Unidade III

5 GERENCIAMENTO DE PROCESSO

A menos que suas instruções sejam executadas por uma CPU, um programa não pode fazer nada. Um programa em execução, de usuário de tempo compartilhado (como um compilador), de processamento de texto, sendo executado por um usuário individual em um PC, uma tarefa do sistema (como o envio de saída para uma impressora), todos esses variados tipos de programas aplicativos são um processo.

Por ora, podemos considerar um processo como um job ou um programa de tempo compartilhado, mas, posteriormente, veremos que o conceito é mais genérico, sendo possível fornecer chamadas de sistema que possibilitem aos processos criar subprocessos a serem executados concorrentemente.

Um processo precisa de certos recursos, incluindo tempo de CPU, memória, arquivos e dispositivos de I/O para cumprir sua tarefa. Esses recursos são fornecidos ao processo quando ele é criado, ou são alocados a ele durante sua execução. Além dos diversos recursos físicos e lógicos que um processo obtém quando é criado, vários dados (entradas) de inicialização também podem ser passados. Por exemplo, considere um processo cuja função seja exibir o status de um arquivo na tela de um terminal. O processo receberá como entrada o nome do arquivo e executará as instruções e chamadas de sistema apropriadas para obter e exibir as informações desejadas no terminal. Quando o processo terminar, o sistema operacional reclamará os recursos reutilizáveis.

É preciso ter em mente que um programa por si só não é um processo e, sim, uma entidade passiva, como os conteúdos de um arquivo armazenado em disco, enquanto um processo é uma entidade ativa. Um processo de um único thread tem um contador de programa especificando a próxima instrução a ser executada. A execução de tal processo deve ser sequencial: a CPU executa uma instrução do processo após a outra, até o processo ser concluído. Além disso, a qualquer momento, no máximo uma instrução é executada em nome do processo. Portanto, embora dois processos possam estar associados ao mesmo programa, eles são considerados como duas sequências de execução separadas. Um processo com vários threads tem múltiplos contadores de programa, e cada um aponta para a próxima instrução a ser executada para determinado thread.

Um processo é a unidade de trabalho de um sistema, sendo que um sistema consiste em um conjunto de processos, alguns dos quais sendo processos do sistema operacional (os que executam código do sistema), e o resto processos de usuário (aqueles que executam código de usuário). Todos esses processos podem ser executados concorrentemente, pela multiplexação em uma única CPU, por exemplo.

O sistema operacional é responsável pelas seguintes atividades relacionadas com o gerenciamento de processos:

- Executar o scheduling de processos e threads nas CPUs.
- Criar e excluir processos de usuário e do sistema.
- Suspender e retomar processos.
- Fornecer mecanismos de sincronização de processos e de comunicação entre processos.

Gerenciamento de memória

A memória principal é essencial para a operação de um sistema de computação moderno. A memória principal é um grande array de bytes que variam em tamanho de centenas de milhares a bilhões.

Cada byte tem seu próprio endereço. A memória principal é um repositório de dados rapidamente acessáveis compartilhado pela CPU e dispositivos de I/O. O processador central lê instruções na memória principal durante o ciclo de busca de instruções, e lê e grava dados a partir dela durante o ciclo de busca de dados (em uma arquitetura von Neumann). Como mencionado anteriormente, a memória principal costuma ser o único grande dispositivo de armazenamento que a CPU consegue endereçar e acessar diretamente. Por exemplo, para a CPU processar dados do disco, primeiro esses dados devem ser transferidos para a memória principal por chamadas de I/O geradas pela CPU. Da mesma forma, as instruções devem estar na memória para a CPU executá-las. Para que um programa seja executado, ele deve ser mapeado para endereços absolutos e carregado na memória. Quando o programa é executado, ele acessa suas instruções e dados na memória gerando esses endereços absolutos.

Eventualmente, o programa é encerrado, seu espaço de memória é declarado disponível e o próximo programa pode ser carregado e executado.

Para melhorar tanto a utilização da CPU quanto a velocidade de resposta do computador para seus usuários, computadores de uso geral devem manter vários programas na memória, criando a necessidade de gerenciá-la. Muitos esquemas diferentes de gerenciamento da memória são usados. Esses esquemas refletem diversas abordagens, e a eficácia de determinado algoritmo depende da situação. Ao selecionar um esquema de gerenciamento da memória para um sistema específico, devemos levar em consideração muitos fatores, principalmente o projeto de hardware do sistema, pois cada algoritmo requer seu próprio suporte de hardware.

O sistema operacional é responsável por atividades relacionadas com o gerenciamento da memória, tais como:

- Controlar quais partes da memória estão sendo correntemente utilizadas e quem as está utilizando.
- Decidir que processos (ou partes de processos) e dados devem ser transferidos para dentro e para fora da memória.
- Alocar e desalocar espaço de memória conforme necessário.
- Gerenciar o armazenamento de massa.

A memória principal é pequena demais para acomodar todos os dados e programas, e já que os dados que ela armazena são perdidos quando não há energia, o sistema de computação deve fornecer memória secundária como backup da memória principal. A maioria dos sistemas de computação modernos usa discos como o principal meio de armazenamento on-line, tanto para programas quanto para dados. Grande parte dos programas, incluindo compiladores, montadores, processadores de texto, editores e formatadores, é armazenada em um disco até ser carregada na memória. Eles, então, usam o disco como fonte e destino de seu processamento. Portanto, o gerenciamento apropriado do armazenamento em disco é de importância primordial para um sistema de computação. O sistema operacional é também responsável por atividades relacionadas com o gerenciamento de disco, como gerenciar o espaço livre, alocar espaço de armazenamento e fazer o scheduling de disco.

Já que a memória secundária é utilizada com frequência, ela deve ser usada de maneira eficiente. A velocidade total de operação de um computador pode depender das velocidades do subsistema de disco e dos algoritmos que manipulam esse subsistema.

No entanto, há muitas utilidades para uma memória mais lenta e mais barata (e às vezes de maior capacidade) do que a memória secundária. Backups de dados de disco, armazenamento de dados pouco usados e armazenamento de longo prazo são alguns exemplos. Os drives de fita magnética e suas fitas e os drives de CD e DVD e discos são típicos dispositivos de memória terciária. A mídia (fitas e discos óticos) varia entre os formatos WORM (grava uma vez, lê várias vezes) e RW (lê e grava).

Algumas das funções que os sistemas operacionais podem fornecer incluem a montagem e desmontagem da mídia em dispositivos, a alocação e liberação dos dispositivos para uso exclusivo pelos processos e a migração de dados da memória secundária para a terciária.

Observação

A memória terciária não é crucial para o desempenho do sistema, mas mesmo assim deve ser gerenciada. Alguns sistemas operacionais assumem essa tarefa, enquanto outros deixam o gerenciamento da memória terciária para programas aplicativos.

5.1 Condições de corrida e regiões críticas

Condições de corrida, também conhecidas como race conditions, em inglês, são um fenômeno crucial no campo da computação paralela e sistemas operacionais. Essas situações ocorrem quando dois ou mais processos ou threads competem pelo acesso a recursos compartilhados de forma não sincronizada, resultando em comportamentos imprevisíveis ou incorretos do sistema.

Em essência, uma condição de corrida surge quando o resultado de uma operação depende da ordem ou tempo de execução de múltiplos processos. Isso acontece porque os processos "correm" uns contra os outros para acessar ou modificar dados compartilhados, sem um mecanismo adequado de controle ou sincronização.

As condições de corrida são particularmente problemáticas em sistemas multiprocessados ou multithreaded, onde várias unidades de execução operam simultaneamente. Nesses ambientes, a interleaving (intercalação) das instruções de diferentes processos pode levar a resultados inesperados se não for adequadamente gerenciada.

As condições de corrida são:

- **Não determinísticas**: o resultado pode variar a cada execução.
- **Difíceis de reproduzir**: podem ocorrer esporadicamente, dificultando a depuração.
- **Potencialmente prejudiciais**: podem corromper dados ou causar falhas no sistema.
- **Dependentes do timing**: ocorrem devido a diferenças sutis na ordem de execução.

Compreender as condições de corrida é fundamental para desenvolver sistemas operacionais robustos e confiáveis. Elas representam um desafio significativo na programação concorrente e exigem técnicas específicas de sincronização e controle de acesso para serem evitadas ou mitigadas. Para melhor compreender as condições de corrida, é crucial examinar exemplos concretos que ilustrem como esses problemas podem se manifestar em sistemas operacionais reais. Portanto, vamos explorar alguns cenários comuns onde as condições de corrida podem ocorrer.

Imagine uma conta bancária acessada simultaneamente por dois processos: um depósito e um saque. Se ambos os processos lerem o saldo atual ao mesmo tempo, realizarem suas operações independentemente e, em seguida, atualizarem o saldo, o resultado final pode não refletir corretamente ambas as transações. Por exemplo, se o saldo inicial for R$ 1.000 e um depósito de R$ 500 e um saque de R$ 200 ocorrerem concorrentemente, o saldo final poderia ser incorretamente R$ 1.500 ou R$ 800, em vez do valor correto de R$ 1.300.

Outro exemplo envolve a alocação de recursos do sistema. Em um sistema operacional, quando múltiplos processos solicitam a alocação de recursos (como memória ou dispositivos de E/S) simultaneamente, pode ocorrer uma condição de corrida. Se dois processos verificarem a disponibilidade de um recurso ao mesmo tempo e ambos concluírem que está livre, ambos podem tentar alocá-lo, levando a um conflito e possível corrupção de dados ou falha do sistema.

Há, ainda, o exemplo clássico do problema do produtor-consumidor, em que um processo (produtor) gera dados que são consumidos por outro processo (consumidor). Se não houver sincronização adequada, o consumidor pode tentar ler dados que ainda não foram completamente produzidos, ou o produtor pode sobrescrever dados que ainda não foram consumidos.

As condições de corrida também podem ocorrer em sistemas de arquivos. Por exemplo, se dois processos tentarem criar um arquivo com o mesmo nome simultaneamente, ambos podem verificar que o arquivo não existe, levando à criação de dois arquivos idênticos ou à sobrescrita de um pelo outro.

Em sistemas de controle de tráfego aéreo, condições de corrida podem ter consequências catastróficas. Se dois controladores atualizarem simultaneamente a posição de uma aeronave, baseando-se em informações ligeiramente defasadas, podem tomar decisões conflitantes que comprometem a segurança do voo.

Estes exemplos ilustram a importância crítica de identificar e prevenir condições de corrida em sistemas operacionais e aplicações concorrentes. A compreensão desses cenários ajuda os desenvolvedores a projetar sistemas mais robustos e implementar mecanismos de sincronização adequados para evitar comportamentos indesejados e potencialmente perigosos.

Impacto das condições de corrida em sistemas operacionais

As condições de corrida têm um impacto significativo e multifacetado nos sistemas operacionais, afetando sua estabilidade, desempenho e confiabilidade. Como já dissemos, compreender esses impactos é crucial para o desenvolvimento e manutenção de sistemas operacionais robustos e eficientes. Vamos a alguns deles:

- **Instabilidade do sistema**: condições de corrida podem levar a comportamentos erráticos e imprevisíveis do sistema operacional. Isso pode resultar em travamentos, congelamentos ou reinicializações inesperadas, comprometendo a experiência do usuário e a confiabilidade do sistema.

- **Corrupção de dados**: um dos impactos mais graves é a potencial corrupção de dados. Quando processos concorrentes acessam e modificam dados compartilhados sem sincronização adequada, podem ocorrer inconsistências nos dados, levando a resultados incorretos ou perda de informações críticas.

- **Degradação de desempenho**: paradoxalmente, as tentativas de evitar condições de corrida através de mecanismos de sincronização excessivos podem levar à degradação do desempenho. O uso intensivo de locks e semáforos pode criar gargalos, reduzindo a eficiência geral do sistema.

- **Falhas de segurança**: condições de corrida podem ser exploradas por atacantes para comprometer a segurança do sistema. Por exemplo, um atacante pode tirar proveito do intervalo entre a verificação e o uso de um recurso (conhecido como Time-of-Check to Time-of-Use, ou TOCTOU) para realizar ações maliciosas.

Além desses impactos diretos, condições de corrida também afetam o processo de desenvolvimento e manutenção de sistemas operacionais. A natureza não determinística dessas condições torna a depuração extremamente desafiadora, pois os problemas podem não se manifestar consistentemente durante os testes. Isso pode levar a ciclos de desenvolvimento mais longos e custos mais elevados.

Em sistemas de tempo real, onde o timing é crítico – como controle de tráfego aéreo, equipamentos médicos ou sistemas de controle industrial –, as condições de corrida podem ter consequências particularmente graves. Elas podem causar atrasos imprevisíveis ou falhas no cumprimento de prazos críticos, comprometendo a funcionalidade e segurança de sistemas.

O impacto das condições de corrida também se estende à confiabilidade percebida do sistema operacional. Usuários e administradores podem perder a confiança em um sistema que exibe comportamentos inconsistentes ou falhas inexplicáveis, mesmo que essas ocorrências sejam raras.

Esses e outros impactos evidenciam a importância de implementar técnicas robustas de sincronização, a realização de testes rigorosos em ambientes concorrentes e a adoção de práticas de programação que minimizem a ocorrência de condições de corrida. Além disso, destaca-se a necessidade de ferramentas avançadas de análise e depuração capazes de detectar e diagnosticar condições de corrida em sistemas complexos.

Regiões críticas: conceito e identificação

Regiões críticas são partes fundamentais de um programa ou sistema operacional onde ocorre o acesso a recursos compartilhados. A compreensão e a correta identificação dessas regiões são cruciais para prevenir condições de corrida e garantir a integridade dos dados em sistemas concorrentes.

Uma região crítica, também conhecida como seção crítica, é um segmento de código que acessa recursos compartilhados e que deve ser executado como uma operação atômica. Isso significa que, quando um processo ou thread está executando uma região crítica, nenhum outro deve ter permissão para executar sua própria região crítica que acesse os mesmos recursos compartilhados. Além disso, as regiões críticas contam com características como a exclusão mútua, em que apenas um processo pode estar na região crítica por vez, e a atomicidade, que determina que as operações dentro da região crítica devem ser indivisíveis. Ainda, a consistência da região crítica garante que os dados compartilhados permaneçam em um estado consistente e, junto a isso, o tempo limitado dessa região define que a execução deve ter uma duração finita.

A identificação correta das regiões críticas é um passo crucial no desenvolvimento de sistemas operacionais e aplicações concorrentes. Isso envolve uma análise cuidadosa do código para determinar onde ocorrem acessos a recursos compartilhados. Algumas técnicas para identificar regiões críticas incluem:

- **Análise de fluxo de dados**: implica examinar como os dados são acessados e modificados ao longo do programa para identificar pontos de interação entre processos.

- **Revisão de código**: envolve revisões sistemáticas do código-fonte, buscando por acessos a variáveis globais, estruturas de dados compartilhadas ou chamadas de sistema que manipulam recursos compartilhados.

- **Ferramentas de análise estática**: utiliza ferramentas automatizadas que podem detectar potenciais regiões críticas analisando o código-fonte em busca de padrões de acesso concorrente.

- **Testes de concorrência**: executa testes específicos que simulam condições de alta concorrência para revelar possíveis regiões críticas não identificadas anteriormente.

Uma vez identificadas, as regiões críticas devem ser cuidadosamente gerenciadas para evitar condições de corrida. Isso geralmente envolve o uso de mecanismos de sincronização como semáforos, mutexes (mutual exclusion) ou monitores para garantir a exclusão mútua.

É importante notar que a identificação excessiva de regiões críticas pode levar a um desempenho subótimo devido ao overhead de sincronização. Por outro lado, a não identificação de uma região crítica pode resultar em condições de corrida. Portanto, é crucial encontrar um equilíbrio entre segurança e desempenho.

Em sistemas operacionais modernos, a identificação e a gestão de regiões críticas tornam-se ainda mais complexas devido à natureza distribuída e altamente concorrente dos sistemas. Técnicas avançadas, como modelagem de concorrência e análise de dependência de dados são frequentemente empregadas para lidar com essa complexidade crescente.

Técnicas de prevenção de condições de corrida

A prevenção de condições de corrida é uma parte essencial do desenvolvimento de sistemas operacionais robustos e confiáveis. Diversas técnicas foram desenvolvidas ao longo dos anos para abordar este desafio, cada uma com suas próprias vantagens e limitações. Vamos explorar algumas das principais técnicas utilizadas para prevenir condições de corrida em sistemas operacionais:

- **Exclusão mútua**: utiliza mecanismos como mutexes e semáforos para garantir que apenas um processo por vez possa acessar um recurso compartilhado. Isso cria uma região crítica protegida, evitando acessos simultâneos.

- **Operações atômicas**: implementa operações que são executadas como uma única unidade indivisível, sem a possibilidade de interrupção, sendo frequentemente suportada por hardware em instruções específicas.

- **Filas de mensagens**: utiliza filas para comunicação entre processos, onde as mensagens são enfileiradas e desenfileiradas de forma atômica, evitando conflitos de acesso concorrente.

- **Controle de versão**: implementa técnicas como Controle de Concorrência Multiversão (MVCC), onde cada transação trabalha com uma versão consistente dos dados, evitando conflitos diretos.

