Programação Concorrente e Paralela

Capítulo 8 – Propriedades de Concorrência

Marcial Porto Fernández
marcial@larces.uece.br

Semestre 2018.2

Sumário

- Propriedades de métodos concorrentes
  - Prioridade, justiça e Intercepção
- Leitores e Escritores
  - Políticas em Leitores e Escritores

Sumário

- Propriedades de métodos concorrentes
  - Prioridade, justiça e Intercepção
- Leitores e Escritores
  - Políticas em Leitores e Escritores

Métodos Concorrentes

- Objetivo:
  - Ao criar métodos paralelos que funcionem concorrentemente é possível criar mecanismos de implementação de políticas para habilitar e desabilitar os métodos com base no estado atual do sistema e dos métodos em execução.

Métodos Concorrentes

- Aplicabilidade
  - Objetos do host são acessados por vários threads diferentes.
  - Serviços do host não são totalmente interdependentes, por isso precisa ser realizado sob exclusão mútua.
  - Você precisa melhorar a eficiência de alguns métodos eliminando algum bloqueio não essencial.
  - Você quer evitar uma starvation acidental ou maliciosa de algum cliente que mantenha o bloqueio.
  - Evitar a sincronização completa fazendo objetos de diversos threads causem impasse ou problemas de progresso.

Métodos Concorrente - Projeto

- Criar uma camada com **política de controle** sobre o mecanismo de concorrência.
- Definição da Política:
  - Quando é que os métodos podem ser executados simultaneamente?
  - O que acontece quando um método desabilitado é invocado?
  - Que prioridade é atribuída às tarefas de espera?

Métodos Concorrente - Projeto

- Instrumentação:
  - Definir variáveis de estado para detectar e aplicar a política.
- Interceptação:
  - Fazer com que um objeto do host intercepte as mensagens, e em seguida, retransmiti-las para os métodos protegidos que realmente vão executar as ações.

Sumário

- Propriedades de métodos concorrentes
  - Prioridade, justiça e Intercepção
- Leitores e Escritores
  - Políticas em Leitores e Escritores

Prioridade (1)

- Prioridade pode depender de:
  - Atributos intrínsecos de tarefas (classe e instância de variáveis).
  - Representações da prioridade da tarefa, custo, preço, ou urgência.
  - O número de tarefas à espera de alguma condição.
  - O momento em que cada tarefa é adicionada a fila de execução.

Prioridade (2)

- Prioridade pode depender de:
  - Tempo estimado de espera ou tempo de execução para conclusão de cada tarefa.
  - O tempo de conclusão desejada de cada tarefa.
  - Dependências de terminação das tarefas.
  - O número de tarefas já concluídas.
  - **ATENÇÃO:** Tomar cuida com a imparcialidade.
    - Garantir que toda tarefa em espera, eventualmente vai ser executada.

Justiça ou Equidade (1)

- Há diferenças sutis entre as definições de justiça:
- Equidade fraca:
  - Se um processo continuamente faz uma solicitação, eventualmente ela será concedida.
    - Pode provocar starvation.
- Equidade forte:
  - Se um processo faz uma solicitação esporadicamente, eventualmente ela será concedida.

Justiça ou Equidade (2)

- Espera linear:
  - Se um processo faz uma solicitação, ela será concedida antes de qualquer outra solicitação seja concedida mais de uma vez.
- FIFO (First-In First-Out):
  - Se um processo faz uma solicitação, esta será concedida antes que de qualquer outro processo faça uma solicitação depois.

Intercepção (1)

- Estratégias de intercepção incluem:
  - Pass-through:
    - O host mantém um conjunto de referências para auxiliar objetos e simplesmente retransmite todas as mensagens para os métodos não sincronizados.
  - Lock-Spliting:
    - Usar diferentes bloqueios, um para cada funcionalidade da classe. Por exemplo, um bloqueio para leitura e outro bloqueio para a escrita.

