


UNIVALI
Universidade do Vale do Itajaí
Escola Politécnica

Ferramentas de sincronização

1




UNIVALI

Tópicos

- Conceitos
- O problema da seção crítica
- Solução da Peterson
- Suporte de hardware para sincronização
- Bloqueios por Mutex
- Semáforos

2




UNIVALI

Conceitos

- Os processos podem ser executados simultaneamente
 - Pode ser interrompido a qualquer momento, concluindo parcialmente a execução
- O acesso simultâneo a dados compartilhados pode resultar em inconsistência de dados
- Manter a consistência dos dados requer mecanismos para garantir a execução ordenada dos processos cooperantes
- Vimos o problema quando consideramos o problema do buffer limitado com o uso de um contador que é atualizado simultaneamente pelo produtor e pelo consumidor
 - O que leva à condição da corrida

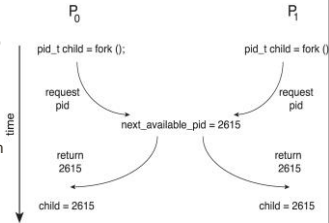
3



UNIVALI

Condição de Corrida

- Os processos P0 e P1 estão criando processos filho usando a chamada do sistema fork()
- Condição de corrida na variável do kernel `next_available_pid` que representa o próximo identificador de processo disponível (pid)
- A menos que haja um mecanismo para impedir que P0 e P1 acessem a variável `next_available_pid`, o mesmo pid pode ser atribuído a dois processos diferentes!



4

Problema da seção crítica

- Considere o sistema de n processos $\{p_0, p_1, \dots, p_{n-1}\}$
 - Cada processo tem segmento de seção crítica de código
 - O processo pode ser alterar variáveis comuns, atualizar tabela, escrever arquivo, etc
- Quando um processo em seção crítica, nenhum outro pode estar em sua seção crítica
- O problema de seção crítica é projetar o protocolo para resolver isso
- Cada processo deve pedir permissão para entrar na seção crítica, executar, sair e prosseguir nas instruções restantes

```
while (true) {
    entry section
    critical section
    exit section
    remainder section
}
```



5

Problema de Seção Crítica

- Requisitos para solução de problema de seção crítica

1. **Exclusão Mútua** - Se o processo P_i estiver sendo executado em sua seção crítica, nenhum outro processo poderá ser executado em suas seções críticas
2. **Progresso** - Se nenhum processo estiver sendo executado em sua seção crítica e existirem alguns processos que desejam entrar em sua seção crítica, a seleção do processo que entrará na seção crítica a seguir não poderá ser adiada indefinidamente
3. **Espera limitada** - Um vinculado deve existir no número de vezes que outros processos têm permissão para entrar em suas seções críticas depois que um processo tiver feito uma solicitação para entrar em sua seção crítica e antes que a solicitação seja concedida
 - Suponha que cada processo seja executado a uma velocidade diferente de zero
 - Nenhuma suposição sobre a velocidade relativa dos n processos



6

Solução baseada em interrupção

- Seção de entrada: desativar interrupções
- Seção de saída: ativar interrupções
- Isso resolverá o problema?
 - E se a seção crítica for um código que é executado por uma hora?
 - Alguns processos podem morrer de fome (conceito de starvation) – nunca entrar em sua seção crítica (conseguir o recurso demandado)
 - E se houver duas CPUs?