Além dessas técnicas fundamentais, existem abordagens mais avançadas e específicas para prevenir condições de corrida. Uma delas é o locking em duas fases, em que todos os locks são adquiridos antes de qualquer operação ser realizada e liberados apenas após todas as operações serem concluídas, ajudando a prevenir deadlocks e garantindo consistência. As transações atômicas são outra abordagem e utilizam o conceito de transações do banco de dados, em que um conjunto de operações é tratado como uma única unidade atômica; se qualquer parte falhar, toda a transação é revertida. Ainda, temos a programação sem bloqueio, na qual técnicas como Compare-And-Swap (CAS) permitem operações concorrentes sem o uso de locks tradicionais, melhorando o desempenho em cenários de alta concorrência. Por fim, o isolamento de recursos é uma abordagem que divide recursos compartilhados em partições isoladas, reduzindo a necessidade de sincronização entre processos que operam em diferentes partições.

A escolha da técnica adequada depende de vários fatores, incluindo a natureza do sistema, os requisitos de desempenho e a complexidade da implementação. Muitas vezes, uma combinação de várias técnicas é necessária para abordar eficazmente todos os cenários potenciais de condições de corrida.

É importante notar que a implementação dessas técnicas requer um cuidado especial. O uso excessivo ou incorreto de mecanismos de sincronização pode levar a problemas como deadlocks, livelocks ou degradação significativa do desempenho. Portanto, é crucial encontrar um equilíbrio entre a prevenção de condições de corrida e a manutenção de um bom desempenho do sistema.

Além disso, o desenvolvimento de novas arquiteturas de hardware e paradigmas de programação continua a influenciar as técnicas de prevenção de condições de corrida. Por exemplo, o advento de processadores multicore e sistemas distribuídos tem levado ao desenvolvimento de técnicas mais sofisticadas e escaláveis para lidar com a concorrência em larga escala.

Sincronização de acesso a recursos compartilhados

A sincronização de acesso a recursos compartilhados é um aspecto crítico na prevenção de condições de corrida em sistemas operacionais. Ela envolve a coordenação do acesso a recursos que podem ser utilizados por múltiplos processos ou threads simultaneamente. O objetivo principal é garantir a integridade dos dados e a consistência das operações, evitando conflitos e comportamentos imprevisíveis. Alguns dos recursos empregados pela sincronização de acesso são:

- **Mecanismos de sincronização**: existem vários mecanismos de sincronização comumentemente utilizados em sistemas operacionais.
- **Semáforos**: permitem controlar o acesso a um número limitado de recursos.
- **Mutexes**: garantem que apenas um processo por vez possa acessar um recurso.
- **Monitores**: combinam exclusão mútua com a capacidade de esperar por uma condição.
- **Variáveis de condição**: permitem que processos esperem por uma condição específica antes de prosseguir.

No entanto, a implementação eficaz da sincronização enfrenta vários desafios, como o overhead de desempenho, que ocorre quando a sincronização introduz atrasos e reduz a concorrência; os deadlocks, que correspondem a situações onde processos ficam permanentemente bloqueados, esperando uns pelos outros; a starvation, que ocorre quando um processo tem continuamente negado acesso a um recurso; e a complexidade que advém de mecanismos de sincronização que podem tornar o código mais difícil de entender e manter.

Uma estratégia eficaz de sincronização deve equilibrar a necessidade de proteção contra condições de corrida com o desejo de manter um alto grau de concorrência e desempenho. Isso geralmente envolve uma análise cuidadosa dos padrões de acesso aos recursos compartilhados e a seleção dos mecanismos de sincronização mais apropriados para cada situação. A seguir, elencamos as principais etapas dessa estratégia:

- **Identificação de recursos compartilhados**: o primeiro passo é identificar todos os recursos que podem ser acessados concorrentemente por múltiplos processos.
- **Análise de padrões de acesso**: examinar como e quando os recursos são acessados para determinar o tipo de sincronização necessário.
- **Implementação de mecanismos**: aplicar os mecanismos de sincronização apropriados, como semáforos ou mutexes, nos pontos de acesso críticos.
- **Testes e otimização**: realizar testes rigorosos para garantir a eficácia da sincronização e otimizar o desempenho conforme necessário.

Técnicas avançadas de sincronização incluem o uso de algoritmos lock-free e wait-free, que visam reduzir o overhead de sincronização em sistemas de alta concorrência. Essas técnicas utilizam operações atômicas de hardware para implementar sincronização sem o uso de locks tradicionais, potencialmente melhorando o desempenho e a escalabilidade. Outro aspecto importante é a granularidade da sincronização, já que uma granularidade muito fina pode levar a um overhead excessivo, enquanto uma muito grossa pode reduzir a concorrência. Encontrar o equilíbrio certo é crucial para o desempenho do sistema.

Em sistemas operacionais modernos, a sincronização de acesso a recursos compartilhados é um desafio contínuo, especialmente com o advento de arquiteturas multicore e sistemas distribuídos em larga escala. Isso tem levado ao desenvolvimento de novas técnicas e paradigmas de sincronização, como programação transacional e modelos de consistência relaxada, que visam melhorar a escalabilidade e o desempenho em ambientes altamente concorrentes.

Algoritmos de exclusão mútua

Os algoritmos de exclusão mútua são fundamentais para garantir o acesso seguro e ordenado a recursos compartilhados em sistemas operacionais. Esses algoritmos visam prevenir condições de corrida, assegurando que apenas um processo ou thread por vez possa acessar um recurso crítico.

Alguns dos principais algoritmos de exclusão mútua e suas características são:

- **Algoritmo de Peterson**: um dos algoritmos clássicos de exclusão mútua para dois processos. Esse algoritmo utiliza variáveis de flag e turno para coordenar o acesso à região crítica. Simples, é limitado a dois processos e pode sofrer de busy-waiting.

- **Algoritmo da padaria de Lamport**: projetado para N processos, simula um sistema de filas de padaria. Cada processo recebe um número de ticket e espera sua vez, garantindo justiça, mas podendo ser ineficiente para um grande número de processos.

- **Algoritmo de Dekker**: outro algoritmo clássico para dois processos. Usa flags e uma variável de turno para determinar qual processo pode entrar na região crítica. É mais complexo do que o de Peterson, e oferece garantias mais fortes de exclusão mútua.

- **Algoritmo de Eisenberg e McGuire**: um algoritmo para N processos que utiliza um array de estados e uma variável de turno. Oferece uma solução mais eficiente do que o algoritmo da padaria para um número maior de processos.

Além desses algoritmos clássicos, existem implementações modernas e otimizadas de exclusão mútua em sistemas operacionais contemporâneos. Os spinlocks são uma delas, sendo uma forma de exclusão mútua onde um processo continua verificando repetidamente se um lock está disponível; são eficientes para esperas curtas em sistemas multiprocessados. Outra implementação moderna são os semáforos, uma abstração mais avançada que permite controlar o acesso a múltiplos recursos, podendo ser binários (mutex) ou contadores. Ainda temos os monitores**,** um mecanismo de alto nível que combina exclusão mútua com a capacidade de esperar por condições específicas, e o Read-Copy-Update (RCU), uma técnica avançada que permite leituras sem locks, sendo útil em cenários com muitas leituras e poucas escritas.

A escolha do algoritmo de exclusão mútua apropriado depende de vários fatores, incluindo o número de processos envolvidos, a natureza do recurso compartilhado, os requisitos de desempenho e a arquitetura do sistema. Em sistemas modernos, é comum utilizar uma combinação de diferentes técnicas para otimizar o desempenho e a escalabilidade.

Um aspecto crucial da implementação de algoritmos de exclusão mútua é evitar problemas como deadlocks, livelocks e starvation.

Lembrete

Deadlocks ocorrem quando processos ficam permanentemente bloqueados, esperando uns pelos outros. Livelocks são situações nas quais os processos mudam de estado continuamente, mas não progridem. Starvation acontece quando um processo tem repetidamente negado acesso ao recurso.

Os sistemas operacionais modernos frequentemente implementam versões otimizadas desses algoritmos, aproveitando características específicas do hardware, como instruções atômicas e cache coerente. Além disso, técnicas como lock-free e wait-free programming estão ganhando popularidade, especialmente em sistemas de alta performance e baixa latência.

É importante notar que, embora a exclusão mútua seja essencial para prevenir condições de corrida, seu uso excessivo pode levar a gargalos de desempenho. Portanto, os desenvolvedores de sistemas operacionais devem buscar um equilíbrio entre segurança e eficiência, utilizando técnicas de análise e otimização para minimizar o overhead de sincronização enquanto mantêm a integridade do sistema.

Observação

A principal vantagem do algoritmo de Peterson é sua simplicidade e eficiência. Ele é um algoritmo de baixo custo, com um baixo número de instruções de sincronização. Além disso, ele é livre de starvation, ou seja, garante que threads bloqueadas aguardem o tempo necessário para acessar a seção crítica. Isso é importante em sistemas multithread, onde a eficiência e a justiça na alocação de recursos são cruciais.

Monitoramento e detecção de condições de corrida

O monitoramento e a detecção de condições de corrida são aspectos cruciais no desenvolvimento e manutenção de sistemas operacionais robustos. Dada a natureza elusiva das condições de corrida, técnicas especializadas e ferramentas sofisticadas são necessárias para identificar e diagnosticar esses problemas.

A análise estática é umas das principais abordagens e ferramentas utilizadas para monitorar e detectar condições de corrida. Nela, a análise envolve a examinação do código-fonte sem executá-lo. Ferramentas de análise estática procuram por padrões e estruturas que possam indicar potenciais condições de corrida. Embora eficazes para identificar muitos problemas, há algumas ferramentas desse tipo de análise que podem gerar falsos positivos e têm limitações na detecção de condições de corrida mais complexas, sendo elas: análise de fluxo de dados; verificação de modelos (model checking); análise de dependência de threads; e as técnicas de análise dinâmica.

A análise dinâmica, por exemplo, envolve a execução do programa e o monitoramento de seu comportamento em tempo real. Essas técnicas são mais eficazes na detecção de condições de corrida reais, mas podem ter um impacto significativo no desempenho do sistema, principalmente ao abordar certos pontos, como: instrumentação de código; monitoramento de acesso à memória; e análise de traços de execução.

Ainda, algumas ferramentas específicas desempenham um papel crucial na detecção de condições de corrida. Algumas das mais populares e eficazes incluem:

- **Valgrind**: uma ferramenta de instrumentação de código que inclui o Helgrind, um detector de condições de corrida para programas C/C++ que usam a biblioteca POSIX threads.

- **ThreadSanitizer**: desenvolvido pelo Google, é um detector de condições de corrida e data races para C/C++ e Go, que utiliza instrumentação em tempo de compilação.

- **Intel Inspector**: uma ferramenta de análise dinâmica que detecta erros de threading, incluindo condições de corrida em aplicações para Windows e Linux.

- **CHESS**: uma ferramenta da Microsoft para testar aplicações concorrentes, que sistematicamente explora diferentes intercalações de threads para encontrar bugs de concorrência.

Além dessas ferramentas específicas, existem técnicas avançadas de monitoramento e detecção que estão ganhando proeminência. Uma delas é a análise de execução simbólica, que explora múltiplos caminhos de execução simultaneamente, permitindo a detecção de condições de corrida que podem não se manifestar em execuções normais. Há ainda outras ferramentas que são dignas de menção, como:

- **Aprendizado de máquina**: utiliza algoritmos de IA para analisar padrões de execução e identificar anomalias que podem indicar condições de corrida.

- **Análise de traços distribuídos**: coleta e analisa traços de execução de sistemas distribuídos para identificar condições de corrida em ambientes complexos de microserviços.

- **Verificação em tempo de execução**: implementa verificações dinâmicas no código para detectar violações de propriedades de concorrência durante a execução.

É importante notar que o monitoramento e a detecção de condições de corrida apresentam desafios significativos, como:

- **Overhead de desempenho**: muitas técnicas de detecção podem causar uma degradação significativa no desempenho do sistema durante o monitoramento.

- **Falsos positivos**: algumas ferramentas podem reportar condições de corrida que não são reais ou que não têm impacto prático.

- **Complexidade de análise**: em sistemas grandes e complexos, a análise dos resultados pode ser desafiadora e requerer expertise significativa.

- **Cobertura incompleta**: é praticamente impossível testar todas as possíveis intercalações de threads, especialmente em sistemas de grande escala.

Para uma detecção eficaz de condições de corrida, é recomendável adotar uma abordagem multifacetada que combine análise estática, testes dinâmicos e monitoramento em produção. Além disso, a integração dessas ferramentas e técnicas no ciclo de desenvolvimento contínuo (CI/CD) pode ajudar na identificação de problemas de concorrência mais cedo no processo de desenvolvimento.

À medida que os sistemas operacionais e as aplicações se tornam mais complexos e distribuídos, a importância do monitoramento e detecção eficazes de condições de corrida continuam a crescer. O desenvolvimento de novas técnicas e ferramentas nesta área permanece um campo ativo de pesquisa e inovação na ciência da computação e engenharia de software.

5.2 Concorrência e sincronização

Concorrência em sistemas operacionais refere-se à capacidade de executar múltiplos processos ou threads simultaneamente. Esse conceito é crucial para a otimização do uso de recursos computacionais, permitindo que o sistema execute várias tarefas de forma eficiente. No entanto, a concorrência também introduz complexidades, principalmente no gerenciamento do acesso compartilhado a recursos.

A concorrência, em sua essência, permite que múltiplas tarefas sejam executadas de forma interrompida. Em um sistema operacional moderno, o processador alterna rapidamente entre diferentes processos, dando a ilusão de que todos estão sendo executados ao mesmo tempo. Essa técnica, conhecida como multiprogramação, maximiza o uso do hardware e permite que o sistema opere de forma mais eficiente.

No entanto, quando processos compartilham recursos como memória, arquivos ou periféricos, podem surgir conflitos. Imagine dois processos tentando acessar o mesmo arquivo ao mesmo tempo, o resultado pode ser dados corrompidos ou resultados inesperados. Para evitar tais situações, precisamos de mecanismos de sincronização que garantam o acesso ordenado e seguro a recursos compartilhados.

Um processo é uma unidade básica de execução em um sistema operacional. Cada processo possui seu próprio espaço de endereço, conjunto de recursos e um contador de programa, os quais determinam o ponto de execução atual. Um processo é frequentemente considerado um programa em execução. Ele pode ter vários threads, que são unidades de execução ainda menores dentro dele.

Observação

Threads compartilham o mesmo espaço de endereço, mas possuem seu próprio contador de programa e pilha. Além disso, permitem que um processo execute diferentes tarefas simultaneamente, aproveitando o processamento paralelo.

O conceito de thread é fundamental para a programação concorrente, pois oferece uma forma eficiente de dividir tarefas dentro de um processo. Imagine um aplicativo de edição de texto: um thread poderia lidar com a interface do usuário enquanto outro thread processa sua digitação. Essa separação de tarefas permite que o aplicativo seja mais responsivo e eficiente.

Uma das principais vantagens da utilização de threads é a redução do overhead de criação e gerenciamento de processos. Criar um novo thread é geralmente mais rápido e menos exigente em termos de recursos do que criar um novo processo. Além disso, threads podem compartilhar recursos como memória e arquivos com outros threads do mesmo processo, otimizando a comunicação e o acesso a dados.

Problemas de concorrência: deadlocks, condições de corrida e starvation

A concorrência, apesar de seus benefícios, traz desafios, especialmente no gerenciamento do acesso a recursos compartilhados. Se não gerenciados adequadamente, podem surgir problemas como deadlocks, condições de corrida e starvation.

Um deadlock ocorre quando dois ou mais processos ficam presos em um estado de espera, cada um aguardando um recurso que o outro possui. Imagine dois carros em uma estrada estreita, cada um esperando que o outro se mova para que possa passar. Nessa situação, nenhum dos carros consegue prosseguir, resultando em um deadlock. Em sistemas operacionais, deadlocks podem ocorrer quando dois ou mais processos requisitam recursos em uma ordem diferente, impedindo a liberação dos recursos e criando uma situação de espera infinita.

Uma condição de corrida ocorre quando o resultado de um processo depende da ordem em que os processos compartilham um recurso. Imagine dois processos atualizando um contador simultaneamente. Se um processo lê o valor do contador, outro processo altera-o e o primeiro escreve o valor atualizado, o contador pode acabar com um valor incorreto. Condições de corrida são difíceis de depurar, pois o comportamento do sistema pode variar de forma imprevisível dependendo da ordem de execução dos processos.

Starvation ocorre quando um processo é continuamente negligenciado e não consegue acessar o recurso necessário para progredir. Imagine um grupo de processos esperando por um único recurso. Se um processo sempre é priorizado e obtém acesso ao recurso antes dos outros, eles podem acabar em starvation, nunca conseguindo executar. Starvation pode ser um problema particularmente difícil de detectar e resolver, pois pode ocorrer de forma sutil e não óbvia.

Mecanismos de sincronização: semáforos, monitores e mutexes

Para resolver os problemas de concorrência, os sistemas operacionais fornecem mecanismos de sincronização que permitem que os processos coordenem seu acesso a recursos compartilhados. Alguns dos mecanismos mais comuns incluem semáforos, monitores e mutexes.

