Processes and Thread III
Concurrent Execution I

Lecture 4

Alma Rahat

CS-210: Concurrency

3 February, 2021

Questions and comments: shorturl.at/dDLMY

What did we do in the last session?

- For implementing choices we use the following syntax: `(x -> P | y-> Q)`.
- Animation or looking at traces can indicate potential issues with a system.

In this lecture...

Learning outcomes.

1. To describe a thread's lifecycle using FSP.
2. To simplify FSP using indices.
3. To describe how concurrent processes are executed on a platform.

Outline.

1. Java `Thread` lifecycle.
2. Indexed processes and actions.
3. Guarded actions.
4. Execution of Concurrent Processes.
5. Modelling Concurrency.

Java Thread lifecycle

`Thread` class has a property called `State`. It can take the following values:

NEW	A thread that is yet to `start` is in this state.
RUNNABLE	A thread executing in the Java virtual machine is in this state.
BLOCKED	A thread that is blocked waiting for a monitor lock is in this state.
WAITING	A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
TIMED_WAITING	A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
TERMINATED	A thread that has existed is in this state.

Java Thread lifecycle

More on waiting states:

BLOCKED Imagine trying to play squash when someone is hogging the court. You have to wait until the court becomes available (i.e. you are waiting for resources to become available).

WAITING Imagine having a conversation with your mates while you wait for your food at the restaurant. Until the chef prepares the food you cannot eat (i.e. you are waiting for products from other threads).

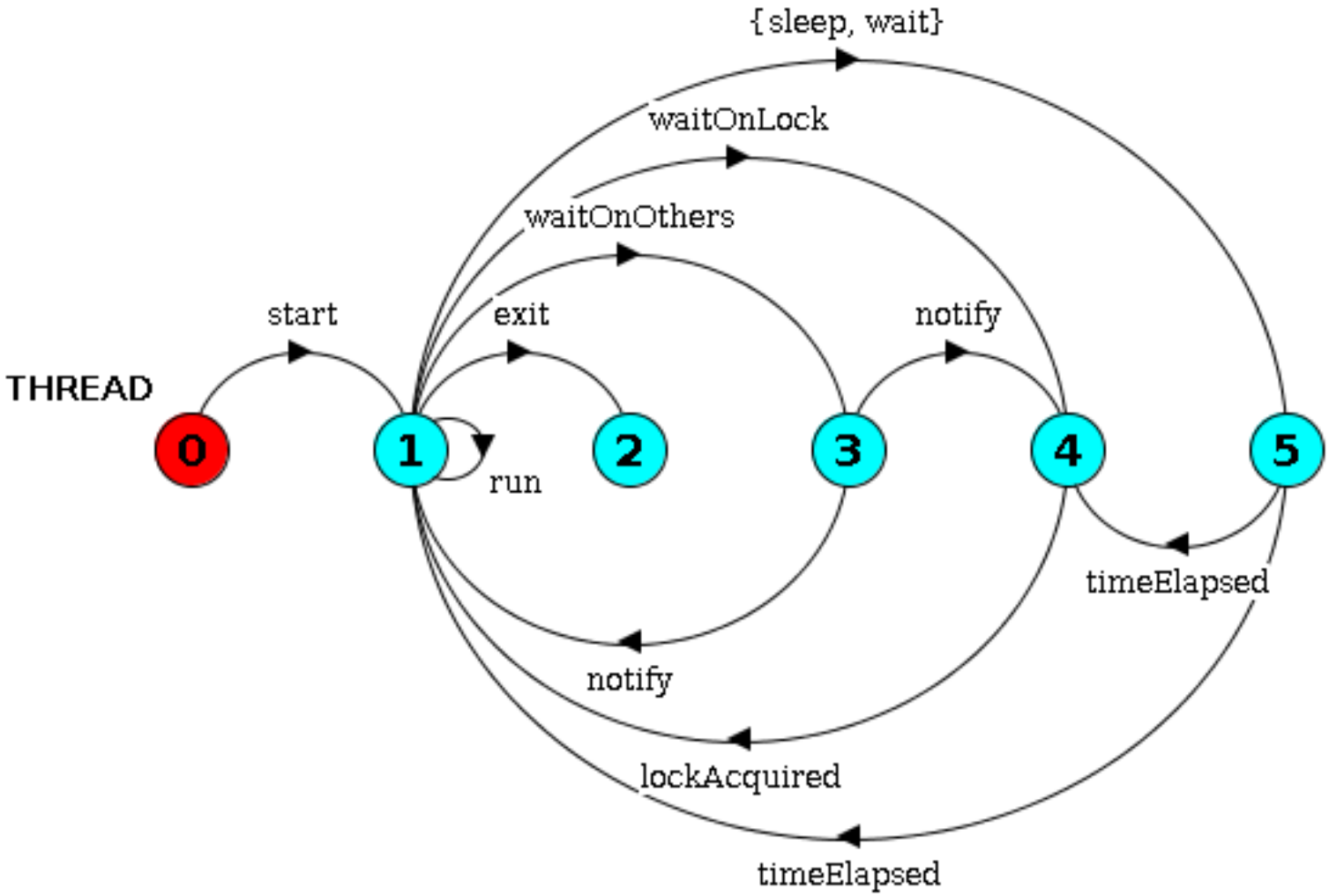
TIMED_WAITING You are told to wait for a specified time.

