

Capítulo 6: Sincronismo de processos



Capítulo 6: Sincronismo de processo

- ❑ Fundamentos
- ❑ O problema da seção crítica
- ❑ Solução de Peterson
- ❑ Hardware de sincronismo
- ❑ Semáforos
- ❑ Problemas clássicos de sincronismo
- ❑ Monitores
- ❑ Exemplos de sincronismo
- ❑ Transações indivisíveis



Fundamentos

- ❑ Acesso concorrente aos dados compartilhados pode resultar em inconsistência de dados
- ❑ Manutenção da consistência dos dados requer mecanismos para garantir a execução ordenada dos processos em cooperação
- ❑ Suponha que queiramos oferecer uma solução para o problema de consumidor-produtor com buffer compartilhado. Podemos fazer isso com um **contador** inteiro que indica o número de espaços ocupados no buffer (inicialmente, contador = 0).



Produtor

```
while (count == BUFFER_SIZE)
    ; // do nothing

// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
```



Consumidor

```
while (count == 0)
    ; // do nothing

// remove an item from the buffer
--count;
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
```



Condição de race

- ❑ `count++` poderia ser implementado como

```
register1 = count
register1 = register1 + 1
count = register1
```
- ❑ `count--` poderia ser implementado como

```
register2 = count
register2 = register2 - 1
count = register2
```
- ❑ Considere esta execução intercalando com “count = 5” inicialmente:
 - S0: produtor executa `register1 = count` {register1 = 5}
 - S1: produtor executa `register1 = register1 + 1` {register1 = 6}
 - S2: consumidor executa `register2 = count` {register2 = 5}
 - S3: consumidor executa `register2 = register2 - 1` {register2 = 4}
 - S4: produtor executa `count = register1` {count = 6}
 - S5: consumidor executa `count = register2` {count = 4}



Solução do problema de seção crítica

1. **Exclusão mútua** – Se o processo P_i estiver executando em sua seção crítica, então nenhum outro processo poderá estar executando em suas seções críticas.
2. **Progresso** – Se nenhum processo estiver executando em sua seção crítica e houver alguns processos que queiram entrar em sua seção crítica, então a seleção dos processos que entrarão na seção crítica em seguida não poderá ser adiada indefinidamente.
3. **Espera limitada** - É preciso haver um limite sobre o 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 essa solicitação seja concedida



Problema de seção crítica

Condição de race – Quando há acesso simultâneo aos dados compartilhados e o resultado final depende da ordem de execução.

Seção crítica – Seção do código onde os dados compartilhados são acessados.

Seção de entrada - Código que solicita permissão para entrar em sua seção crítica.

Seção de saída – Código executado após a saída da seção crítica.



Estrutura de um processo típico

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



Solução de Peterson

- ❑ Solução em dois processos
- ❑ Os dois processos compartilham duas variáveis:
 - int **turn**;
 - Boolean **flag[2]**
- ❑ A variável **turn** indica de quem é a vez de entrar na seção crítica.
- ❑ O array **flag** é usado para indicar se um processo está pronto para entrar na seção crítica. **flag[i] = true** implica que o processo P_i está pronto!



Algoritmo para o processo P_i

```
while (true) {
```

```
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);
```

critical section

```
    flag[i] = FALSE;
```

remainder section

```
}
```



Seção crítica usando locks

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



Hardware de sincronismo

- ❑ Muitos sistemas oferecem suporte do hardware para o código da seção crítica
- ❑ Uniprocessadores – poderiam desativar interrupções
 - Código atualmente em execução poderia ser executado sem preempção
 - Geralmente muito ineficaz em sistemas multiprocessados
- ❑ Máquinas modernas oferecem instruções de hardware especiais indivisíveis
 - ❑ Indivisível = não interrompida



Estrutura de dados para soluções de hardware

```
public class HardwareData
{
    private boolean value = false;

    public HardwareData(boolean value) {
        this.value = value;
    }

    public boolean get() {
        return value;
    }

    public void set(boolean newValue) {
        value = newValue;
    }

    public boolean getAndSet(boolean newValue) {
        boolean oldValue = this.get();
        this.set(newValue);

        return oldValue;
    }

    public void swap(HardwareData other) {
        boolean temp = this.get();

        this.set(other.get());
        other.set(temp);
    }
}
```



Solução usando GetAndSet

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

while (true) {
    while (lock.getAndSet(true))
        Thread.yield();

    criticalSection();
    lock.set(false);
    remainderSection();
}
```



Solução usando Swap

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

// each thread has a local copy of key
HardwareData key = new HardwareData(true);

while (true) {
    key.set(true);

    do {
        lock.swap(key);
    }
    while (key.get() == true);

    criticalSection();
    lock.set(false);
    remainderSection();
}
```