Um semáforo é um objeto que é usado para controlar o acesso a recursos compartilhados. Ele tem um valor inteiro que representa o número de recursos disponíveis. Um processo pode solicitar acesso a um recurso realizando uma operação de wait (esperar) no semáforo. Se houver recursos disponíveis, o valor do semáforo é decrementado e o processo obtém acesso. Porém, caso não tenha recursos disponíveis, o processo fica em espera até que um recurso seja liberado. Quando um processo termina de usar um recurso, ele realiza uma operação de signal (sinalizar) no semáforo, incrementando seu valor e notificando outro processo em espera.

Um monitor é um objeto que encapsula recursos compartilhados e as operações que podem ser realizadas sobre eles. Monitores oferecem uma forma mais estruturada de sincronização do que semáforos, pois fornecem uma única entrada para acessar os recursos compartilhados. O acesso a um monitor é controlado por uma variável de condição que permite que os processos esperem por

eventos específicos. Os monitores são frequentemente usados para implementar estruturas de dados compartilhadas, garantindo que as operações sejam executadas de forma atômica e segura.

Um mutex, exclusão mútua (mutual exclusion), é um tipo de semáforo que permite que apenas um processo tenha acesso a um recurso compartilhado por vez. Quando um processo solicita acesso a um recurso protegido por um mutex, ele realiza uma operação de lock (bloqueio) no mutex. Se o mutex estiver desbloqueado, o processo obtém acesso ao recurso. Caso contrário, ele fica em espera até que o mutex seja liberado. Quando um processo termina de usar o recurso, ele realiza uma operação de unlock (desbloqueio) no mutex, liberando-o para que outro processo possa acessá-lo.

Algoritmos de exclusão mútua: algoritmo de Peterson e algoritmo de padaria

Algoritmos de exclusão mútua são usados para garantir que apenas um processo tenha acesso a um recurso compartilhado por vez. Diversos algoritmos foram propostos para implementar a exclusão mútua, cada um com suas próprias características e vantagens. Anteriormente, mencionamos brevemente sobre os algoritmos de Peterson e de padaria de Lamport; agora, nos determos mais neles.

O algoritmo de Peterson, um algoritmo clássico de exclusão mútua, é relativamente simples e eficiente. Ele usa uma variável de flag para cada processo e uma variável compartilhada para indicar qual processo tem prioridade. Cada processo define sua flag como verdadeira quando deseja entrar na seção crítica e define o turno como outro processo. Com isso, o processo entra em um loop de espera, verificando se o outro processo tem a flag definida como verdadeira e se seu turno é maior do que o do outro. Se ambas as condições forem verdadeiras, o processo espera. Caso contrário, ele entra na seção crítica. Após sair da seção crítica, ele define sua flag como falsa.

O algoritmo de padaria de Lamport é outro algoritmo de exclusão mútua que garante a exclusão mútua. Ele é baseado no conceito de número de ticket e permite que os processos entrem na seção crítica em uma ordem "justa". Cada processo obtém um número de ticket e espera em uma fila, acessando a seção crítica de acordo com a ordem dos números de ticket. O algoritmo usa uma variável compartilhada, "lastTicket", para gerar números de ticket únicos e uma fila compartilhada, "queue", para armazenar os processos em espera.

Problemas clássicos de sincronização: problema do produtor-consumidor, problema dos filósofos

A sincronização de processos é fundamental para resolver problemas clássicos de concorrência em sistemas operacionais. Esses problemas servem como modelos para entender e testar mecanismos de sincronização.

O problema do produtor-consumidor é um exemplo clássico de sincronização, onde um processo produtor gera dados e outro consumidor, justamente, consome esses dados. Os processos compartilham um buffer limitado, onde o produtor coloca os dados e o consumidor os remove. O problema reside em garantir que o produtor não escreva dados em um buffer cheio e que o consumidor não leia dados de um buffer vazio. A sincronização é necessária para coordenar as ações do produtor e do consumidor, evitando condições de corrida e deadlocks.

O problema dos filósofos é outro problema clássico de sincronização, funcionando da seguinte forma: cinco filósofos sentados ao redor de uma mesa compartilham cinco garfos; cada filósofo precisa de dois garfos para comer, mas apenas um garfo está disponível para cada um. O problema consiste em garantir que os filósofos não fiquem em um deadlock, onde todos estão esperando por um garfo que outro filósofo está usando. A sincronização é necessária para que os filósofos possam pegar os garfos e comer sem bloquear o acesso aos garfos pelos outros filósofos.

Implementação de sincronização em sistemas operacionais modernos

Sistemas operacionais modernos fornecem uma variedade de recursos para gerenciar a concorrência e a sincronização. Esses recursos incluem:

- **Primitivas de sincronização**: os sistemas operacionais fornecem primitivas de sincronização como semáforos, mutexes, condições de espera e variáveis de condição. Essas primitivas são usadas por programas para sincronizar o acesso a recursos compartilhados e evitar condições de corrida e deadlocks.

- **Gerenciamento de threads**: sistemas operacionais modernos fornecem um mecanismo para gerenciar threads, que são unidades de execução leves que podem ser usadas para melhorar o desempenho e a responsividade do sistema. O gerenciamento de threads inclui sua criação, seu escalonamento e a sua destruição.

- **Escalonamento de processos**: é o processo de gerenciar os recursos do sistema e determinar quais processos serão executados em um determinado momento. O objetivo do escalonamento é maximizar o uso do sistema e garantir que todos os processos tenham uma oportunidade de execução.

Os sistemas operacionais modernos também oferecem bibliotecas e frameworks para programação concorrente. Essas ferramentas fornecem estruturas e abstrações que facilitam a implementação de programas concorrentes e garantem a segurança e a eficiência do código.

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:

- **Minimizar o tamanho da seção crítica**: que 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.

- **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.

- **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.

- **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. Por isso, é importante implementar mecanismos para detecta-los e resolvê-los.

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

A concorrência e a sincronização são conceitos fundamentais em sistemas operacionais, permitindo a execução de múltiplos processos e threads, otimizando o uso de recursos e melhorando a responsividade do sistema. No entanto, a concorrência apresenta desafios, como deadlocks, condições de corrida e starvation, que exigem mecanismos de sincronização para garantir a segurança e a corretude do sistema.

A escolha e implementação adequadas de mecanismos de sincronização são cruciais para o bom funcionamento do sistema. Semáforos, monitores e mutexes são apenas alguns exemplos dessas ferramentas, cada uma com suas próprias vantagens e desvantagens. É importante entender as necessidades do sistema e escolher a melhor solução para cada cenário.

Boas práticas de programação concorrente garantem a segurança, a eficiência e a robustez do código. Minimizar seções críticas, usar primitivas de sincronização apropriadas, evitar condições de corrida, detectar e resolver deadlocks e testar cuidadosamente o código são aspectos essenciais para a construção de programas concorrentes eficientes e confiáveis.

Com o avanço da tecnologia e o aumento da complexidade dos sistemas, a importância da concorrência e da sincronização continua a crescer. É essencial que os desenvolvedores de sistemas operacionais e os programadores estejam familiarizados com esses conceitos e com as melhores práticas para garantir o bom funcionamento e a eficiência dos sistemas modernos.

Programação lock-free

A programação lock-free é uma técnica de programação concorrente que garante que um processo em execução nunca seja bloqueado, independentemente das ações de outros processos. Isso significa que um processo sempre pode fazer progresso, mesmo que outros estejam em estado de espera ou falhem. Essa propriedade é fundamental para construir sistemas que sejam resilientes a falhas e possam lidar com alto nível de concorrência.

Em contraste com a programação com bloqueios, onde processos podem ser bloqueados enquanto aguardam a liberação de um recurso compartilhado, a programação lock-free evita completamente o uso de bloqueios. Em vez disso, os processos competem atomicamente por acesso aos recursos compartilhados, garantindo que apenas um processo tenha acesso exclusivo a um recurso por vez.

Para entender a programação lock-free, é fundamental conhecer os conceitos de espera e bloqueio em programação concorrente. Em um ambiente multi-processo, os processos frequentemente precisam acessar recursos compartilhados, como variáveis globais ou estruturas de dados. O problema surge quando múltiplos processos tentam acessar o mesmo recurso ao mesmo tempo.

A programação com bloqueios utiliza mecanismos como mutexes (mutual exclusion) para garantir que apenas um processo tenha acesso exclusivo ao recurso compartilhado por vez. Outros processos que tentam acessar o recurso ficam bloqueados até que o processo atual libere o recurso. Isso pode levar a problemas de desempenho, especialmente se o processo atual ficar bloqueado por tempo indeterminado.

A espera ocorre quando um processo está aguardando um evento ou condição que pode ser causada por outro processo. Em alguns casos, a espera pode ser bloqueada, como quando um processo fica bloqueado esperando a liberação de um recurso por outro processo. Em outros casos, a espera pode ser não-bloqueada, como quando um processo espera a conclusão de uma operação atômica.

A programação lock-free oferece várias vantagens em relação à programação com bloqueios, tornando-a uma abordagem ideal para cenários de alta concorrência e tolerância a falhas, como:

- **Sem deadlocks**: a programação lock-free elimina o risco de deadlocks, pois não há bloqueios ou espera mútua entre processos.
- **Progresso garantido**: sempre que um processo é executado em um sistema lock-free, ele pode fazer progresso em sua tarefa. Isso significa que ele pode realizar alguma operação útil, mesmo que outros processos estejam em estado de espera ou falhem.
- **Tolerância a falhas**: se um processo falhar em um sistema lock-free, os outros processos podem continuar operando sem serem afetados pela falha. Isso garante que o sistema como um todo permaneça disponível.
- **Escalabilidade**: a programação lock-free é altamente escalável, pois permite que um grande número de processos compartilhe recursos de forma eficiente. Isso é especialmente importante em sistemas com alto grau de concorrência.

No entanto, a implementação de algoritmos lock-free pode ser complexa e exigir um profundo conhecimento de mecanismos de sincronização e arquitetura de memória.

A implementação de estruturas de dados lock-free envolve o uso de técnicas de sincronização atômica para garantir que as operações de leitura e escrita em recursos compartilhados sejam realizadas de forma segura e eficiente. Essas técnicas incluem:

- **Operações atômicas**: são instruções de baixo nível que garantem que um conjunto de operações seja executado como uma única unidade indivisível. Essas operações são essenciais para a programação lock-free, pois permitem que os processos atualizem recursos compartilhados de forma segura, mesmo que outros processos estejam acessando os mesmos recursos ao mesmo tempo.

- **Comparar e trocar (CAS)**: é uma operação atômica que compara o valor atual de uma localização de memória com um valor esperado e, se os valores forem iguais, substitui o valor atual com um novo valor. A operação CAS é frequentemente utilizada em algoritmos lock-free para atualizar estruturas de dados de forma atômica.

- **Load-linked/store-conditional**: esse par de instruções fornece uma forma de garantir a atomicidade de operações de leitura e escrita em um sistema multiprocessado. A instrução load-linked lê um valor de uma localização de memória e a instrução store-conditional grava um novo valor na localização de memória, apenas se o valor original não tiver sido modificado desde a leitura inicial.

São vários os exemplos de estruturas de dados lock-free, incluindo listas ligadas, filas, mapas e árvores. A implementação dessas estruturas de dados requer o uso cuidadoso de técnicas de sincronização atômica e algoritmos que garantem a consistência da estrutura de dados.

Técnicas de sincronização lock-free

A programação lock-free depende fortemente de técnicas de sincronização atômica para garantir que as operações em recursos compartilhados sejam executadas de forma segura e eficiente. As técnicas mais comuns incluem:

- **Operações atômicas de baixo nível**: como CAS, load-linked/store-conditional e instruções de troca atômica são usadas para atualizar recursos compartilhados de forma indivisível. Essas operações garantem que as mudanças no estado compartilhado sejam visíveis para todos os processos de forma consistente.

- **Algoritmos de consenso distribuído**: são usados para alcançar um acordo entre múltiplos processos sobre um valor compartilhado. Esses algoritmos são essenciais para a programação lock-free, pois permitem que os processos coordenem suas ações e garantam a consistência do estado compartilhado.

- **Memória de transação**: é uma técnica que simplifica a programação lock-free, permitindo que os processos executem operações em um estado compartilhado como se estivessem em uma transação atômica. Essas transações são garantidas para serem atômicas ou falhar de forma atômica.

Ao implementar técnicas de sincronização lock-free, é essencial garantir que as operações sejam atômicas e que o estado compartilhado seja consistente. A complexidade dos algoritmos lock-free geralmente aumenta à medida que o número de processos e recursos compartilhados aumentam.

Desafios e limitações da programação lock-free

Embora a programação lock-free ofereça vantagens significativas, ela também apresenta desafios e limitações que precisam ser cuidadosamente considerados:

- **Complexidade**: a implementação de algoritmos lock-free é geralmente mais complexa do que a programação com bloqueios. A otimização de desempenho e a garantia de corretude exigem um profundo conhecimento de mecanismos de sincronização e arquitetura de memória.

- **Sobrecarga de desempenho**: em alguns casos, as operações lock-free podem ter sobrecarga de desempenho, especialmente se os recursos compartilhados forem frequentemente acessados por um grande número de processos. Em sistemas de baixa concorrência, a programação com bloqueios pode ser mais eficiente.

- **Difícil de depurar**: os algoritmos lock-free são mais difíceis de depurar do que os algoritmos com bloqueios, pois o comportamento do sistema pode ser imprevisível devido à natureza concorrente das operações.

- **Limitação de recursos**: a programação lock-free pode ser limitada por recursos de hardware, como o número de núcleos de CPU ou a largura de banda da memória. Em sistemas com recursos limitados, a programação com bloqueios pode ser mais adequada.

Apesar desses desafios, a programação lock-free continua sendo uma técnica promissora para construir sistemas escaláveis, resilientes a falhas e eficientes.

Algoritmos wait-free e suas propriedades

A programação wait-free é um subconjunto da programação lock-free que garante que cada processo possa concluir sua operação em um número finito de passos, independentemente das ações dos outros processos. Isso significa que nenhum processo é obrigado a esperar por outros, mesmo que eles estejam em estado de espera ou falhem. Algoritmos wait-free oferecem um nível ainda maior de tolerância a falhas e escalabilidade em comparação com algoritmos lock-free. As principais propriedades dos algoritmos wait-free incluem:

- **Não bloqueio**: nenhum processo é bloqueado enquanto espera por outro processo.

- **Progresso garantido**: cada processo pode completar sua operação em um número finito de passos, independentemente das ações dos outros processos.

- **Tolerância a falhas**: os processos podem completar suas operações mesmo que outros falhem ou parem de responder.

- **Escalabilidade**: os algoritmos wait-free são altamente escaláveis, pois permitem que um grande número de processos compartilhe recursos de forma eficiente.

A implementação de algoritmos wait-free é ainda mais complexa do que a implementação de algoritmos lock-free. A complexidade de implementação depende do tipo de estrutura de dados e do nível de concorrência no sistema. Assim, a programação lock-free e wait-free tem encontrado ampla aplicação em várias áreas, incluindo:

- **Sistemas operacionais**: a programação lock-free é usada em sistemas operacionais modernos para gerenciar recursos compartilhados, como gerenciadores de memória, planejadores de processos e sistemas de arquivos. Esses sistemas precisam ser altamente confiáveis e tolerantes a falhas.

- **Bancos de dados**: sistemas de gerenciamento de bancos de dados utilizam algoritmos lock-free e wait-free para garantir a consistência dos dados e a tolerância a falhas em ambientes altamente concorrentes.

- **Processamento de transações**: é aplicada em sistemas de processamento de transações para garantir a atomicidade e a consistência das transações, mesmo que múltiplos processos estejam acessando o mesmo recurso ao mesmo tempo.

- **Computação de alto desempenho**: é essencial em sistemas de computação de alto desempenho, onde é preciso obter o máximo desempenho de múltiplos núcleos de CPU ou GPUs, sem sacrificar a precisão ou a confiabilidade.

- **Simulações de física**: é utilizada em simulações complexas para modelar sistemas com múltiplos objetos interagindo em um ambiente concorrente.

A capacidade de lidar com altos níveis de concorrência, garantir a consistência dos dados e tolerar falhas, torna a programação lock-free e wait-free uma técnica fundamental para construir sistemas robustos e eficientes em várias áreas da computação.

Implementar algoritmos lock-free e wait-free exige um profundo conhecimento de mecanismos de sincronização, arquitetura de memória e técnicas de otimização. Algumas boas práticas para implementar algoritmos lock-free e wait-free incluem:

- **Usar técnicas de sincronização atômicas**: as operações atômicas são essenciais para a programação lock-free. As operações CAS, load-linked/store-conditional e instruções de troca atômica são ferramentas importantes para atualizar recursos compartilhados de forma segura.

- **Garantir a corretude e o desempenho**: certificar-se de que os algoritmos lock-free estejam corretos e funcionem como esperado em cenários concorrentes, avaliando o desempenho dos algoritmos e otimizando-os para obter o máximo desempenho.

- **Utilizar ferramentas de depuração**: as ferramentas de depuração podem ajudar a identificar erros sutis e raças de dados em algoritmos lock-free, é recomendável utilizar essas ferramentas para garantir a corretude e a confiabilidade dos seus algoritmos.

- **Documentar seu código**: documentar claramente os algoritmos lock-free para que outros desenvolvedores possam entender e manter o código, incluindo explicações detalhadas das técnicas de sincronização e da lógica dos algoritmos.

