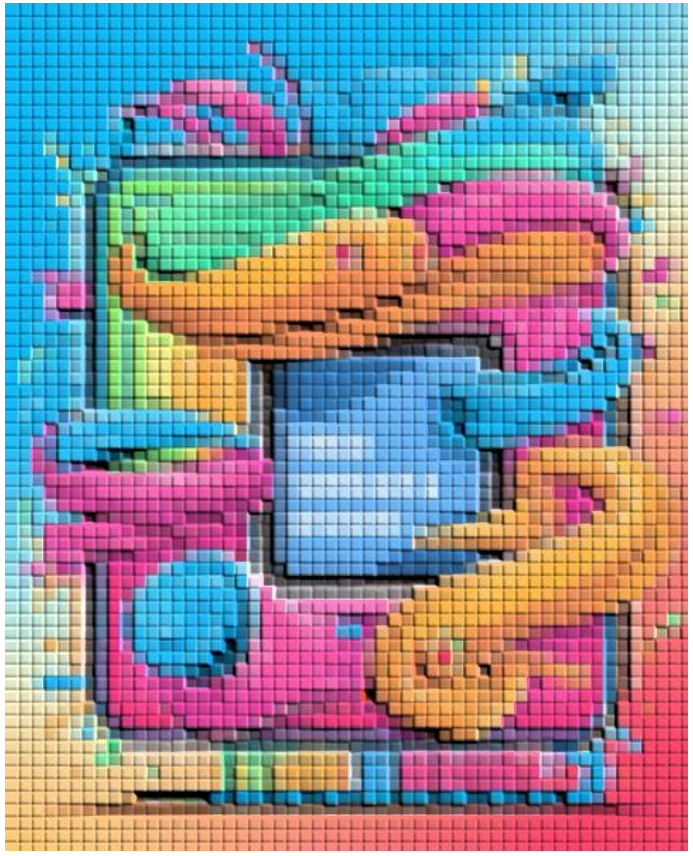
- **Detectar e resolver deadlocks:** deadlocks podem ocorrer quando dois ou mais processos ficam presos em um estado de espera, cada um aguardando um recurso que o outro possui. É importante implementar mecanismos para detectar e resolver deadlocks.



Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, quais são as boas práticas de programação concorrente.

Boas práticas de programação concorrente

- **Testar e depurar código concorrente:** é essencial testar e depurar cuidadosamente o código concorrente para garantir que ele funcione corretamente em vários cenários de concorrência.



Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, quais são as boas práticas de programação concorrente.

Bibliotecas e frameworks para programação concorrente

Abstrações de Alto Nível

- Fornecem interfaces simplificadas que ocultam a complexidade dos mecanismos de sincronização de baixo nível, permitindo que os desenvolvedores se concentrem na lógica de negócios em vez de detalhes de implementação.

Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, as bibliotecas e frameworks para programação concorrente.

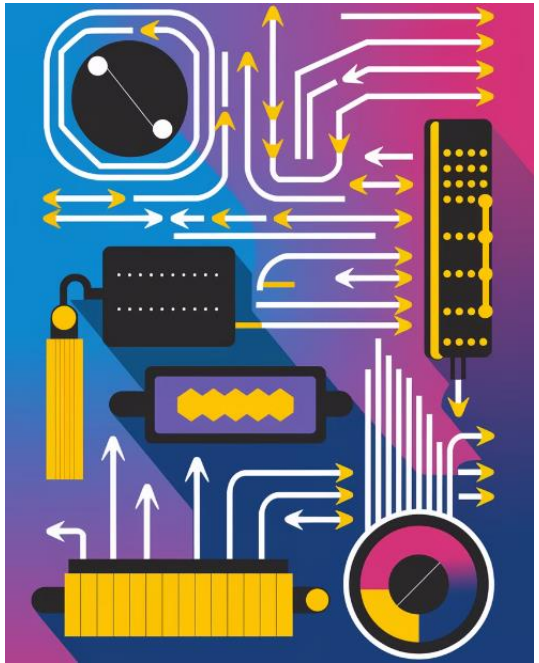
Bibliotecas e frameworks para programação concorrente

Padrões de Concorrência

- Implementam soluções padronizadas para problemas comuns de concorrência, como pools de threads, filas de trabalho e barreiras de sincronização, reduzindo a necessidade de reinventar soluções.

