Programação Concorrente e Paralela

Capítulo 4 – Modelagem e Threads

Marcial Porto Fernández
marcial@larces.uece.br

Semestre 2018.2

Sumário

- Modelagem
  - Processo Estado Finito (FSP)
  - Sistema de Transição de Processos (LTS)
- Java Threads
  - Sincronização em Java

Sumário

- Modelagem
  - Processo Estado Finito (FSP)
  - Sistema de Transição de Processos (LTS)
- Java Threads
  - Sincronização em Java

Modelagem de simultaneidade

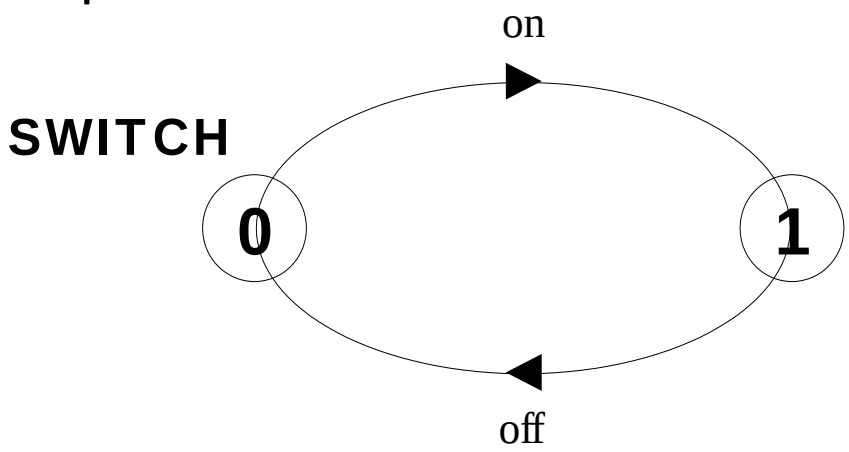
- Como sistemas concorrentes são não-deterministicos, pode ser difícil raciocinar sobre suas propriedades.
- Um modelo é uma abstração do mundo real que torna mais fácil analisar os pontos de interesse.
- Abordagem:
  - Modelar sistemas simultâneos como um conjunto de estados finitos sequenciais de processos

Processos de Estado Finito

- FSP (Finit State Process) é uma notação textual para a especificação de um processo de estado finito:

```
SWITCH = (on -> off-> SWITCH).
```

- LTS (Labelled Transition System) é uma notação gráfica para interpretar um processo como um sistema de transição rotulados:



- A significado de um processo é um conjunto de traços possíveis:
  - on →off → on → off → on → off → on ...

Sumário

- Modelagem
  - Processo Estado Finito (FSP)
  - Sistema de Transição de Processos (LTS)
- Java Threads
  - Sincronização em Java

Resumo de FSP

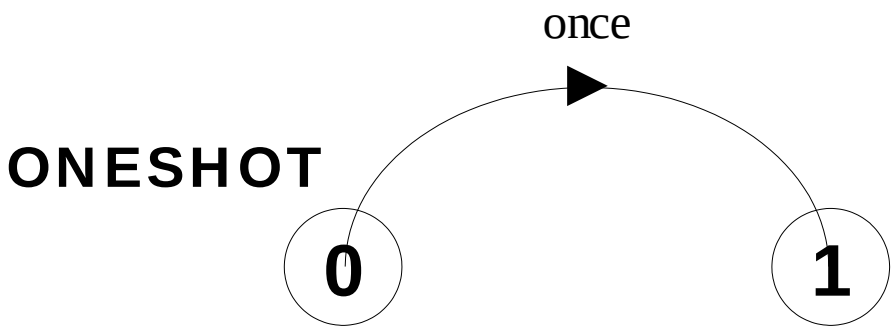
Prefixo de ação	`(x->P)`	***Composição paralela***	`(P||Q)`
Escolha	`(x->P|y->Q)`	***Replicador***	`forall [I:1..N] P(I)`
Ação vigiada	`(when B x->P|y->Q)`	***Rótulo processo***	`a:P`
Extensão alfabeto	`P + S`	***Processo compartilhado***	$\{a_1,...,a_n\}::P$
Condicional	`If B then P else Q`	***Prioridade alta***	$\|\|C=(P\|\|Q)<<\{a_1,\ldots,a_n\}$
Remarcação	$/\{new_1/old_1,\ldots\}$	***Prioridade baixa***	$\|\|C=(P\|\|Q)>>\{a_1,\ldots,a_n\}$
Escondido	$\backslash\{a_1,\ldots,a_n\}$	***Propriedade***	`property P`
Interface	$@\{a_1,\ldots,a_n\}$	***Progresso***	`progress P =` $\{a_1,\ldots,a_n\}$

Identificadores Minúsculo: ação Identificadores Maiúsculo: processo

FSP - Prefixo de Ação

- Se **x** é uma ação e **P** um processo, então (x -> P) é um processo P que é disparado com a ação x e, em seguida, se comporta como um estado P.

```
ONESHOT = (once -> STOP).
```