Semáforo

- ❑ Ferramenta de sincronismo que não exige espera ocupada
- ❑ Semáforo S – variável inteira
- ❑ Duas operações padrão modificam S : `acquire()` e `release()`
- ❑ Só pode ser acessado por duas operações indivisíveis

```
acquire() {  
    while value <= 0  
        ; // no-op  
    value--;  
}  
  
release() {  
    value++;  
}
```



Semáforo como ferramenta geral de sincronismo

- ❑ **Semáforo de contagem** – valor inteiro pode variar por um domínio irrestrito
- ❑ **Semáforo binário** – valor inteiro só pode variar entre 0 e 1; pode ser mais simples de implementar
 - Também conhecidos como **locks mutex**

```
Semaphore S = new Semaphore();  
  
S.acquire();  
  
    // critical section  
  
S.release();  
  
    // remainder section
```



Exemplo Java usando semáforos

```
public class Worker implements Runnable
{
    private Semaphore sem;
    private String name;

    public Worker(Semaphore sem, String name) {
        this.sem = sem;
        this.name = name;
    }

    public void run() {
        while (true) {
            sem.acquire();
            MutualExclusionUtilities.criticalSection(name);
            sem.release();
            MutualExclusionUtilities.remainderSection(name);
        }
    }
}
```



Exemplo Java usando semáforos

```
public class SemaphoreFactory
{
    public static void main(String args[]) {
        Semaphore sem = new Semaphore(1);
        Thread[] bees = new Thread[5];

        for (int i = 0; i < 5; i++)
            bees[i] = new Thread(new Worker
                (sem, "Worker " + (new Integer(i)).toString() ));

        for (int i = 0; i < 5; i++)
            bees[i].start();
    }
}
```



Implementação de semáforo

- ❑ Precisa garantir que dois processos não poderão executar `acquire ()` e `release ()` no mesmo semáforo ao mesmo tempo



Implementação de semáforo

- ❑ Com cada semáforo existe uma fila de espera associada (elimina espera ocupada). Cada entrada em uma fila de espera possui dois itens de dados:
 - valor (do tipo int)
 - ponteiro para próximo registro na lista
- ❑ Duas operações:
 - **block** – coloca o processo que chama a operação na fila de espera apropriada.
 - **wakeup** – remove um dos processos na fila de espera e o coloca na fila de prontos.



Implementação de semáforo sem espera ocupada (cont.)

□ Implementação de **acquire()**:

```
acquire(){  
    value--;  
    if (value < 0) {  
        add this process to list  
        block;  
    }  
}
```

□ Implementação de **release()**:

```
release(){  
    value++;  
    if (value <= 0) {  
        remove a process P from list  
        wakeup(P);  
    }  
}
```



Deadlock e starvation

- ❑ **Deadlock** – dois ou mais processos estão esperando indefinidamente por um evento que pode ser causado somente por um dos processos esperando
- ❑ Exemplo: sejam **S** e **Q** dois semáforos inicializados com 1

P_0	P_1
S.acquire();	Q.acquire();
Q.acquire();	S.acquire();
.	.
.	.
.	.
S.release();	Q.release();
Q.release();	S.release();

- ❑ **Starvation** – lock indefinido. Um processo pode nunca ser removido da fila de semáforo em que ele é suspenso.



Problemas clássicos de sincronismo

- ❑ Problema de buffer limitado
- ❑ Problema de leitores-escretores
- ❑ Problema do jantar dos filósofos



Problema de buffer limitado

- ❑ Buffer com capacidade para N itens
- ❑ **Mutex** inicializado com o valor 1
- ❑ Semáforo **cheio** inicializado com o valor 0
- ❑ Semáforo **vazio** inicializado com o valor N.