Java Thread lifecycle

FSP for Java Thread lifecycle.

```
THREAD = NEW, \\An instance of a thread.
NEW = (start -> RUNNABLE), \\start method makes it runnable
(running or ready to be scheduled).
RUNNABLE = (waitOnLock -> BLOCKED
       | sleep -> TIMED_WAITING
       | wait -> TIMED_WAITING
       | waitOnOthers -> WAIT
       | run -> RUNNABLE
       | exit -> TERMINATED),
BLOCKED = (lockAcquired -> RUNNABLE),
TIMED_WAITING = (timeElapsed -> RUNNABLE
       | timeElapsed -> BLOCKED),
WAIT = (notify -> RUNNABLE
       | notify -> BLOCKED),
TERMINATED = STOP.\\STOP is a special keyword for termination.
```

Java Thread lifecycle



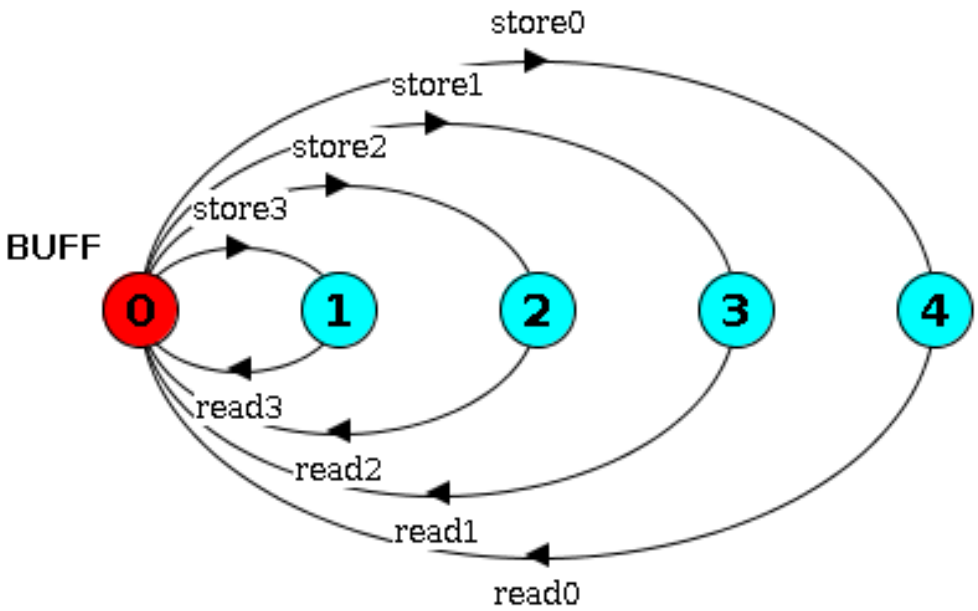
Any questions?

Indexed Processes and Actions

Consider a four slot buffer: imagine a four slot RAM, you can store a value and read it back.

```
BUFF = (store0-> read0 -> BUFF
      | store1 -> read1 -> BUFF
      | store2 -> read2 -> BUFF
      | store3 -> read3 -> BUFF)
```

Tedious?

