Processes and Threads

Abhik Roychoudhury
CS 3211
National University of Singapore

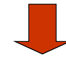
Reading material: Chapter 2 of Textbook.

Concurrent processes

We structure complex systems as sets of simpler activities, each represented as a **sequential process**. Processes can be concurrent, so as to reflect the concurrency inherent in the physical world, or to offload time-consuming tasks, or to manage communications or other devices.

Designing concurrent software can be complex and error prone. A rigorous engineering approach is essential.

Concept of a process as a sequence of actions.



Model processes as finite state machines.

Program processes as threads in Java.

Warm-up exercises at beginning of Lecture 3

Abhik Roychoudhury
National University of Singapore

A question asked by your friend

```
Thread 0                                   Thread 1
1. lock0 = 1;                              A. lock1 = 1;
2. turn = 1;                               B. turn = 0;
3. while(1){                               C. while(1) {
4.        if (lock1!=1)||(turn==0)            D.   if
(lock0!=1)||(turn==1)
5.            break; }                     E.          break;}
6. counter++;                              F.  counter++;
7. lock0 = 0;                              G. lock1 = 0;
```

Question: You said in class if we mark all shared variables as volatile in Peterson's mutual exclusion algorithm – still mutual exclusion may be violated – due to volatile write -> volatile read re-orderings. How does the situation change – if I did not mark the shared variables as volatile?

Answer: If the shared variables are not marked volatile and not protected by lock – clearly there is no expectation of not having data races in the program.

However, if the shared variables are marked as volatile – one may have a misplaced expectation that the program will work "correctly". This is because you treat the Java volatile variables as a substitute for locks.

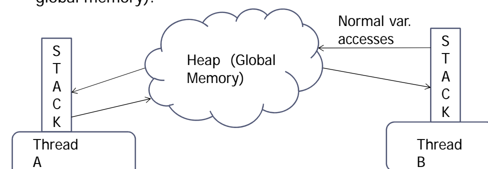
This expectation is wrong. Locks ensure (i) atomicity of the code protected by lock (ii) mutual exclusion, and (iii) visibility of the updates made in critical section once you release lock. Note that volatile variables only ensure atomicity, and not mutual exclusion.

My question for you

- Do the volatile variables in Java ensure state visibility? If I make a volatile variable operation, after the operation is the state change visible to all threads?
- What do you think? If we follow the main property of volatile variables – we can get to the answer! See next slide and decide!!

Volatile Variables

- Consider any simple Java stmt, e.g., x=0
  - Translates to a sequence of bytecodes, not atomically executed.
- One way of ensuring atomic execution in Java –
  - Mark variables as volatile.
  - reads/writes of volatile variables are atomic (directly update global memory).



A quick teaser

Consider an atomic instruction flip --- always executed ***atomically***. It flips a 0 to 1 and a 1 to 0. Suppose 2 processes are each executing the following code. Initially lock == 0

```
/* Lock acquisition*/
while ( flip(lock) != 1)
      while (lock != 0)
{};
CRITICAL SECTION
/* Lock release */
lock = 0;
```

Is there any possible **violation of mutual exclusion**? Why or why not?

Answer to the teaser

```
/* Lock acquisition*/
while ( flip(lock) != 1)
      while (lock != 0)
{};
CRITICAL SECTION
/* Lock release */
lock = 0;
```

```
Process X                                  Process Y
------------------------------------------------------------------------
flip(lock) != 1
Outer loop exit (lock == 1)
                                            flip(lock) != 1  (lock == 0)
                                              lock ! = 0   (false)
                                             Inner loop exit
                                            flip(lock) != 1 (lock == 1)
                                            Outer loop exit
CRITICAL SECTION                           CRITICAL SECTION
```

End of Warm-up exercises

- Discussed in class, and posted after class as lecture follow-up in IVLE.

What are we doing today?

In this lecture, we are tie-ing up the following.

Concepts: processes - units of sequential execution.

Models: **finite state processes (FSP)** to model processes as sequences of actions. **labelled transition systems (LTS)** to analyse, display and animate behavior. [***Promela models we saw in the first lecture correspond to finite state processes.***]

Practice: Java threads [***discussed in second lecture.***]

Conceptually what is a process, is implemented as a Java thread.

Going back to Concurrency

Sequential program (also use the term process)	Concurrent program (Concurrent composition of processes)
Fundamental Constructs:	***Fundamental Constructs:***
-> Prefixing of an action	Parallel Composition
| Choice	Relabeling of action names (while connecting processes, connect the disparate set of action names)
Iterative repetition	Hiding of actions (internal to a process, not visible to the concurrent composition)
[These are the ones used in a modeling language like Promela or programming language like Java]	

Modelling Processes

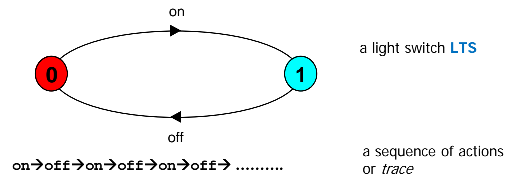
Models are described using state machines, known as Labelled Transition Systems LTS. These are described textually as finite state processes (FSP) and displayed and analysed by the ***LTSA*** analysis tool.