Problema de buffer limitado

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;
    private Object[] buffer;
    private int in, out;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;

    public BoundedBuffer() {
        // buffer is initially empty
        in = 0;
        out = 0;
        buffer = new Object[BUFFER_SIZE];

        mutex = new Semaphore(1);
        empty = new Semaphore(BUFFER_SIZE);
        full = new Semaphore(0);
    }

    public void insert(Object item) {
        // Figure 6.9
    }

    public Object remove() {
        // Figure 6.10
    }
}
```



Problema de buffer limitado

```
public void insert(Object item) {  
    empty.acquire();  
    mutex.acquire();  
  
    // add an item to the buffer  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
  
    mutex.release();  
    full.release();  
}
```



Problema de buffer limitado

```
public Object remove() {  
    full.acquire();  
    mutex.acquire();  
  
    // remove an item from the buffer  
    Object item = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
  
    mutex.release();  
    empty.release();  
  
    return item;  
}
```

--



Problema de buffer limitado (cont.)

□ A estrutura do processo produtor

```
public class Producer implements Runnable
{
    private Buffer buffer;

    public Producer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
        Date message;

        while (true) {
            // nap for awhile
            SleepUtilities.nap();
            // produce an item & enter it into the buffer
            message = new Date();
            buffer.insert(message);
        }
    }
}
```



Problema de buffer limitado (cont.)

□ A estrutura do processo consumidor

```
public class Consumer implements Runnable
{
    private Buffer buffer;

    public Consumer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
        Date message;

        while (true) {
            // nap for awhile
            SleepUtilities.nap();
            // consume an item from the buffer
            message = (Date)buffer.remove();
        }
    }
}
```



Problema de buffer limitado (cont.)

```
public class Factory
{
    public static void main(String args[]) {
        Buffer buffer = new BoundedBuffer();

        // now create the producer and consumer threads
        Thread producer = new Thread(new Producer(buffer));
        Thread consumer = new Thread(new Consumer(buffer));

        producer.start();
        consumer.start();
    }
}
```



Problema de leitores-escritores

- Um conjunto de dados é compartilhado entre diversos processos concorrentes
 - Leitores – só lêem o conjunto de dados; eles não realizam quaisquer atualizações
 - Escritores – podem ler e escrever.
- Problema – permite que vários leitores leiam ao mesmo tempo. Apenas um único escritor pode acessar os dados compartilhados de uma só vez.
- Dados compartilhados
 - Conjunto de dados
 - **Mutex** inicializado com 1.
 - Semáforo **db** inicializado com 1.
 - Inteiro **readerCount** inicializado com 0.
 - Solução apresentada pode dar starvation!



Problema de leitores-escritores

Interface para locks de leitura-escrita

```
public interface RWLock
{
    public abstract void acquireReadLock();
    public abstract void acquireWriteLock();
    public abstract void releaseReadLock();
    public abstract void releaseWriteLock();
}
```



Problema de leitores-escritores

```
public class DataBase implements RWLock {  
    private int readerCount;  
    private Semaphore mutex;  
    private Semaphore db;  
  
    public Database() {  
        readerCount = 0;  
        mutex = new Semaphore(1);  
        db = new Semaphore(1);  
    }  
  
    //métodos acquire and release read/write lock  
}
```



Problema de leitores-escritores

Métodos chamados por escritores.

```
public void acquireWriteLock() {  
    db.acquire();  
}  
  
public void releaseWriteLock() {  
    db.release();  
}
```



Problema de leitores-escritores

Métodos chamados por leitores.

```
public void acquireReadLock() {  
    mutex.acquire();  
    readerCount++;  
  
    if(readerCount == 1)  
        db.acquire();  
  
    mutex.release();  
}
```

```
public void releaseReadLock() {  
    mutex.acquire();  
    readerCount--;  
  
    if(readerCount == 0)  
        db.release();  
  
    mutex.release();  
}
```



Problema de leitores-escritores

□ A estrutura de um processo escritor

```
public class Writer implements Runnable
{
    private RWLock db;

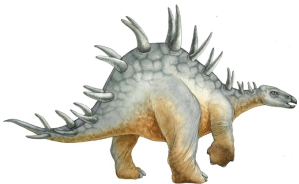
    public Writer(RWLock db) {
        this.db = db;
    }

    public void run() {
        while (true) {
            // nap for awhile
            SleepUtilities.nap();

            db.acquireWriteLock();

            // you have access to write to the database
            SleepUtilities.nap();

            db.releaseWriteLock();
        }
    }
}
```



Problema de leitores-escritores

□ A estrutura de um processo leitor

```
public class Reader implements Runnable
{
    private RWLock db;

    public Reader(RWLock db) {
        this.db = db;
    }

    public void run() {
        while (true) {
            // nap for awhile
            SleepUtilities.nap();