7

Solução de Software 1

- Solução de dois processos
- Suponha que as instruções de linguagem de máquina de carregamento e armazenamento sejam atômicas; ou seja, não pode ser interrompido
- Os dois processos compartilham uma variável:

```
int turn;
• A variável turn indica de quem é a vez de entrar na seção crítica
• inicialmente, o valor de turn é definido como i

while (true){
    while (turn == j);

    /* critical section */

    turn = j;
    /* remainder section */
}
```



8

Barreira de memória



- Modelo de memória são as garantias de memória que uma arquitetura de computador faz para programas de aplicação
- Os modelos de memória podem ser:
 - Fortemente ordenado – onde uma modificação de memória de um processador é imediatamente visível para todos os outros processadores
 - Fracamente ordenado – onde uma modificação de memória de um processador pode não ser imediatamente visível para todos os outros processadores
- Uma barreira de memória é uma instrução que força qualquer alteração na memória a ser propagada (tornada visível) para todos os outros processadores

9

Instruções de barreira de memória



- Quando uma instrução de barreira de memória é executada, o sistema garante que todas as operações de carregamento e armazenamento sejam concluídas antes que qualquer carregamento subsequente ou operações de armazenamento sejam executadas
- Portanto, mesmo que as instruções tenham sido reordenadas, a barreira da memória garante que as operações de armazenamento sejam concluídas na memória e visíveis para outros processadores antes que futuras operações de carregamento ou armazenamento sejam executadas

10

Exemplo de barreira de memória



- Poderíamos adicionar uma barreira de memória às seguintes instruções para garantir que o Thread 1 produza 100:
- O thread 1 agora executa


```
while (!flag)
  memory_barrier();
print x
```
- O thread 2 agora executa


```
x = 100;
memory_barrier();
flag = true
```
- Para o Thread 1, temos a garantia de que o valor do sinalizador é carregado antes do valor de x
- Para o Thread 2, garantimos que a atribuição a x ocorra antes do sinalizador de atribuição

11

Hardware de sincronização



- Muitos sistemas fornecem suporte de hardware para implementar o código de seção crítica.
- Processadores de único núcleo – podem desativar interrupções
- O código atualmente em execução seria executado sem preempção
- Geralmente muito ineficiente com multiprocessadores
- Os sistemas operacionais que usam isso não são amplamente escalonáveis
- Veremos três formas de suporte de hardware:
 1. Instruções de hardware
 2. Variáveis atômicas

12

Instruções de hardware

- Instruções especiais de hardware que nos permitem testar e modificar o conteúdo de uma palavra, ou duas trocam o conteúdo de duas palavras atômica e ininterruptamente (ininterruptamente)
 - Test-and-Set
 - Compare-and-Swap

13

A Instrução compare_and_swap

Definição

```
int compare_and_swap(int *value, int expected, int new_value){
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

Propriedades

- Executado atômica e ininterruptamente
- Retorna o valor original do valor do parâmetro passado
- Define o valor da variável o valor do parâmetro passado new_value mas somente se *value == expected for true. Ou seja, a troca ocorre apenas sob essa condição

14

Solução usando compare_and_swap

- Bloqueio de inteiro compartilhado inicializado como 0;
- Solução:

```
while (true){
    while (compare_and_swap(&lock,0, 1) != 0); /*do nothing */
    /* critical section */
    lock = 0;
    /* remainder section */
}
```

- Isso resolve o problema da seção crítica?

15

Variáveis atômicas

- Normalmente, instruções compare-and-swap são usadas como blocos de construção para outras ferramentas de sincronização.
- Uma ferramenta é uma variável atômica que fornece atualizações atômicas (ininterruptas) em tipos de dados básicos, como inteiros e booleanos.
- Por exemplo:
 - Seja a sequência uma variável atômica
 - Seja increment() a operação na sequência de variáveis atômicas
- O comando: increment(&sequence); garante que a sequência seja incrementada sem interrupção

16

Bloqueios por Mutex

- As soluções anteriores são complicadas e geralmente inacessíveis aos programadores
- São criadas ferramentas para resolver problemas críticos de seção
- O mais simples é o bloqueio mutex
 - Variável booleana que indica se o bloqueio está disponível ou não
- Proteja uma seção crítica
 - Primeiro adquira() um bloqueio
 - Em seguida, libere() o bloqueio
- As chamadas para adquirir() e liberar() devem ser atômicas
 - Geralmente implementado através de instruções atômicas de hardware, como comparar e trocar
- Mas esta solução requer uma espera atarefada
 - Essa trava é chamada de spinlock

17

Solução para o problema de SC usando bloqueios por Mutex