Level of details (↓)

- LTS - graphical form (state machines)
- FSP – textual form close to state machines
- Promela – imperative textual form (closer to programming)
- Java - programming language supporting multi-threading

Modeling Processes

A process is the execution of a sequential program. It is modeled as a finite state machine which transits from state to state by executing a sequence of atomic actions.



a light switch **LTS**

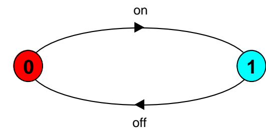
on→off→on→off→on→off→ ……….

a sequence of actions or *trace*

FSP - action prefix & recursion

Repetitive behaviour uses recursion:

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



Substituting to get a more succinct definition:

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

And again:

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

Concise equational description of non-terminating behavior.

In Promela (discussed earlier)

```
proctype switch()
{
   bit on;

   do
   :: on =1;   // equivalent to the action "on";
      on = 0;  // equivalent to the action "off";
   od
}
```

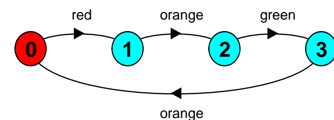
Reasonably close to implementation.
Yet supported by formal analysis !!

FSP - action prefix

FSP model of a traffic light :

```
TRAFFICLIGHT = (red->orange->green->orange
                 -> TRAFFICLIGHT).
```

LTS generated :



Trace:

red→orange→green→orange→red→orange→green …

FSP - choice

If **x** and **y** are actions then **(x-> P | y-> Q)** describes a process which initially engages in either of the actions **x** or **y**. After the first action has occurred, the subsequent behavior is described by **P** if the first action was **x** and **Q** if the first action was **y**.

Who makes the choice?

Is there a difference between **input** and **output** actions?

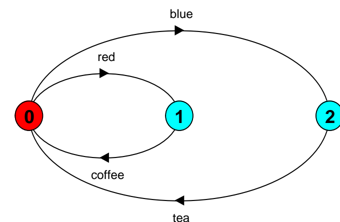
FSP - choice

FSP model of a drinks machine :

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

Input actions: red, blue
Output actions: coffee, tea

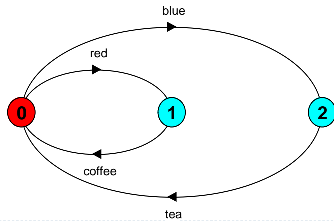
LTS generated :



Possible traces?

Spot Exercise

Traces??



Input and output

Input actions: red, blue
Output actions: coffee, tea

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

```
proctype DRINKS()
{
   do
   ::ch_in? color;
     if
     :: color == red -> ch_out!coffee;
     :: color == blue -> ch_out!tea;
     fi
   od
}
```

Promela description

Input and output are marked.

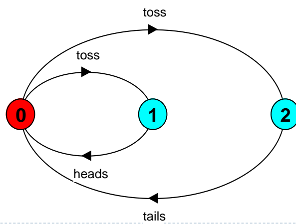
Non-deterministic choice

Process (x-> P | x -> Q) describes a process which engages in **x** and then behaves as either P or Q.

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

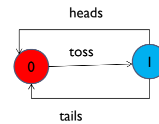
Tossing a coin.

Possible traces?



Another encoding

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



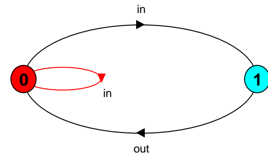
Possible traces ?
(as sequence of action labels)

Modeling failure in environment

How do we model an unreliable communication channel which accepts **in** actions and if a failure occurs it produces no output, otherwise performs an **out** action?

Use non-determinism...

```
CHAN = (in->CHAN
       |in->out->CHAN
       ).
```



Non-determinism in the physical world modeled by non-determinism in process description.

FSP - indexed processes and actions

Single slot buffer that inputs a value in the range 0 to 3 and then outputs that value. *Value passing – brings in variables.*

```
BUFF = (in[i:0..3]->out[i]-> BUFF).
```

equivalent to

```
BUFF = (in[0]->out[0]->BUFF
       |in[1]->out[1]->BUFF
       |in[2]->out[2]->BUFF
       |in[3]->out[3]->BUFF
       ).
```

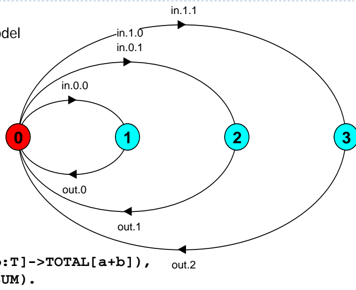
or using a process parameter with default value:

```
BUFF(N=3) = (in[i:0..N]->out[i]-> BUFF).
```

Input and output actions are clarified at the initiative of the programmer.