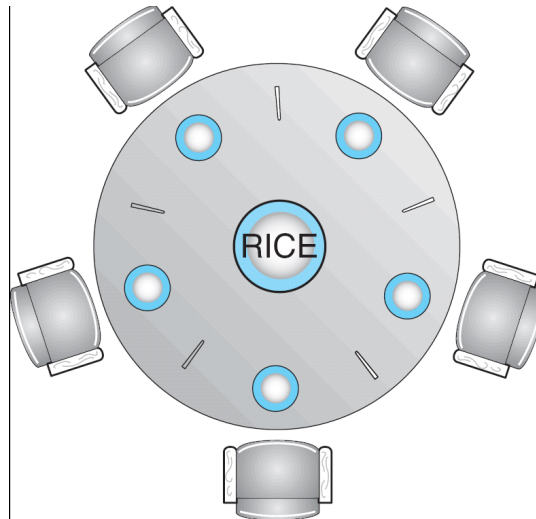
            db.acquireReadLock();

            // you have access to read from the database
            SleepUtilities.nap();

            db.releaseReadLock();
        }
    }
}
```



Problema do jantar dos filósofos



- Dados compartilhados
 - Tigela de arroz (conjunto de dados)
 - Semáforo **chopStick [5]** inicializado com 1
 - Solução apresentada pode dar deadlock!



Problema do jantar dos filósofos (cont.)

- A estrutura do Filósofo *i*:

```
while (true) {  
    // get left chopstick  
    chopStick[i].acquire();  
    // get right chopstick  
    chopStick[(i + 1) % 5].acquire();  
  
    eating();  
  
    // return left chopstick  
    chopStick[i].release();  
    // return right chopstick  
    chopStick[(i + 1) % 5].release();  
  
    thinking();  
}
```



Problemas com semáforos

- Uso correto das operações de semáforo:
 - `mutex.acquire()` `mutex.release()`
 - Omissão de `mutex.release()` pode gerar problema



Monitores

- Uma abstração de alto nível que oferece um mecanismo conveniente e eficaz para o sincronismo de processo
- Somente um processo pode estar ativo dentro do monitor em determinado momento



Sintaxe de um monitor

```
monitor monitor name
{
    // shared variable declarations

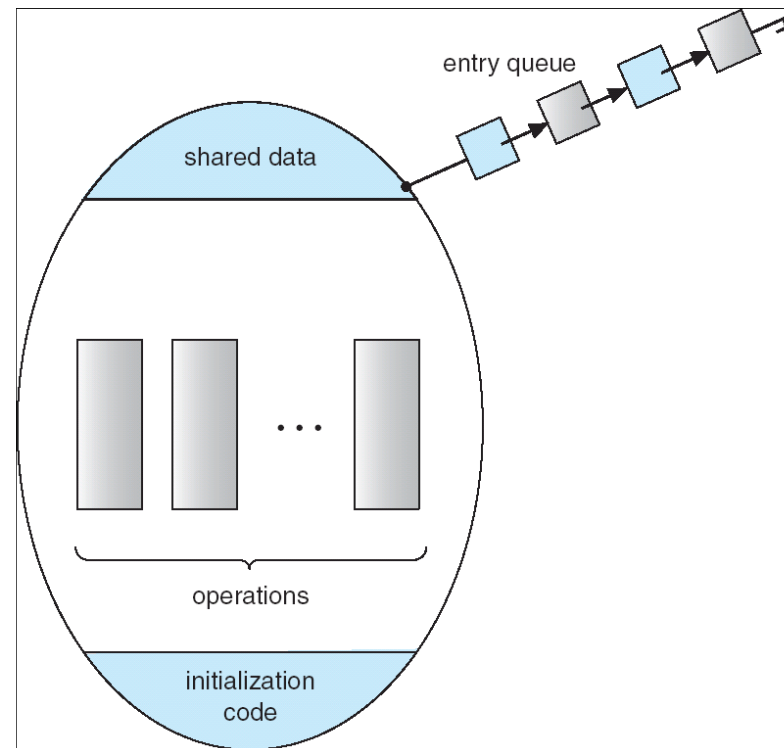
    initialization code ( . . . ) {
        . . .
    }