Garantias de Segurança

- Oferecem mecanismos para evitar condições de corrida e deadlocks automaticamente, através de verificações em tempo de compilação ou execução, aumentando a robustez dos programas concorrentes.



Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, as bibliotecas e frameworks para programação concorrente.

Evitando condições de corrida

- **Identificação de Recursos Compartilhados:** Mapear todos os recursos que podem ser acessados por múltiplos processos simultaneamente, incluindo variáveis globais, arquivos e estruturas de dados compartilhadas.
- **Implementação de Proteções:** Aplicar mecanismos de sincronização apropriados para garantir acesso exclusivo ou controlado aos recursos compartilhados, utilizando mutexes, semáforos ou outras primitivas.
- **Verificação de Atomicidade:** Garantir que operações críticas sejam executadas como unidades indivisíveis, utilizando instruções atômicas do processador ou mecanismos de sincronização para operações complexas.

Testes de Concorrência

- Realizar testes específicos para detectar condições de corrida, utilizando ferramentas de análise estática e dinâmica, além de testes de estresse com múltiplas threads.

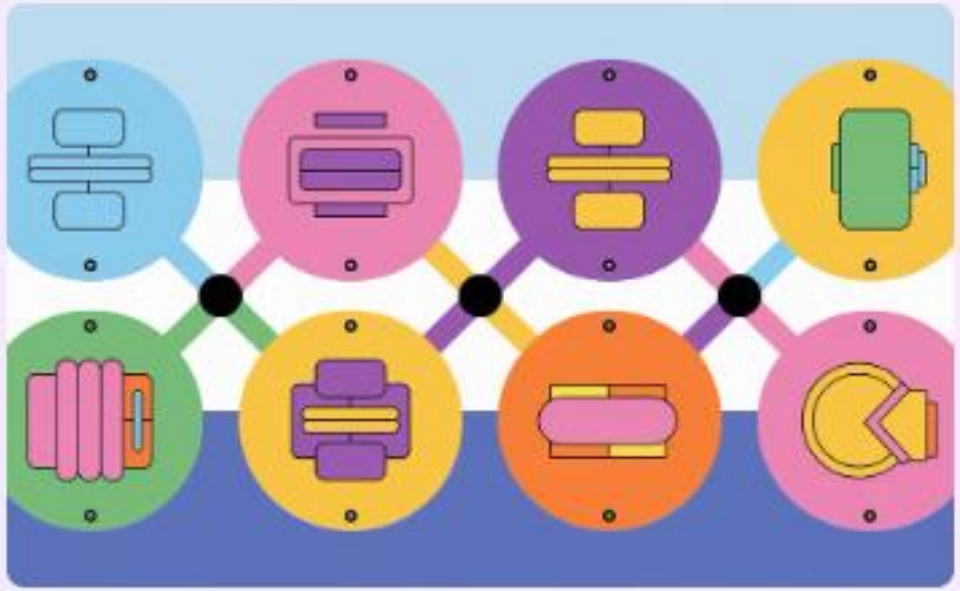
Evitando condições de corrida

Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, as bibliotecas e frameworks para programação concorrente.

Algoritmos de Detecção

- Sistemas modernos implementam algoritmos que analisam o grafo de alocação de recursos para identificar ciclos que indicam deadlocks. Estes algoritmos podem ser executados periodicamente ou quando o sistema detecta uma possível situação de impasse.

Evitando condições de corrida



Estratégias de Prevenção

- Técnicas como ordenação de recursos, alocação completa e verificação de segurança são implementadas para evitar que deadlocks ocorram, garantindo que o sistema nunca entre em estados que possam levar a impasses.

Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, as bibliotecas e frameworks para programação concorrente.

Evitando condições de corrida

Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, as bibliotecas e frameworks para programação concorrente.

Mecanismos de Recuperação

- Quando um deadlock é detectado, o sistema pode implementar estratégias como terminação de processos, preempção de recursos ou rollback de transações para restaurar o sistema a um estado operacional.

Teste e depuração de código concorrente

Algoritmos de Detecção

- O teste e a depuração de código concorrente exigem ferramentas especializadas que podem identificar problemas difíceis de reproduzir. Analisadores estáticos verificam o código antes da execução, enquanto ferramentas de análise dinâmica monitoram o comportamento em tempo real para detectar condições de corrida, deadlocks e outros problemas de concorrência.
- Técnicas como execução determinística, injeção de falhas e testes de estresse são essenciais para garantir a robustez de sistemas concorrentes em diferentes cenários de carga e condições de execução.

Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, as bibliotecas e frameworks para programação concorrente.

Interatividade

Qual mecanismo de sincronização é frequentemente usado para garantir acesso exclusivo a uma seção crítica?

a) Variáveis globais sem qualquer proteção.
b) Mutex (Mutual Exclusion).
c) Aumento do número de threads para evitar acesso concorrente.
d) Desativação de interrupções do sistema operacional.
e) Uso de ponteiros inteligentes (smart pointers).

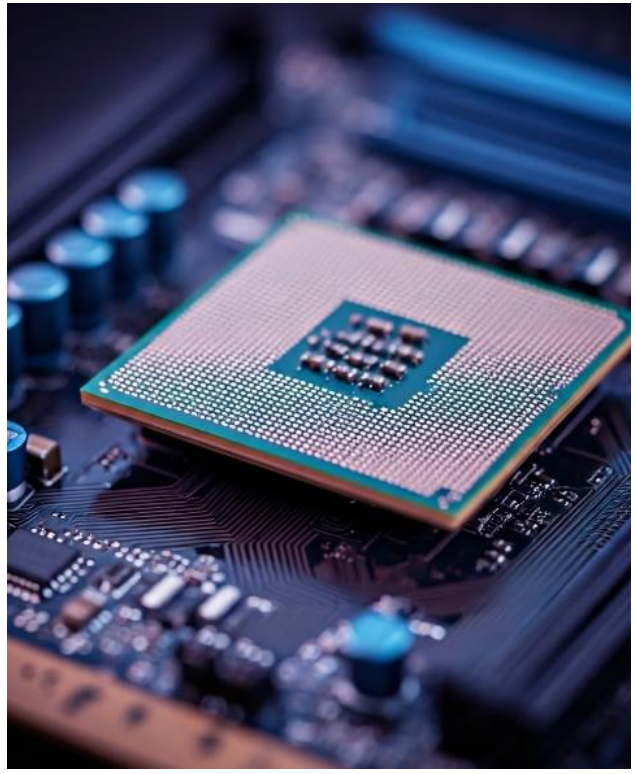
Resposta

Qual mecanismo de sincronização é frequentemente usado para garantir acesso exclusivo a uma seção crítica?

a) Variáveis globais sem qualquer proteção.
b) Mutex (Mutual Exclusion).
c) Aumento do número de threads para evitar acesso concorrente.
d) Desativação de interrupções do sistema operacional.
e) Uso de ponteiros inteligentes (smart pointers).

Gerenciamento de processador em sistemas Linux

- O gerenciamento de processador em sistemas Linux lida com processos, as chamadas de sistema fundamentais e os mecanismos que tornam o Linux único em comparação com outros sistemas UNIX.
- Examinaremos o modelo de processos do Linux, desde sua concepção até implementação, incluindo os aspectos técnicos que são essenciais para estudantes e profissionais de ciência da computação compreenderem o funcionamento interno dos sistemas operacionais modernos.



Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, o gerenciamento do processador.

Processos como unidades fundamentais

Definição de Processo

- Um processo é o contexto básico dentro do qual toda atividade solicitada pelos usuários é atendida no sistema operacional. Ele representa uma unidade de execução com recursos alocados pelo sistema operacional.

Compatibilidade UNIX

- Para manter compatibilidade com outros sistemas UNIX, o Linux implementa um modelo de processo semelhante, porém com diferenças em aspectos-chave que melhoram o desempenho e a flexibilidade.

Contexto de Execução

- Cada processo mantém seu próprio contexto de execução, incluindo registradores, contador de programa, espaço de endereçamento e recursos do sistema alocados exclusivamente para ele.

O modelo de processos Linux

1. Separação de Conceitos

- O Linux implementa uma separação clara entre a criação de processos e a execução de programas, diferente de outros sistemas operacionais que combinam essas operações.

2. Flexibilidade Operacional

- Esta separação fornece maior flexibilidade aos desenvolvedores, permitindo a implementação de estruturas complexas de aplicações e serviços com controle preciso sobre o ambiente de execução.

3. Modelo Fork-Exec

- O paradigma consagrado de fork() para criar processos e exec() para executar programas forma a base do gerenciamento de processos no Linux, seguindo a tradição UNIX.

