Sincronização e Comunicação entre Processos (IPC)

Introdução

Em sistemas operacionais que suportam concorrência, seja através de multiprogramação com múltiplos processos ou multithreading dentro de um único processo, a necessidade de coordenar as atividades dessas entidades concorrentes é crucial. Processos e threads frequentemente precisam cooperar para realizar uma tarefa comum ou competir por recursos compartilhados. Sem mecanismos adequados de coordenação, podem surgir problemas como condições de corrida (race conditions), onde o resultado de uma computação depende da ordem imprevisível de execução das operações, e deadlocks, onde dois ou more processos ficam permanentemente bloqueados, cada um esperando por um recurso que o outro detém. Este capítulo aborda os conceitos fundamentais de sincronização e comunicação entre processos (Inter-Process Communication - IPC), explorando soluções históricas e atuais para o problema da exclusão mútua, discutindo problemas clássicos de IPC, e detalhando mecanismos como passagem de mensagens e barreiras. Além disso, examinaremos como a sincronização e a IPC são implementadas nos sistemas operacionais Linux e Windows.

Conceitos Fundamentais

Concorrência: Ocorre quando múltiplos processos ou threads parecem executar simultaneamente. Em sistemas monoprocessadores, isso é alcançado pela

intercalação rápida da execução (multiprogramação). Em sistemas multiprocessadores, a concorrência pode envolver paralelismo real, com diferentes processos/threads executando em diferentes CPUs ao mesmo tempo.

Recursos Compartilhados: São recursos do sistema que podem ser acessados por múltiplos processos ou threads, como variáveis em memória compartilhada, arquivos, dispositivos de E/S, etc.

Seção Crítica: É uma porção de código dentro de um processo/thread onde um recurso compartilhado é acessado. Para garantir a integridade dos dados, é essencial que apenas um processo/thread possa executar sua seção crítica (para um determinado recurso compartilhado) por vez.

Condição de Corrida (Race Condition): Uma situação onde múltiplos processos/threads acessam e manipulam dados compartilhados concorrentemente, e o resultado final da manipulação dos dados depende da ordem particular em que os acessos ocorrem. Condições de corrida podem levar a resultados incorretos e inconsistentes.

Exclusão Mútua: É um mecanismo que garante que, se um processo está executando em sua seção crítica, nenhum outro processo pode estar executando em sua seção crítica (relativa ao mesmo recurso compartilhado). É a principal forma de evitar condições de corrida.

Soluções Históricas e Atuais para Exclusão Mútua

Para garantir a exclusão mútua, várias soluções foram propostas, variando de abordagens baseadas em software a suporte de hardware.

Soluções de Software

- **Algoritmo de Peterson (para dois processos):** Uma solução clássica baseada em software que usa variáveis compartilhadas (`turn` e `flag[]`) para coordenar o acesso à seção crítica entre dois processos. Garante exclusão mútua, progresso e espera limitada (bounded waiting).
- **Algoritmo da Padaria de Lamport (para N processos):** Uma solução mais geral para N processos, onde cada processo recebe um número (como em uma padaria) e o processo com o menor número entra na seção crítica.

Soluções com Suporte de Hardware

Instruções de hardware podem simplificar a implementação da exclusão mútua.

- **Desabilitar Interrupções:** Em sistemas monoprocessadores, um processo pode desabilitar todas as interrupções antes de entrar na seção crítica e reabilitá-las ao sair. Isso impede que o processo seja preemptado, garantindo a exclusão mútua. No entanto, essa abordagem não funciona em sistemas multiprocessadores e pode ser perigosa se o processo permanecer na seção crítica por muito tempo.
- **Instruções Atômicas (Test-and-Set, Compare-and-Swap):** São instruções de máquina que executam múltiplas operações (como ler um valor, testá-lo e

modificá-lo) de forma indivisível (atômica). A instrução TestAndSet(lock) testa o valor de lock e o define como true em uma única operação atômica. A instrução CompareAndSwap(lock, expected, new) compara atomicamente o conteúdo de lock com expected; se forem iguais, lock é modificado para new.

Primitivas de Sincronização de Alto Nível

Para facilitar a programação concorrente, os sistemas operacionais e bibliotecas de programação fornecem primitivas de sincronização de mais alto nível.