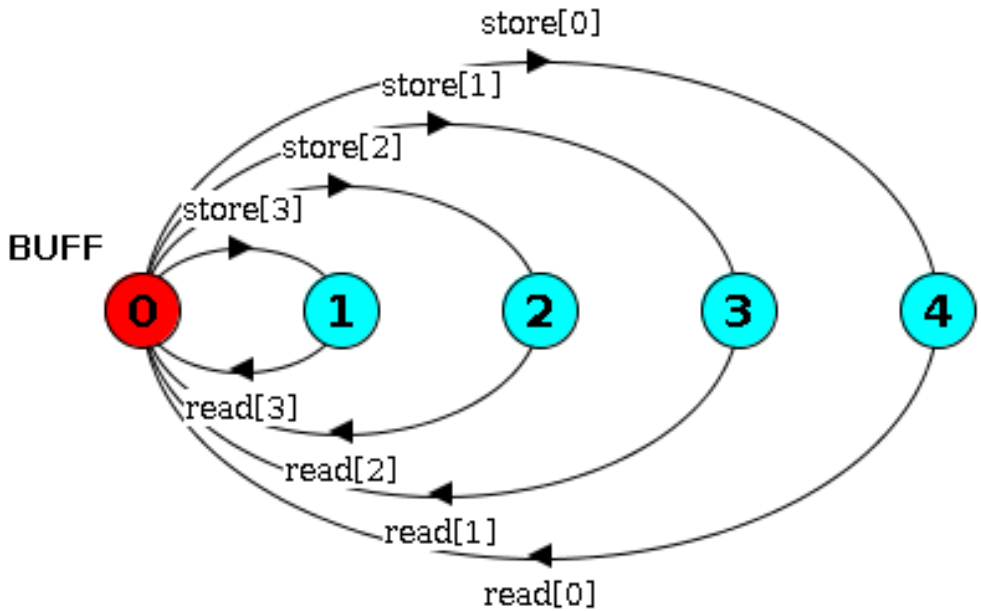


Indexed Processes and Actions

Consider a four slot buffer: imagine a four slot RAM, you can store a value and read it back.

```
BUFF = (store[0]-> read[0] -> BUFF
      | store[1] -> read[1] -> BUFF
      | store[2] -> read[2] -> BUFF
      | store[3] -> read[3] -> BUFF)
```

Indices instead.



Indexed Processes and Actions

Consider a four slot buffer: imagine a four slot RAM, you can store a value and read it back.

Simplification of indices.

```
BUFF = (store[i:0..3]-> read[i] -> BUFF).\\x..y   is   a
```
range from x to y.

Use of constants.

```
const N = 3 \\const is a keyword for defining a constant.
BUFF = (store[i:0..N]-> read[i] -> BUFF).
```

Use of process parameters.

```
BUFF(N = 3) = (store[i:0..N]-> read[i] -> BUFF).\\N is
```
a parameter.

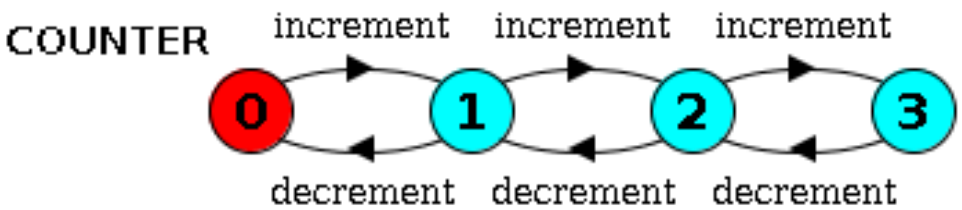
Another keyword: `range T = 0..3`.

Guarded Actions

The choice (when B x ->P | y ->Q) describes a process that is like (x ->P | y ->Q) except that the action x can only be chosen when the guard B is true.

Consider a counter: it can count up to *N* and then counts down to 0.

```
COUNTER(N = 3) = COUNTER[0],
COUNTER[i:0..N] = (when (i < N) increment
-> COUNTER[i+1] | when (i > 0) decrement ->
COUNTER[i-1]).
```