A chamada de sistema fork()

Definição

- A chamada de sistema fork() é responsável por criar um novo processo no Linux, duplicando o processo chamador (processo pai) para criar um processo filho quase idêntico.

Duplicação de Recursos

- Durante o fork(), o kernel cria uma cópia do espaço de endereçamento do processo pai, incluindo código, dados, pilha e heap, tornando-os disponíveis ao processo filho.

Identificação

- O valor de retorno de fork() permite identificar se o código está sendo executado no contexto do processo pai (retorna o PID do filho) ou filho (retorna 0), possibilitando comportamentos distintos.

Particularidades do fork() no Linux

Copy-on-Write

- O Linux implementa o mecanismo Copy-on-Write (CoW) que, em vez de duplicar imediatamente todo o espaço de endereçamento, apenas marca as páginas como compartilhadas e só as copia quando uma delas é modificada.

Otimização de Desempenho

- Esta técnica reduz significativamente o overhead de criação de processos, tornando a operação fork() muito mais eficiente em termos de uso de memória e tempo de execução.

Identidade do Processo

- Após o fork(), o processo filho recebe um novo PID (Process ID), mas herda quase todas as características do pai, incluindo descritores de arquivos abertos e variáveis de ambiente.

A chamada de sistema exec()

Função Principal

- A chamada exec() substitui o programa em execução por um novo programa dentro do mesmo processo, mantendo o PID e recursos alocados, mas carregando um novo executável na memória.

Carregamento de Binários

- Durante a execução de exec(), o kernel limpa o espaço de endereçamento atual do processo e carrega o novo programa executável, configurando seus segmentos de texto, dados e pilha.

Estado do Processo

- Após a chamada bem-sucedida a exec(), a execução começa no ponto de entrada do novo programa (geralmente main()), com novos registradores e pilha, mas mantendo recursos como arquivos abertos.

Separação fork() e exec(): Vantagens

Flexibilidade de Configuração

- A separação permite que o processo filho configure seu ambiente antes de executar o novo programa, ajustando descritores de arquivo, sinais, privilégios ou outras características.

Redirecionamentos

- Facilita implementações de redirecionamento de E/S como os utilizados em shells, permitindo manipular stdin/stdout antes de chamar exec() para o comando desejado.

Controle de Execução

- Possibilita que o processo pai mantenha controle sobre o filho, implementando monitores, wrappers ou sistemas de logging que precisam intervir antes da execução do programa final.

Implementação do exec() no Linux

Fase da Execução	Ações do Kernel
Validação	Verifica permissões do arquivo, formato do binário e compatibilidade com a arquitetura.
Liberação de Recursos	Libera estruturas de memória do programa anterior, mantendo descritores abertos.
Carregamento	Carrega seções do executável (.text, .data, etc.) em novas áreas de memória.
Preparação da Pilha	Configura a pilha com argumentos (argv) e variáveis de ambiente (envp).
Execução	Transfere controle para o ponto de entrada do novo programa.

Fonte: autoria própria.

Threads vs. Processos no Linux

- No Linux, threads são implementadas usando a mesma estrutura de processos, porém, com recursos compartilhados. A chamada de sistema clone() permite criar threads como processos leves, onde diferentes níveis de compartilhamento de recursos podem ser especificados através de flags.
- Threads compartilham o mesmo espaço de endereçamento virtual, arquivos abertos e outros recursos, enquanto processos possuem espaços independentes. Isso torna a troca de contexto entre threads significativamente mais rápida do que entre processos, mas introduz desafios de sincronização e proteção de dados compartilhados.

A chamada de sistema clone()

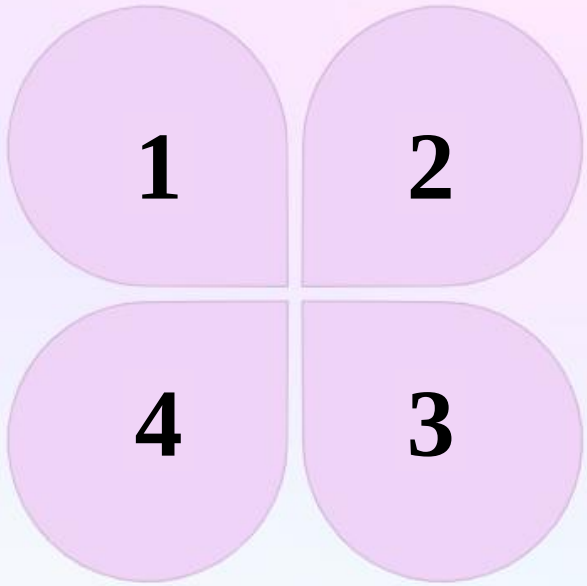
1. **Controle Total:** Oferece controle granular sobre compartilhamento.
2. **Flags de Compartilhamento:** Especifica recursos a serem compartilhados.
3. **Implementação Flexível:** Base para bibliotecas de threads.

