SISTEMAS OPERACIONAIS

Threads

Execução Assíncrona Concorrente

- **Execução concorrente (Threads Concorrentes):**
  - Pode haver mais de um thread no sistema simultaneamente.
  - Os threads podem ser executados independentemente ou cooperativamente.
  - Diz-se que processos (threads) operam independentemente um do outro, contudo, de quando em quando, devem se comunicar e se sincronizar para executar tarefas cooperativas executam (funcionam) ***assincronamente***.
  - **Execução assíncrona**
    - Os threads em geral são independentes.
    - Podem se comunicar ou sincronizar ocasionalmente.
    - O gerenciamento dessas interações é complexo e difícil.

Problema: dois threads acessando dados ao mesmo tempo

- Os dados podem ficar inconsistentes.
  - O chaveamento de contexto pode ocorrer a qualquer momento. Por exemplo, antes de o thread terminar de mudar um valor.
- Esses dados precisam ser acessados de uma maneira mutuamente exclusiva.
  - Deve-se permitir o acesso a apenas um thread por vez.
  - Outros threads devem esperar até que o recurso seja desbloqueado.
  - O acesso é serializado.
  - Esse processo precisa ser gerenciado de modo que o tempo de espera não seja exagerado.

Problema: dois threads acessando dados ao mesmo tempo

Ou seja:
Quando um thread lê dados que um outro thread está escrevendo, ou quando um thread escreve dados que um outro thread também está escrevendo, podem ocorrer resultados indeterminados.

- Pode-se resolver esse problema concedendo a cada thread acesso exclusivo à variável compartilhada.
- Enquanto um thread incrementa a variável compartilhada, todos os outros threads que desejam fazer o mesmo terão que esperar.

Essa operação é denominada: ***EXCLUSÃO MÚTUA***

Serialização de acesso

- Quando o thread que está em execução terminar de acessar a variável compartilhada, o sistema permitirá que um dos processos à espera prossiga.

Essa operação é denominada: ***Serialização do acesso*** à variável compartilhada.

Desta maneira, threads não poderão acessar dados compartilhados simultaneamente.

Relacionamento produtor/consumidor

- Um thread produtor cria dados para armazená-los em um objeto compartilhado.
- O thread consumidor lê os dados desse objeto.
  - Devido a erros de lógica com acesso não sincronizados, há grande possibilidade de corrupção de dados.

- ***Dados podem ser perdidos se o produtor colocar novos dados no buffer compartilhado antes que o consumidor consuma (leia) os dados anteriores.***
- ***Dados podem ser incorretamente duplicados se o consumidor consumir dados novamente antes que o produtor produza o próximo valor.***

Se essa lógica fizesse parte de uma aplicação de contrôle de tráfego aéreo, vidas humanas poderiam estar em risco !!

Seção Crítica

- Exclusão Mútua precisa ser imposta SOMENTE quando threads acessam dados modificáveis compartilhados.

- Quando estão executando operações que não conflitam umas com as outras (lendo dados, por exemplo), o sistema deve permitir que os threads executem (acessem) concorrentemente.

Quando um thread acessa dados modificáveis compartilhados, diz-se que está em uma ***seção crítica*** (ou região crítica).

Seção crítica

Para evitar os tipos de erros mencionados anteriormente, o sistema deve garantir que somente um thread por vez possa executar instruções em sua seção crítica.

- *Considerações:*
  - *Se um thread qualquer tentar entrar em sua seção crítica equanto outro estiver executando sua própria seção crítica, o primeiro deverá esperar até que o thread que está em execução saia de sua seção crítica.*
  - *Assim que o thread sair da sua seção criítica, o thread que esteja esperando (ou um dos threads à espera, se houver vários) poderá entrar e executar sua seção crítica.*
  - *Se um thread que estiver dentro de uma seção crítica terminar (voluntária ou involuntáriamente), o sistema operacional, ao realizar sua "limpeza final", deverá liberar a exclusão mútua para que outros threads possam entrar em suas seções críticas.*

Exemplo 1: Problemas de concorrência

- O primeiro problema é analisado a partir do programa Conta_Corrente, que atualiza o saldo bancário de um cliente após um lançamento de debito ou crédito no arquivo de contas-correntes Arq_Contas. Neste arquivo são armazenados os saldos de todos os correntistas do banco. O programa lê o registro do cliente no arquivo (Reg_Cliente), lê o valor a ser depositado ou retirado (Valor_Dep_Ret) e, em seguida, atualiza o saldo no arquivo de contas.
    - *PROGRAM Conta_Corrente;*
    - *..*
    - *READ (Arq_Contas, Reg_Cliente);*
    - *READLN (Valor_Dep_Ret);*
    - *Reg_Cliente.Saldo := Reg_Cliente.Saldo + Valor_Dep_Ret;*
    - *WRITE (Arq_Contas, Reg_Cliente);*
    - *. .*
    - *END.*
- Considerando processos concorrentes pertencentes a dois funcionários do banco que atualizam o saldo de um mesmo cliente simultaneamente, a situação de compartilhamento do recurso pode ser analisada no exemplo da Tabela 7.1. O processo do primeiro funcionário (Caixa 1) lê o registro do cliente e soma ao campo Saldo o valor do lançamento de débito. Antes de gravar o novo saldo no arquivo, o processo do segundo funcionário (Caixa 2) lê o registro do mesmo cliente, que está sendo atualizado, para realizar outro lançamento, desta vez de crédito. Independentemente de qual dos processos atualize primeiro o saldo no arquivo, o dado gravado estará inconsistente.