FSP - constant & range declaration

index expressions to model calculation:



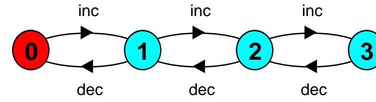

```
const N = 1
range T = 0..N
range R = 0..2*N

SUM         = (in[a:T][b:T]->TOTAL[a+b]),
TOTAL[s:R] = (out[s]->SUM).
```

Computation is described, apart from control flow!

FSP - guarded actions

The choice **(when B x -> P | y -> Q)** means that when the guard **B** is true then the actions **x** and **y** are both eligible to be chosen, otherwise if **B** is false then the action **x** cannot be chosen.

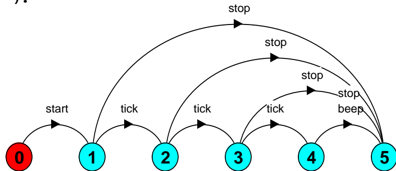
```
COUNT (N=3)         = COUNT[0],
COUNT[i:0..N] = (when(i<N) inc->COUNT[i+1]
                |when(i>0) dec->COUNT[i-1]
                ).
```



FSP - guarded actions

A countdown timer which beeps after N ticks, or can be stopped.

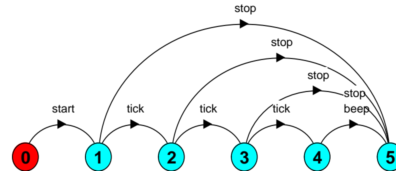
```
COUNTDOWN (N=3)    = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] =
            (when(i>0) tick->COUNTDOWN[i-1]
            |when(i==0)beep->STOP
            |stop->STOP
            ).
```



Traces of COUNTDOWN

We will discuss the Java implementation later!!

View it as a parameterized process, which will be implemented as a Java class.



FSP - guarded actions

What is the following FSP process equivalent to?

```
const False = 0
P = (when (False) doanything->P).
```


FSP - process alphabets

The alphabet of a process is the set of actions in which it can engage.

Alphabet extension can be used to extend the **implicit** alphabet of a process:

```
WRITER = (write[1]->write[3]->WRITER)
          +{write[0..3]}.
```

Alphabet of `WRITER` is the set `{write[0..3]}`

Exercise

In FSP, model a process FILTER, that exhibits the following repetitive behavior:

inputs a value v between 0 and 5, but only outputs it if v <= 2, otherwise it discards it.

```
FILTER = (in[v:0..5] -> DECIDE[v]),

DECIDE[v:0..5] = (     ?     ).
```

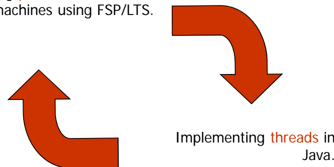

Organization

Modeling Processes (so far)

Implementing Processes in Java (now)

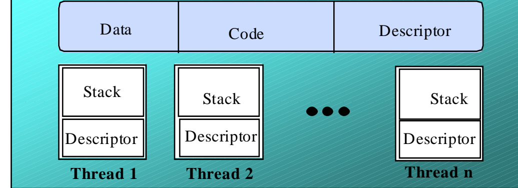
Implementing processes

Modeling processes as finite state machines using FSP/LTS.



Implementing threads in Java.

Implementing processes - the OS view



A (heavyweight) process in an operating system is represented by its code, data and the state of the machine registers, given in a descriptor. In order to support multiple (lightweight) **threads of control**, it has multiple stacks, one for each thread.

Threads and Processes

A Java Virtual Machine (JVM) usually runs as an OS process.

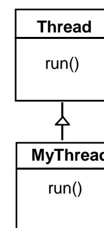
The JVM runs a multi-threaded Java program which has several threads. The thread scheduling may or may not be done by the JVM.

A thread is created by the keyword **new**, somewhat similar to the creation of other Java objects.

We have seen Java thread creation and starting in the lecture of week 2 (*last week's lecture*)

threads in Java

A Thread class manages a single sequential thread of control. Threads may be created and deleted dynamically.



The Thread class executes instructions from its method run(). The actual code executed depends on the implementation provided for run() in a derived class.

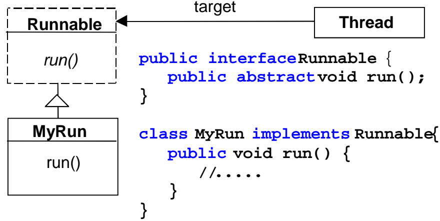
```
class MyThread extends Thread {
      public void run() {
            //......
      }
}
```

Creating a thread object:

```
Thread a = new MyThread();
```

threads in Java

We often implement the **run()** method in a class not derived from Thread but from the interface Runnable.