- A chamada clone() é mais flexível que fork(), permitindo especificar exatamente quais recursos serão compartilhados entre o processo original e o novo. Com flags como CLONE_VM (compartilha memória virtual), CLONE_FS (compartilha informações do sistema de arquivos), CLONE_FILES (compartilha descritores) e CLONE_SIGHAND (compartilha manipuladores de sinais), é possível criar desde threads completas até processos quase independentes.
    - Esta flexibilidade torna clone() a base para implementações de bibliotecas de threads como a POSIX Threads (pthreads), possibilitando diferentes modelos de concorrência adaptados às necessidades específicas das aplicações.

Estados de processos no Linux

Running

Processo está executando ou pronto para executar na CPU. Pode estar usando efetivamente o processador ou aguardando na fila do escalonador.



Waiting (Blocked)

Processo está esperando por algum evento ou recurso, como operação de I/O, sinal ou temporizador. Não pode ser executado até que o evento ocorra.

Zombie

Processo terminou sua execução, mas mantém entrada na tabela de processos até que o processo pai obtenha seus status de saída.

Stopped

Processo foi parado, geralmente por um sinal como SIGSTOP. Pode ser retomado posteriormente com outro sinal (SIGCONT).

Fonte: autoria própria.

Processos Zombie e Órfãos

- **Processos Zombie:** Ocorre quando um processo filho termina sua execução, mas seu processo pai não chama wait() para coletar seu status de saída.

- **Processos Órfãos:** Quando um processo pai termina antes de seus filhos, estes se tornam órfãos e são "adotados" pelo processo init (PID 1) ou systemd nos sistemas modernos.

Coleta de Processos:

- As chamadas wait(), waitpid() e suas variantes permitem que processos pais coletem informações sobre o término de seus filhos, liberando as entradas da tabela de processos e evitando a formação de zombies.

Chamadas de sistema para gerenciamento de processos

Criação e Execução

- fork() – Cria um novo processo.
- clone() – Cria processo com controle granular.
- exec*() – Família de funções para execução.
- posix_spaw() – Combinação otimizada de fork/exec.

Monitoramento

- wait() – Aguarda término de qualquer filho.
- waitpid() – Aguarda término de filho específico.
- waitid() – Versão mais flexível de espera.
- getpid(), getppid() – Obtém IDs de processos.

Controle

- kill() – Envia sinal para processo.
- setpriority() – Ajusta prioridade de escalonamento.
- Sched_setaffinity() – Define afinidade CPU.
- exit() – Termina o processo atual.

Fonte: autoria própria.

Diferenças entre Linux e outros UNIX

- **Implementação de fork():** O Linux utiliza Copy-on-Write de forma mais agressiva que muitos outros UNIX, resultando em fork() significativamente mais rápido. Essa otimização é crucial para o desempenho de aplicações que criam processos frequentemente, como servidores web.
- **Threads e Processos:** O modelo de “tudo é processo” do Linux, com clone() para implementar threads como processos leves, difere de outros UNIX que possuem distinção mais clara entre threads e processos no nível do kernel.

Diferenças entre Linux e outros UNIX

Escalonamento

- O escalonador Completely Fair Scheduler (CFS) do Linux difere significativamente dos escalonadores tradicionais UNIX, proporcionando melhor resposta interativa e justiça na distribuição de tempo de CPU entre processos.

Namespaces

- O Linux implementa namespaces para isolamento de recursos entre grupos de processos, permitindo virtualização em nível de sistema operacional e containerização, recursos não disponíveis em UNIX tradicionais.

