Arquitetura e Organização de Computadores II

Sincronismo

Prof. Nilton Luiz Queiroz Jr.

Sincronismo

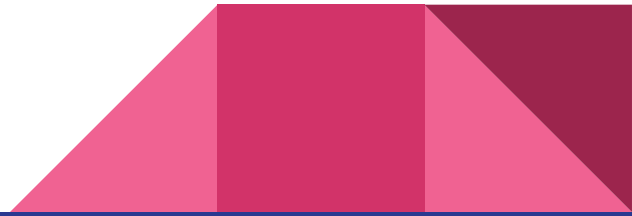
- Mecanismos de sincronia são normalmente construídos com rotinas de software que usam instruções fornecidas pelo hardware;
- Em multiprocessadores menores a capacidade principal do hardware é uma instrução capaz de ler, alterar e salvar o valor de maneira indivisível;
  - Em outras palavras, a instrução é feita de maneira atômica;

Sincronismo

- Implementar a sincronização em multiprocessadores, requer principalmente, a implementação de primitivas atômicas em memória;
- Essas primitivas implementam também mecanismos que informam se a leitura e escrita foram feitas de maneira atômica;
- Em multiprocessadores de larga escala a sincronização pode ser um gargalo;
  - São necessárias técnicas avançadas para reduzir a introdução de atrasos adicionais e a latência de sincronização;

Sincronismo

- As primitivas atômicas são usadas para construir uma grande variedade de operações de sincronia a nível de usuário;
  - Por exemplo:
    - Locks (bloqueios);
    - Barreiras;



Instruções primitivas

- Existem diversas instruções primitivas atômicas que podem ser implementadas para dar suporte a operações de sincronismo:
  - Test-and-set;
  - Fetch-and-increment;
  - Atomic exchange;

Instruções primitivas

- Test-and-set(testar e marcar);
  - Testa um valor e marca caso ele passe no teste;
  - Assim o valor de retorno pode ser armazenado em um registrador, e caso o lock esteja livre, ele será então marcado para que os próximos testes falhem;
  - Após sair da região crítica, deve-se atualizar a memória;

```
Test&Set (m), R:
    R ← M[m];
    if R==0 then
        M[m] ← 1;
```

Instruções primitivas

- Fetch-and-add (buscar e incrementar);
  - Retorna o valor de um local de memória e o incrementa;
  - Usa-se o valor 0 para informar que a variável de sincronismo está livre;

```
Fetch&Add (m), RV, R:
    R ← M[m];
    M[m] ← R + RV;
```

Instruções primitivas

- Atomic Exchange (troca atômica);
  - Permuta um valor em um registrador por um valor em memória;
  - A implementação de um lock simples, usando 0 para lock livre e 1 para o lockindisponível, faria a troca de um registrador, armazenando o valor de lock, pelo endereço de memória correspondente ao lock;
    - Caso o endereço estivesse bloqueado a troca armazenada no registrador o valor 1, que significaria indisponibilidade e “não alteraria” o valor da memória;
    - Caso contrário, o calor a ser armazenado no registrador seria 0, o que significa que o endereço está disponível, além disso seria armazenado 1 no endereço do lock;
  - Quando dois processadores tentarem modificar o mesmo lock, somente um terá retorno 0;

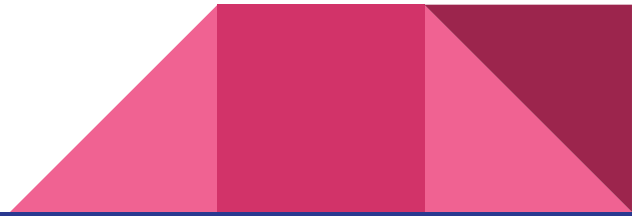
```
Swap (m), R:
    Rt ← M[m];
    M[m] ← R;
    R ← Rt;
```

Instruções primitivas

- Implementações de memória indivisíveis exigem leituras e escritas em memória sem nenhuma outra instrução no meio;
  - Complica a implementação de coerência;
- Uma alternativa é implementar um par de instruções onde a segunda instrução retorne um valor que indique se as instruções foram executadas como se fossem indivisíveis;
  - Nenhum outro processador pode alterar o valor entre o par;

Instruções primitivas

- Assim é necessária a implementação de uma instrução especial para load, e outra para store:
  - Load Linked (carregamento ligado);
  - Store Conditional (armazenamento condicional);
- O armazenamento condicional só funcionará quando:
  - Não houver alterações no endereço carregado;
  - Não houver troca de contexto no processador entre as duas instruções;



Instruções primitivas

- São necessários registradores especiais para armazenar uma flag e os endereços de memória, além de um registrador para armazenar o retorno do store condicional;
  - O registrador pode ser o próprio a ser armazenado na memória;
- O registrador extra usado de flag pode também ser chamado de registrador de link;
- Esse registrador é “apagado” quando:
  - Ocorrem interrupções;
  - O bloco da cache combinado com o registrador de link for invalidado;
- A instrução de store verifica se o endereço está com a flag correta, caso não esteja a operação falha;

Instruções primitivas

-

```
Store-conditional (m), R:
    if <flag, adr> == <1, m>
    then  cancel other procs'
          reservation on m;
          M[m] ← R;
          R ← 1
    else  R ← 0;
```

```
Load-link R, (m):
    <flag, adr> ← <1, m>;
    R ← M[m];
```