Em suma, a programação lock-free e wait-free oferece vantagens significativas em termos de tolerância a falhas, escalabilidade e desempenho, mas exige um profundo conhecimento e cuidado na implementação. Ao seguir as boas práticas e entender as limitações, é possível construir sistemas robustos e eficientes que possam lidar com altos níveis de concorrência em ambientes desafiadores.

Transação atômica

A transação atômica é um conjunto de operações que, em um processo, são tratadas como uma unidade indivisível. Isso significa que todas as operações dentro da transação são executadas com sucesso ou nenhuma delas é executada. Se qualquer uma das operações falhar, a transação inteira é revertida para seu estado original, garantindo a consistência dos dados. As características principais de uma transação atômica são:

- **Atomicidade**: garante que as operações da transação sejam tratadas como uma unidade indivisível. Ou todas as operações são concluídas com sucesso ou nenhuma delas é executada.
- **Consistência**: assegura que a transação deixa o sistema em um estado válido, mesmo que ocorra uma falha.
- **Isolamento**: as transações são isoladas umas das outras. As alterações feitas em uma transação não são visíveis para outras transações até que a primeira seja concluída.
- **Durabilidade**: assegura que as alterações feitas por uma transação concluída sejam persistidas permanentemente, mesmo que ocorra uma falha.

Transações atômicas são amplamente utilizadas em diversos cenários de processo, incluindo: banco de dados, afim de garantir a consistência de dados em operações como transferências bancárias, atualizações de saldo e inserções de registros; em sistemas de transações online (OLTP), no processamento de pedidos de clientes, gerenciamento inventário e realização de transações financeiras em tempo real; nos serviços web, para assegurar que as chamadas API sejam concluídas com sucesso ou revertidas se ocorrerem falhas, garantindo a consistência dos dados entre os serviços web; e no processamento de mensagens, afim de garantir que mensagens sejam processadas com sucesso, mesmo em caso de falhas de rede ou de servidor, o que garante a integridade das mensagens e a consistência dos dados.

Os principais benefícios da implementação de transações atômicas incluem: a consistência dos dados, que assegura que eles permaneçam íntegros e consistentes, mesmo em caso de falhas; a integridade do processo, garantindo que os processos sejam concluídos com sucesso ou revertidos para o estado original, evitando inconsistências; segurança de transações, protegendo os dados de erros e acesso não autorizado e garantindo a integridade das informações.

Apesar dos benefícios presentes, existem desafios na implementação de transações atômicas, como: a possibilidade das operações atômicas afetarem o desempenho do sistema, especialmente em transações complexas; o aumento da complexidade que a implementação das transações atômicas podem exigir no código e na infraestrutura; e o fato de que seria necessário garantir mecanismos robustos para lidar com falhas e garantir a integridade das transações.

Além disso, para uma implementação bem-sucedida dessas transações, é crucial seguir práticas melhores, como definir claramente os escopos das transações, garantindo a atomicidade e a consistência; implementar mecanismos de recuperação de falhas, planejando cenários de falha e implementando mecanismos robustos de recuperação de dados e rollback de transações; monitorar e otimizar o desempenho, ao monitorar o impacto das transações atômicas no desempenho do sistema para garantir a melhor experiência possível; e, por fim, realizar testes rigorosos para garantir que as transações atômicas funcionem corretamente em diferentes cenários e cargas.

A implementação de transações atômicas requer uma análise cuidadosa dos requisitos específicos do sistema e um planejamento estratégico. Seguindo as melhores práticas, é possível garantir a consistência dos dados, a integridade dos processos e a segurança das transações, com mínimo impacto no desempenho do sistema.

Escalonamento de processos

Escalonamento de processos é uma técnica fundamental em sistema operacionais, uma vez que é responsável por determinar a ordem em que os processos compartilham recursos computacionais, como CPU e memória. O objetivo principal do escalonamento é garantir que cada processo tenha acesso à CPU de forma justa e eficiente, maximizando o uso do sistema e o desempenho geral.

Existem diversos algoritmos de escalonamento de processos, cada um com suas próprias características e aplicações. A escolha do algoritmo ideal depende de fatores como o tipo de sistema, a natureza dos processos e as prioridades estabelecidas para cada um deles.

Algoritmo FIFO (First-In, First-Out), algoritmo Round-Robin e algoritmo Multilevel Queue

O algoritmo FIFO, também conhecido como First Come, First Served (FCFS), é um dos algoritmos de escalonamento de processos mais simples. Ele funciona de acordo com o princípio do primeiro a chegar, primeiro a ser atendido. Os processos são colocados em uma fila e são atendidos na ordem em que chegaram ao sistema, sem considerar suas prioridades ou necessidades.

O FIFO é fácil de implementar e entender, mas pode apresentar desvantagens em termos de desempenho, pois um processo com alta prioridade ou curto tempo de execução pode ter que esperar por um processo longo e de baixa prioridade que chegou antes.

Ao tratarmos de um algoritmo de escalonamento preemptivo, temos o algoritmo Round-Robin, que pode interromper um processo em execução para dar oportunidade a outros processos. Ele funciona

distribuindo um tempo de execução fixo, chamado de quantum, para cada processo. Esses processos são colocados em uma fila circular e a CPU alterna entre eles, executando cada um por um quantum de tempo.

O Round-Robin garante um tempo de resposta mais rápido para processos com tempo de execução curto e oferece um bom desempenho geral para sistemas com vários deles. No entanto, o algoritmo pode gerar sobrecarga devido ao contexto switching frequente, o que pode afetar o desempenho para processos longos.

Quanto ao algoritmo Multilevel Queue, ele é uma abordagem mais sofisticada que divide os processos em várias filas com diferentes prioridades. Cada fila pode ter seu próprio algoritmo de escalonamento, como FIFO ou Round-Robin, e os processos são movidos entre as filas de acordo com suas necessidades e prioridades.

Este algoritmo oferece flexibilidade e melhor desempenho para sistemas que requerem diferentes tipos de processos, com diferentes prioridades e tempos de execução. No entanto, o Multilevel Queue pode ser mais complexo de implementar e configurar, exigindo uma gestão cuidadosa das prioridades e das regras de movimentação entre filas.

Ao compararmos esses três tipos de algoritmos – FIFO, Round-Robin e Multilevel Queue –, vemos que eles apresentam características distintas, cada um com suas vantagens e desvantagens. O FIFO é simples e fácil de implementar, mas pode levar a longos tempos de espera para processos de alta prioridade. O Round-Robin garante uma resposta rápida para processos curtos, mas pode gerar sobrecarga devido ao contexto switching frequente. O Multilevel Queue oferece flexibilidade e melhor desempenho para sistemas com diferentes tipos de processos, mas exige uma implementação mais complexa.

A escolha do algoritmo ideal depende do tipo de sistema, das necessidades dos processos e das prioridades estabelecidas. Em sistemas com processos de tempo de execução variável, o Round-Robin ou o Multilevel Queue podem ser mais eficientes do que o FIFO.

No entanto, diversos fatores devem ser considerados ao escolher um algoritmo de escalonamento de processos, incluindo:

- **Tempo de resposta**: o tempo que leva para um processo iniciar e retornar um resultado.
- **Tempo de espera**: o tempo que um processo passa na fila aguardando a sua vez de ser executado.
- **Utilização da CPU**: a porcentagem do tempo que a CPU está ocupada executando processos.
- **Sobrecarga**: a quantidade de tempo gasto no contexto switching entre processos.
- **Prioridade dos processos**: as diferentes prioridades atribuídas aos processos, como tempo real, interativo ou em lote.

A otimização do escalonamento de processos envolve encontrar um equilíbrio entre esses fatores para atingir o melhor desempenho geral do sistema.

Quadro 4 – Vantagens e desvantagens de cada algoritmo

Algoritmo	Vantagens	Desvantagens
FIFO	Simples de implementar e entender	Pode levar a longos tempos de espera para processos de alta prioridade
Round-Robin	Garante um tempo de resposta mais rápido para processos curtos	Pode gerar sobrecarga devido ao contexto switching frequente
Multilevel Queue	Oferece flexibilidade e melhor desempenho para sistemas com diferentes tipos de processos	Pode ser mais complexo de implementar e configurar

Implementação prática dos algoritmos

A implementação dos algoritmos de escalonamento de processos depende do sistema operacional em uso. Os sistemas operacionais modernos geralmente fornecem mecanismos para configurar e gerenciar o escalonamento de processos, incluindo opções para escolher o algoritmo de escalonamento, definir prioridades de processos e gerenciar o tempo de execução.

Alguns sistemas operacionais, como o Linux, oferecem opções para configurar o algoritmo de escalonamento de processos, como "SCHED_FIFO", "SCHED_RR" e "SCHED_OTHER". Essas opções permitem que os usuários escolham o algoritmo mais adequado para as necessidades do sistema.

A escolha do algoritmo de escalonamento de processos adequado depende dos objetivos do sistema e das necessidades dos processos. É fundamental considerar os fatores descritos anteriormente e avaliar as vantagens e desvantagens de cada algoritmo antes de tomar uma decisão. Para otimizar o desempenho do sistema, algumas práticas recomendadas incluem:

- Definir prioridades para os processos de acordo com suas necessidades.
- Gerenciar o tempo de execução dos processos para evitar que um deles monopolize a CPU.
- Monitorar o desempenho do sistema e ajustar o algoritmo de escalonamento se necessário.

Ao implementar e gerenciar o escalonamento de processos, é importante manter uma visão holística do sistema e buscar um equilíbrio entre o tempo de resposta, o tempo de espera e a utilização da CPU.

5.3 Monitores e semáforos

Os monitores, introduzidos por C.A.R. Hoare e Per Brinch Hansen na década de 1970, são uma abstração de alto nível para sincronização de processos em sistemas operacionais. Eles surgiram como uma resposta às dificuldades encontradas no uso de semáforos, oferecendo uma abordagem mais estruturada e menos propensa a erros para lidar com a concorrência.

Um monitor pode ser entendido como um tipo abstrato de dados que encapsula variáveis compartilhadas e os procedimentos que operam sobre essas variáveis. A característica fundamental dos monitores é que eles garantem exclusão mútua automaticamente, ou seja, apenas um processo pode estar ativo dentro do monitor em um determinado momento.

Esta garantia de exclusão mútua é implementada pelo próprio sistema operacional, liberando o programador da responsabilidade de gerenciar explicitamente a sincronização. Isso reduz significativamente a complexidade do código e minimiza a ocorrência de erros comuns em programação concorrente, como condições de corrida e deadlocks.

Os principais benefícios de um monitor são:

- **Encapsulamento**: monitores agrupam dados e operações relacionadas, promovendo modularidade.
- **Exclusão mútua**: garantem que apenas um processo execute as operações do monitor por vez.
- **Sincronização**: oferecem mecanismos para coordenar a execução de processos concorrentes.
- **Abstração**: proporcionam uma interface de alto nível para lidar com concorrência.

Definição e utilidade dos monitores

Monitores são estruturas de programação que facilitam a sincronização entre processos concorrentes em um sistema operacional. Eles são definidos como um conjunto de procedimentos, variáveis e estruturas de dados agrupados em um tipo especial de módulo ou pacote. A característica distintiva dos monitores é que eles garantem que apenas um processo possa estar ativo dentro do monitor em qualquer momento. A utilidade dos monitores se estende por várias áreas críticas do gerenciamento de sistemas operacionais:

- **Controle de concorrência**: simplificam o controle de acesso a recursos compartilhados, evitando conflitos entre processos concorrentes.
- **Sincronização de processos**: fornecem mecanismos para coordenar a execução de múltiplos processos, permitindo que eles aguardem condições específicas antes de prosseguir.
- **Gerenciamento de recursos**: são eficazes na alocação e liberação controlada de recursos do sistema, como memória, dispositivos de E/S e arquivos.
- **Prevenção de deadlocks**: ao estruturar o acesso a recursos de forma organizada, monitores ajudam a prevenir situações de impasse entre processos.

A implementação de monitores em sistemas operacionais modernos tem se mostrado crucial para garantir a integridade e eficiência de sistemas multiprocessados e de multitarefas. Eles oferecem uma abstração de alto nível que permite aos programadores focarem na lógica de negócios, deixando os detalhes complexos de sincronização para o sistema operacional gerenciar.

A fim de nos aprofundarmos no conceito do uso de monitores, é importante abordamos sua estrutura que, conforme descrito por Tanenbaum e Austin (2013), consiste em quatro componentes principais:

- **Variáveis de dados**: são as variáveis compartilhadas que representam o estado do monitor.
- **Procedimentos**: métodos que operam sobre as variáveis de dados e implementam a lógica do monitor.
- **Inicialização**: código executado quando o monitor é criado, inicializando as variáveis de dados.
- **Fila de entrada**: uma fila implícita onde os processos aguardam para entrar no monitor.

Um monitor funciona a partir de um padrão específico. Quando um processo deseja acessar o monitor, ele entra na fila de entrada; caso o monitor esteja livre, o primeiro processo na fila ganha acesso exclusivo; então, ele executa um dos procedimentos do monitor; e, ao concluir, o processo libera o monitor, permitindo que o próximo na fila entre.

Uma característica crucial dos monitores é a capacidade de suspender a execução de um processo dentro do monitor e liberar o acesso para outros processos. Isso é realizado através de variáveis de condição, que permitem que um processo espere por uma condição específica antes de continuar sua execução. As variáveis de condição têm duas operações principais:

- **wait()**: suspende o processo atual e libera o monitor.
- **signal()**: acorda um processo suspenso, se houver algum esperando na variável de condição.

Essa estrutura permite uma sincronização mais flexível e eficiente entre processos, evitando espera ocupada e otimizando o uso dos recursos do sistema.

Condição de espera e condição de progresso nos monitores

As condições de espera e progresso são conceitos fundamentais no funcionamento dos monitores em sistemas operacionais. Elas são implementadas através de variáveis de condição, que permitem uma sincronização mais refinada entre processos.

A condição de espera ocorre quando um processo dentro do monitor precisa aguardar a ocorrência de um evento específico antes de prosseguir. Isso é implementado através da operação **wait()** em uma variável de condição. Nessa operação, o processo é suspenso e colocado em uma fila associada à variável de condição; então, o monitor é liberado, permitindo que outro processo entre; em sua etapa final, o processo permanece suspenso até que seja acordado por uma operação signal().

A condição de progresso é alcançada quando um processo dentro do monitor sinaliza que uma condição específica foi satisfeita, potencialmente permitindo que processos em espera continuem sua execução. Isso é implementado através da operação **signal()** em uma variável de condição. Nessa operação, se houver processos esperando na fila da variável de condição, um deles é acordado. O processo acordado, então, retoma sua execução de onde parou; por fim, o processo que chamou signal() continua sua execução normalmente.

Na operação signal(), é importante notar que existem duas semânticas diferentes para a operação:

- **Hoare's Signaling**: o processo que chama signal() imediatamente cede o controle do monitor para o processo acordado. Isso garante que o processo acordado execute imediatamente após o signal(), mas pode levar a uma troca de contexto adicional.

- **Hansen's Signaling**: o processo que chama signal() continua sua execução até sair do monitor. O processo acordado só ganha o controle do monitor quando o processo sinalizador sai. Isso é mais eficiente em termos de troca de contexto, mas pode levar a condições de corrida se não for usado com cuidado.

A escolha entre essas semânticas depende da implementação específica do sistema operacional e das necessidades da aplicação. Ambas têm suas vantagens e desvantagens em termos de desempenho e previsibilidade.

Conceito de semáforos

Os semáforos, que foram introduzidos por Edsger Dijkstra, em 1965, são uma ferramenta fundamental para sincronização em sistemas operacionais. Eles representam uma abstração de baixo nível que permite o controle de acesso a recursos compartilhados e a coordenação entre processos concorrentes.

Um semáforo é essencialmente uma variável inteira não negativa que, além das operações de inicialização, suporta duas operações atômicas principais:

- **P() ou wait()**: decrementa o valor do semáforo. Se o valor resultante for negativo, o processo é bloqueado e colocado em uma fila de espera.

- **V() ou signal()**: incrementa o valor do semáforo. Se houver processos bloqueados na fila de espera, um deles é desbloqueado.

A atomicidade dessas operações é crucial, garantindo que não ocorram condições de corrida durante a manipulação do semáforo. Isso significa que, uma vez iniciada, uma operação wait() ou signal() será concluída sem interrupção.

Os semáforos são utilizados para resolver diversos problemas clássicos de sincronização, como: exclusão mútua, garantindo que apenas um processo acesse um recurso compartilhado por vez; sincronização de eventos, fazendo com que um processo aguarde até que outro complete uma determinada tarefa; controle de concorrência, limitando o número de processos que podem acessar simultaneamente um conjunto de recursos.

Apesar de sua simplicidade conceitual, os semáforos podem ser usados para construir soluções elegantes para problemas complexos de sincronização. No entanto, seu uso inadequado pode levar a erros sutis e difíceis de detectar, como deadlocks e starvation.

Tipos de semáforos: binários e contadores

Os semáforos em sistemas operacionais são classificados em dois tipos principais: semáforos binários e semáforos contadores (ou gerais). Cada tipo possui características e aplicações específicas no contexto da sincronização de processos.

Também conhecidos como mutex, os semáforos binários podem assumir apenas dois valores: 0 ou 1. Eles são utilizados principalmente para garantir a exclusão mútua no acesso a recursos compartilhados, onde:

- **Valor 1**: indica que o recurso está disponível.
- **Valor 0**: indica que o recurso está em uso ou bloqueado.