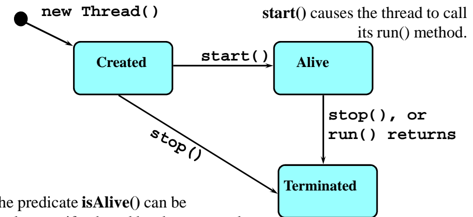



```
public interface Runnable {
    public abstract void run();
}

class MyRun implements Runnable{
    public void run() {
        //.....
    }
}
```

Creating a thread object:
`Thread b = new Thread(new MyRun());`
Starting: `b.start();`

thread life-cycle in Java

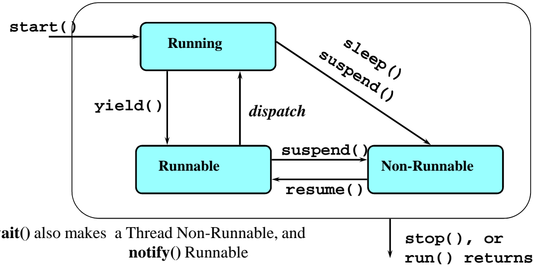
An overview of the life-cycle of a thread as state transitions:



The predicate **isAlive()** can be used to test if a thread has been started but not terminated. Once terminated, it cannot be restarted.

thread alive states in Java

Once started, an **alive** thread has a number of substates :



wait() also makes a Thread Non-Runnable, and **notify()** Runnable

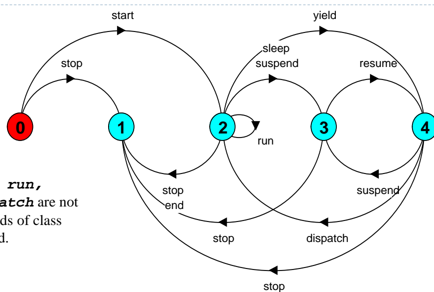
`stop(), or run() returns`

REVISIT: Previous week's discussion on wait(), notify(), notifyAll()

Java thread lifecycle - an FSP specification

```
THREAD        = CREATED,
CREATED       = (start           ->RUNNING
                |stop            ->TERMINATED),
RUNNING       = ({suspend,sleep}->NON_RUNNABLE
                |yield           ->RUNNABLE
                |{stop,end}      ->TERMINATED
                |run             ->RUNNING),
RUNNABLE      = (suspend         ->NON_RUNNABLE
                |dispatch        ->RUNNING
                |stop            ->TERMINATED),
NON_RUNNABLE = (resume           ->RUNNABLE
                |stop            ->TERMINATED),
TERMINATED    = STOP.
```


Java thread lifecycle - an FSP specification



end, run, dispatch are not methods of class Thread.

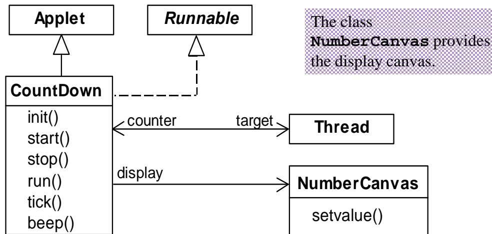
States 0 to 4 correspond to **CREATED, TERMINATED, RUNNING, NON-RUNNABLE**, and **RUNNABLE** respectively.

CountDown timer example

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

Implementation in Java?

CountDown timer - class diagram



The class `CountDown` derives from `Applet` and contains the implementation of the `run()` method which is required by `Thread`.

Countdown timer Implementation

Countdown is a class

The **counter** is itself a Thread within the Countdown class.

The **numberCanvas** is another class to make the implementation more realistic – as if it is a canvas on which the counter value will be displayed.
--- an instance of **numberCanvas** is modified inside the methods of **Countdown** class.

CountDown class

```
public class CountDown extends Applet
                       implements Runnable {
  Thread counter; int i;
  final static int N = 10;
  AudioClip beepSound, tickSound;
  NumberCanvas display;

  public void init()  {...}
  public void start() {...}
  public void stop()  {...}
  public void run()   {...}
  private void tick() {...}
  private void beep() {...}
}
```


CountDown class - start(), stop() and run()

```
public void start() {
  counter = new Thread(this);
  i = N; counter.start();
}

public void stop() {
  counter = null;
}

public void run() {
  while(true) {
    if (counter == null) return;
    if (i>0)  { tick(); --i; }
    if (i==0) { beep(); return;}
  }
}
```