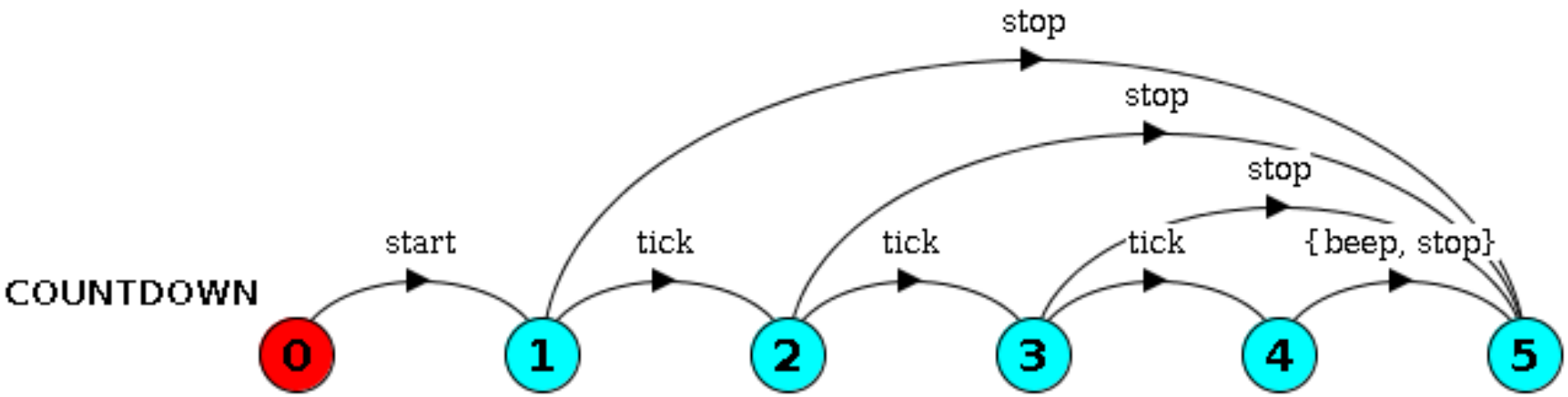
Example: COUNTDOWN

A countdown timer which beeps after 4 ticks and stops, or it can be stopped at any state.

Example: COUNTDOWN

A countdown timer which beeps after 4 ticks and stops, or it can be stopped at any state.

```
const N = 3
range T = 0..N
COUNTDOWN = (start -> COUNTDOWN[N]),
COUNTDOWN[i:T] = (when i>0 tick ->
COUNTDOWN[i-1] | when i==0 beep -> STOP | stop
-> STOP).
```



Homework Exercise

Write the FSP and draw the state diagram.

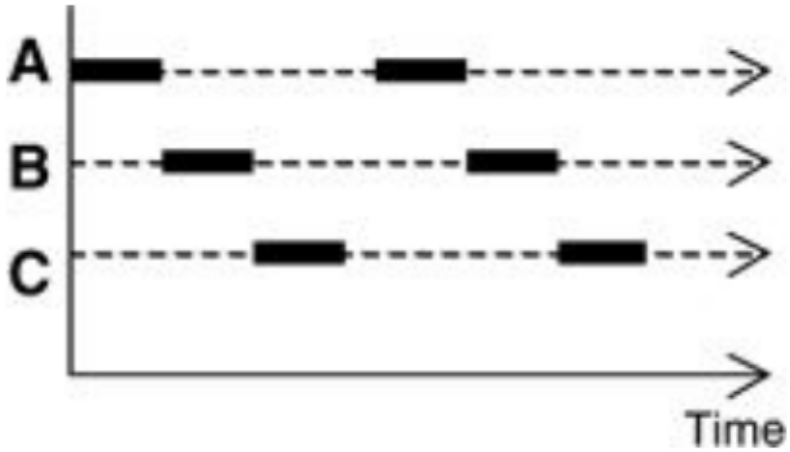
1. A drinks dispensing machine charges 15p for a can of cola. The machine accepts 5p, 10p and 20p, and gives change.

Any questions?

Thus far we have only discussed a single process: a set of tasks that must occur sequentially. We are interested in multiple tasks (or processes) running concurrently. Henceforth, we will concentrate on multiple processes.

Execution of Concurrent Processes

- The term concurrency means logically simultaneous processing, and does not imply multiple processing elements (PEs). Multiple concurrent process may run on a single PE through interleaving.



- Parallelism means that you actually execute multiple processes in parallel.
- Both concurrency and parallelism require controlled access to shared resources: so we logically consider them to be equivalent.

Modelling Concurrency

We are interested in thinking about multiple processes that may be being executed simultaneously (at least logically).

- How should we model process execution speed?
  $\Rightarrow$ Arbitrary speed; we abstract away time. We want to model independent of number of processes or the scheduling strategy.
- How do we model concurrency?
  $\Rightarrow$ Arbitrary relative order of actions from different processes; interleaving but preservation of each process order.
- What is the result?
  $\Rightarrow$ Provides a general model independent of scheduling; asynchronous (in time) model of execution.

Summary

- We can model a thread's lifecycle using FSP.
- We can use indices in FSP: `store[i:0..3]`.
- We can use guarded actions: `(when B x-> | y -> Q)`.
- We model processes as logically parallel, but in reality they may be interleaved.