Memória Transacional

Programação Concorrente e Paralela – 2013.2

Motivação

- dificuldades de se trabalhar com memória compartilhada e locks
  - deadlocks e serialização
- sincronização *wait-free*: soluções específicas para cada estrutura de dados
  - listas, filas, tabelas de hash, ...

Locks – dificuldades

programador pode facilmente se perder com:

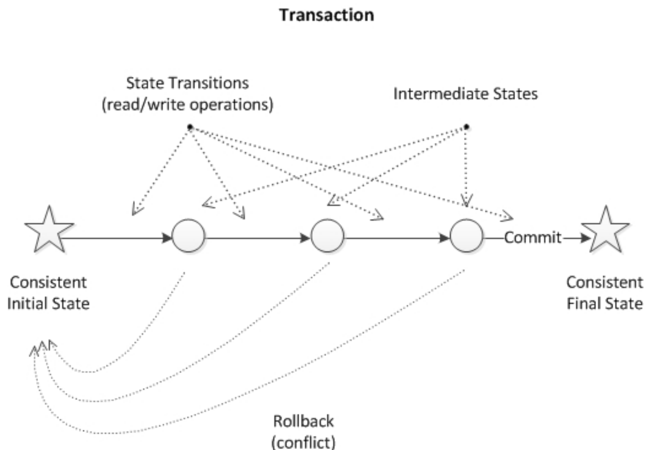
- locks de menos
- locks demais
- locks erradas
- locks pedidas na ordem errada
- tratamento de erros
- sinalizações perdidas e erros em *retries*

- uso de construções similares às de transações de bancos de dados
- transações *linearizáveis*
- extensão de Java (descrita por Herlihy&Shavit):

```
void transfer (account from, int amount) {
  atomic {
    if (from.balance() >= amount) {
      from.withdraw(amount);
      this.deposit(amount);
    }
}
```

- propriedades ACID
  - atomicidade, consistência, isolamento, durabilidade

Transação – Conceito



Implementação

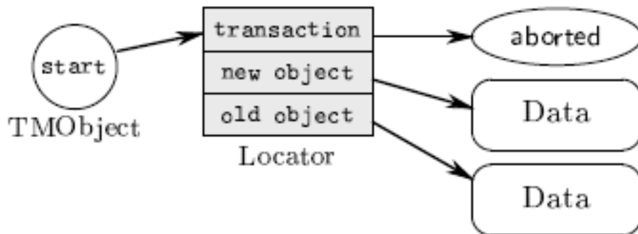
- um lock “guarda-chuva” pedido em todos os trechos atômicos seria uma implementação conceitualmente correta?

- propostas de implementação por hardware
- uso de logging como em BDs
- uso de marcações e objetos intermediários

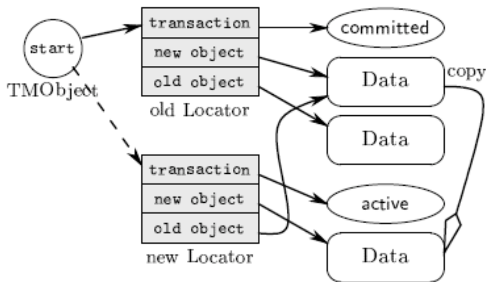
- políticas otimistas e pessimistas

Objetos Atômicos – Indireção

- descritor de objeto atômico:



Objetos Atômicos – Indireção



- instruções do tipo *compareAndSet* usadas para substituição final

Objetos Atômicos – Exemplo

```
public boolean insert(int v)
                    throws TMException {
  List newList = new List(v);
  TMObject newNode = new TMObject(newList);
  TMThread thread =
           (TMThread)Thread.currentThread();
  while (true) {
    thread.beginTransaction();
    boolean result = true;
    try {
      List prevList =
        (List)this.first.open(WRITE);
      List currList =
        (List)prevList.next.open(WRITE);
      while (currList.value < v) {
        prevList = currList;
        currList =
          (List)currList.next.open(WRITE);
      }
      if (currList.value == v) {
        result = false;
      } else {
        result = true;
        newList.next = prevList.next;
        prevList.next = newNode;
      }
    } catch (Denied d){}
    if (thread.commitTransaction())
      return result;
  }
}
```