    public P1 ( . . . ) {
        . . .
    }

    public P2 ( . . . ) {
        . . .
    }
    .
    .
    .
    public Pn ( . . . ) {
        . . .
    }
}
```



Visão esquemática de um monitor

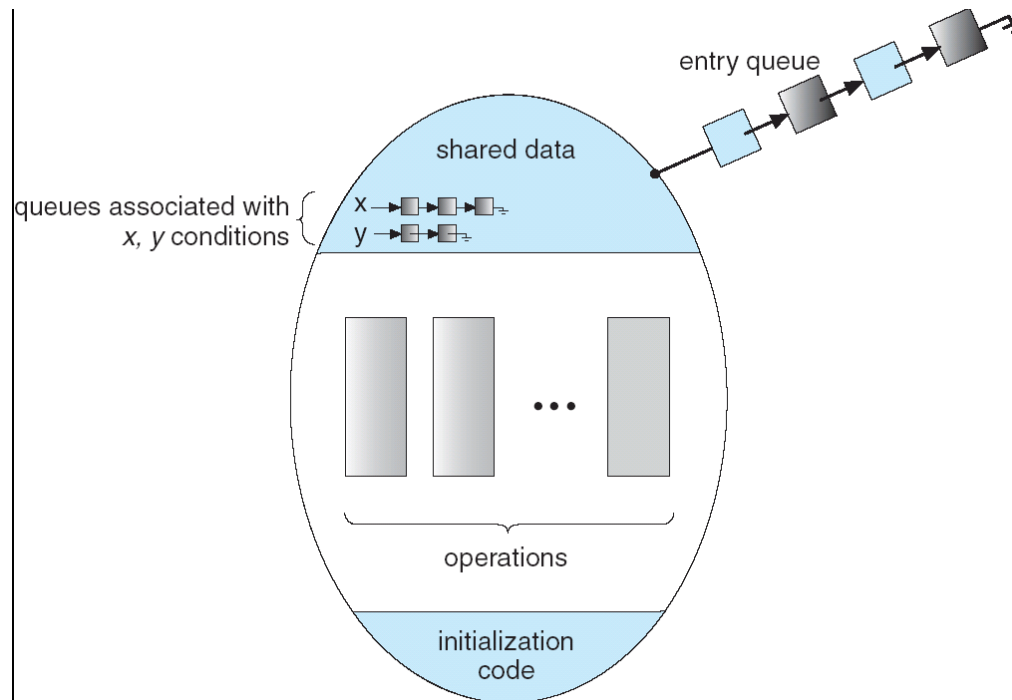


Variáveis de condição

- Condição x, y ;
- Duas operações em uma variável de condição:
 - $x.wait()$ – o processo que invoca a operação é suspenso.
 - $x.signal()$ – retoma um dos processos (se houver) que invocou $x.wait()$



Monitor com variáveis de condição



Solução do jantar dos filósofos

```
monitor DiningPhilosophers
{
    enum State {THINKING, HUNGRY, EATING};
    State[] states = new State[5];
    Condition[] self = new Condition[5];

    public DiningPhilosophers {
        for (int i = 0; i < 5; i++)
            state[i] = State.THINKING;
    }

    public void takeForks(int i) {
        state[i] = State.HUNGRY;
        test(i);
        if (state[i] != State.EATING)
            self[i].wait;
    }

    public void returnForks(int i) {
        state[i] = State.THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    private void test(int i) {
        if ( (state[(i + 4) % 5] != State.EATING) &&
            (state[i] == State.HUNGRY) &&
            (state[(i + 1) % 5] != State.EATING) ) {
            state[i] = State.EATING;
            self[i].signal;
        }
    }
}
```



Solução do jantar dos filósofos (cont.)

- Cada filósofo i invoca as operações `takeForks(i)` e `returnForks(i)` na sequência a seguir:

`dp.takeForks (i)`

COMER

`dp.returnForks (i)`



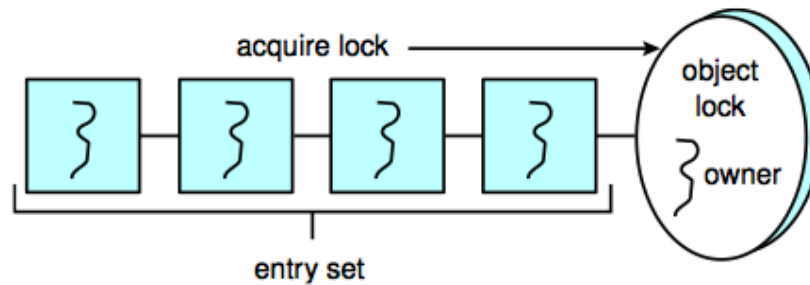
Sincronismo em Java

- ❑ Java oferece sincronismo em nível de linguagem.
- ❑ Cada objeto Java possui um lock associado.
- ❑ Esse lock é adquirido pela invocação de um método (ou bloco) **sincronizado**.
- ❑ Esse lock é liberado na saída do método (ou bloco) sincronizado.
- ❑ Os threads esperando para adquirir o lock do objeto são colocados no **conjunto de entrada** para o lock do objeto.



Sincronismo em Java

Cada objeto possui um conjunto de entrada associado.



Cada cobrinha é uma thread :)



Sincronismo em Java

Métodos sincronizados insert() e remove()