Os semáforos binários são ideais para situações onde apenas um processo pode acessar um recurso por vez, como escrever em um arquivo ou modificar uma estrutura de dados compartilhada. As principais aplicações dos semáforos binários são em proteção de seções críticas, implementação de locks e sincronização simples entre dois processos.

Os semáforos contadores, por outro lado, podem assumir qualquer valor inteiro não negativo. Eles são mais versáteis e são usados para controlar o acesso a recursos que têm múltiplas instâncias. Nos semáforos contadores, o valor inicial representa o número de unidades disponíveis do recurso, sendo que cada operação P() decrementa o contador, representando a alocação de uma unidade do recurso, e cada operação V() incrementa o contador, representando a liberação de uma unidade do recurso.

Semáforos contadores são úteis em cenários como o controle de um pool de conexões de banco de dados ou a gestão de um conjunto de impressoras compartilhadas. As principais aplicações dos semáforos contadores são no controle de recursos com múltiplas instâncias, na implementação de buffers limitados, e na coordenação de grupos de processos.

A escolha entre semáforos binários e contadores depende da natureza do problema de sincronização a ser resolvido. Em muitos casos, os semáforos binários são suficientes e mais simples de usar, mas para cenários mais complexos, envolvendo múltiplos recursos ou produtores/consumidores, os semáforos contadores oferecem maior flexibilidade.

Operações básicas com semáforos: P() e V()

As operações P() e V(), também conhecidas como wait() e signal(), respectivamente, são as operações fundamentais realizadas em semáforos. Essas operações, introduzidas por Dijkstra, são a base para a implementação de mecanismos de sincronização em sistemas operacionais. A operação P(), derivada da palavra holandesa "proberen" (testar), é usada quando um processo deseja adquirir um recurso ou entrar em uma seção crítica.

Sua implementação pode ser descrita como vemos na figura 84.

```
P(semáforo S) {
    S = S - 1;
    if (S < 0) {
        Bloquear o processo chamador;
        Adicionar processo à fila de espera de S;
    }
}
```

Figura 84 – Operação P(), imagem gerada em IA Image Generator

A operação P() é uma operação atômica, ou seja, não pode ser interrompida no meio de sua execução. Assim, se o valor do semáforo se tornar negativo, o processo é bloqueado e colocado em uma fila de espera, o que faz com que o bloqueio ocorra antes que o processo possa continuar sua execução.

A operação V(), derivada da palavra holandesa "verhogen" (incrementar), é usada quando um processo libera um recurso ou sai de uma seção crítica. Sua implementação pode ser descrita como vemos na figura 85.

```
V(semáforo S) {
    S = S + 1;
    if (S <= 0) {
        Remover um processo P da fila de espera de S;
        Desbloquear P;
    }
}
```

Figura 85 – Operação V(), imagem gerada em IA Image Generator

A operação V() também é uma operação atômica. Assim, se o valor do semáforo era negativo ou zero, um processo bloqueado é liberado da fila de espera. No entanto, a operação V() nunca bloqueia o processo que a chama.

A implementação dessas operações é geralmente feita pelo kernel do sistema operacional, garantindo sua atomicidade e eficiência. O uso correto de P() e V() permite a criação de soluções elegantes para problemas complexos de sincronização, como o problema do produtor-consumidor, dos leitores-escritores e o do jantar dos filósofos.

Problema do produtor-consumidor

Este problema envolve dois tipos de processos: produtores, que geram dados, e consumidores, que utilizam esses dados, compartilhando um buffer de tamanho fixo.

```
semaphore mutex = 1;  // Exclusão mútua para o buffer
semaphore empty = N;  // Inicialmente, N slots vazios
semaphore full = 0;   // Inicialmente, nenhum slot cheio

Produtor:
    P(empty);
    P(mutex);
    // Produzir item e adicionar ao buffer
    V(mutex);
    V(full);S

Consumidor:
    P(full);
    P(mutex);
    // Consumir item do buffer
    V(mutex);
    V(empty);
```

Figura 86 – Problema do produtor-consumidor, imagem gerada em IA Image Generator

Problema dos leitores-escritores

Neste cenário, múltiplos processos leitores podem ler simultaneamente, mas escritores precisam de acesso exclusivo.

```
semaphore wrt = 1;     // Exclusão mútua para escritores
semaphore mutex = 1;   // Proteção da variável read_count
int read_count = 0;    // Número de leitores ativos

Escritor:
    P(wrt);
    // Escrever no recurso compartilhado
    V(wrt);

Leitor:
    P(mutex);
    read_count++;
    if (read_count == 1) P(wrt);
    V(mutex);
    // Ler o recurso compartilhado
    P(mutex);
    read_count--;
    if (read_count == 0) V(wrt);
    V(mutex);
```

Figura 87 – Problema dos leitores-escritores, imagem gerada em IA Image Generator

Problema do jantar dos filósofos

Este problema ilustra os desafios de alocação de recursos e prevenção de deadlocks.

```
semaphore fork[5] = {1, 1, 1, 1, 1};
semaphore room = 4;  // Limita o número de filósofos na mesa

Filósofo i:
    P(room);
    P(fork[i]);
    P(fork[(i+1) % 5]);
    // Comer
    V(fork[(i+1) % 5]);
    V(fork[i]);
    V(room);
```

Figura 88 – Problema do jantar dos filósofos, imagem gerada em IA Image Generator

Esses exemplos demonstram como os semáforos podem ser utilizados para coordenar o acesso a recursos compartilhados, garantir exclusão mútua e prevenir condições de corrida. A chave para o uso eficaz de semáforos está em identificar corretamente os recursos compartilhados e as condições de sincronização necessárias.

Comparação entre monitores e semáforos

Monitores e semáforos são duas abordagens fundamentais para lidar com sincronização em sistemas operacionais. Embora ambos sejam utilizados para resolver problemas de concorrência, eles diferem significativamente em sua filosofia, implementação e facilidade de uso.

Os semáforos possuem um mecanismo de sincronização de baixo nível, requerendo uma implementação explícita de exclusão mútua. Além disso, são mais flexíveis, mas propensos a erros, podendo ser usados para sincronização entre processos, onde as operações P() e V() devem ser implementadas corretamente pelo programador. Algumas vantagens dos semáforos são:

- Maior flexibilidade na implementação de soluções de sincronização complexas.
- Podem ser usados em uma variedade maior de situações, incluindo sincronização entre processos.
- Geralmente têm melhor desempenho em sistemas com muita contenção.

Os monitores, por sua vez, possuem uma abstração de alto nível para sincronização, garantindo exclusão mútua automaticamente, além de serem mais seguros e fáceis de usar, no entanto, são menos flexíveis. Geralmente, os monitores são usados para sincronização dentro de um mesmo processo, onde o compilador e o sistema de tempo de execução gerenciam a sincronização. Algumas vantagens dos monitores são:

- Encapsulamento de dados e operações, promovendo modularidade.

- Redução de erros de programação relacionados à sincronização.
- Código mais legível e fácil de manter.
- Suporte a variáveis de condição para sincronização mais refinada.

A escolha entre monitores e semáforos depende de vários fatores, como a complexidade do problema de sincronização, os requisitos de desempenho do sistema, a experiência da equipe de desenvolvimento, e o suporte da linguagem de programação e do sistema operacional.

Em geral, monitores são preferidos em situações onde a clareza e a segurança do código são prioritárias, enquanto semáforos são escolhidos quando é necessário um controle mais fino sobre a sincronização ou quando se lida com sincronização entre processos.

Para Tanenbaum e Austin (2013), os monitores e semáforos em sistemas operacionais são mecanismos que desempenham papéis cruciais na gestão de concorrência e sincronização em sistemas computacionais modernos.

Monitores, com sua abordagem de alto nível e encapsulamento, oferecem uma solução elegante e segura para muitos problemas de sincronização. Eles simplificam o desenvolvimento de software concorrente ao automatizar a exclusão mútua e fornecer um mecanismo estruturado para coordenação entre processos. A capacidade de agrupar dados e operações relacionadas em uma única unidade aumenta a modularidade e reduz a probabilidade de erros de sincronização.

Por outro lado, os semáforos são considerados ferramentas de baixo nível, que proporcionam maior flexibilidade e controle fino sobre a sincronização. Sua simplicidade conceitual e eficiência os tornam ideais para uma ampla gama de problemas de sincronização, especialmente quando se trata de coordenação entre processos em nível de sistema operacional. Ao comparar ambos mecanismos, algumas considerações devem ser analisadas, tais como:

- A evolução contínua desses mecanismos para atender às demandas de sistemas cada vez mais complexos e distribuídos.
- O desenvolvimento de ferramentas e frameworks que facilitam o uso correto de monitores e semáforos, minimizando erros comuns.
- A integração desses conceitos em paradigmas de programação emergentes, como computação paralela e distribuída em larga escala.

Em última análise, a escolha entre monitores e semáforos deve ser baseada nas necessidades específicas do projeto, considerando fatores como complexidade do problema, requisitos de desempenho, facilidade de manutenção e experiência da equipe de desenvolvimento. Ambos os mecanismos continuarão a ser fundamentais no arsenal de ferramentas para o desenvolvimento de sistemas operacionais e aplicações concorrentes eficientes e confiáveis.

6 GERENCIAMENTO DO PROCESSADOR

Um processo é o contexto básico dentro do qual toda a atividade solicitada por usuários é atendida dentro do sistema operacional. Para ser compatível com outros sistemas Unix, o Linux deve usar um modelo de processo semelhante ao de outras versões do Unix. O Linux opera diferentemente do Unix em alguns aspectos chave.

O modelo de processos com fork() e exec()

O princípio básico do gerenciamento de processos no Unix é separar em dois passos duas operações que são usualmente combinadas em uma: a criação de um novo processo e a execução de um novo programa. Um novo processo é criado pela chamada de sistema fork() e um novo programa é executado após uma chamada a exec(); essas são duas funções totalmente distintas. Podemos criar um novo processo com fork() sem executar um novo programa, onde o novo subprocesso simplesmente continua a executar exatamente o mesmo programa que o primeiro processo (pai) estava executando, precisamente no mesmo ponto.

Da mesma forma, a execução de um novo programa não requer que um novo processo seja criado primeiro. Qualquer processo pode chamar exec() a qualquer momento. Um novo objeto binário é carregado no espaço de endereçamento do processo e o novo executável começa a sua execução no contexto do processo existente.

Esse modelo tem a vantagem de ser muito simples, sendo que, nele, não é necessário especificar cada detalhe do ambiente de um novo programa na chamada de sistema que o executa. Simplesmente, o novo programa é executado no ambiente existente. Se um processo-pai quer modificar o ambiente em que um novo programa deve ser executado, ele pode criar uma ramificação e, então, ainda executando o original em um processo-filho, fazer quaisquer chamadas de sistema necessárias à modificação desse processo-filho antes de finalmente executar o novo programa.

Assim, no Unix, um processo inclui todas as informações que o sistema operacional deve manter para rastrear o contexto de execução individual de um único programa. De modo geral, as propriedades dos processos classificam-se em três grupos: a identidade, o ambiente e o contexto do processo.

6.1 Identidade do processo

ID do processo (PID – process ID)

Cada processo tem um identificador exclusivo. O PID é usado para especificar o processo para o sistema operacional quando uma aplicação faz uma chamada de sistema para notificar o processo, modificá-lo ou esperar por ele. Identificadores adicionais associam o processo a um grupo de processos (tipicamente, uma árvore de processos criados por um único comando de usuário) e a uma sessão de login.

Credenciais

Cada processo deve ter um ID de usuário associado e um ou mais IDs de grupo, que determinem seus direitos de acesso a recursos e arquivos do sistema.

Personalidade

As personalidades de processos não são tradicionalmente encontradas em sistemas Unix, mas no Linux cada processo tem um identificador de personalidade associado que pode modificar ligeiramente a semântica de certas chamadas de sistema. As personalidades são usadas principalmente por bibliotecas de emulação para solicitar que as chamadas de sistema sejam compatíveis com certas versões do Unix.

Espaço de nomes

Cada processo é associado a uma visão específica da hierarquia do sistema de arquivos, denominada de espaço de nomes. A maioria dos processos compartilha um espaço de nomes comum e, portanto, opera em uma hierarquia de sistema de arquivos compartilhada. No entanto, os processos e seus filhos podem ter espaços de nomes diferentes, cada um com uma hierarquia de sistema de arquivos exclusiva, tendo seu próprio diretório raiz e conjunto de sistemas de arquivos montados.

A maioria desses identificadores fica sob o controle limitado do próprio processo. Os identificadores de grupo de processos e de sessão podem ser alterados se o processo quiser iniciar um novo grupo ou sessão. Suas credenciais podem ser alteradas, estando sujeitas às verificações de segurança apropriadas. No entanto, o PID primário de um processo é inalterável e identifica de maneira exclusiva esse processo até o seu encerramento.

Ambiente do processo

O ambiente de um processo é herdado do pai e é composto por dois vetores terminados em nulo: o vetor de argumentos e o vetor de ambiente. O vetor de argumentos simplesmente lista os argumentos da linha de comando usados para invocar o programa em execução, e, por convenção, ele começa com o nome do próprio programa. O vetor de ambiente é uma lista de pares "NOME=VALOR" que associam variáveis de ambiente nomeadas a valores textuais arbitrários. O ambiente não é mantido na memória do kernel, mas armazenado no próprio espaço de endereçamento de modalidade de usuário do processo, como o primeiro dado no topo da pilha do processo.

Os vetores de argumentos e de ambiente não são alterados quando um novo processo é criado. O novo processo-filho herdará o ambiente de pai. No entanto, um ambiente completamente novo é estabelecido quando um novo programa é invocado. Ao chamar exec(), um processo deve fornecer o ambiente para o novo programa. O kernel passa essas variáveis de ambiente para o próximo programa, substituindo o ambiente corrente do processo. Alternativamente, o kernel deixa os vetores de ambiente e linha de comando inalterados, deixando sua interpretação inteiramente a cargo das bibliotecas e aplicações de modalidade de usuário.

A passagem de variáveis de ambiente de um processo para o próximo e a herança dessas variáveis pelos filhos de um processo fornecem maneiras flexíveis de passar informações para componentes de software do sistema de modalidade de usuário. Diversas variáveis de ambiente têm significados convencionais para partes relacionadas do software do sistema. Por exemplo, a variável TERM é posicionada para nomear o tipo de terminal conectado a uma sessão de login de usuário. Muitos programas usam essa variável para determinar como executar operações na tela do usuário, tais como mover o cursor e rolar uma região de texto. Programas com suporte multilíngue usam a variável LANG para determinar em que idioma devem exibir mensagens do sistema para programas que também incluam esse suporte.

O mecanismo de variáveis de ambiente personaliza o sistema operacional por processo, permitindo aos usuários a escolha de seus idiomas ou de seus editores independente uns dos outros.

Contexto do processo

As propriedades de identidade e de ambiente do processo são usualmente estabelecidas quando um processo é criado e não são alteradas durante a existência dele. Um processo pode optar por alterar alguns aspectos de sua identidade se precisar fazê-lo ou pode alterar seu ambiente. Por outro lado, o contexto do processo é o estado do programa em execução em determinado momento, mudando constantemente. O contexto do processo inclui as seguintes partes:

- **Contexto de scheduling**: a parte mais importante do contexto do processo é seu contexto de scheduling, que são as informações que o scheduler precisa para suspender e reiniciar o processo. Essas informações incluem as cópias salvas de todos os registradores do processo. Registradores de ponto flutuante são armazenados separadamente e são restaurados apenas quando necessário. Assim, processos que não usam aritmética de ponto flutuante não incorrem no overhead de salvar esse estado. O contexto de scheculing também inclui informações sobre a prioridade do scheduling e sobre quaisquer sinais pendentes em espera para serem distribuídos ao processo. Uma parte chave do contexto de scheduling é a pilha do kernel do processo, uma área separada da memória do kernel reservada para uso pelo código em modalidade de kernel. Tanto as chamadas de sistema quanto as interrupções que ocorrem enquanto o processo está em execução usarão essa pilha.

- **Contabilidade**: o kernel mantém informações de contabilidade sobre os recursos que estão sendo correntemente consumidos pelos processos e o total de recursos consumidos pelos processos, em todo o seu tempo de vida, até o momento.

- **Tabela de arquivos**: a tabela de arquivos é um array de ponteiros para estruturas de arquivos do kernel representando arquivos abertos. Ao fazer chamadas de sistema de I/O de arquivo, os processos referenciam os arquivos por um inteiro, conhecido como descrito de arquivo, File Descriptor (FD), que o kernel usa como índice nessa tabela.

- **Contexto do sistema de arquivos**: enquanto a tabela de arquivos lista os arquivos abertos existentes, o contexto do sistema de arquivos é aplicado a solicitações de abertura de novos arquivos. O contexto do sistema de arquivos inclui o diretório raiz, o diretório de trabalho corrente e o espaço de nomes do processo.

- **Tabela de manipuladores de sinais**: os sistemas Unix podem distribuir sinais assíncronos para um processo em resposta a vários eventos externos. A tabela de manipuladores de sinais define a ação a ser tomada em resposta a um sinal específico. Ações válidas incluem ignorar o sinal, encerrar o processo e invocar uma rotina no espaço de endereçamento do processo.