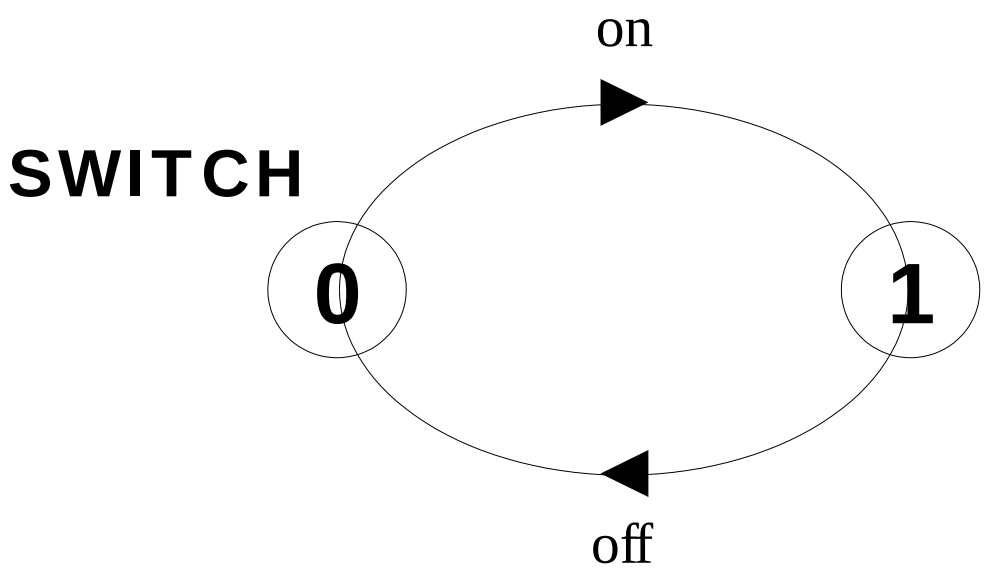


- Convenção:
  - Processos começar com MAIÚSCULAS, As ações começam com minúsculas.

FSP - Recursão

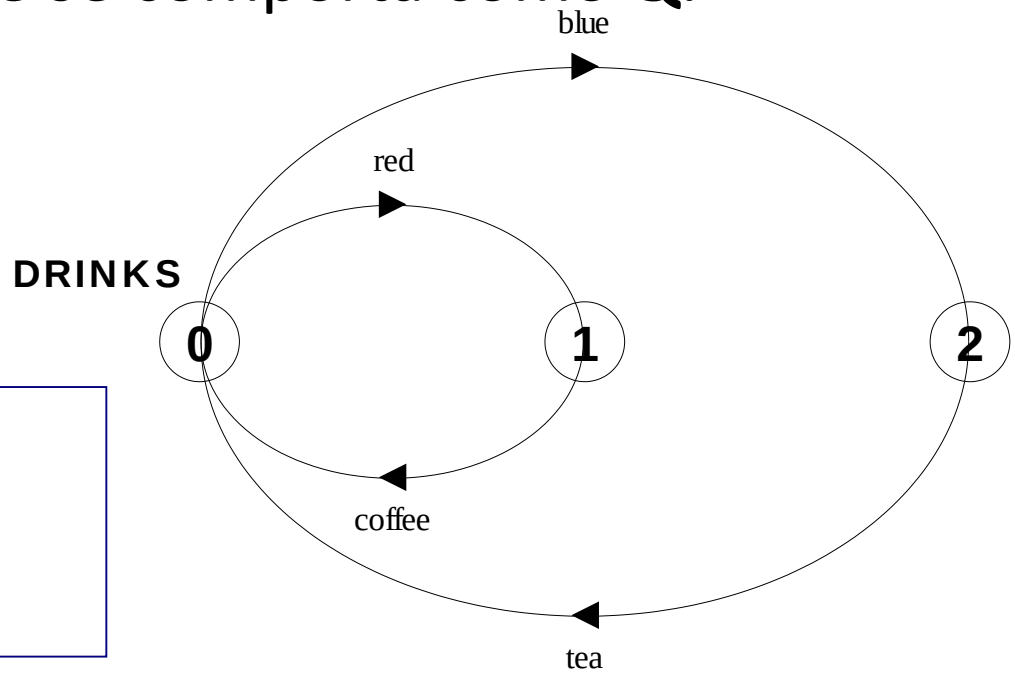
- Comportamento repetitivo utiliza recursão

```
SWITCH = OFF,
OFF = (on -> ON),
ON  = (off-> OFF).
```