Instruções primitivas

- Essas duas operações juntas podem implementar outras primitivas, tais como:
  - Atomic Exchange:

```
try:  MOV    R3,R4
      LL     R2,0(R1)
      SC     R3,0(R1)
      BEQZ   R3,try
      MOV    R4,R2
```

  - Fetch and add:

```
try:  LL     R2,0(R1)
      DADDUI R3,R2,#1
      SC     R3,0(R1)
      BEQZ   R3,try
```

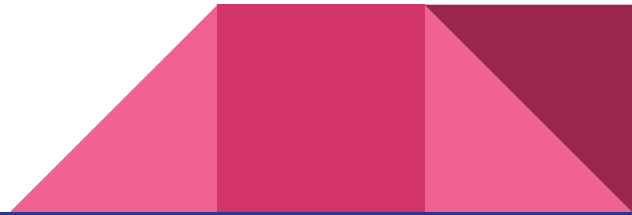
Implementação de locks usando coerência

- Pode-se usar mecanismos de coerência para implementar spin locks;
  - O processador tenta obter os locks dentro de um loop, e fica no laço até obtelos;
- Sem coerência de cache poderia-se implementar da seguinte maneira:

```
        DADDUI  R2,R0,#1
lockit: EXCH    R2,0(R1)
        BNEZ    R2,lockit
```

Implementação de locks usando coerência

- Processadores de um multiprocessadores com coerência podem trazer os locks para a memória até conseguirem o acesso;
  - Acesso à cache local ao invés de acessar a memória global;
  - Tirar vantagem do princípio da localidade
    - O lock muito provavelmente será usado a seguir;
- Desse modo a latência de acesso pode ser reduzida;



Implementação de locks usando coerência

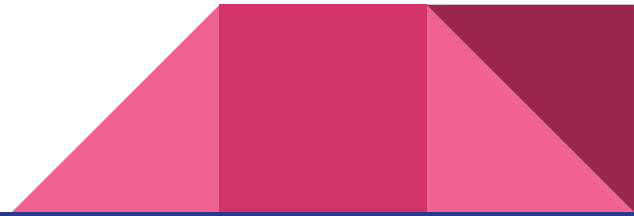
- Ser capaz de iterar em uma cópia local da cache exige uma mudança no procedimento;
  - Se múltiplos processadores tentarem obter o lock, todos geraram uma escrita;
- O procedimento deve seguir os passos:
  - Iterar até que o lock esteja disponível;
  - Tentar ler o valor do lock com uma troca atômica;
  - Caso não vença a corrida deve voltar a iterar para obter o lock;
  - Depois de executar o código exclusivo o processador deve armazenar 0 na variável do lock;

Implementação de locks usando coerência

- Em outras palavras, os processadores que desejam o lock devem aguardar até que ele fique disponível;
- Em seguida, todos eles tentam fazer uma instrução de atomic exchange;
  - Irá ocorrer uma corrida entre esses processadores para obtenção do recurso;
- Somente um conseguira, os demais irão voltar a aguardar a disponibilidade do lock;
- O processador vencedor deverá liberar o lock para os demais;

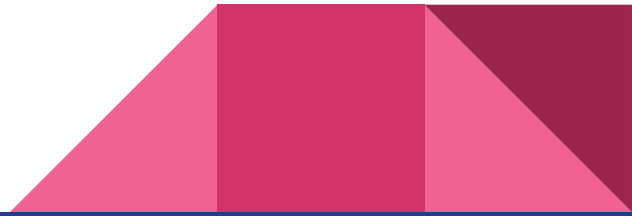
Implementação de locks usando coerência

```
lockit:LD       R2,0(R1)
       BNEZ     R2,lockit
       DADDUI   R2,R0,#1
       EXCH     R2,0(R1)
       BNEZ     R2,lockit
```



Exemplo

- Imagine uma situação com 3 processadores, P0, P1 e P2, onde P0 inicialmente tem o lock, P1 e P2 estão aguardando por ele
- A partir do momento que P0 retirar o bloqueio irá ocorrer uma corrida pelo lock entre P1 e P2;
- O vencedor da corrida irá escrever no lock e entrará no código exclusivo;
- O “perdedor” irá esperar o bloqueio liberar;



Exemplo

Step	P0	P1	P2	Coherence state of lock at end of step	Bus/directory activity
1	Has lock	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	Shared	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock variable from P0.
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write-back from P0; state shared.
4		(Waits while bus/ directory busy)	Lock = 0 test succeeds	Shared	Cache miss for P2 satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets lock = 1	Exclusive (P2)	Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.
7		Swap completes and returns 1, and sets lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		Spins, testing if lock = 0			None

Implementação de locks usando coerência

- Também é possível implementar usando as primitivas de load linked e store store conditional:

```
lockit:LL       R2,0(R1)
       BNEZ     R2,lockit
       DADDUI   R2,R0,#1
       SC       R2,0(R1)
       BEQZ     R2,lockit
```

Referências

HENNESSY, John L.;PATTERSON, David A. Computer architecture: a quantitative approach. Elsevier, 2011.