COUNTDOWN Model

`start ->`

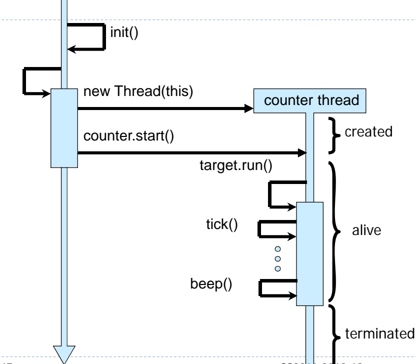
`stop ->`

`COUNTDOWN[i]` process
`recursion` as a `while` loop

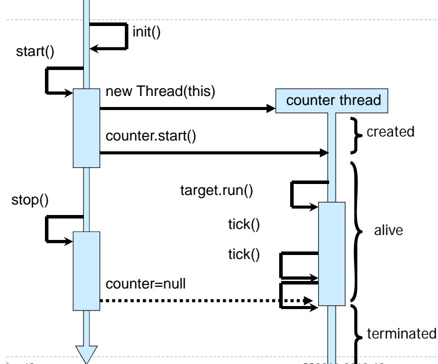
`when(i>0) tick -> CD[i-1]`
`when(i==0)beep -> STOP`

`STOP` when `run()` `returns`

CountDown execution



CountDown execution

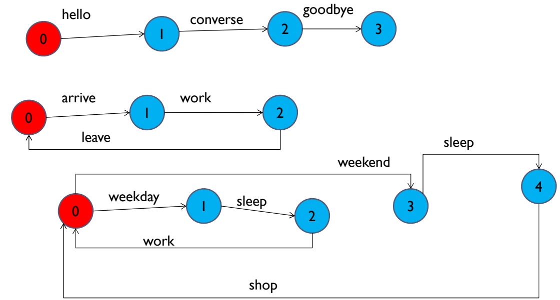


Summary

- Concepts
  - process - unit of concurrency, execution of a program
- Models
  - LTS to model processes as state machines - sequences of atomic actions
  - FSP to specify processes using
    - prefix "->"
    - choice " | "
    - recursion.
- Practice
  - Java threads to implement processes.
  - Thread lifecycle - created, running, runnable, ...

Quick Follow-up Exercises

Describe the following as FSP processes, or in Promela or as Java threads.



Next week ...

I am away.

Tutorial (Monday 4th Feb) as usual
Dr. Jooyong Lee will take my group, for next week only.

Lecture (Thursday 7th Feb) cancelled in next week.

Concurrent Execution of Processes and Threads

Abhik Roychoudhury
CS 3211
National University of Singapore

Reading material: Chapter 3 of Textbook.

So Far ...

How to model individual processes?

Implementing such processes as Java threads.

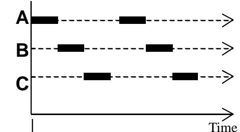
Now

Modeling composition of processes and their interaction/communication.

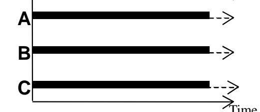
Implementing such a composition as a multi-threaded Java program.

Definitions

- ***Concurrency***
  - *Logically* simultaneous processing. Does not imply multiple processing elements (PEs). Requires interleaved execution on a single PE.



- ***Parallelism***
  - *Physically* simultaneous processing. Involves multiple PEs and/or independent device operations.



Both concurrency and parallelism require controlled access to shared resources . We use the terms parallel and concurrent interchangeably and generally do not distinguish between real and pseudo-concurrent execution.

3.1 Modeling Concurrency

- How should we model process execution speed?
  - arbitrary speed

    (we abstract away time)
- How do we model concurrency?
  - arbitrary relative order of actions from different processes (**interleaving** but preservation of each process order)
- What is the result?
  - provides a general model independent of scheduling (**asynchronous** model of execution)

parallel composition - action interleaving

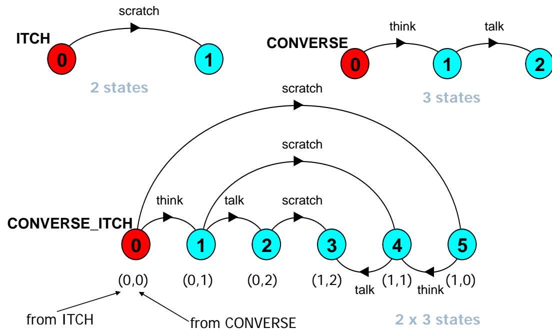
If P and Q are processes then (P||Q) represents the concurrent execution of P and Q. The operator || is the parallel composition operator.

```
ITCH     = (scratch->STOP).
CONVERSE = (think->talk->STOP).

||CONVERSE_ITCH = (ITCH || CONVERSE).
```

```
think→talk→scratch
think→scratch→talk
scratch→think→talk
```

Possible traces as a result of action interleaving.

parallel composition - action interleaving



parallel composition - algebraic laws

Commutative: (P||Q) = (Q||P)
Associative: (P||(Q||R)) = ((P||Q)||R) = (P||Q||R).

Clock radio example:

```
CLOCK = (tick->CLOCK).
RADIO = (on->off->RADIO).

||CLOCK_RADIO = (CLOCK || RADIO).
```

LTS? Traces? Number of states?