```
public synchronized void insert(Object item) {  
    while (count == BUFFER_SIZE)  
        Thread.yield();  
  
    ++count;  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
}
```

```
public synchronized Object remove() {  
    Object item;  
  
    while (count == 0)  
        Thread.yield();  
  
    --count;  
    item = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
  
    return item;  
}
```



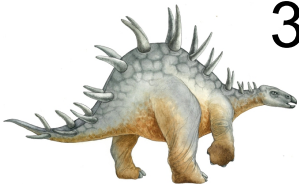
Sincronismo em Java: wait/notify()

- Quando uma thread invoca **wait()**:

1. A thread libera o lock do objeto;
2. O estado da thread é definido como Blocked;
3. A thread é colocada na fila de espera associada ao objeto.

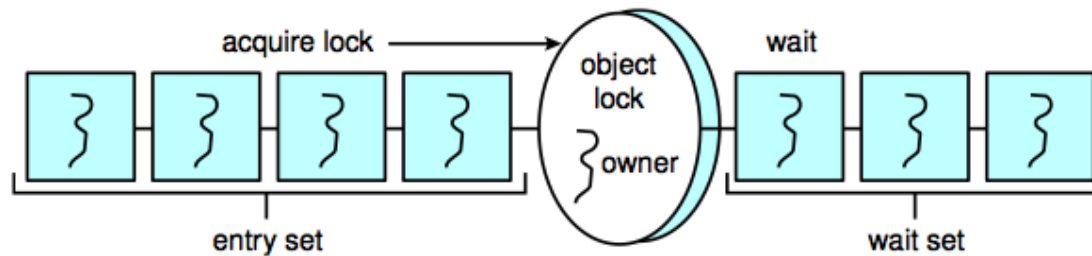
- Quando uma thread invoca **notify()**:

1. Uma thread qualquer T do conjunto de espera é selecionada;
2. T é movida da espera para o conjunto de entrada;
3. O estado de T é definido como Runnable.



Sincronismo em Java

Entrada e conjuntos de espera



Sincronismo em Java – buffer limitado

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;

    private int count, in, out;
    private Object[] buffer;

    public BoundedBuffer() {
        // buffer is initially empty
        count = 0;
        in = 0;
        out = 0;
        buffer = new Object[BUFFER_SIZE];
    }

    public synchronized void insert(Object item) {
        // Figure 6.28
    }

    public synchronized Object remove() {
        // Figure 6.28
    }
}
```



Sincronismo em Java - buffer limitado

```
public synchronized void insert(Object item) {
    while (count == BUFFER_SIZE) {
        try {
            wait();
        }
        catch (InterruptedException e) { }
    }

    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    notify();
}

public synchronized Object remove() {
    Object item;

    while (count == 0) {
        try {
            wait();
        }
        catch (InterruptedException e) { }
    }

    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

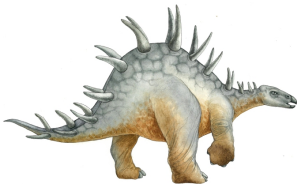
    notify();

    return item;
}
```



Sincronismo em Java

- ❑ A chamada a **notify()** seleciona uma thread qualquer do conjunto de espera. É possível que a thread selecionada não esteja de fato esperando pela condição para a qual foi notificada.
- ❑ A chamada **notifyAll()** seleciona todas as threads no conjunto de espera e as move para o conjunto de entrada.



Sincronismo em Java – leitores-escritores

```
public class Database implements RWLock
{
    private int readerCount;
    private boolean dbWriting;

    public Database() {
        readerCount = 0;
        dbWriting = false;
    }

    public synchronized void acquireReadLock() {
        // Figure 6.33
    }

    public synchronized void releaseReadLock() {
        // Figure 6.33
    }

    public synchronized void acquireWriteLock() {
        // Figure 6.34
    }

    public synchronized void releaseWriteLock() {
        // Figure 6.34
    }
}
```



Sincronismo em Java – leitores-escritores

Métodos chamados pelos leitores

```
public synchronized void acquireReadLock() {  
    while (dbWriting == true) {  
        try {  
            wait();  
        }  
        catch (InterruptedException e) { }  
    }  
  
    ++readerCount;  
}  
  
public synchronized int releaseReadLock() {  
    --readerCount;  
  
    // if I am the last reader tell writers  
    // that the database is no longer being read  
    if (readerCount == 0)  
        notify();  
}
```