- **Semáforos:** Propostos por Dijkstra, um semáforo é uma variável inteira acessada apenas através de duas operações atômicas: wait() (ou P, do holandês *proberen*, tentar) e signal() (ou V, *verhogen*, incrementar). Semáforos de contagem podem ter qualquer valor inteiro não negativo, enquanto semáforos binários (mutexes) só podem ter os valores 0 ou 1, sendo usados para implementar exclusão mútua.
- **Mutexes (Mutual Exclusion Locks):** São essencialmente semáforos binários usados para proteger seções críticas. Um processo adquire o mutex antes de entrar na seção crítica e o libera ao sair. Apenas um processo pode deter o mutex por vez.
- **Variáveis de Condição:** Usadas em conjunto com mutexes, permitem que threads esperem por uma condição específica se tornar verdadeira. Uma thread pode esperar (wait) em uma variável de condição, liberando o mutex associado. Outra thread, após alterar o estado que satisfaz a condição, pode sinalizar (signal ou broadcast) a variável de condição para acordar uma ou todas as threads em espera.
- **Monitores:** Uma construção de linguagem de programação de alto nível que fornece um mecanismo conveniente e eficaz para sincronização de processos. Um monitor é um módulo que encapsula dados compartilhados, procedimentos que operam nesses dados e mecanismos de sincronização para esses procedimentos. Apenas uma thread pode estar ativa dentro de um monitor por vez, garantindo exclusão mútua implicitamente. Variáveis de condição são frequentemente usadas dentro de monitores.

Problemas Clássicos de Comunicação entre Processos

Diversos problemas clássicos são usados para testar e demonstrar a eficácia das primitivas de sincronização:

- **Problema do Produtor-Consumidor (Bounded-Buffer Problem):** Um ou mais processos produtores geram dados e os colocam em um buffer compartilhado de tamanho finito. Um ou mais processos consumidores retiram dados do buffer. É preciso sincronizar o acesso ao buffer para evitar que produtores adicionem dados a um buffer cheio ou que consumidores tentem remover dados de um buffer vazio.
- **Problema dos Leitores-Escritores:** Múltiplos processos acessam um recurso de dados compartilhado. Alguns processos (leitores) apenas leem os dados, enquanto outros (escritores) modificam os dados. Múltiplos leitores podem acessar os dados

simultaneamente. No entanto, se um escritor está acessando os dados, nenhum outro processo (leitor ou escritor) pode acessá-los. Variações existem, priorizando leitores ou escritores, ou buscando evitar a inanição (starvation) de qualquer um deles.

- **Problema do Jantar dos Filósofos:** Cinco filósofos estão sentados em volta de uma mesa circular. Entre cada par de filósofos adjacentes há um garfo (hashi). Cada filósofo alterna entre pensar e comer. Para comer, um filósofo precisa de dois garfos: o da sua esquerda e o da sua direita. O problema é projetar um protocolo que permita aos filósofos comerem sem causar deadlock (todos pegam um garfo e esperam pelo outro) ou inanição.

Passagem de Mensagens (Message Passing)

A passagem de mensagens é um mecanismo de IPC onde os processos se comunicam e sincronizam suas ações sem compartilhar o mesmo espaço de endereçamento. Em vez disso, eles trocam mensagens. Duas operações básicas são necessárias: send(message) e receive(message).

Características

- **Comunicação Direta vs. Indireta:**
  - **Direta:** Cada processo deve nomear explicitamente o destinatário ou remetente da comunicação (e.g., send(P, message) envia para o processo P; receive(Q, message) recebe de Q).
  - **Indireta:** As mensagens são enviadas e recebidas de caixas de correio (mailboxes) ou portas. Múltiplos processos podem compartilhar uma caixa de correio.
- **Sincronização (Bloqueante vs. Não-Bloqueante):**
  - **Send Bloqueante:** O remetente é bloqueado até que a mensagem seja recebida pelo destinatário ou pela caixa de correio.
  - **Send Não-Bloqueante:** O remetente envia a mensagem e continua sua execução.
  - **Receive Bloqueante:** O receptor é bloqueado até que uma mensagem esteja disponível.
  - **Receive Não-Bloqueante:** O receptor tenta receber uma mensagem; se nenhuma estiver disponível, ele retorna imediatamente (com um indicador de falha ou uma mensagem nula).
- **Buffering:** As mensagens trocadas residem em uma fila temporária. As filas podem ter capacidade zero (sem buffer, o remetente deve esperar pelo receptor - rendezvous), capacidade limitada ou capacidade ilimitada.

Barreiras (Barriers)

Uma barreira é um ponto de sincronização onde múltiplos processos ou threads devem esperar até que todos os membros de um grupo cheguem a esse ponto antes que qualquer um deles possa prosseguir. Barreiras são comumente usadas em computação

paralela, onde um cálculo é dividido em fases e todos os threads devem completar uma fase antes de iniciar a próxima.