Class Exercise

```
CLOCK = (tick->CLOCK).
RADIO = (on->off->RADIO).

||CLOCK_RADIO = (CLOCK || RADIO).
```

LTS? Traces? Number of states?

The components CLOCK and RADIO run independently i.e. no communication.

modeling interaction - shared actions

If processes in a composition have actions in common, these actions are said to be *shared*. Shared actions are the way that process communication is modelled. While unshared actions may be arbitrarily interleaved, a shared action must be executed at the same time by all processes that participate in the shared action.

```
MAKER = (make->ready->MAKER).
USER  = (ready->use->USER).

||MAKER_USER = (MAKER || USER).
```

MAKER synchronizes with **USER** when **ready**.

LTS? Traces? Number of states?

Class Exercise

```
MAKER = (make->ready->MAKER).
USER  = (ready->use->USER).

||MAKER_USER = (MAKER || USER).
```

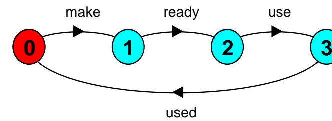
LTS? Traces? Number of states?

Need to consider the communication via shared actions.

modeling interaction - handshake

A handshake is an action acknowledged by another:

```
MAKERv2 = (make->ready->used->MAKERv2).   3 states
USERv2  = (ready->use->used ->USERv2).    3 states

||MAKER_USERv2 = (MAKERv2 || USERv2).     3 x 3 states?
```



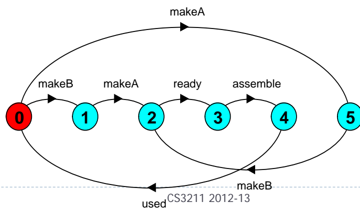
4 states

Interaction constrains the overall behaviour.

modeling interaction - multiple processes

Multi-party synchronization:

```
MAKE_A   = (makeA->ready->used->MAKE_A).
MAKE_B   = (makeB->ready->used->MAKE_B).
ASSEMBLE = (ready->assemble->used->ASSEMBLE).

||FACTORY = (MAKE_A || MAKE_B || ASSEMBLE).
```



composite processes

A composite process is a parallel composition of primitive processes. These composite processes can be used in the definition of further compositions.

```
||MAKERS = (MAKE_A || MAKE_B).

||FACTORY = (MAKERS || ASSEMBLE).
```

Substituting the definition for **MAKERS** in **FACTORY** and applying the **commutative** and **associative** laws for parallel composition results in the original definition for **FACTORY** in terms of primitive processes.

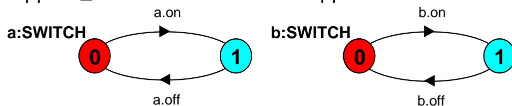
```
||FACTORY = (MAKE_A || MAKE_B || ASSEMBLE).
```


process labeling

a:P prefixes each action label in the alphabet of P with a.

Two **instances** of a switch process:

```
SWITCH = (on->off->SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```



An array of **instances** of the switch process:

```
||SWITCHES(N=3) = (forall[i:1..N] s[i]:SWITCH).
||SWITCHES(N=3) = (s[i:1..N]:SWITCH).
```

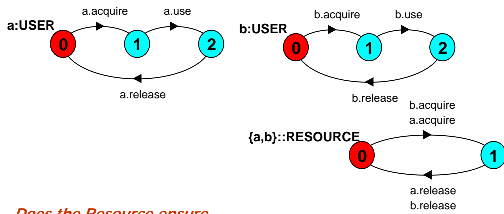

process labeling by a set of prefix labels

$\{a_1,..,a_x\}$::P replaces every action label n in the alphabet of P with the labels a_1.n,…,a_x.n. Further, every transition (n->X) in the definition of P is replaced with the transitions ($\{a_1$.n,…,a_x.n$\}$ ->X).

Process prefixing is useful for modeling **shared** resources:

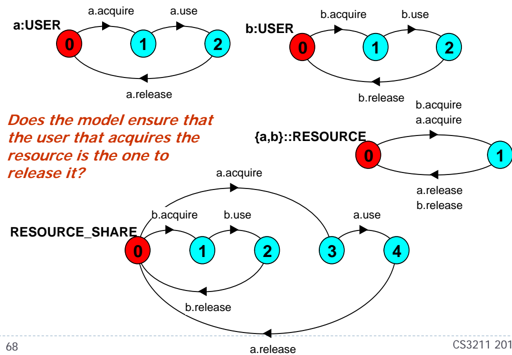
```
RESOURCE = (acquire->release->RESOURCE).
USER = (acquire->use->release->USER).