FSP - Escolha

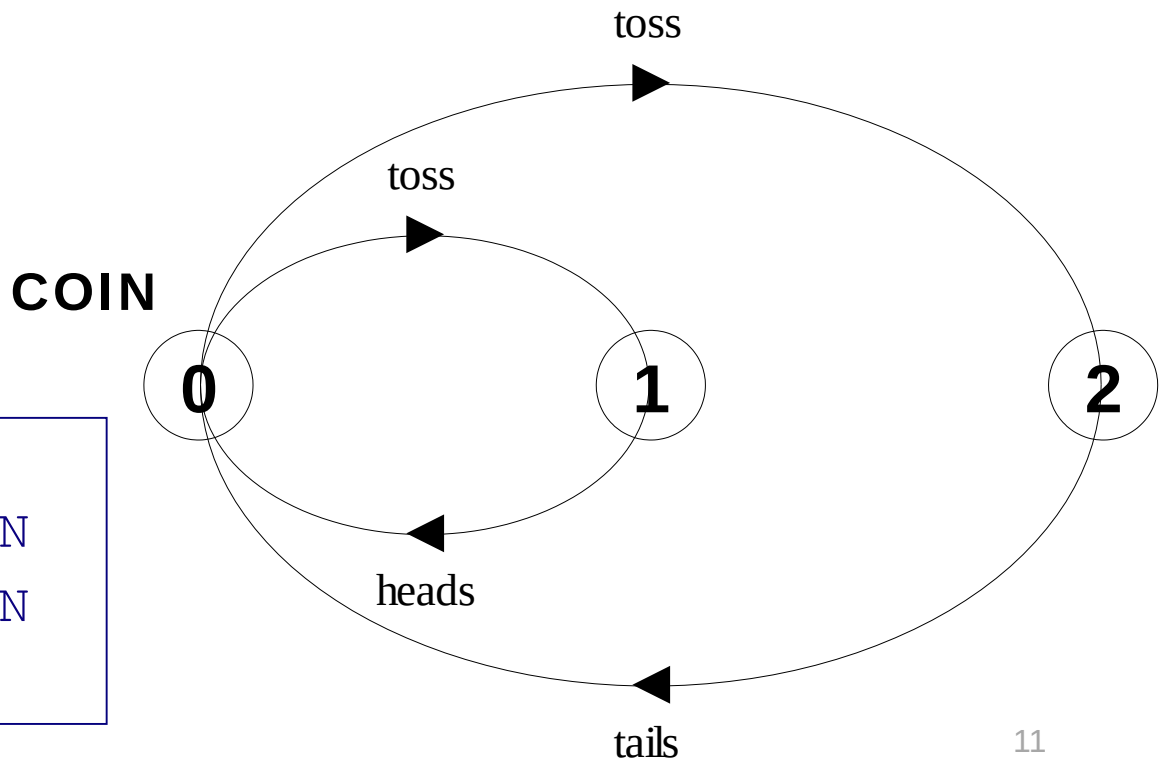
- Se x e y são ações, então (x-> P | y-> Q) é um processo que inicia com qualquer das ações x ou y.
- Se **x** ocorre, o processo então se comporta como **P**, caso contrário, se **y** ocorre, ele se comporta como **Q**.




```
DRINKS =
  ( red -> coffee -> DRINKS
  | blue -> tea -> DRINKS
  ).
```

FSP - não-determinismo

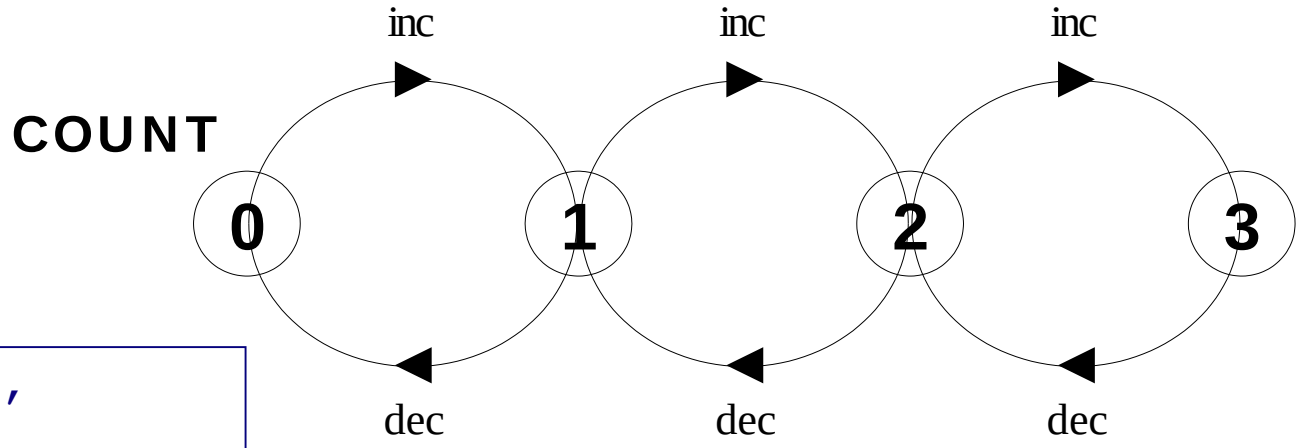
- (x-> P | x-> Q) ao executar **x**, em seguida, se comporta como **P** ou **Q**.




```
COIN =
 ( toss   -> heads  -> COIN
 | toss   -> tails  -> COIN
 ).
```

Ações vigiadas - FSP

- (when B x->P | y->Q) significa que quando a guarda B é verdadeira em seguida, qualquer um **x** ou **y** podem ser escolhidos, caso contrário, se B é falso então só **y** pode ser escolhido.




```
COUNT (N=3)   = COUNT[0],
COUNT[i:0..N]=
 ( when(i<N) inc->COUNT[i+1]
 | when(i>0) dec->COUNT[i-1]
 ).
```

Sumário

- Modelagem
  - Processo Estado Finito (FSP)
  - Sistema de Transição de Processos (LTS)
- Java Threads
  - Sincronização em Java