Sincronismo em Java – leitores-escritores

Métodos chamados pelos escritores

```
public synchronized void acquireWriteLock() {  
    while (readerCount > 0 || dbWriting == true) {  
        try {  
            wait();  
        }  
        catch (InterruptedException e) { }  
    }  
  
    // once there are either no readers or writers  
    // indicate that the database is being written  
    dbWriting = true;  
}  
  
public synchronized void releaseWriteLock() {  
    dbWriting = false;  
  
    notifyAll();  
}
```



Sincronismo em Java

Ao invés de sincronizar um método inteiro, blocos de código podem ser declarados como sincronizados

```
Object mutexLock = new Object();  
.  
.  
.  
public void someMethod() {  
    nonCriticalSection();  
  
    synchronized(mutexLock) {  
        criticalSection();  
    }  
  
    remainderSection();  
}
```



Sincronismo em Java

Sincronismo de bloco usando wait()/notify()

```
Object mutexLock = new Object();  
.  
.  
.  
synchronized(mutexLock) {  
    try {  
        mutexLock.wait();  
    }  
    catch (InterruptedException ie) { }  
}  
  
synchronized(mutexLock) {  
    mutexLock.notify();  
}
```



Transações indivisíveis

- ❑ Modelo do sistema
- ❑ Recuperação baseada em log
- ❑ Pontos de verificação
- ❑ Transações indivisíveis simultâneas



Modelo do sistema

- ❑ Garante que as operações acontecem como uma única unidade lógica de trabalho, em sua inteireza, ou não acontecem
- ❑ Relacionado ao campo de sistemas de banco de dados
- ❑ Desafio é assegurar a indivisibilidade apesar das falhas do sistema de computação
- ❑ **Transação** – coleção de instruções ou operações que realiza uma única função lógica
 - Aqui, nos preocupamos com mudanças no armazenamento estável – disco
 - A transação é uma série de operações **read** e **write**
 - Terminado com operação **commit** (transação bem sucedida) ou **abort** (transação falhou)
 - Transação abortada deve ser **rolled back** para desfazer quaisquer mudanças feitas por ela



Tipos de meio de armazenamento

- Armazenamento volátil – informação armazenada aqui não sobrevive a falhas do sistema
 - Exemplo: memória principal, cache
- Armazenamento não volátil – A informação normalmente sobrevive a falhas
 - Exemplo: disco e fita
- Armazenamento estável – Informação nunca se perde
 - Não é realmente possível, e por isso é aproximado pela replicação ou RAID para dispositivos com modos de falha independentes

Objetivo é garantir indivisibilidade da transação onde as falhas causam perda de informações no armazenamento volátil



Recuperação baseada em log

- ❑ Registro de informações em armazenamento estável sobre todas as modificações feitas por uma transação
- ❑ Mais comum é o **logging write-ahead**
 - Log em armazenamento estável, cada registro de log descreve operação de escrita de única transação, incluindo
 - ❑ Nome da transação
 - ❑ Nome do item de dados
 - ❑ Valor antigo
 - ❑ Valor novo
 - $\langle T_i \text{ inicia} \rangle$ escrito no log quando a transação T_i inicia
 - $\langle T_i \text{ confirma} \rangle$ escrito quando T_i confirma
- ❑ Entrada de log precisa alcançar o armazenamento estável antes que a operação ocorra



Algoritmo de recuperação baseado em log

- Usando o log, o sistema pode tratar de quaisquer erros de memória volátil
 - **Undo(T_i)** restaura o valor de todos os dados atualizados por T_i
 - **Redo(T_i)** define valores de todos os dados na transação T_i para novos valores
- Undo(T_i) e redo(T_i) devem ser **idempotentes**
 - Múltiplas execuções devem ter o mesmo resultado que uma execução
- Se o sistema falhar, restaura estado de todos os dados atualizados em log
 - Se log contém $\langle T_i \text{ starts} \rangle$ sem $\langle T_i \text{ commits} \rangle$, **undo(T_i)**
 - Se log contém $\langle T_i \text{ starts} \rangle$ e $\langle T_i \text{ commits} \rangle$, **redo(T_i)**



Pontos de verificação

- ❑ Log poderia se tornar longo, e recuperação poderia levar muito tempo
- ❑ Pontos de verificação encurtam log e tempo de recuperação.
- ❑ Esquema de ponto de verificação:
 1. Enviar todos os registradores de log atualmente no armazenamento volátil para o armazenamento estável
 2. Enviar todos os dados modificados do armazenamento volátil para o estável
 3. Enviar um registro de log <checkpoint> para o log no armazenamento estável
- ❑ Agora a recuperação só inclui Ti, tal que Ti iniciou a execução antes do ponto de verificação mais recente, e todas as transações após Ti. Todas as outras transações já estão em armazenamento estável