RESOURCE_SHARE = (a:USER || b:USER
                  || {a,b}::RESOURCE).
```

process prefix labels for shared resources



Does the Resource ensure that the user that acquires the resource is the one to release it?

process prefix labels for shared resources



Does the model ensure that the user that acquires the resource is the one to release it?

action relabeling

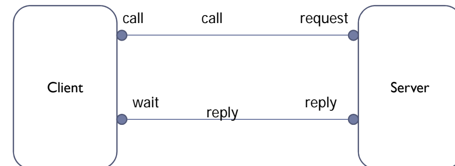
Relabeling functions are applied to processes to change the names of action labels. The general form of the relabeling function is:

/{*newlabel_1*/*oldlabel_1*,… *newlabel_n*/*oldlabel_n*}.

Relabeling to ensure that composed processes synchronize on particular actions.

```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).
```


Client – Server (Port View)

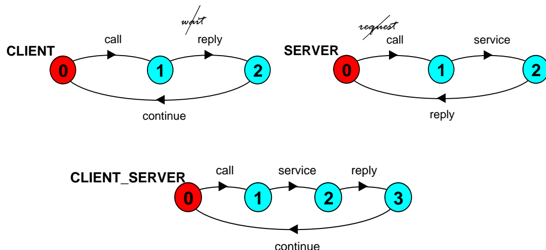


These "connections" hint at action ***relabeling***.

Actions not mentioned in this diagram are ***internal*** to a process! While they are not shared actions, they are still visible without explicit ***hiding***.

Action relabeling

```
CLIENT_SERVER = (CLIENT || SERVER)
                /{call/request, reply/wait}.
```



Action relabeling - prefix labels

An alternative formulation of the client server system is described below using qualified or prefixed labels:

```
SERVERv2 = (accept.request
           ->service->accept.reply->SERVERv2).
CLIENTv2 = (call.request
           ->call.reply->continue->CLIENTv2).

CLIENT_SERVERv2 = (CLIENTv2 || SERVERv2)
                  /{call/accept}.
```

action hiding - abstraction to reduce complexity

When applied to a process P, the hiding operator $\backslash\{a_1 \ldots a_x\}$ removes the action names $a_1 \ldots a_x$ from the alphabet of P and makes these concealed actions "silent". These silent actions are labelled tau. Silent actions in different processes are not shared.

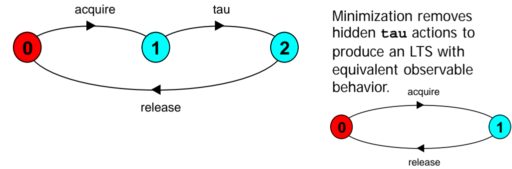
Sometimes it is more convenient to specify the set of labels to be exposed....

When applied to a process P, the interface operator $@\{a_1 \ldots a_x\}$ hides all actions in the alphabet of P not labelled in the set $a_1 \ldots a_x$.

action hiding

The following definitions are equivalent:

```
USER = (acquire->use->release->USER)
       \{use}.

USER = (acquire->use->release->USER)
       @{acquire,release}.
```



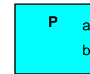
Class Exercise

```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).
```

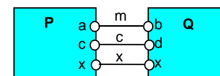
```
CLIENT_SERVERv3 = ((CLIENT || SERVER)
                  /{call/request, reply/wait}
                  )\{call, reply}
```

Construct the LTS for CLIENT_SERVERv3

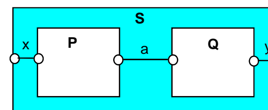
structure diagrams



Process P with alphabet {a,b}.



Parallel Composition (P||Q) / {m/a,m/b,c/d}



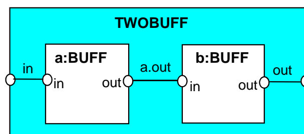
Composite process ||S = (P||Q) @ {x,y}

structure diagrams

We use structure diagrams to capture the structure of a model expressed by the fundamental operations: *parallel composition*, *relabeling* and *hiding*.

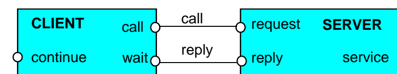
```
range T = 0..3
BUFF = (in[i:T]->out[i]->BUFF).
||TWOBUF = ?
```

Do it in class



structure diagrams

Structure diagram for **CLIENT_SERVER** ?

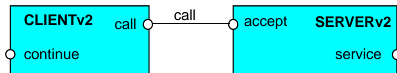



```
CLIENT_SERVER = (CLIENT || SERVER)
                /{call/request, reply/wait}.

CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).
```

Structure diagram for CLIENT_SERVERv2??

```
SERVERv2 = (accept.request
            ->service->accept.reply->SERVERv2).
CLIENTv2 = (call.request
            ->call.reply->continue->CLIENTv2).

CLIENT_SERVERv2 = (CLIENTv2 || SERVERv2)
                  /{call/accept}.
```

Structure diagram for **CLIENT_SERVERv2** ?



Structure diagram for CLIENT_SERVERv3??


```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).