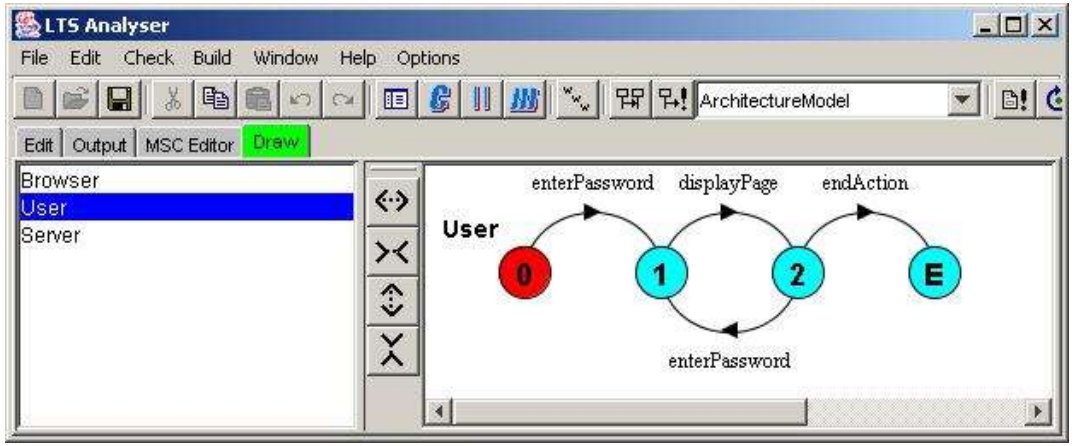
Definição de LTS

- Um LTS (Labelled Transition System) **T** consiste nos seguintes elementos:
  - um conjunto **S** de estados
  - um conjunto **L** de ações
  - um conjunto de transições **->** de forma que: s->a->t, onde $s,t \in S$ e $a \in L$.
  - um estado inicial $s_0 \in S$

- Assim, **T= (S,L,->, s_0).**

LTSA - Labelled Transition System Analyser

- LTSA é uma ferramenta de verificação de sistemas concorrentes desenvolvida no Imperial College London
- Ela verifica automaticamente a especificação de um sistema concorrente descrita em FSP ou LTS para verificar se satisfaz as propriedades requeridas.
- Além disso, exibe uma animação da especificação para facilitar o desenvolvimento interativo do sistema.



LTSA - Labelled Transition System Analyser

- Um sistema em LTS é modelado como um conjunto de máquinas de estados finitos iterativas. As propriedades requeridas também são modelados como máquinas de estado.
- LTSA realiza análise de acessibilidade exaustivamente para procurar violações das propriedades.
- Cada componente de uma especificação é descrita como um Labelled Transition System (LTS), que contém todos os estados que um componente pode chegar e todas as transições que podem executar.

LTSA - Labelled Transition System Analyser

- No entanto, a descrição explícita de um LTS em termos de seus estados, conjunto de rótulos de ação e relação de transições é complicado, a não ser para pequenos sistemas.
- O sistema LTSA suporta a descrição em notação de álgebra de processos (FSP) para a descrição concisa do comportamento.
- A ferramenta converte a especificação FSP em LTS para permitir ser visualizados graficamente.
- http://www.doc.ic.ac.uk/ltsa/

Sumário

- Modelagem
  - Processo Estado Finito (FSP)
  - Sistema de Transição de Processos (LTS)
- Java Threads
  - Sincronização em Java

Implementação Java

- Modelo de simultaneidade baseado em monitores
  - Palavra-chave *synchronized*
  - Métodos wait() e *notify()*
  - Classe *Thread* e interface *Runnable*

- Pacote java.util.concurrency (Java > 1.5)
  - Implementa muitas expressões comuns para simultaneidade

Java Threads

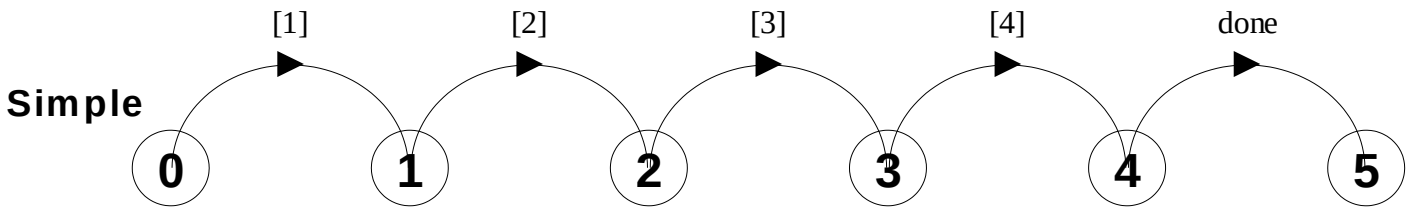
- Um Thread Java tem um método *run* para definir o comportamento:

```
class SimpleThread extends Thread {
      public static void main (String[] args) { ... }
      public SimpleThread(String str) {
            super(str);       // Call Thread constructor
      }
      public void run() {     // What the thread does
            for (int i=0; i<5; i++) {
                  System.out.println(i + " " + getName());
                  try {sleep((int)(Math.random()*1000));}
                  catch(InterruptedException e) { }
            }
            System.out.println("DONE! " + getName());
      }
  }
```

SimpleThread FSP

SimpleThread pode ser modelado como um único processo de estado finito sequencial :

```
Simple = ([1] -> [2] -> [3] -> [4] -> done -> STOP).
```



Ou, mais genericamente:

```
const N       =  5
Simple        =  Print[1],
Print[n:1..N] =  ( when(n<N)  [n] -> Print[n+1]
                 | when(n==N) done -> STOP).
```

Threads múltiplos..