B., MACHADO, F., MAIA, Paulo. Arquitetura de Sistemas Operacionais, 5ª edição. LTC, 03/2013.

Tabela 7.1 Problema de Concorrência I

Caixa	Instrução	Saldo arquivo	Valor dep/ret	Saldo memória
1	READ	1.000	*	1.000
1	READLN	1.000	-200	1.000
1	:=	1.000	-200	800
2	READ	1.000	*	1.000
2	READLN	1.000	+300	1.000
2	:=	1.000	+300	1.300
1	WRITE	800	-200	800
2	WRITE	1.300	+300	1.300

Exemplo 2: Problemas de concorrência

- Outro exemplo, ainda mais simples, onde o problema da concorrência pode levar a resultados inesperados é a situação onde dois processos (A e B) executam um comando de atribuição. O Processo A soma 1 à variável X e o Processo B diminui 1 da mesma variável que está sendo compartilhada. Inicialmente, a variável X possui o valor 2.

Processo A	Processo B
X := X + 1;	X := X − 1;

- Seria razoável pensar que o resultado final da variável X, após a execução dos Processos A e B, continuasse 2, porém isto nem sempre será verdade. Os comandos de atribuição, em uma linguagem de alto nível, podem ser decompostos em comandos mais elementares, como visto a seguir:

Processo A	Processo B
LOAD x, R_a	LOAD x, R_b
ADD 1, R_a	SUB 1, R_b
STORE R_a, x	STORE R_b, x

Exemplo 2: Problemas de concorrência

- Utilizando o exemplo da Tabela 7.2, considere que o Processo A carregue o valor de X no registrador Ra, some 1 e, no momento em que vai armazenar o valor em X, seja interrompido. Nesse instante, o Processo B inicia sua execução, carrega o valor de X em Rb e subtrai 1. Dessa vez, o Processo B é interrompido e o Processo A volta a ser processado, atribuindo o valor 3 à variável X e concluindo sua execução. Finalmente, o Processo B retorna a execução, atribui o valor 1 a X, e sobrepõe o valor anteriormente gravado pelo Processo A. O valor final da variável X é inconsistente em função da forma concorrente com que os dois processos executaram.
  - *B., MACHADO, F., MAIA, Paulo. Arquitetura de Sistemas Operacionais, 5ª edição.*

Tabela 7.2 Problema de Concorrência II

Processo	Instrução	X	R_a	R_b
A	LOAD X,R_a	2	2	*
A	ADD 1,R_a	2	3	*
B	LOAD X,R_b	2	*	2
B	SUB 1,R_b	2	*	1
A	STORE R_a,X	3	3	*
B	STORE R_b,X	1	*	1

Algoritmo de Dekker

- Solução apropriada.
- Usa a noção de threads favorecidos para determinar a entrada em seções críticas.
  - Resolve o conflito sobre qual thread deveria ser executado em primeiro lugar.
  - Todo thread desconfigura temporariamente o flag de solicitação de seção crítica.
  - O status favorecido alterna entre threads.
- Garante exclusão mútua.
- Evita problemas anteriores de deadlock e adiamento indefinido.

Algoritmo de Peterson

- Menos complicado que o algoritmo de Dekker
- Também usa espera ociosa e threads favorecidos.
- Requer menos etapas para executar primitivas de exclusão mútua.
- É fácil de demonstrar sua precisão.
- Não apresenta adiamento indefinido ou deadlock.

Algoritmo da padaria de Lamport

- Aplicável a qualquer quantidade de threads.
  - Cria uma fila de threads em espera distribuindo "fichas" numeradas.
  - O thread é executado quando, dentre todos os outros threads, o número de sua ficha é o menor.
  - Diferentemente do algoritmo de Dekker e do algoritmo de Peterson, o algoritmo da padaria funciona em sistemas multiprocessadores e para *n* threads.
  - É relativamente simples de entender por ser uma condição análoga à do mundo real

Semáforos

- Arquitetura de software que pode ser usada para impor a exclusão mútua.
- Contém uma variável protegida.
  - Essa variável pode ser acessada apenas por meio dos comandos esperar e sinalizar.
  - Também chamado de operações *P* e *V*, respectivamente.

Exclusão Mútua com Semáforos:

Semáforo binário: permite apenas um thread por vez em sua seção crítica.

Operação esperar
- Se nenhum thread estiver esperando, permite que o thread permaneça em sua seção crítica.
- Diminui a variável protegida (a 0 nesse caso).
- Do contrário, coloca em fila de espera.

Operação sinalizar
- Indica que o thread está fora de sua seção crítica.
- Aumenta a variável protegida (de 0 para 1).
- O thread que estiver aguardando (se houver) então pode entrar.

- **Os semáforos podem ser implementados na aplicação ou no núcleo.**
  - Na aplicação: normalmente são implementados com espera ociosa.
    - Ineficiente.
  - No núcleo podem evitar espera ociosa.
    - Os threads em espera são bloqueados até que estejam prontos.
  - As implementações no núcleo podem desabilitar interrupções.
    - Garantem acesso exclusivo ao semáforo.
    - É necessário cuidado para evitar baixo desempenho e deadlock.
    - As implementações para sistemas multiprocessadores precisam usar uma abordagem mais aprimorada.

Quizzzzzzzzzz....

- Cite algumas razões porque o estudo da concorrência é apropriado e importante para estudantes de sistemas operacionais.

- Explique porque a seguinte afirmativa é falsa:

"Quando diversos threads acessam informações compartilhadas na memória principal, a exclusão mútua deve ser imposta para evitar a produção de resultados indeterminados".

- Qual o significado do algoritmo de Dekker?