Interatividade

Qual é a principal função da chamada de sistema clone() no contexto de threads no Linux?

a) Criar processos completamente independentes, sem nenhum compartilhamento de recursos.

b) Criar threads como processos leves, permitindo diferentes níveis de compartilhamento de recursos.

c) Encerrar a execução de um processo pai imediatamente.

d) Duplicar apenas o espaço de endereçamento virtual de um processo, sem compartilhar mais nada.

e) Carregar dinamicamente um módulo do kernel em tempo de execução.

Resposta

Qual é a principal função da chamada de sistema clone() no contexto de threads no Linux?

a) Criar processos completamente independentes, sem nenhum compartilhamento de recursos.

b) Criar threads como processos leves, permitindo diferentes níveis de compartilhamento de recursos.

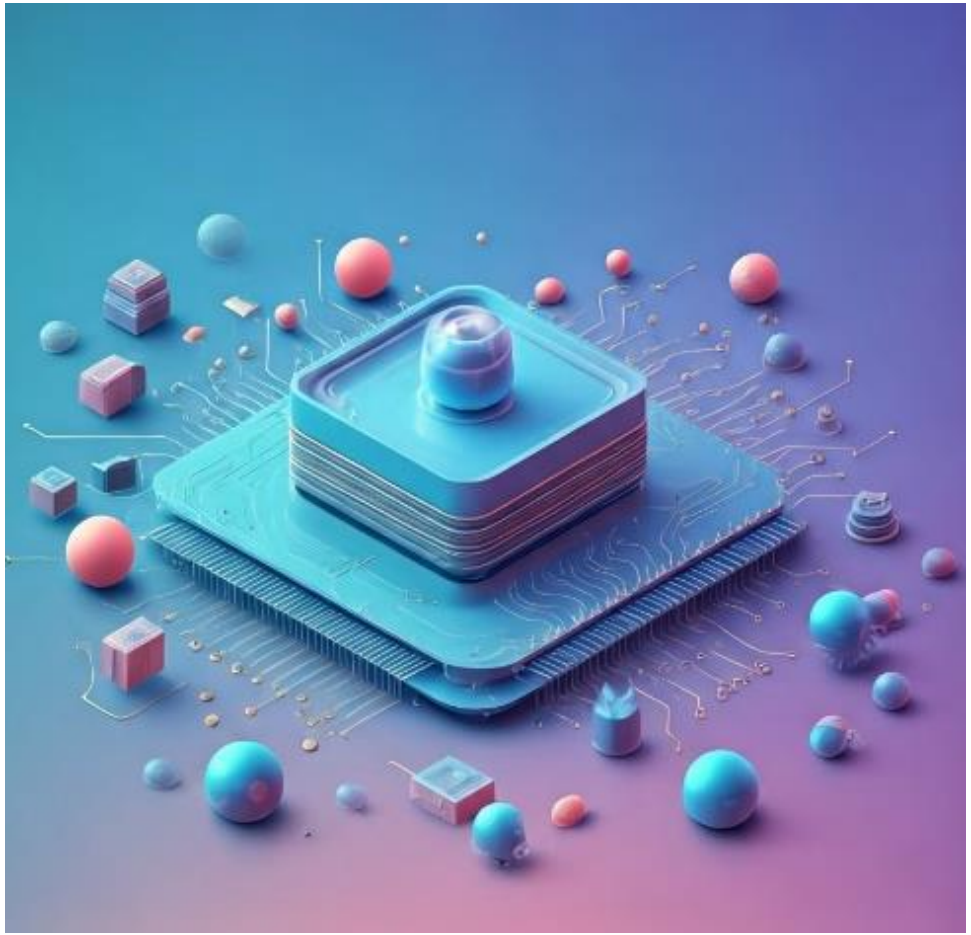
c) Encerrar a execução de um processo pai imediatamente.

d) Duplicar apenas o espaço de endereçamento virtual de um processo, sem compartilhar mais nada.

e) Carregar dinamicamente um módulo do kernel em tempo de execução.

Sincronização do kernel: Organizando operações seguras

- Exploraremos como o kernel gerencia suas próprias operações internas, garantindo a integridade dos dados compartilhados quando múltiplas tarefas tentam acessar simultaneamente as mesmas estruturas.
- Abordaremos os desafios fundamentais de sincronização, conceitos de seções críticas, mecanismos de proteção e soluções implementadas por kernels modernos. Este conhecimento é essencial para compreender o funcionamento interno dos sistemas operacionais e desenvolver software de sistema confiável.



Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, a sincronização do kernel.