- O método *run* de Thread nunca é chamado diretamente, mas é executado quando o thread é iniciado:

```
class SimpleThread {
      public static void main (String[] args) {
            // Instantiate a Thread, then start it:
            new SimpleThread("Jamaica").start();
            new SimpleThread("Fiji").start();
      }
      …
}
```

Executando o TwoThreadsDemo

Nesta implementação de Java, a execução dos dois threads é intercalada.

Isto não é garantido para todas as implementações!

Porque é que as linhas de saída nunca saem truncadas?

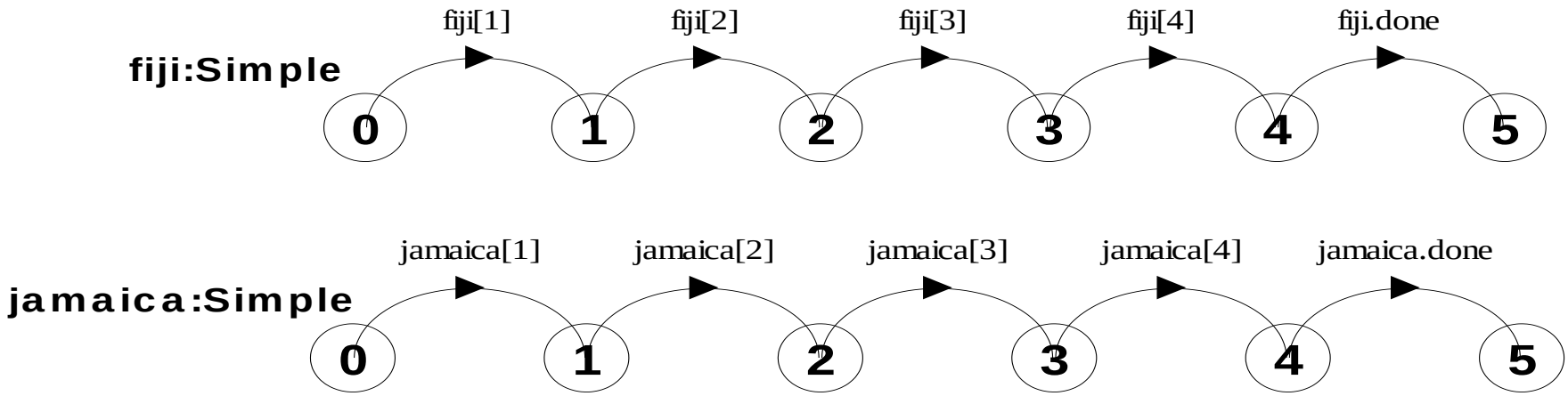
```
0 Ja0 Fimajiica ...
```

```
0 Jamaica
0 Fiji
1 Jamaica
1 Fiji
2 Fiji
3 Fiji
2 Jamaica
4 Fiji
3 Jamaica
DONE! Fiji
4 Jamaica
DONE! Jamaica
```

FSP - Concorrência

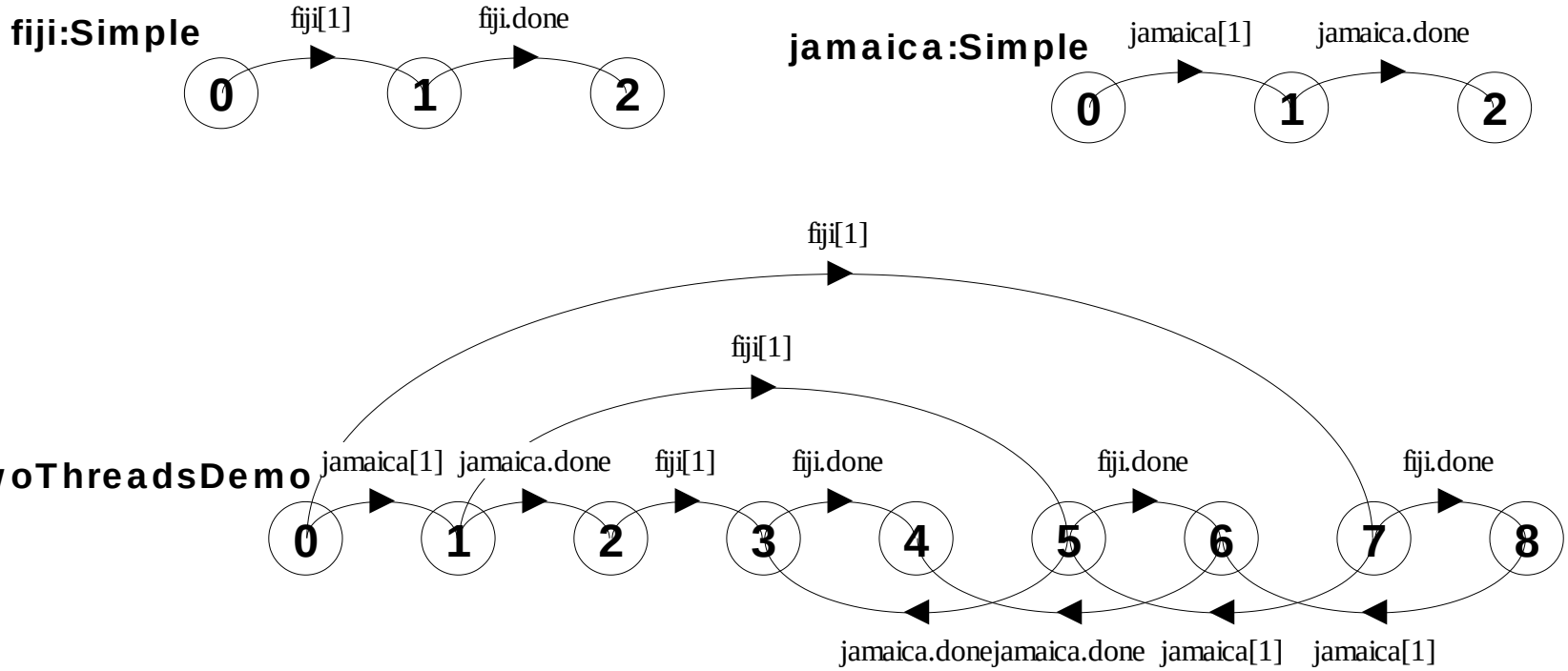
- Nós podemos reetiquetar as transições de *Simple* e ao mesmo tempo compor duas cópias do mesmo:

```
||TwoThreadsDemo = ( fiji:Simple
                   || jamaica:Simple
                   ).
```