- **Contexto da memória virtual**: o contexto da memória virtual descreve o conteúdo completo do espaço de endereçamento privado de um processo.

Processos e threads

O Linux fornece a chamada de sistema fork() que duplica um processo sem carregar uma nova imagem executável. Além disso, fornece o recurso de criação de threads por meio da chamada de sistema clone(). No entanto, o Linux não faz a distinção entre processos e threads. Na verdade, ele geralmente usa o termo tarefa, em vez de processo ou thread, quando se refere a um fluxo de controle dentro de um programa. A chamada de sistema clone() comporta-se de forma idêntica a fork(), exceto por aceitar como argumentos um conjunto de flags que definem os recursos que são compartilhados entre pai e filho (enquanto um processo criado com fork() não compartilha recursos com seu pai). Os flags incluem:

Quadro 5 – Flags em um sistema clone()

Flag	Significado
CLONE_FS	Informações do sistema de arquivos são compartilhadas
CLONE_VM	O mesmo espaço de memória é compartilhado
CLONE_SIGHAND	Manipuladores de sinais são compartilhados
CLONE_FILES	O conjunto de arquivos abertos é compartilhado

Portanto, se clone() receber os flags CLONE_FS, CLONE_VM, CLONE_SIGHAND e CLONE_FILES, a tarefa-pai e a tarefa-filha compartilharão as mesmas informações do sistema de arquivos (tal como o diretório de trabalho corrente), o mesmo espaço de memória, os mesmos manipuladores de sinais e o mesmo conjunto de arquivos abertos. O uso de clone(), dessa forma, é equivalente à criação de um thread em outros sistemas, já que a tarefa-pai compartilha a maioria de seus recursos com sua tarefa-filha. No entanto, se nenhum desses flags estiver posicionado quando clone() for invocada, os recursos associados não serão compartilhados, resultando em uma funcionalidade semelhante a da chamada de sistema fork().

A falta de diferenciação entre processos e threads é possível, porque o Linux não mantém o contexto inteiro de um processo dentro da estrutura de dados do processo principal. Em vez disso, ele mantém o contexto dentro de subcontextos independentes. Assim, o contexto do sistema de arquivos, a tabela de descritores de arquivos, a tabela de manipuladores de sinais e o contexto de memória virtual de um processo são mantidos em estruturas de dados separadas.

A estrutura de dados do processo contém ponteiros para essas outras estruturas. Desse modo, qualquer número de processos pode facilmente compartilhar um subcontexto apontando para o mesmo subcontexto e incrementando uma contagem de referência.

Os argumentos da chamada de sistema clone() informam quais subcontextos ela deve copiar e quais ela deve compartilhar. O novo processo recebe sempre uma nova identidade e um novo contexto de scheduling, sendo essas as bases de um processo no Linux. Dependendo dos argumentos passados, no entanto, o kernel pode criar novas estruturas de dados de subcontexto inicializadas para serem cópias da estrutura do pai — ou configurar o novo processo para usar as mesmas estruturas de dados de subcontexto usadas pelo pai. A chamada de sistema fork() não é nada mais do que um caso especial de clone() que copia todos os subcontextos sem compartilhar qualquer um deles.

6.2 Critérios e tipos de escalonamento

O Linux tem dois algoritmos diferentes de scheduling de processos: um deles é o algoritmo de compartilhamento de tempo para a obtenção de scheduling justo e preemptivo entre múltiplos processos; o outro foi projetado para tarefas de tempo real com prioridades absolutas de execução.

O algoritmo de scheduling usado para tarefas rotineiras de compartilhamento de tempo recebeu uma revisão maior na versão 2.6 do kernel. Versões anteriores executavam uma variação do algoritmo de scheduling tradicional do Unix. Esse algoritmo não fornece suporte adequado a sistemas SMP, não escala bem à medida que o número de tarefas no sistema cresce e não mantém a justiça entre tarefas interativas, principalmente em sistemas como computadores desktop e dispositivos móveis.

O scheduler de processos foi revisado, pela primeira vez, na versão 2.5 do kernel. Essa versão implementou um algoritmo de scheduling que seleciona a tarefa a ser executada em tempo constante, conhecido como O (1), independentemente do número de tarefas ou processadores no sistema. O novo scheduler também fornecia um suporte maior ao SMP, incluindo a afinidade com o processador e o balanceamento de carga. Mesmo melhorando a escalabilidade, essas alterações não melhoraram o desempenho ou a justiça em ambiente interativo. Na verdade, pioraram esses problemas para certas cargas de trabalho. Consequentemente, o scheduler de processos foi revisado uma segunda vez, na versão 2.6 do kernel do Linux. Essa versão introduziu o Completely Fair Scheduler (CFS).

O scheduler do Linux é um algoritmo preemptivo baseado em prioridades com dois intervalos de prioridades separados: um intervalo de tempo real de 0 a 99 e um valor de ajuste (nice value) variando de –20 a 19. Valores de ajuste menores indicam prioridades mais altas. Assim, aumentando o valor de ajuste, você estará diminuindo sua prioridade e sendo "bom" para o resto do sistema.

O CFS é um afastamento significativo do scheduler de processos tradicional is do Unix. Neste último, as principais variáveis do algoritmo de scheduling são a prioridade e a fatia de tempo, que é o período de tempo (a fatia do processador) que pode ser dado a um processo. Sistemas Unix tradicionais dão aos processos uma fatia de tempo fixa, podendo haver um aumento ou uma queda para processos de alta ou baixa prioridade, respectivamente. Um processo pode ser executado pelo período de sua fatia

de tempo, e processos de prioridade mais alta são executados antes de processos de prioridade mais baixa. É um algoritmo simples que muitos sistemas não Unix empregam. Essa simplicidade funcionava bem para os primeiros sistemas de tempo compartilhado, mas mostrou-se incapaz de fornecer bom desempenho e justiça em ambientes interativos dos desktops modernos e dispositivos móveis atuais.

O CFS introduziu um novo algoritmo de scheduling chamado scheduling justo, que aplica fatias de tempo no sentido tradicional. Em vez de delas, todos os processos recebem uma proporção do tempo do processador. Com isso, o CFS calcula por quanto tempo um processo deve ser executado em função do número total de processos executáveis.

Para começar, o CFS diz que, se houver *N* processos executáveis, cada um deve receber 1/N do tempo do processador. O CFS ajusta então essa divisão, avaliando a alocação atribuída a cada processo por seu valor de ajuste. Processos com o valor de ajuste default têm peso 1 e sua prioridade não é alterada. Processos com um valor de ajuste menor (prioridade mais alta) recebem um peso maior, enquanto aqueles com valor de ajuste maior (prioridade mais baixa) recebem um peso menor. Em seguida, o CFS executa cada processo durante uma fatia de tempo proporcional ao peso do processo dividido pelo peso total de todos os processos executáveis.

Para calcular o período de tempo real durante o qual um processo é executado, o CFS conta com uma variável configurável chamada latência-alvo, que é o intervalo de tempo durante o qual cada tarefa executável deve ser executada pelo menos uma vez. Por exemplo, suponha que a latência-alvo seja de 10 milissegundos; imagine também, que existam dois processos executáveis com a mesma prioridade. Os dois processos têm o mesmo peso e, portanto, recebem a mesma proporção de tempo do processador. Nesse caso, com uma latência-alvo de 10 milissegundos, o primeiro processo é executado por 5 milissegundos; então, o outro processo é executado por 5 milissegundos; depois, o primeiro processo é executado por 5 milissegundos novamente, e assim por diante. Se tivermos 10 processos executáveis, o CFS executará cada um deles por um milissegundo antes de repetir o procedimento.

E se tivermos, por exemplo, 1.000 processos? Cada processo seria executado por 1 microssegundo se seguirmos o procedimento que acaba de ser descrito. Em razão dos custos das mudanças de um processo para outro, o scheduling de processos por períodos de tempo tão curtos é ineficiente. Consequentemente, o CFS conta com uma segunda variável configurável, a granularidade mínima, que é um período de tempo mínimo durante o qual um processo recebe o processador. Todos os processos, independentemente da latência-alvo, serão executados durante pelo menos a granularidade mínima. Dessa forma, o CFS assegura que os custos das mudanças não fiquem inaceitavelmente altos quando o número de processos executáveis aumenta muito. Ao fazê-lo, ele viola suas tentativas de ser justo.

Normalmente, no entanto, o número de processos executáveis permanece razoável, e tanto a justiça quanto os custos das mudanças são maximizados.

Com a mudança para o scheduling justo, o CFS comporta-se diferentemente dos shedulers de processos tradicionais do Unix de vários modos. Como vimos, o mais evidente é que o CFS elimina o conceito de uma fatia de tempo estática.

Em vez disso, cada processo recebe uma proporção do tempo do processador, onde a duração dessa alocação vai depender de quantos processos mais são executáveis. Essa abordagem resolve vários problemas de mapeamento de prioridades para fatias de tempo inerentes a algoritmos de scheduling preemptivos baseados em prioridades. É claro que é possível resolver esses problemas de outras formas, sem ser preciso abandonar o scheduler clássico do UNIX. No entanto, o CFS resolve os problemas com um algoritmo simples que tem bom desempenho com cargas de trabalho interativas, como as dos dispositivos móveis, sem comprometer o desempenho do throughput, mesmo no maior dos servidores.

Revolucionando o design de algoritmos, o algoritmo O(1) permitiu que os programadores criassem algoritmos mais eficientes e escaláveis. Ao garantir que o tempo de execução permaneça constante, independentemente do tamanho da entrada, o algoritmo O(1) torna possível lidar com grandes conjuntos de dados sem afetar o desempenho. Isso é crucial para aplicações que exigem processamento rápido, como bancos de dados, sistemas de busca e aplicações de tempo real.

O algoritmo O(1) é frequentemente usado em operações de baixo nível, como acesso à memória, operações de disco e comunicação de rede. Essas operações são essenciais para o desempenho de qualquer sistema computacional e o algoritmo O(1) garante que elas sejam executadas de forma rápida e eficiente, independentemente do tamanho do conjunto de dados.

Uma das principais vantagens do algoritmo O(1) é sua capacidade de escalar com eficiência. À medida que o tamanho dos conjuntos de dados aumenta, o algoritmo O(1) continua a funcionar com a mesma velocidade, enquanto algoritmos com complexidade de tempo linear ou logarítmica se tornam mais lentos. Isso é crucial para o escalonamento de processos, pois permite que os sistemas lidem com grandes quantidades de dados e usuários simultâneos sem afetar o desempenho.

Embora o algoritmo O(1) seja extremamente útil, ele também possui algumas limitações. A complexidade de tempo constante se aplica apenas a operações específicas e nem todos os algoritmos podem ser implementados com essa complexidade. Além disso, o algoritmo O(1) pode exigir mais recursos, como memória ou espaço de armazenamento, para garantir o desempenho constante. Portanto, é importante analisar cuidadosamente as necessidades do sistema e o custo-benefício antes de implementar o algoritmo O(1).

O algoritmo O(1) é uma ferramenta poderosa para o design de algoritmos eficientes e escaláveis. Sua complexidade de tempo constante permite que os programadores criem algoritmos que possam lidar com grandes conjuntos de dados sem afetar o desempenho. As aplicações do algoritmo O(1) se estendem a diversas áreas, desde operações de baixo nível até o escalonamento de processos, tornando-o um conceito fundamental na ciência da computação. Ao entender as vantagens e limitações do algoritmo O(1), os desenvolvedores podem tomar decisões mais informadas sobre sua aplicação em seus projetos, garantindo o desempenho e a escalabilidade de seus sistemas.

Considerando, justamente, tais características, o algoritmo O(1) é considerado um dos algoritmos mais eficientes, com a complexidade de tempo mais baixa. Outros tipos de complexidade de tempo incluem O(n), O(log n) e O(n^2), conforme indica a figura a seguir.

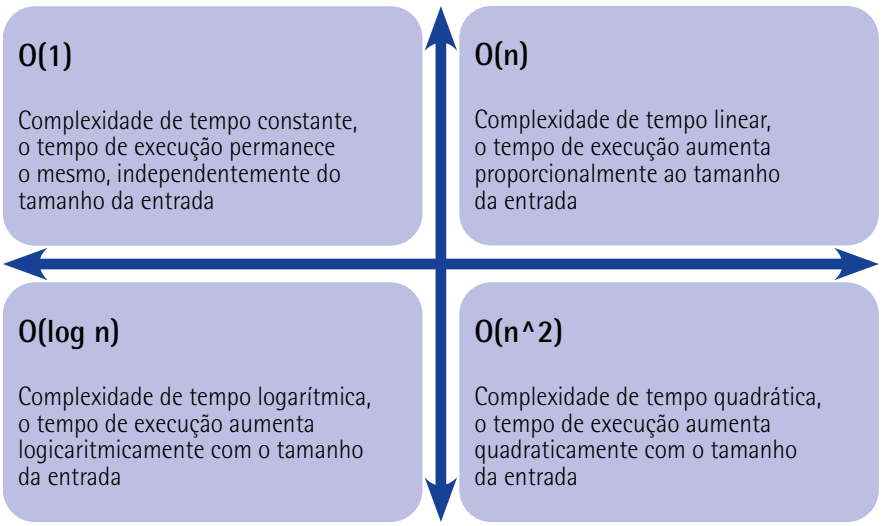

Figura 89 – Complexidades de tempo do algoritmo O(1)

Lembrete

Um algoritmo O(1) é um algoritmo que leva um tempo constante, independentemente do tamanho da entrada. Isso significa que o algoritmo sempre levará o mesmo tempo para concluir sua tarefa, não importando quão grande seja a entrada.

Scheduling de tempo real

O algoritmo de scheduling de tempo real do Linux é significativamente mais simples do que o scheduling justo empregado para processos de tempo compartilhado padrão. O Linux implementa as duas classes de scheduling de tempo real requeridas pelo POSIX.1b: primeiro-a-chegar/primeiro-a-ser-atendido (FCFS) e Round-Robin, respectivamente. Nos dois casos, cada processo tem uma prioridade além de sua classe de scheduling. O scheduler sempre executa o processo com a prioridade mais alta. Entre processos de prioridade igual, ele executa aquele que esteja esperando há mais tempo. A única diferença entre os schedulings FCFS e Round-Robin é que os processos no scheduling FCFS continuam sendo executados até que saiam ou sejam bloqueados, enquanto um processo no scheduling Round-Robin sofre preempção depois de algum tempo e é transferido para o fim da fila de scheduling. Assim, processos do scheduling Round-Robin com prioridades iguais compartilham o tempo entre eles automaticamente.

O scheduling de tempo real do Linux não oferece garantias rígidas de tempo real em um ambiente multitarefa preemptivo. O scheduler oferece garantias rigorosas em relação às prioridades relativas dos processos de tempo real, mas o kernel não oferece nenhuma garantia relativa à rapidez com que esse tipo de processo será incluído no schedule, uma vez que o processo se torne executável. Por outro lado, um sistema de tempo compartilhado crítico pode garantir uma latência mínima entre o momento em que um processo se torna executável e o momento em que ele é realmente executado.

Sincronização do kernel

A maneira como o kernel organiza o schedule de suas próprias operações é fundamentalmente diferente da maneira como ele organiza o schedule de processos. Uma solicitação de execução em modalidade de kernel pode ocorrer de duas formas: um programa em execução pode solicitar um serviço do sistema operacional explicitamente por meio de uma chamada de sistema ou implicitamente, por exemplo, quando ocorre um erro de página; alternativamente, um controlador de dispositivos pode distribuir uma interrupção de hardware que faça a CPU começar a executar um manipulador definido pelo kernel para essa interrupção.

O problema que se apresenta para o kernel é que todas essas tarefas podem tentar acessar as mesmas estruturas de dados internos. Se uma tarefa do kernel estiver no meio do acesso a alguma estrutura de dados quando uma rotina de serviço de interrupção for executada, então, essa rotina de serviço não poderá acessar ou modificar os mesmos dados sem correr o risco de corrompê-los. Esse fato está relacionado com a ideia das seções críticas, as partes do código que acessam dados compartilhados e que, portanto, não devem ter permissão para serem executadas concorrentemente. Como resultado, a sincronização do kernel envolve muito mais do que apenas o scheduling de processos. É necessária uma estrutura que permita que tarefas do kernel sejam executadas sem violar a integridade dos dados compartilhados.

Antes da versão 2.6, o Linux era um kernel não preemptivo, significando que um processo em execução em modalidade de kernel não podia sofrer preempção, mesmo se um processo de prioridade mais alta ficasse disponível para execução. Na versão 2.6, o kernel do Linux tornou-se totalmente preemptivo. Agora, uma tarefa pode sofrer preempção quando está sendo executada no kernel.

Vale ressaltarmos que o kernel Linux 5.k representa um grande avanço em relação às versões anteriores, incorporando uma série de recursos e melhorias significativas. Uma das novidades mais notáveis é a introdução do suporte a novos sistemas de arquivos, como o ZFS, que oferece recursos avançados de replicação, instantâneos e snapshots, tornando o gerenciamento de dados mais eficiente e seguro. Além disso, o kernel 5.k inclui suporte aprimorado a tecnologias de virtualização, como o KVM, permitindo uma melhor performance e estabilidade em ambientes virtualizados.