```
while (true) {
    acquire lock
    critical section
    release lock
    remainder section
}
```

18

Semáforo

- Ferramenta de sincronização que fornece maneiras mais sofisticadas (do que os bloqueios Mutex) para que os processos sincronizem suas atividades.
- Semaphore S – integer variable
- Só pode ser acessado através de duas operações indivisíveis (atômicas)
 - wait() e signal()
 - Originalmente chamado de P() e V()
- Definição da operação wait() e signal()

wait(S) {	signal(S) {
while (S <= 0); // busy wait	S++;
S--;	}
}	

19

Semáforo (Cont.)

- Semáforo contador – o valor inteiro pode variar em um valor a ser definido
- Semáforo binário – o valor inteiro pode variar apenas entre 0 e 1
 - Literalmente igual mutex
- Pode implementar um semáforo binário S como um semáforo contador
- Com semáforos podemos resolver vários problemas de sincronização

20

Exemplo de uso de semáforo



- Solução para o problema CS

- Criar um semáforo "mutex" inicializado para 1

```
wait(mutex);
SC
signal(mutex);
```

- Considere P1 e P2 que com duas afirmações S1 e S2 e a exigência de que S1 aconteça antes de S2

- Criar um semáforo "synch" inicializado em 0

```
P1:      P2:
S1;      wait(synch);
signal(synch);  S2;
```

21

Implementação do Semáforo



- Deve garantir que não haja dois processos que possam executar o wait() e o signal() no mesmo semáforo ao mesmo tempo
- Assim, a implementação se torna o problema da seção crítica, onde o código de espera e sinal é colocado na seção crítica
- Agora poderia ter ocupado aguardando na implementação de seção crítica
 - Mas o código de implementação é curto
 - Pouco ocupado esperando se a seção crítica raramente ocupada
- Observe que os aplicativos podem gastar muito tempo em seções críticas e, portanto, essa não é uma boa solução

22

Problemas com semáforos



- Uso incorreto de operações de semáforo:

- signal(mutex) wait(mutex)
 - wait(mutex) ... wait(mutex)
 - Omissão do wait (mutex) e/ou signal (mutex)

- Estes – e outros – são exemplos do que pode ocorrer quando semáforos e outras ferramentas de sincronização são usados incorretamente..

23

Liveness



- Os processos podem ter que esperar indefinidamente ao tentar adquirir uma ferramenta de sincronização, como um bloqueio mutex ou semáforo
- Esperar indefinidamente viola o progresso e os critérios de espera limitada
- Liveness refere-se a um conjunto de propriedades que um sistema deve satisfazer para garantir que os processos progridam
- A espera indefinida é um exemplo de um fracasso

24

Liveness



- Impasse – dois ou mais processos estão aguardando indefinidamente por um evento que pode ser causado por apenas um dos processos em espera
- Seja S e Q dois semáforos inicializados para 1

P0	P1
wait(S);	wait(Q);
wait(Q);	wait(S);
...	...
signal(S);	signal(Q);
signal(Q);	signal(S);

- Considere se P0 executa wait(S) e P1 wait(Q). Quando P0 executa wait(Q), ele deve aguardar até que P1 execute signal(Q)
- No entanto, P1 está aguardando até que P0 execute sinal(S)
- Como essas operações signal() nunca serão executadas, P0 e P1 estão bloqueadas

25

Liveness



- Outras formas de impasse:
- Fome (starvation) – bloqueio indefinido
- Um processo nunca pode ser removido da fila de semáforos em que está suspenso
- Inversão de prioridade – Problema de agendamento quando o processo de prioridade mais baixa mantém um bloqueio necessário para o processo de prioridade mais alta
- Resolvido por meio do protocolo de herança de prioridade

26