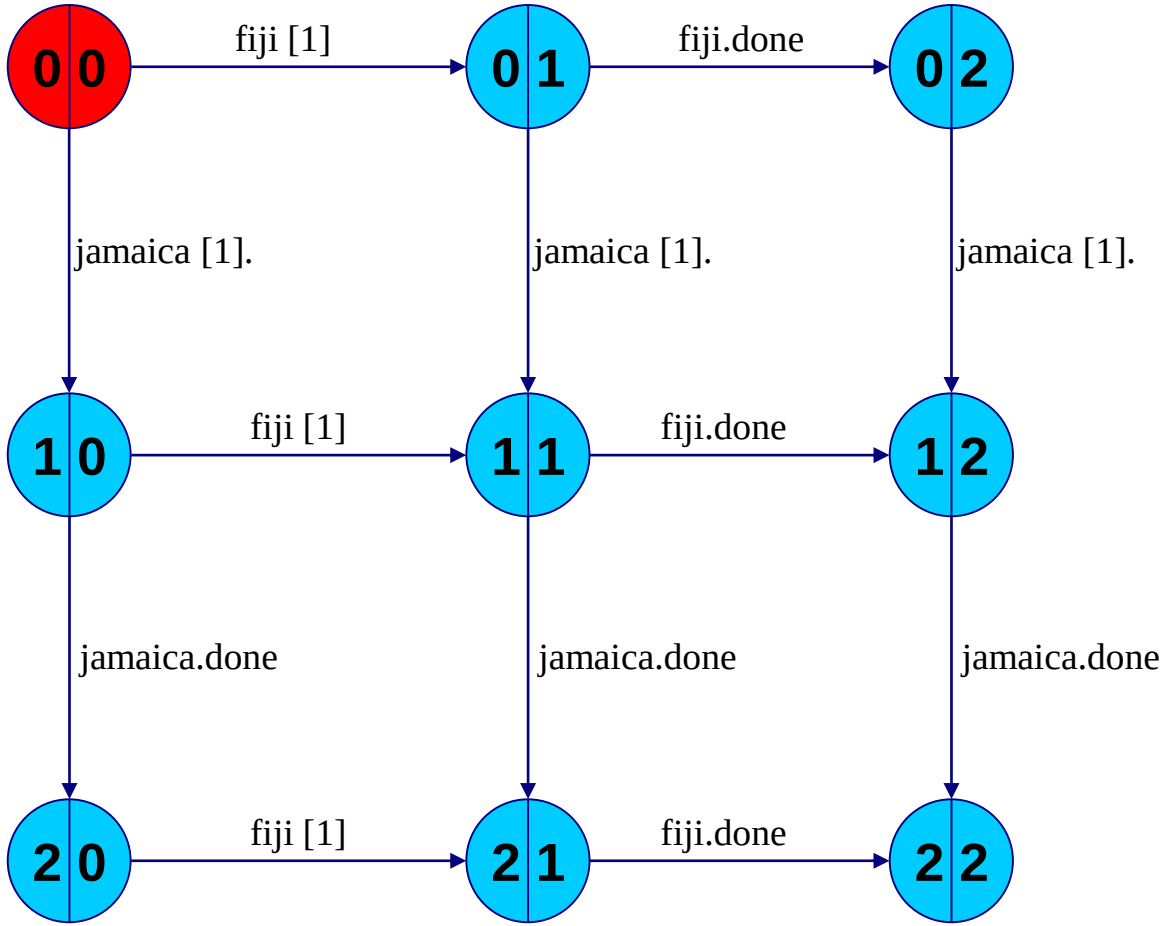
FSP - Composição

- Se nos limitarmos a duas etapas, a composição terá nove estados:



Composição de espaço de estados

O espaço de estado de dois processos compostos é (no máximo) o produto cartesiano dos espaços de cada estado



java.lang.Thread (criação)

- Um Thread Java tanto pode herdar de java.lang.Thread, ou conter um objeto *Runnable*:

```
public class java.lang.Thread
      extends java.lang.Object
      implements java.lang.Runnable
{
      public Thread();
      public Thread(Runnable target);
      public Thread(Runnable target, String name);
      public Thread(String name);
...
```

java.lang.Thread (métodos)

O Thread deve ser criado, e então **start()**:

```
...
      public void run();
      public synchronized void start();
      public static void sleep(long millis)
            throws InterruptedException;
      public static void yield();
      public final String getName();
...
}
```

java.lang.Runnable

```
public interface java.lang.Runnable {
     public abstract void run();
}
```

Como Java não suporta herança múltipla, é impossível herdar simultaneamente de Thread e outra classe.