Outra melhoria importante é a implementação de um novo sistema de gerenciamento de energia, que otimiza seu consumo, prolongando a autonomia de dispositivos móveis e reduzindo o consumo de energia em servidores.

O kernel 5.k também traz avanços em termos de segurança, com a inclusão de novas medidas de proteção contra ataques e exploits, tornando o sistema operacional mais resiliente e seguro. Essas melhorias abrangem desde a proteção de dados e a detecção de intrusões até a mitigação de vulnerabilidades conhecidas, garantindo um ambiente mais seguro para os usuários.

Suporte aprimorado a hardware

O kernel Linux 5.k oferece suporte aprimorado a uma ampla gama de hardwares, incluindo processadores modernos, como os da linha AMD Ryzen e Intel Core, e placas gráficas de última geração e dispositivos de armazenamento mais rápidos, como os NVMe. Essa compatibilidade garante que o kernel Linux seja capaz de aproveitar ao máximo o poder de processamento e os recursos de hardware disponíveis, oferecendo melhor desempenho e eficiência.

Além disso, o kernel 5.k introduziu suporte a novos dispositivos e interfaces, como o Wi-Fi 6, Bluetooth 5.2 e USB4, garantindo que o sistema operacional esteja pronto para as tecnologias mais recentes e que os usuários possam aproveitar ao máximo os benefícios de seus dispositivos. Outro ponto importante é a melhoria no suporte a drivers de dispositivos, o que significa que os usuários podem ter certeza de que seus periféricos, como impressoras, scanners e webcams, funcionarão de forma confiável com o kernel Linux 5.k. O suporte aprimorado a hardware é essencial para garantir que o kernel Linux possa ser usado em uma ampla variedade de dispositivos e que os usuários possam ter acesso a todos os recursos de hardware disponíveis.

Avanços em segurança e estabilidade

A segurança e a estabilidade são aspectos cruciais para qualquer sistema operacional, e o kernel Linux 5.k dá um passo importante nessa direção. O kernel inclui um conjunto de melhorias que visam aumentar a segurança do sistema, incluindo novas medidas de proteção contra ataques de ransomware e exploits de kernel, além de um sistema de detecção de intrusões aprimorado.

As melhorias na estabilidade do sistema, com correções para diversos bugs e falhas que podem afetar o desempenho e a confiabilidade do sistema, foi outro avanço promovido pelo kernel 5.k, garantindo que ele seja mais resistente a falhas e que os usuários possam ter uma experiência mais suave e confiável.

O kernel Linux 5.k também inclui um novo sistema de gerenciamento de memória, que visa melhorar a segurança e a estabilidade do sistema, reduzindo o risco de ataques de memória ao passo em que a otimiza, garantindo um desempenho mais eficiente e confiável.

Os avanços em segurança e estabilidade são essenciais para garantir que o kernel Linux seja um sistema operacional confiável e seguro, apto a atender às necessidades dos usuários em um ambiente cada vez mais complexo e desafiador.

Aperfeiçoamentos no sistema de arquivos

O kernel Linux 5.k apresenta uma série de aperfeiçoamentos no sistema de arquivos, visando aumentar a performance, a segurança e a confiabilidade. Uma das melhorias mais significativas é a implementação do suporte a novos sistemas de arquivos, como o ZFS, que oferece recursos avançados de replicação, instantâneos e snapshots, tornando o gerenciamento de dados mais eficiente e seguro.

Outro aperfeiçoamento foi a inclusão de otimizações no sistema de arquivos ext4, o sistema de arquivos padrão do Linux, visando melhorar o desempenho de leitura e escrita, além de reduzir a fragmentação de arquivos e aumentar a capacidade de lidar com grandes quantidades de dados.

Além dessas melhorias, o kernel 5.k oferece suporte a recursos avançados de criptografia de arquivos, garantindo a confidencialidade e a integridade dos dados armazenados no sistema. Os aperfeiçoamentos no sistema de arquivos garantem que o kernel Linux possa lidar de forma eficiente com os desafios de gerenciamento de dados em um mundo cada vez mais digital, com grandes quantidades de informação sendo geradas e armazenadas.

Otimizações de desempenho e eficiência

O kernel Linux 5.k inclui diversas otimizações de desempenho e eficiência, visando aumentar a velocidade e a eficiência do sistema em diversas áreas. Uma das otimizações mais importantes é a melhoria no gerenciamento de memória, que otimiza o uso da memória RAM e reduz a fragmentação de memória, resultando em um desempenho mais rápido e eficiente.

As otimizações feitas no escalonador de processos, que gerencia a execução de tarefas no sistema, também garantem que as tarefas sejam executadas de forma eficiente e que os recursos do sistema sejam utilizados da melhor maneira possível.

Além disso, o kernel 5.k inclui otimizações de desempenho em diversas áreas, como o gerenciamento de rede, o suporte a dispositivos de armazenamento e a execução de processos em sistemas multicore, garantindo um desempenho geral mais rápido e eficiente. Tais otimizações de desempenho e eficiência asseguram que o kernel Linux seja capaz de lidar com as demandas de um ambiente cada vez mais exigente, com aplicativos cada vez mais complexos e consumidores de recursos.

O kernel Linux 5.k inclui diversas atualizações em drivers e módulos, garantindo que o sistema seja compatível com o hardware mais recente e que os usuários possam aproveitar ao máximo os recursos de seus dispositivos. Uma das atualizações mais importantes é a inclusão de novos drivers para placas gráficas de última geração, como as da linha NVIDIA GeForce RTX e AMD Radeon RX, garantindo melhor desempenho e compatibilidade com os jogos e aplicativos mais recentes.

Ao incluir atualizações em drivers para dispositivos de rede, como placas de rede Wi-Fi e Bluetooth, o kernel 5.k garante melhor conectividade e performance, além de suporte a novas tecnologias, como Wi-Fi 6 e Bluetooth 5.2. Junto a isso, atualizações em drivers para dispositivos de armazenamento, garantem, também, melhor performance e compatibilidade com os dispositivos mais rápidos, como os

NVMe. Essas atualizações em drivers e módulos garantem que o kernel Linux esteja sempre atualizado e compatível com os últimos avanços em hardware, oferecendo uma experiência mais suave e confiável para os usuários.

O kernel Linux 5.k representa um grande passo no desenvolvimento do sistema operacional Linux, com uma série de recursos e melhorias que visam aperfeiçoar a segurança, a estabilidade, o desempenho e a compatibilidade do sistema. As novas funcionalidades, como o suporte a novos sistemas de arquivos, as otimizações de desempenho e as atualizações em drivers garantem que o kernel Linux continue a ser uma plataforma robusta e confiável para os usuários.

O kernel 5.k fornece spinlocks e semáforos (assim como versões de leitor-gravador desses dois locks) para trancamento no kernel. Em máquinas SMP, o mecanismo básico de trancamento é um spinlock, e o kernel é projetado de modo que eles sejam mantidos apenas por períodos curtos. Em máquinas com um único processador, os spinlocks não são apropriados para uso e são substituídos pela habilitação e desabilitação da preempção do kernel. Isto é, em vez de manter um spinlock, a tarefa desabilita a preempção do kernel. No momento em que a tarefa liberaria o spinlock, ela habilita a preempção do kernel. Esse padrão é resumido no quadro abaixo:

Quadro 6 – Padrão de habilitação e desabilitação da preempção do kernel

Um processador	Múltiplos processadores
Desabilita a preempção do kernel	Adquirem spinlock
Habilita a preempção do kernel	Liberam spinlock

O Linux usa uma abordagem interessante para desabilitar e habilitar a preempção do kernel. Ele fornece duas interfaces simples: preempt_disable() e preempt_enable(). Além disso, o kernel não pode sofrer preempção se uma tarefa em modalidade de kernel estiver mantendo um spinlock. Para que essa regra seja imposta, cada tarefa no sistema tem uma estrutura thread-info que inclui o campo preempt_count (um contador indicando o número de locks sendo mantidos pela tarefa). O contador é incrementado quando um lock é adquirido e decrementado quando um lock é liberado. Se o valor de preempt_count para a tarefa em execução corrente for maior do que zero, não será seguro causar a preempção do kernel, já que essa tarefa mantém correntemente um lock. Se a contagem for igual a zero, o kernel poderá ser interrompido com segurança, supondo que não haja chamadas pendentes a preempt_disable().

Os spinlocks e a habilitação e desabilitação da preempção do kernel são usados no kernel somente quando o lock é mantido por curtos períodos. Quando um lock deve ser mantido por períodos mais longos, são usados os semáforos.

A segunda técnica de proteção usada pelo Linux aplica-se às seções críticas que ocorrem em rotinas de serviço de interrupção. A ferramenta básica é o hardware de controle de interrupções do processador. Desabilitando interrupções (ou usando spinlocks) durante uma seção crítica, o kernel garante que poderá prosseguir sem o risco de acesso concorrente às estruturas de dados compartilhadas.

No entanto, há desvantagens na desabilitação de interrupções. Na maioria das arquiteturas de hardware, instruções de habilitação e desabilitação de interrupções não são baratas. O mais importante é que, enquanto as interrupções permanecerem desabilitadas, todo o I/O será suspenso e qualquer dispositivo em espera por atendimento terá que esperar até que as interrupções sejam reabilitadas; assim, o desempenho é degradado. Para resolver esse problema, o kernel do Linux usa uma arquitetura de sincronização que permite que seções críticas longas sejam executadas por toda a sua duração sem que as interrupções sejam desabilitadas. Esse recurso é particularmente útil para o código de conexão de rede. Uma interrupção em um driver de dispositivo de rede pode sinalizar a chegada de um pacote de rede inteiro, o que pode resultar em um grande volume de código sendo executado para desmontagem, roteamento e encaminhamento desse pacote dentro da rotina de serviço de interrupção.

O Linux implementa essa arquitetura separando as rotinas de serviço de interrupção em duas seções: a metade do topo e a metade da base. Na metade do topo, está uma rotina de serviço de interrupção padrão, a qual é executada com as interrupções recursivas desabilitadas. As interrupções de mesmo número (ou nível) são desabilitadas, mas outras interrupções podem ser executadas. Uma rotina de serviço da metade da base é executada, com todas as interrupções habilitadas, por um scheduler miniatura que assegura que as rotinas da metade da base nunca interrompam a si mesmas.

O scheduler da metade da base é invocado automaticamente sempre que uma rotina de serviço de interrupção é encerrada. Essa separação significa que o kernel pode concluir qualquer processamento complexo que tenha que ser executado em resposta a uma interrupção, sem preocupações quanto a ser interrompido. Se outra interrupção ocorrer enquanto uma rotina da metade da base estiver em execução, então ela poderá solicitar que a mesma rotina seja executada, mas isso será adiado até que a rotina em execução corrente seja concluída. Cada execução de uma rotina da metade da base pode ser interrompida por uma rotina da metade do topo, mas nunca por uma rotina semelhante da metade da base.

A arquitetura da metade do topo/metade da base é complementada por um mecanismo para desabilitação de rotinas da metade da base selecionadas enquanto o código de foreground normal do kernel está em execução. O kernel pode codificar as seções críticas facilmente usando esse sistema. Os manipuladores de interrupções podem codificar suas seções críticas como metades da base; e, quando o kernel de foreground quiser entrar em uma seção crítica, ele poderá desabilitar qualquer rotina relevante da metade da base para impedir que alguma outra seção crítica o interrompa. No fim da seção crítica, o kernel pode reabilitar as rotinas da metade da base e executar qualquer tarefa da metade da base que tenha sido enfileirada por rotinas de serviço de interrupção da metade do topo durante a seção crítica.

A figura 90 resume os diversos níveis de proteção de interrupções dentro do kernel. Cada um deles pode ser interrompido por código em execução em um nível mais alto, mas nunca será interrompido por código em execução no mesmo nível ou em um nível inferior, exceto para códigos de modalidade de usuário. Os processos de usuário sempre poderão sofrer preempção por outro processo quando ocorrer uma interrupção de scheduling de compartilhamento de tempo.

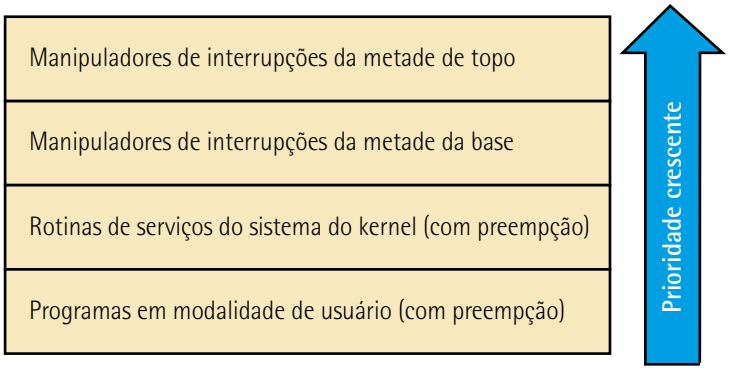

Figura 90 – Níveis de proteção de interrupções

Fonte: Silberschartz (2015, p. 623).

Observação

A rotina do sistema operacional tem como principal função implementar os critérios da política de scheduler. Em um sistema multiprogramável, o escalonador é fundamental, pois todo o compartilhamento do processador depende desta rotina.

Multiprocessamento simétrico

O kernel 2.0 do Linux foi o primeiro kernel estável do Linux a dar suporte ao hardware multiprocessador simétrico (SMP), permitindo que processos separados fossem executados em paralelo e em processadores separados. A implementação original do SMP impunha a restrição de que apenas um processador de cada vez poderia executar o código do kernel.

Na versão 2.2 do kernel, um único spinlock, big kernel lock (BKL), foi criado para permitir que múltiplos processos (sendo executados em diferentes processadores) estivessem ativos no kernel concorrentemente. No entanto, o BKL fornecia um nível muito baixo de granularidade de trancamento, resultando em escalabilidade pobre para máquinas com muitos processadores e processos. Versões posteriores do kernel tornaram a implementação do SMP mais escalável, dividindo esse spinlock único em múltiplos locks, onde cada um protege apenas um pequeno subconjunto das estruturas de dados do kernel. O kernel 3.0 fornece melhorias adicionais ao SMP, incluindo trancamento mais refinado, afinidade de processadores e algoritmos de balanceamento de carga.

O BKL foi introduzido no Linux como uma solução simples para proteger o kernel de acesso concorrente. Ele era uma maneira eficaz de lidar com a concorrência em sistemas com um único processador. No entanto, conforme o Linux se tornou mais popular e os sistemas multiprocessadores se tornaram mais comuns, as limitações do BKL se tornaram cada vez mais evidentes.

O BKL tornou-se um gargalo de desempenho, pois impedia que os núcleos acessassem o kernel simultaneamente. A demanda por uma abordagem de sincronização mais eficiente levou à introdução de mecanismos de sincronização mais sofisticados, como spinlocks e semáforos. Ainda, o BKL apresenta vários problemas que o tornam inadequado para sistemas modernos. Alguns dos problemas mais significativos incluem:

- **Gargalo de desempenho**: o BKL atua como um único ponto de acesso para o kernel, criando um gargalo que limita o desempenho em sistemas multiprocessadores.
- **Escalabilidade limitada**: conforme o número de núcleos aumenta, o desempenho do BKL diminui, pois os núcleos ficam presos na fila esperando o acesso ao BKL.
- **Complexidade do código**: o BKL adiciona complexidade ao código do kernel, dificultando a manutenção e o desenvolvimento de novos recursos.

Devido a tais problemas e defeitos em seu funcionamento, o BKL teve um impacto negativo no desempenho do Linux em sistemas multiprocessadores. O gargalo criado pelo BKL restringia a capacidade do sistema de utilizar todos os seus núcleos de forma eficiente.

Em sistemas com vários núcleos, o BKL impede que eles trabalhem em paralelo, resultando em uma redução significativa do desempenho geral do sistema. Isso é especialmente perceptível em tarefas intensivas de CPU, onde a concorrência pelo BKL torna-se um fator limitante.

Frente a essas dificuldades, várias abordagens foram desenvolvidas para superar as limitações do BKL e substituí-lo. Algumas das técnicas mais proeminentes incluem:

- **Spinlocks**: são travamentos de baixo nível que permitem que os núcleos girem em um loop até que a trava seja liberada. Eles são mais eficientes que o BKL em situações onde a espera é curta.
- **Semáforos**: são uma estrutura de sincronização que permite que os núcleos se comuniquem entre si e coordenem o acesso a recursos compartilhados.
- **RCU (Read-Copy-Update)**: é uma técnica de sincronização que permite que os dados sejam modificados de forma segura, mesmo que outros núcleos estejam lendo os dados.

Com o tempo, o BKL foi gradualmente eliminado do kernel Linux. A adoção de mecanismos de sincronização mais sofisticados, como spinlocks, semáforos e RCU, tornou o Linux mais escalável e eficiente.

A transição para técnicas de sincronização modernas permitiu que o Linux tirasse proveito da capacidade de processamento paralela de sistemas multiprocessadores. Assim, o kernel foi otimizado para lidar com a concorrência de forma eficiente, melhorando o desempenho do sistema como um todo.

Gerenciamento de memória

O gerenciamento da memória no Linux tem dois componentes: o primeiro lida com a alocação e a liberação de memória física, como páginas, grupos de páginas e pequenos blocos de RAM; o segundo manipula a memória virtual, que é a memória mapeada para o espaço de endereçamento de processos em execução.