Visão geral: O desafio da sincronização

- **Acesso Concorrente:** Múltiplas tarefas do kernel podem tentar acessar e modificar as mesmas estruturas de dados simultaneamente, criando potenciais condições de corrida.
- **Interrupções:** Interrupções de hardware podem ocorrer a qualquer momento, introduzindo novas tarefas que precisam ser executadas imediatamente pelo kernel.
- **Preempção:** Eventos de alta prioridade podem interromper a execução de tarefas do kernel em andamento, aumentando a complexidade da sincronização.
- **Integridade de Dados:** Manter a consistência das estruturas de dados compartilhadas é crucial para evitar comportamentos imprevisíveis e falhas do sistema.

Schedule de Processos vs. Schedule do Kernel

- **Schedule de Processos:** Gerencia a execução de processos de usuário, alternando entre eles para compartilhar a CPU. Utiliza algoritmos, como Round Robin, prioridades e tempo compartilhado.
- A preempção ocorre em pontos bem definidos, geralmente após a expiração de um quantum de tempo ou quando um processo de maior prioridade fica pronto.

- **Schedule do Kernel:** Lida com a execução de código dentro do próprio kernel, incluindo chamadas de sistema e rotinas de serviço de interrupção.
    - A execução pode ser interrompida a qualquer momento por interrupções de hardware, apresentando desafios únicos de sincronização que vão além do simples revezamento entre processos.

Como o kernel é invocado

1. **Chamadas de Sistema**

- Solicitações explícitas de um programa em execução para acessar recursos do sistema operacional. Exemplos: open(), read(), write(), fork().

2. **Exceções**

- Eventos que ocorrem durante a execução de um programa, como erros de página, violações de acesso à memória ou instruções ilegais que requerem intervenção do kernel.

3. **Interrupções de Hardware**

- Sinais assíncronos enviados por dispositivos de hardware que necessitam de atenção imediata do sistema operacional, como chegada de dados em uma interface de rede.

Fonte: Flux Fast 1.1, 2025 – A imagem representa, de forma figurativa, a sincronização do kernel.

O problema das seções críticas

Identificação da Seção Crítica

- Seções críticas são partes do código que acessam recursos compartilhados e não podem ser executadas simultaneamente por múltiplas tarefas sem risco de inconsistência.

Condições de Corrida

- Ocorrem quando o resultado da execução depende da ordem ou timing de eventos não controláveis, como interrupções durante acesso a dados compartilhados.

Consequências Potenciais

- Corrupção de dados, comportamento imprevisível do sistema, falhas de segurança e crashes do sistema operacional podem resultar de seções críticas mal protegidas.

Seções críticas no contexto do kernel

Antes da Seção Crítica

- O código do kernel identifica que precisará acessar uma estrutura de dados compartilhada e deve adquirir mecanismos de sincronização apropriados.

Durante a Seção Crítica

- O código tem acesso exclusivo à estrutura de dados compartilhada. Neste momento, interrupções ou outras tarefas do kernel que tentam acessar a mesma estrutura devem ser bloqueadas ou atrasadas.

Após a Seção Crítica

- O código libera os mecanismos de sincronização, permitindo que outras tarefas acessem a estrutura de dados compartilhada novamente.

Mecanismos de sincronização: Desabilitando interrupções

Funcionamento

- Antes de entrar em uma seção crítica, o kernel desabilita temporariamente as interrupções na CPU, impedindo que rotinas de serviço de interrupção sejam executadas e interfiram com a operação em andamento.

Vantagens

- Mecanismo simples e eficiente para CPUs únicas. Garante proteção completa contra interrupções inesperadas durante operações críticas do kernel.

Limitações

- Não resolve o problema em sistemas multiprocessados, onde outras CPUs podem acessar os mesmos dados simultaneamente. Períodos longos com interrupções desabilitadas podem afetar a responsividade do sistema.

Mecanismos de sincronização: Mutexes

- **Definição e Propósito:** Mutex (mutual exclusion) é um caso especial de semáforo binário, especificamente projetado para proteger seções críticas garantindo que apenas uma tarefa possa executá-la por vez.
- **Aquisição do Mutex:** Uma tarefa tenta obter o mutex antes de entrar na seção crítica. Se estiver disponível, a tarefa obtém acesso exclusivo; caso contrário, a tarefa é bloqueada até que o mutex seja liberado.
- **Liberação do Mutex:** Após concluir a seção crítica, a tarefa libera o mutex, permitindo que outras tarefas em espera possam adquiri-lo e acessar a seção crítica.
    - **Proprietário e Aninhamento:** Diferente dos semáforos gerais, os mutexes têm o conceito de proprietário e podem suportar operações de bloqueio aninhadas pelo mesmo thread sem causar deadlocks.