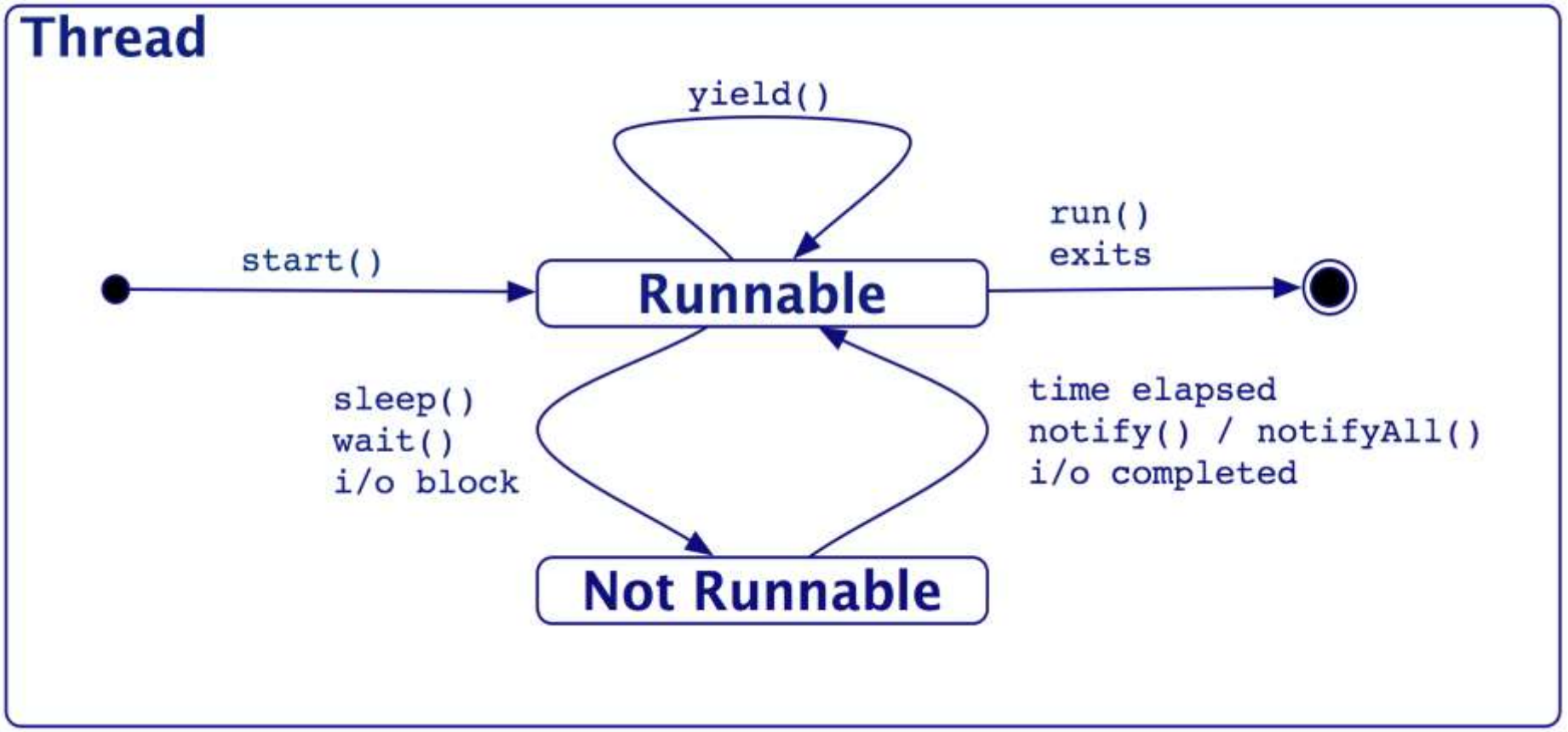
Em vez disso, basta definir:

```
class MyStuff extends UsefulStuff
     implements Runnable ...
```

e criar uma instância:

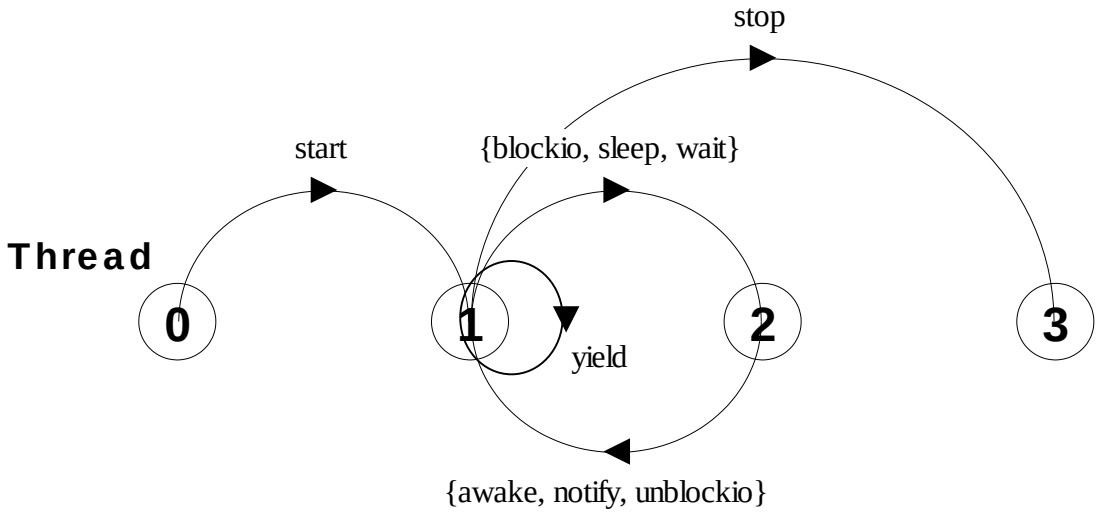
```
new Thread(new MyStuff).start();
```