Intercepção (2)

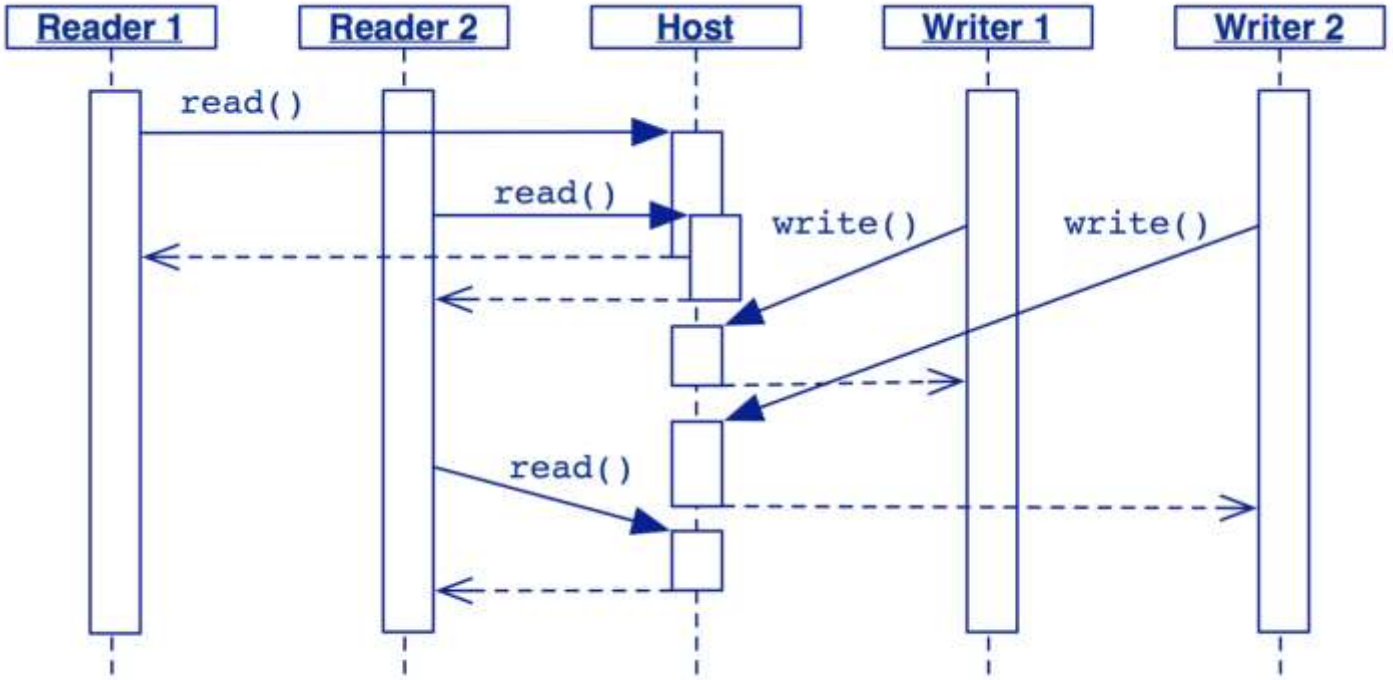
- Estratégias de intercepção incluem:
  - Métodos antes/depois :
    - Métodos públicos contêm chamadas à métodos não-públicos no host que executam os serviços antes e depois do processamento.
    - O método **antes** é chamado toda vez que vai executar a ação, e este método decide qual método interno será chamado.
    - O método **depois** atualiza o estado e libera os bloqueios.

Sumário

- Propriedades de métodos concorrentes
  - Prioridade, justiça e Intercepção
- Leitores e Escritores
  - Políticas em Leitores e Escritores

Leitores e Escritores concorrentes

- "Leitores e Escritores" é uma família de modelos de controle de concorrência em que "Leitores" (entes que não alteram conteúdo) pode acessar concorrentemente os recursos e "Escritores" (entes que alteram conteúdo) requerem acesso exclusivo.