Problemas clássicos de sincronização

- Os problemas clássicos de sincronização ajudam a entender os desafios fundamentais enfrentados pelo kernel.

- O problema do produtor-consumidor ilustra a coordenação entre tarefas que produzem e consomem dados através de um buffer compartilhado.

- O problema dos leitores-escritores demonstra como priorizar acesso simultâneo de leitura versus acesso exclusivo de escrita. Já o problema dos filósofos jantando representa desafios de alocação de recursos e prevenção de deadlocks.

    - Estes cenários são frequentemente encontrados no kernel, onde drivers de dispositivos produzem dados consumidos por processos de usuário, múltiplas operações leem estruturas de dados enquanto atualizações ocorrem, e diferentes subsistemas competem por recursos limitados.

Desafios específicos do kernel

Deadlocks

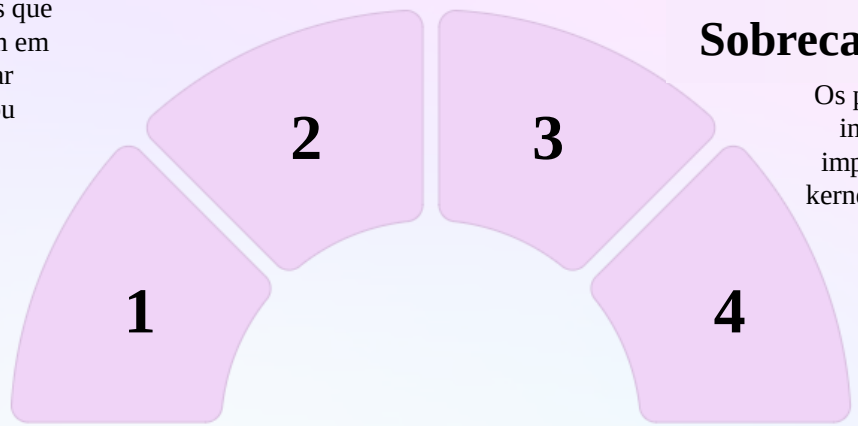
Situação em que duas ou mais tarefas estão esperando indefinidamente por recursos que estão retidos por outras tarefas, também em espera. O kernel precisa implementar estratégias de prevenção, detecção ou recuperação de deadlocks.

Starvation

Ocorre quando a uma tarefa é constantemente negado acesso a recursos necessários, devido a algoritmos de sincronização inadequados. O kernel precisa garantir justiça na alocação de recursos.

Inversão de Prioridade

Problema onde uma tarefa de alta prioridade é bloqueada esperando por um recurso mantido por uma tarefa de baixa prioridade, que por sua vez está preemptada por tarefas de média prioridade.



Sobrecarga de Sincronização

Os próprios mecanismos de sincronização introduzem overhead, potencialmente impactando o desempenho do sistema. O kernel deve balancear proteção e eficiência.

Fonte: autoria própria.

Interatividade

Em um cenário de acesso concorrente em estruturas de dados, qual é o principal risco que pode surgir?

a) Aumento linear do desempenho por conta da paralelização das tarefas.
b) Possibilidade de condições de corrida, levando a inconsistências e resultados imprevisíveis.
c) Redução total do número de interrupções de hardware, pois o sistema compartilha recursos.
d) Eliminação da necessidade de bloqueios e mecanismos de sincronização.
e) Garantia automática da integridade de dados, pois o kernel gerencia tudo de forma centralizada.

Resposta

Em um cenário de acesso concorrente em estruturas de dados, qual é o principal risco que pode surgir?

a) Aumento linear do desempenho por conta da paralelização das tarefas.
b) Possibilidade de condições de corrida, levando a inconsistências e resultados imprevisíveis.
c) Redução total do número de interrupções de hardware, pois o sistema compartilha recursos.
d) Eliminação da necessidade de bloqueios e mecanismos de sincronização.
e) Garantia automática da integridade de dados, pois o kernel gerencia tudo de forma centralizada.

ATÉ A PRÓXIMA!