Memória Transacional – condições de espera

- retry: aborta transação e volta a tentar quando *algum dos valores lidos no trecho tiver sido alterado*

```
public void enq (T x) {
  atomic {
    if (count == items.length)
      retry;
    items [tail] = x;
    if (++tail == items.length)
      tail = 0;
    ++count;
  }
}
```

Memória Transacional – escolha

- construção *orelse*: espera por alguma condição

```
atomic {
  x = q0.deq(); /* se chamar retry tenta o outro */
} orElse {
  x = q1.deq();
}
```

Memória transacional – Implementações

- Haskell
- linguagem *Fortress* proposta pela Sun
- biblioteca .Net

  Back in January Joe Duffy, Microsoft's best known researcher on parallel and concurrent programming, cited four reasons why he becomes disillusioned with STM in his Brief Retrospective on Transactional Memory.
  (http://www.infoq.com/news/2010/05/STM-Dropped)

- C++11

Memória Transacional – Dificuldades

- implementações por hardware: não adotadas
- implementações por software:
  - custo alto
  - retry: problemas semelhantes aos dos monitores com testes implícitos:
    - quando tentar de novo?
  - dificuldades com entrada e saída
- API é apropriada?
  - o mundo é transacional...?
  - capacidade de rollback não necessariamente ligada a concorrência
  - ações irreversíveis como comunicação com outra thread ou E/S
  - rollback em caso de falhas: como detectar motivos da falha?

- especificação de transações (*Draft Specification of Transactional Memory Constructs for C++*)
- construções têm comportamento bem definido *apenas para programas sem condições de corrida*.
- novas palavras chave `__transaction_atomic`, `__transaction_relaxed` e `__transaction_cancel`
  - também `__transaction_safe` e `__transaction_unsafe` (atributos de funções)

C++11 – atomic transactions

__transaction_atomic

- isolamento rígido
- atomicidade: ou tem efeito por completo ou não tem efeito algum
- diversas restrições sobre código protegido

```
__transaction_atomic {
    stmt1
     __transaction_cancel;
  }
  stmt2
```

C++11 – relaxed transactions

__transaction_relaxed

- executa sem observar alterações realizadas por outras transações durante sua execução
- podem executar ações *irrevogáveis*

```
void sched_wait (void) {
  int no_active_lp = FALSE;
  while (no active lp == FALSE) {
    __transaction_relaxed {if (lpcount == 0) {
      no_active_lp = TRUE;
    }
  }
}
```

Exemplo em C++11

```
int add(int value)
{
  node_t *prev, *next;

  __transaction_atomic {
    prev = set->head; next = prev->next;
    while (next->val < val) {
      prev = next;
      next = prev->next;
    }
    if (next->val != val) prev->next = new_node(val, next);
  }
  return result;
}

static __attribute__((transaction_safe))
node_t *new_node(val_t val, node_t *next)
{
  (...)
}
```

C++11 – Modelo de Memória

- 2 operações estão em conflito se uma delas modifica uma posição de memória e a outra acessa ou modifica a mesma posição
- a execução de um programa contém uma *condição de corrida* se contém operações conflitantes em threads distintos, pelo menos uma das quais não é uma operação atômica, e nenhuma *acontece antes* da outra

data-race free

Um programa é dito *livre de condições de corrida* se nenhuma de suas execuções contém uma condição de corrida

Referências

- M. Herlihy, N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
- Maurice Herlihy, Victor Luchango, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. PODC 2003: 92-101, Jul 2003.
- Ali-Reza Adl-Tabatabai and others (eds). Draft Specification of Transactional Language Constructs for C++. 02/2012.
- Hans-J. Boehm and Sarita V. Adve. 2012. You don't know jack about shared variables or memory models. Commun. ACM 55, 2 (February 2012), 48-54. http://doi.acm.org/10.1145/2076450.2076465.