Sincronização e Comunicação nos Sistemas Operacionais Linux e Windows

Linux

O Linux fornece um rico conjunto de mecanismos de sincronização e IPC, tanto no nível do kernel quanto para aplicações de espaço de usuário, aderindo em grande parte aos padrões POSIX.

- **Sincronização no Kernel:** Spinlocks, mutexes, semáforos, read-write locks, completion variables.
- **Sincronização no Espaço do Usuário (POSIX):**
  - **Mutexes (Pthreads):** pthread_mutex_t
  - **Variáveis de Condição (Pthreads):** pthread_cond_t
  - **Semáforos (POSIX):** Nomeados (sem_open) e não nomeados/baseados em memória (sem_init).
  - **Barreiras (Pthreads):** pthread_barrier_t
  - **Read-Write Locks (Pthreads):** pthread_rwlock_t
- **IPC no Linux:**
  - **Pipes (Anônimos e Nomeados - FIFOs):** Fluxos de bytes unidirecionais.
  - **Filas de Mensagens (POSIX e System V):** Permitem que processos troquem mensagens formatadas.
  - **Memória Compartilhada (POSIX e System V):** Permite que múltiplos processos compartilhem uma região de memória.
  - **Sockets (Berkeley Sockets):** Para comunicação em rede e também IPC local (Unix Domain Sockets).
  - **Sinais:** Mecanismo assíncrono para notificar processos sobre eventos.

Windows

O Windows oferece um conjunto abrangente de objetos de sincronização e mecanismos de IPC através da Win32 API.

- **Objetos de Sincronização:**
  - **Mutexes:** CreateMutex, ReleaseMutex, WaitForSingleObject.
  - **Semáforos:** CreateSemaphore, ReleaseSemaphore, WaitForSingleObject.
  - **Eventos (Auto-reset e Manual-reset):** CreateEvent, SetEvent, ResetEvent, PulseEvent, WaitForSingleObject. Usados para sinalizar a ocorrência de uma condição.

  - **Critical Sections:** Um mecanismo de sincronização mais leve e rápido para threads dentro do mesmo processo. InitializeCriticalSection, EnterCriticalSection, LeaveCriticalSection.
  - **Slim Reader/Writer (SRW) Locks:** Otimizados para cenários com muitos leitores e poucos escritores.
  - **Condition Variables:** Usadas com critical sections ou SRW locks. InitializeConditionVariable, SleepConditionVariableCS, SleepConditionVariableSRW, WakeConditionVariable, WakeAllConditionVariable.
- **IPC no Windows:**
  - **Pipes (Anônimos e Nomeados):** Semelhantes aos do Linux.
  - **Mailslots:** Comunicação unidirecional baseada em mensagens, tipicamente em rede local.
  - **Memória Compartilhada (Memory-Mapped Files):** Permite que processos mapeiem a mesma seção de um arquivo (ou da memória do sistema) em seus espaços de endereço.
  - **Sockets (Winsock):** Para comunicação em rede e IPC local.
  - **Chamada de Procedimento Remoto (RPC):** Mecanismo de alto nível para comunicação entre processos, inclusive em máquinas diferentes.
  - **Mensagens do Windows (Window Messages):** Usadas primariamente para comunicação com interfaces gráficas, mas podem ser usadas para IPC geral entre threads que possuem filas de mensagens.

Conclusão

A sincronização e a comunicação entre processos são aspectos fundamentais do design de sistemas operacionais e da programação concorrente. A escolha correta e o uso adequado de primitivas de sincronização como mutexes, semáforos e variáveis de condição são essenciais para evitar problemas como condições de corrida e deadlocks, garantindo a corretude e a eficiência de aplicações multithreaded e sistemas multiprocesso. Mecanismos de IPC, como passagem de mensagens, pipes e memória compartilhada, fornecem os meios para que processos cooperem e troquem dados. Tanto Linux quanto Windows oferecem um conjunto robusto e diversificado de ferramentas para sincronização e IPC, permitindo que os desenvolvedores construam aplicações concorrentes complexas e eficientes. A compreensão desses mecanismos é vital para o desenvolvimento de software robusto e de alto desempenho em ambientes modernos.

Referências

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.
- Tanenbaum, A. S., & Bos, H. (2015). *Modern Operating Systems* (4th ed.). Pearson Education.
- Downey, A. B. (2016). *The Little Book of Semaphores* (2nd ed.). Green Tea Press.

- Hart, J. M. (2005). *Windows System Programming* (3rd ed.). Addison-Wesley Professional.
- Love, R. (2010). *Linux Kernel Development* (3rd ed.). Addison-Wesley Professional.