Gerenciamento da memória física

Em razão de restrições específicas de hardware, o Linux separa a memória física em quatro zonas, ou regiões, diferentes: ZONE_DMA; ZONE_DMA32; ZONE_NORMAL; e ZONE_HIGHMEM.

Essas zonas são específicas da arquitetura. Por exemplo, na arquitetura Intel x86-32, certos dispositivos ISA (industry standard architecture) podem acessar somente os 16 MB inferiores de memória física usando o DMA. Nesses sistemas, os primeiros 16 MB de memória física compõem a ZONE_DMA. Em outros sistemas, certos dispositivos podem acessar apenas os primeiros 4 GB de memória física, apesar de suportarem endereços de 64 bits. Em tais sistemas, os primeiros 4 GB de memória física compõem a ZONE_DMA32.

A ZONE_HIGHMEM (high memory) refere-se à memória física que não é mapeada para o espaço de endereçamento do kernel. Por exemplo, na arquitetura Intel de 32 bits (em que 232 fornecem um espaço de endereçamento de 4 GB), o kernel é mapeado para os primeiros 896 MB do espaço de endereçamento. A memória restante é chamada memória alta e é alocada a partir da ZONE_HIGHMEM.

Por fim, a ZONE_NORMAL compreende todo o resto — as páginas normais mapeadas regularmente. As restrições de uma arquitetura é que definem se ela tem determinada zona. Uma arquitetura moderna de 64 bits, tal como a do Intel x86-64, tem uma pequena ZONE_DMA de 16 MB (para dispositivos legados) e todo o resto de sua memória fica na ZONE_NORMAL, sem "memória alta".

O relacionamento entre zonas e endereços físicos na arquitetura do Intel x86-32 faz com que o kernel mantenha uma lista de páginas livres para cada zona. Quando chega uma solicitação de memória física, o kernel atende à solicitação usando a zona apropriada.

O principal gerenciador de memória física no kernel do Linux é o alocador de páginas. Cada zona tem seu próprio alocador, que é responsável por alocar e liberar todas as páginas físicas para a zona, podendo alocar intervalos de páginas fisicamente contíguas e sob demanda. O alocador usa um sistema de pares (buddy system) para rastrear páginas físicas disponíveis. Nesse esquema, unidades adjacentes de memória alocável são reunidas em pares, daí seu nome. Cada região de memória alocável tem um parceiro adjacente (ou buddy). Sempre que duas regiões parceiras alocadas são liberadas, elas são combinadas para formar uma região maior, um heap de pares (buddy heap). Essa região maior também tem um parceiro com o qual ela pode se unir para formar uma região livre ainda maior.

Inversamente, se uma solicitação de pouca memória não puder ser atendida pela alocação de uma pequena região livre existente, então uma região livre maior será subdividida em dois parceiros para atender à solicitação. Listas encadeadas separadas são usadas para registrar as regiões de memória livres de cada tamanho permitido. No Linux, o menor tamanho alocável por meio desse mecanismo é uma única página física, onde uma região de 4 kB está sendo alocada, mas a menor região disponível tem 16 kB. A região é dividida recursivamente até que um bloco do tamanho desejado esteja livre.

Por fim, todas as alocações de memória no kernel do Linux são feitas, estaticamente, por drivers que reservam uma área contígua de memória durante o tempo de inicialização do sistema ou, dinamicamente, pelo alocador de páginas. No entanto, as funções do kernel não precisam usar o alocador básico para reservar memória. Vários subsistemas especializados de gerenciamento da memória usam o alocador de páginas subjacente para gerenciar seus próprios pools de memória. Os mais importantes são o sistema de memória virtual; o alocador de tamanho variável kmalloc(); o de slabs, usado na alocação de memória para estruturas de dados do kernel; e o cache de páginas, usado para armazenar em cache páginas pertencentes a arquivos.

Saiba mais

Para saber mais sobre BKL, leia o capítulo 18 do livro a seguir:

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Fundamentos de sistemas operacionais*. 9. ed. Rio de Janeiro: LTC, 2015.

6.3 Escalonamento com múltiplos processadores

Sistemas com múltiplos processadores são arquiteturas que possuem duas ou mais UCPs interligadas e que funcionam em conjunto na execução de tarefas independentes ou no processamento simultâneo de uma mesma tarefa. Inicialmente, os computadores eram vistos como máquinas sequenciais, em que o processador executava as instruções de um programa uma de cada vez. Com a implementação de sistemas com múltiplos processadores, o conceito de paralelismo pôde ser expandido a um nível mais amplo.

A evolução desses sistemas se deve, em grande parte, ao elevado custo de desenvolvimento de processadores mais rápidos. Em função disso, passou-se a dar ênfase a computadores com múltiplos processadores, em vez de arquiteturas com um único processador de alto desempenho. Outra motivação são aplicações que necessitam de grande poder computacional, como sistemas de previsão do tempo, dinâmica dos fluidos, genoma humano, modelagens e simulações. Com múltiplos processadores, é possível reduzir drasticamente o tempo de processamento destas aplicações. Inicialmente, as configurações limitavam-se a poucos processadores, mas atualmente existem sistemas com milhares deles.

Saiba mais

Para conhecer um pouco mais sobre Escalonamento First-In-First-Out (FIFO), leia o capítulo 8 do livro a seguir:

MACHADO, F. B.; MAIA, L. P. *Arquitetura de sistemas operacionais*. 5. ed. Rio de Janeiro: LTC, 2013.

Resumo

A unidade III abordou os conceitos essenciais de processos e as técnicas de gerenciamento que garantem a eficiência e a estabilidade dos sistemas. Estudamos a conceituação de processos, o problema de concorrência, as ferramentas de sincronização, a importância de evitar conflitos em sistemas multitarefa e, finalmente, a importância da eficiência e estabilidade dos sistemas. Além disso, tivemos a oportunidade de adentrar os conceitos de monitores e semáforos, que são utilizados para garantir a sincronização entre processos concorrentes, explorando como essas são ferramentas essenciais para nossa área de estudo.

A multitarefa foi outro assunto abordado nessa unidade, essa que é uma técnica fundamental em sistemas operacionais modernos, permitindo que múltiplos processos compartilhem os recursos do sistema e sejam executados concorrentemente. No entanto, a multitarefa aumenta a complexidade do gerenciamento de processos e torna a concorrência ainda mais crítica. Vimos, também, as técnicas e estratégias essenciais para evitar conflitos em sistemas multitarefa, utilizando monitores, semáforos e outras técnicas de sincronização, ao passo em que analisamos como essas ferramentas garantem a integridade dos dados, a consistência das operações e a ordem correta de execução de tarefas.

Os conceitos de processos e as técnicas de gerenciamento que aprendemos são fundamentais para garantir a eficiência e a estabilidade dos sistemas. Um sistema eficiente utiliza os recursos de hardware e software de forma otimizada, minimizando o tempo de resposta e maximizando o desempenho. Um sistema estável é robusto e confiável, livre de erros e falhas, garantindo que as operações sejam executadas de forma previsível e segura. As técnicas de gerenciamento de processos, incluindo a sincronização, a comunicação interprocessos e o gerenciamento de recursos compartilhados, são essenciais para construir sistemas que atendam a esses requisitos de eficiência e estabilidade.

Abordamos, ainda, temas como a importância da alocação eficiente dos recursos de processamento, com ênfase no escalonamento (seus critérios e tipos), além de técnicas específicas para múltiplos processadores.

Exercícios

Questão 1. As condições de corrida (race conditions) e a necessidade de sincronizar processos ou threads são temas centrais nos sistemas operacionais modernos. Quando múltiplas unidades de execução disputam acesso a dados ou a recursos compartilhados, torna-se imprescindível garantir exclusão mútua e evitar erros sutis, como a leitura e a escrita simultâneas em uma variável ou em uma estrutura de dados compartilhada. Para isso, surgiram mecanismos como semáforos, mutexes e monitores, cada um com características de implementação próprias e aplicações específicas. Contudo, a simples adoção de um mecanismo de sincronização não é suficiente para evitar problemas como deadlocks, starvation e mesmo possíveis degradações de desempenho. Em certos casos, o uso inadequado de exclusão mútua pode criar gargalos que impactam a eficiência global do sistema. Assim, é preciso conhecer não apenas os conceitos de monitores e semáforos, mas também as boas práticas de programação concorrente, a identificação correta de regiões críticas e a forma como o sistema operacional oferece serviços de criação, de suspensão e de retomada de processos.

Com base nessas considerações, assinale a alternativa que melhor descreve de que forma essas ferramentas (monitores e semáforos) contribuem para lidar com as condições de corrida, levando em conta tanto a robustez da exclusão mútua quanto a complexidade de implementação e uso.

A) Monitores encapsulam dados e procedimentos, promovem modularidade e garantem exclusão mútua de forma automática, já que o sistema operacional não permite que dois processos acessem o monitor simultaneamente. Contudo, essa solução não oferece suporte a variáveis de condição, e não há como suspender temporariamente a execução de um processo dentro do monitor para aguardar um evento. Com isso, é impossível lidar com cenários em que processos precisam esperar por atualizações e retomar do ponto exato onde pararam. Desse modo, mesmo que monitores ajudem na exclusão mútua, eles não se mostram úteis em coordenações mais complexas que envolvam sinalização.

B) Semáforos permitem controlar o acesso a recursos compartilhados por meio de contadores e de operações atômicas de decremento (wait) e de incremento (signal). Esse mecanismo suporta tanto a exclusão mútua (quando o semáforo é binário) quanto a coordenação entre múltiplos processos (quando o semáforo é um contador geral). Entretanto, o uso de semáforos é inevitavelmente limitado a situações de kernel monolítico e não pode ser aproveitado em outros tipos de arquitetura de sistema operacional. Além disso, semáforos exigem sempre spinlocks, forçando processos a ficarem em busy-wait até receberem acesso, o que impede qualquer forma de bloqueio inteligente ou fila de espera.

C) Monitores fornecem uma abstração de alto nível, em que variáveis de condição permitem que processos esperem por eventos específicos dentro do monitor enquanto a exclusão mútua é oferecida de modo automático pelo compilador ou pelo próprio sistema. Já semáforos, sejam binários ou de contador, são uma ferramenta de mais baixo nível, mas flexível, capaz de gerenciar tanto a exclusão mútua quanto os múltiplos recursos simultâneos. Em ambos os casos, a ideia central é evitar que duas unidades de execução modifiquem dados compartilhados de forma não sincronizada, o que reduz os riscos de corrupção de dados e de obtenção de resultados inconsistentes. Ao mesmo tempo, esses mecanismos exigem disciplina do programador para evitar problemas como deadlocks e starvation.

D) Para eliminar completamente as condições de corrida, basta alocar uma única CPU para cada processo. Dessa forma, não há risco de execução paralela real, pois cada processador executaria apenas um processo por vez e não haveria nenhum compartilhamento de recursos. Nessa abordagem, monitores e semáforos tornam-se dispensáveis, já que a concorrência se torna inteiramente uma ilusão de multiprogramação. Além disso, tal estratégia também melhora a eficiência, pois se elimina qualquer overhead de sincronização, e cada processo ficaria isolado em seu próprio processador, sem depender de filas ou de escalonadores do sistema operacional.

E) Uma das maneiras de lidar com condições de corrida é transformar todas as variáveis globais em variáveis locais de cada processo, o que elimina a necessidade de exclusão mútua. Nesse modelo, os processos não compartilham memória e se comunicam apenas via mensagens (comunicação assíncrona). Assim, semáforos e monitores podem ser substituídos por filas de mensagens não bloqueantes, o que resolve completamente a concorrência. Essa abordagem, entretanto, requer necessariamente sistemas operacionais distribuídos, pois, em ambientes de um único nó, não é possível organizar processos sem memória global ou disco compartilhado.

Resposta correta: alternativa C.

Análise da questão

A alternativa C expõe a função dos monitores, que encapsulam dados e oferecem variáveis de condição para esperar/sinalizar eventos, e dos semáforos, que controlam acessos concorrentes a recursos por meio de contadores ou de flags binários. Ela mostra, ainda, que ambos exigem cuidado do programador para evitar problemas clássicos de concorrência, como deadlocks e starvation. Essa descrição captura a essência de como monitores e semáforos agem contra condições de corrida e ressalta a importância de disciplina no uso dessas ferramentas.

Questão 2. Considere o modelo de criação de processos em sistemas Unix e Linux, que se baseia principalmente nas chamadas de sistema fork() e exec(). No contexto do Linux, sabe-se que o processo é definido por um conjunto de subcontextos independentes (como tabelas de arquivos, manipuladores de sinais, espaço de endereçamento etc.) e que a chamada clone() permite controlar quais desses subcontextos serão compartilhados entre o processo-pai e o processo-filho. Além disso, cada processo apresenta uma identidade (PID, credenciais e grupos) e um ambiente (vetores de argumentos e variáveis de ambiente) que podem ser manipulados, em certos limites, pelo próprio processo. Assim, um novo processo pode surgir como uma simples cópia exata do pai ou pode carregar outro programa executável em seu espaço de endereçamento, substituindo completamente o código antigo. Suponha que um desenvolvedor deseje criar um processo que compartilhe manipuladores de sinais e descritores de arquivos, mas que não compartilhe o espaço de memória do pai. A partir desta situação, analise as alternativas a seguir, considerando como as flags de clone() e a lógica de fork()/exec() interagem para definir o comportamento do processo resultante, e assinale a correta.

A) Esse desenvolvedor poderia simplesmente usar fork(), já que fork() gera um processo completamente independente em todos os subcontextos (incluindo tabelas de arquivos e manipuladores de sinais). Para que haja compartilhamento de descritores de arquivos, seria suficiente que, após o fork(), o processo-filho realizasse chamadas de sistema adicionais que copiassem as estruturas de arquivo do pai, a fim de obter equivalência completa em relação a esses descritores. Além disso, o compartilhamento de manipuladores de sinais não é possível via clone(), pois essa chamada de sistema foi projetada apenas para manipular a questão do espaço de memória e dos diretórios de trabalho. Dessa forma, não haveria necessidade de recorrer à chamada exec(), que apenas substituiria o binário, pois a funcionalidade pretendida já estaria atendida exclusivamente pela combinação de fork() com cópias explícitas de cada descritor.

B) A maneira de proceder seria criar um processo por clone() utilizando as flags CLONE_SIGHAND e CLONE_FILES, a fim de garantir que o processo-filho compartilhe os manipuladores de sinais e a tabela de arquivos abertos. Ao mesmo tempo, seria preciso omitir a flag CLONE_VM, para que o espaço de endereçamento não fosse compartilhado. Dessa forma, pai e filho compartilham sinalizadores e descritores de arquivos, mas cada um tem memória independente. Posteriormente, se desejado, o processo-filho ainda poderá chamar exec() para carregar um novo binário, pois manterá o compartilhamento definido pelas flags de clone() até que ele mesmo altere alguma configuração ou saia.

C) O modelo de processos do Linux não admite que dois processos compartilhem descritores de arquivos sem compartilhar também o espaço de memória, porque a tabela de arquivos abertos está necessariamente contida no mesmo subcontexto de memória virtual. Logo, se o desenvolvedor quiser que pai e filho acessem as mesmas conexões de arquivo, forçosamente terá de usar CLONE_VM, o que faz com que ambos executem as mesmas instruções na mesma memória. Em consequência, qualquer tentativa de manter duas instâncias de código diferentes — por exemplo, chamando exec() — implicaria encerrar de vez o processo anterior, pois não há como duas tarefas rodarem binários distintos em um subcontexto único de memória compartilhada.

D) A relação entre fork() e clone() é tal que não há diferença efetiva de comportamento entre ambas, pois cada uma cria um processo completamente independente em todos os aspectos. A única dissimilaridade prática é que, para rodar um programa diferente do pai, qualquer um dos processos gerados por essas chamadas deve obrigatoriamente reiniciar a sua lista de descritores de arquivos, a fim de não manter nada do ambiente anterior. Assim, o compartilhamento de manipuladores de sinal ou de tabelas de arquivo não é algo configurável e, portanto, depende exclusivamente dos mecanismos internos do kernel que definem a herança padrão.

E) Uma alternativa plausível seria usar fork() para criar um processo-filho idêntico, mas não compartilhar o espaço de memória. Em seguida, o processo-filho realizaria exec() para executar outro binário, o que automaticamente sincronizaria as suas credenciais e seus IDs de grupo com o processo-pai. Contudo, manipular descritores de arquivo e manipuladores de sinais exige que se configure explicitamente uma zona de namespace diferenciada, pois, em condição padrão, qualquer mudança de estado de arquivo em um processo não se reflete no outro e vice-versa. Assim, o compartilhamento de sinais e de arquivos demandaria sobrescrever a estrutura de PID do pai, algo que não é suportado na API nativa do Linux.

Resposta correta: alternativa B.

Análise da questão

A única combinação plausível para compartilhar manipuladores de sinais (CLONE_SIGHAND) e de descritores de arquivo (CLONE_FILES), mas não compartilhar o espaço de endereçamento (omitir CLONE_VM), é justamente o uso de clone() configurado dessa forma. O fork() padrão não oferece a granularidade de escolha de subcontextos, e a chamada exec() apenas substitui o código em execução, sem interferir diretamente no compartilhamento prévio estabelecido. Por isso, a alternativa B descreve com exatidão o procedimento para obter esse resultado.