Transições entre Estados de Thread

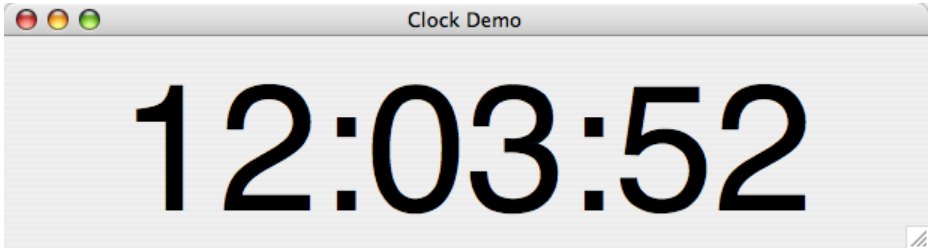


LTS de Threads

```
Thread = ( start -> Runnable ),
Runnable =
     ( yield -> Runnable
     | {sleep, wait, blockio} -> NotRunnable
     | stop -> STOP ),
NotRunnable =
     ( {awake, notify, unblockio} -> Runnable ).
```



Criando Threads



Este aplicativo Clock usa um thread para atualizar a hora:

```
public class Clock extends Canvas implements Runnable {
     private Thread clockThread = null;

     public Clock() {
          super();
          if (clockThread == null) {
               clockThread = new Thread(this, "Clock");
               clockThread.start();
          }
     }
```

Criando Threads ...

```
...
public void run() {
   // stops when clockThread is set to null
   while(Thread.currentThread() == clockThread) {
      repaint();
      try {clockThread.sleep(1000); }
      catch (InterruptedException e){ }
   }
}
...
```

... e detê-los

```
...
     public void stopThread() {
          clockThread = null;
     }

     public void paintComponent(Graphics g) {
          ...
          String time = dateFormat.format(new Date());
          g2d.drawString(...);
     }
...
```

Sumário

- Modelagem
  - Processo Estado Finito (FSP)
  - Sistema de Transição de Processos (LTS)
- Java Threads
  - Sincronização em Java

Sincronização

- Sem **sincronização**, um número arbitrário de threads pode ser executado a qualquer momento dentro dos métodos de um objeto.
  - Invariantes da classe não pode ser garantida quando o método inicia!
  - Portanto, não pode garantir qualquer condição futura!
- **Uma solução:** considerar um método para ser uma seção crítica que bloqueia o acesso ao objeto enquanto ele está executando.
  - Isso funciona, desde que os métodos cooperem no bloqueio e desbloqueio de acesso!

Métodos sincronizados

- **Ou:** declarar um método completo para ser sincronizado com outros métodos de sincronização de um objeto:

```
public class PrintStream extends FilterOutputStream
{
     ...
     public synchronized void println(String s);
     public synchronized void println(char c);
     ...
}
```

Blocos Sincronizados

- **Ou ainda:** sincronizar um bloco individual dentro de um método em relação a algum objeto (recurso):

```
public Object aMethod() {
     // unsynchronized code
     ...
     synchronized(resource) { // lock resource
          ...
     }                          // unlock resource
     ...
}
```

Wait e Notify

Sincronização por vezes têm de ser interrompida:

```
public class Account {
	protected int assets = 0;
	public synchronized void withdraw(int amount) {
		while (amount > assets) {
			try {
				wait();
			} catch(InterruptedException e) { }
		}
		assets -= amount;
	}
	public synchronized void deposit(int amount) {
		assets += amount;
		notifyAll();
	}
}
```

Wait e Notify em ação ...

```
final Account myAccount = new Account();
new Thread() {
      public void run() {
            int amount = 100;
            System.out.println("Waiting to withdraw " + amount + " units ...");
            myAccount.withdraw(amount);
            System.out.println("I withdrew " + amount + " units!");
      }
}.start();
try { Thread.sleep(1000); }
catch (InterruptedException e){ }
new Thread() {
      public void run() {
            int amount = 200;
            System.out.println("Depositing " + amount + " units ...");
            myAccount.deposit(amount);
            System.out.println("I deposited " + amount + " units!");
      }
}.start();
```

java.lang.Object

NB: wait() e notify() são métodos ao invés de palavras-chaves:

```
public class java.lang.Object
{
     ...
     public final void wait()
          throws InterruptedException;
     public final void notify();
     public final void notifyAll();
     ...
}
```

Fim do Capítulo 4

- Esse capítulo apresentou a modelagem de sistemas concorrente e Java Threads.
- Revise os conceitos apresentados.
- Resolver os exercícios da Lista.
- Instale o programa LTSA e teste alguns exemplos.