Modelo Leitores/Escritores

- Só estamos interessados em capturar quem tem acesso:

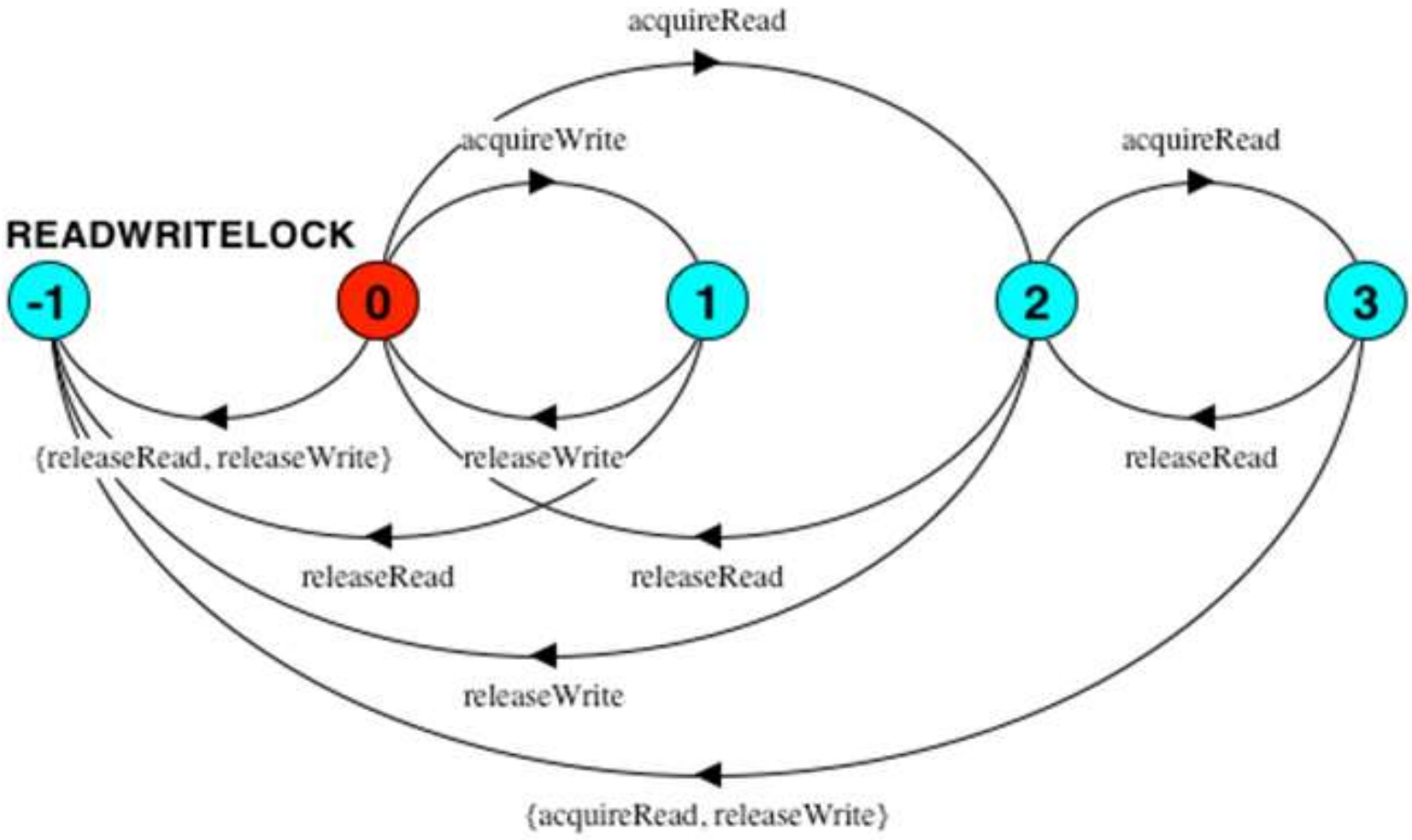
```
set Actions = {acquireRead, releaseRead, acquireWrite, releaseWrite}

READER      = ( acquireRead -> examine -> releaseRead -> READER)
                        +Actions \{examine}.

WRITER      = (acquireWrite -> modify -> releaseWrite ->WRITER).
```

Propriedades de segurança ...

```
||READWRITELOCK = (RW_LOCK || SAFE_RW).
```



Compondo Leitores e Escritores

- Compomos leitores e escritores com um protocolo e verificamos se há violações de segurança:

```
||READERS_WRITERS=
    (  reader[1..Nread]:READER
  ||  writer[1..Nwrite]:WRITER
  || {reader[1..Nread], writer[1..Nwrite]}::READWRITELOCK).
```

```
No deadlocks/errors
```