Transações concorrentes

- ❑ Devem ser equivalentes à execução serial -
serialização
- ❑ Poderiam executar todas as transações na seção crítica
 - Ineficiente, muito restritivo
- ❑ **Algoritmos de controle de concorrência** oferecem serialização



serialização

- ❑ Considere dois itens de dados A e B
- ❑ Considere as transações T_0 e T_1
- ❑ Execute T_0 , T_1 de forma indivisível
- ❑ Seqüência de execução chamada **schedule**
- ❑ Ordem de transação executada de forma indivisível chamada **schedule serial**
- ❑ Para N transações, existem $N!$ schedules seriais válidos



Schedule 1: T_0 depois T_1

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)



Schedule não serial

- **Schedule não serial** permite execução sobreposta
 - Execução resultante não necessariamente incorreta
- Considere schedule S , operações O_i, O_j
 - **Conflito** se acessar mesmo item de dados, com pelo menos uma escrita
- Se O_i, O_j consecutivos e operações de diferentes transações & O_i e O_j não em conflito
 - Então S' com ordem trocada $O_j O_i$ equivalente a S
- Se S puder se tornar S' via swapping de operações não em conflito
 - S é **seriável em conflito**



Schedule 2: Schedule seriável simultâneo

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)



Protocolo de lock

- Garante serialização pela associação de lock a cada item de dados
 - Segue protocolo de lock para controle de acesso
- locks
 - **Compartilhado** – T_i tem lock de modo compartilhado (S) no item Q, T_i pode ler Q mas não escrever Q
 - **Exclusivo** – T_i tem lock em modo exclusivo (X) em Q, T_i pode ler e escrever Q
- Exige que cada transação no item Q adquira lock apropriado
- Se lock já for mantido, nova solicitação pode ter que esperar
 - Semelhante ao algoritmo de leitores-escretores



Protocolo de lock em duas fases

- ❑ Geralmente, garante serialização em conflito
- ❑ Cada transação emite solicitações de lock e unlock em duas fases
 - Crescimento – obtendo locks
 - Encolhimento – liberando locks
- ❑ Não impede deadlock



Protocolos baseados em estampa de tempo

- ❑ Selecione a ordem entre transações com antecedência – ordenação por estampa de tempo
- ❑ Transação T_i associada à estampa de tempo $TS(T_i)$ antes que T_j inicie
 - $TS(T_i) < TS(T_j)$ se T_i entrou no sistema antes de T_j
 - TS pode ser gerado pelo clock do sistema ou como contador lógico incrementado em cada entrada de transação
- ❑ Estampas de tempo determinam a ordem de serialização
 - Se $TS(T_i) < TS(T_j)$, sistema deve garantir schedule produzido equivalente a schedule serial, onde T_i aparece antes de T_j



Implementação do protocolo baseado em estampa de tempo

- Item de dados Q recebe duas estampas de tempo
 - estampa-W(Q) – maior estampa de tempo de qualquer transação que executou write(Q) com sucesso
 - estampa-R(Q) – maior estampa de tempo de read(Q) bem sucedido
 - Atualizado sempre que read(Q) ou write(Q) é executado
- Protocolo de ordenação por estampa de tempo garante que qualquer read e write em conflito sejam executados na ordem da estampa de tempo
- Suponha que T_i execute read(Q)
 - Se $TS(T_i) < \text{estampa-W}(Q)$, T_i precisa ler valor de Q que já foi modificado
 - operação read rejeitada e T_i cancelada
 - Se $TS(T_i) \geq \text{estampa-W}(Q)$
 - read executado, estampa-R(Q) definido como $\max(\text{estampa-R}(Q), TS(T_i))$



Protocolo de ordenação de estampa de tempo

- Suponha que T_i execute $\text{write}(Q)$
 - If $\text{TS}(T_i) < \text{estampa-R}(Q)$, valor Q produzido por T_i foi necessário anteriormente e T_i assumiu que nunca seria produzido
 - Operação **write** rejeitada, T_i cancelada
 - Se $\text{TS}(T_i) < \text{estampa-W}(Q)$, T_i tentando escrever valor obsoleto de Q
 - Operação **write** rejeitada e T_i cancelada
 - Caso contrário, **write** executado
- Qualquer transação cancelada T_i recebe nova estampa de tempo e é reiniciada
- Algoritmo garante serialização por conflito e ausência de deadlock



Schedule possível sob protocolo de estampa de tempo

T_2	T_3
read(B)	read(B) write(B)
read(A)	read(A) write(A)



Final do Capítulo 6