Propriedade Progresso

- Analogamente podemos especificar a propriedade progresso:

```
progress WRITE[i:1..Nwrite] = writer[i].acquireWrite
progress READ[i:1..Nwrite] = reader[i].acquireRead
```

- Assumindo escolha justa, não teremos problemas de progresso

```
Progress Check...
No progress violations detected.
```

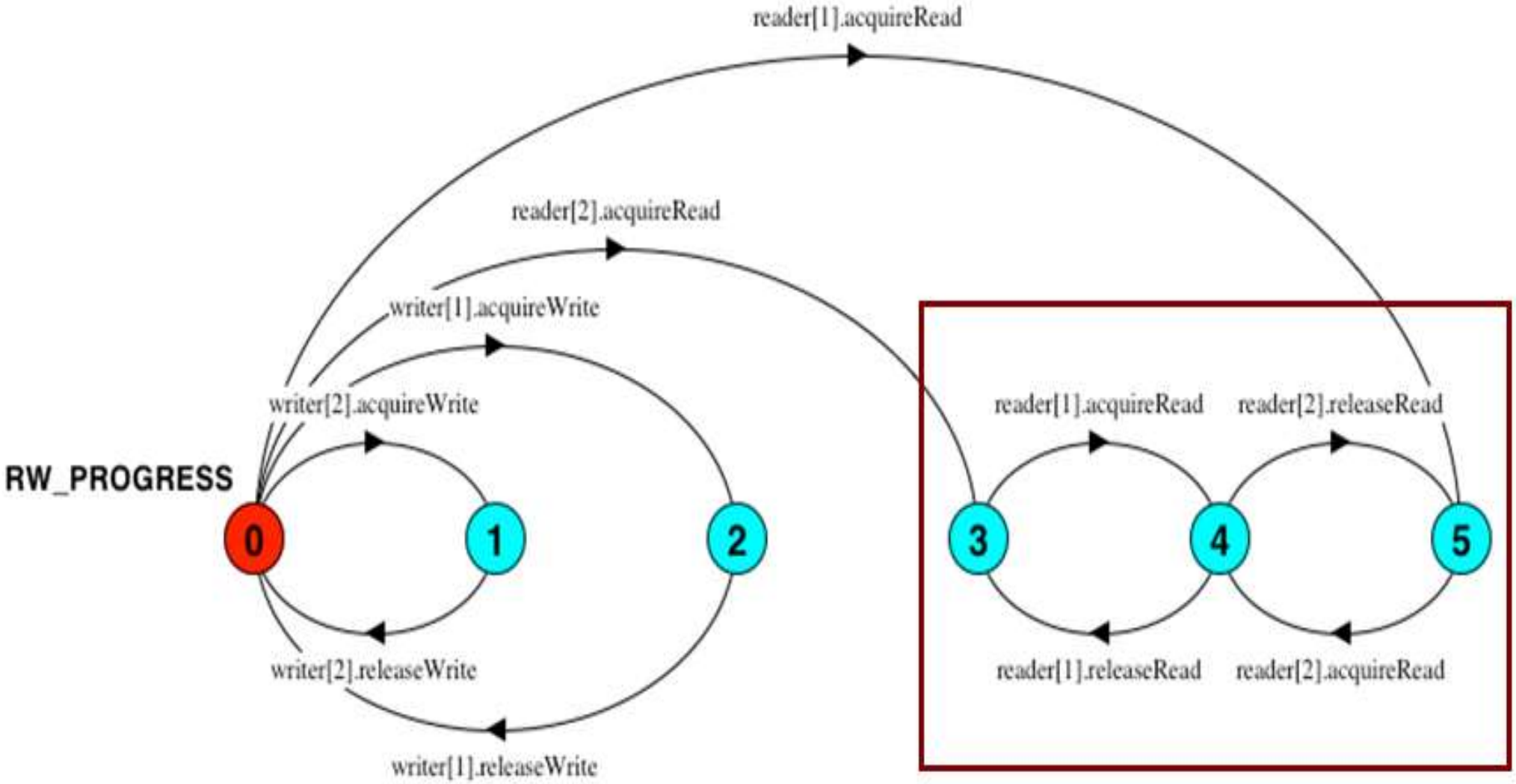
Prioridade

Se definirmos uma prioridade maior para leitores ou escritores podemos criar uma starvation!

```
||RW_PROGRESS =
      READERS_WRITERS
      >>{reader[1..Nread].releaseRead,
      writer[1..Nread].releaseWrite}.
```

```
Progress violation: WRITE.1 WRITE.2
Trace to terminal set of states:
     reader.1.acquireRead
Actions in terminal set:
     reader[1..2].{acquireRead, releaseRead}
```

Starvation


reader[1].acquireRead
reader[2].acquireRead
writer[1].acquireWrite
writer[2].acquireWrite
RW_PROGRESS
0
1
2
writer[2].releaseWrite
writer[1].releaseWrite
3
4
5
reader[1].acquireRead
reader[2].releaseRead
reader[1].releaseRead
reader[2].acquireRead

Sumário

- Propriedades de métodos concorrentes
  - Prioridade, justiça e Intercepção
- Leitores e Escritores
  - Políticas em Leitores e Escritores

Políticas Leitores e Escritores

- As políticas individuais devem abordar:

- Pode-se adicionar novos Leitores mesmo que um escritor esteja esperando?
  - Em caso afirmativo, os escritores podem morrer de fome
  - Se não, o throughput dos leitores diminui
- Se ambos leitores e escritores estão à espera de um escritor terminar, quem deve assumir em primeiro lugar?
  - Os leitores? Um escritor? Random? Alternado?
  - A mesma política quando algum dos leitores terminar?
- É possível que Leitores passem para Escritores sem tentar dar o acesso a outro?

Políticas ...

- Opções típicas:
  - Bloqueia entrada de Leitores se há Escritores em espera.
    - CUIDADO: Apenas se houver poucas escritas...
  - Escolher "aleatoriamente" entre os threads em entrada.
  - Não permitir ações consecutivas do mesmo thread.

Os métodos antes ou depois são a maneira mais simples de implementar políticas em Leitores e Escritores.

Leitores e Escritores: exemplo

- Implementação de variáveis de controle do estado

```
public abstract class ReadersWritersStateTracking {
     protected int activeReaders = 0;      // zero or more
     protected int activeWriters = 0;      // always zero or one
     protected int waitingReaders = 0;
     protected int waitingWriters = 0;
     protected abstract void doRead();     // defined by subclass
     protected abstract void doWrite();
...
```

Leitores e Escritores: exemplo

- Usa método antes/depois para proteger ação

```
...
      public void read() {          // unsynchronized
            beforeRead();           // obtain access
            doRead();
            afterRead();            // release access
      }
      public void write() {
            beforeWrite();
            doWrite();
            afterWrite();
      }
...
```

Leitores e Escritores: exemplo

- Sincronizar métodos antes/depois para manter estado de variáveis

```
protected synchronized void beforeRead() {
      ++waitingReaders;
      while (!allowReader()) {
            try { wait(); }
            catch (InterruptedException ex) {}
      }
      --waitingReaders;
      ++activeReaders;
}
protected synchronized void afterRead() {
      --activeReaders;
      notifyAll();
}
```

Leitores e Escritores: exemplo

- Implementa a política de acesso

```
...
protected boolean allowReader() {     // default policy
  return waitingWriters == 0 && activeWriters == 0;
}
...
```

Leitores e Escritores demo

```
class ReadWriteDemo extends ReadersWritersStateTracking {
  ...
  public void doit() {
      new Reader(this).start();
      ...
  }
  ...
  protected void doRead() {
      System.out.print("(");
      ...
      System.out.print(")");
  }
  protected void doWrite() {
      System.out.print("[");
      ...
      System.out.print(")");
    }
}
```

```
(()())[][][][][][][]
[][][][](()()()()()
()()()()()())[][]
[][][][][][][][][]
(()()()()())
```

Fim do Capítulo 8

- Esse capítulo apresentou as propriedade de prioridade, justiça e intercepção e problema de leitores e escritores.
- Revise os conceitos apresentados.
- Resolver os exercícios da Lista.