CLIENT_SERVERv3 = ((CLIENT || SERVER)
                  /{call/request, reply/wait}
                  )\{call, reply}
```

Exercise: Can you now construct the Structure diagram for CLIENT_SERVERv3??

Exercise

```
RESOURCE = (acquire->release->RESOURCE).
USER =     (printer.acquire->use
            ->printer.release->USER).

||PRINTER_SHARE
  = (a:USER||b:USER||{a,b}::printer:RESOURCE).
```

Draw the state machine of the composed process defined above.

What do we gain from such diagrams?

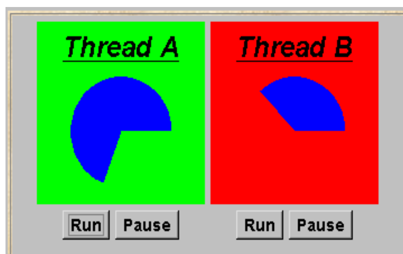
A "port" view of a concurrent system.

It clearly marks the following information.

- the processes inside the system.
- the ports or input/output channels for each process.
- the connections among the ports
  [who talks to whom and exchanges what information?]
- the "external environment" for each process
  [the other processes, consider the physical environment too!]
- visibility of actions to the external environment.

Multi-threaded Programs in Java

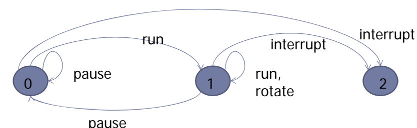
Concurrency in Java occurs when more than one thread is alive. ThreadDemo has two threads which rotate displays.



ThreadDemo Example

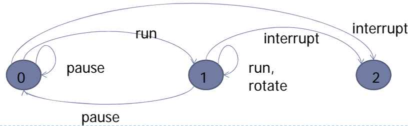
Two threads in the program A, B
--- (do not forget the main thread of course)

Lifecycle of thread A, B
Running – display associated with it rotates (background = green)
Paused – Rotation stops (background = red)
Communication
Thread A with main thread.
Thread B with main thread.

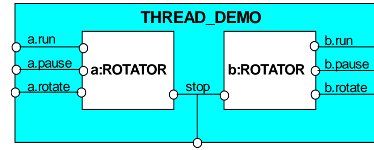


The two descriptions

```
ROTATOR = PAUSED,
PAUSED  = (run->RUN | pause->PAUSED
          |interrupt->STOP),
RUN     = (pause->PAUSED |{run,rotate}->RUN
          |interrupt->STOP).

||THREAD_DEMO = (a:ROTATOR || b:ROTATOR)
 /{stop/{a,b}.interrupt}.
```



ThreadDemo – Structure Diagram




```
ROTATOR = PAUSED,
PAUSED  = (run->RUN | pause->PAUSED
          |interrupt->STOP),
RUN     = (pause->PAUSED |{run,rotate}->RUN
          |interrupt->STOP).

||THREAD_DEMO = (a:ROTATOR || b:ROTATOR)
 /{stop/{a,b}.interrupt}.
```

Interpret* `run`, `pause`, `interrupt` *as inputs,* `rotate` *as an output.

Rotator class

```
class Rotator implements Runnable {

  public void run() {
    try {
      while(true) ThreadPanel.rotate();
    } catch(InterruptedException e) {}
  }
}
```

`Rotator` implements the `runnable` interface, calling `ThreadPanel.rotate()` to move the display.

`run()` finishes if an exception is raised by `Thread.interrupt()`.

ThreadPanel class

```
public class ThreadPanel extends Panel {

  // construct display with title and segment  color c
  public ThreadPanel(String title, Color c) {…}

  // rotate display of currently running thread 6 degrees
  // return value not used in this example
  public static boolean rotate()
         throws InterruptedException {…}

  // create a new thread with target r and start it running
  public void start(Runnable r) {
        thread = new DisplayThread(canvas,r,…);
        thread.start();
  }

  // stop the thread using Thread.interrupt()
  public void stop() {thread.interrupt();}
}
```

`ThreadPanel` manages the display and control buttons for a thread.

Calls to `rotate()` are delegated to `DisplayThread`.

Threads are created by the `start()` method, and terminated by the `stop()` method.

ThreadDemo class

```
public class ThreadDemo extends Applet {
  ThreadPanel A; ThreadPanel B;

  public void init() {
    A = new ThreadPanel("Thread A",Color.blue);
    B = new ThreadPanel("Thread B",Color.blue);
    add(A); add(B);
  }

  public void start() {
    A.start(new Rotator());
    B.start(new Rotator());
  }

  public void stop() {
    A.stop();
    B.stop();
  }
}
```

`ThreadDemo` creates two `ThreadPanel` displays when initialized and two threads when started.

`ThreadPanel` is used extensively in later demonstration programs.

Summary

- Concepts
  - concurrent processes and process interaction
- Models
  - Asynchronous (arbitrary speed) & interleaving (arbitrary order).
  - Parallel composition as a finite state process with action interleaving.
  - Process interaction by shared actions.
  - Process labeling and action relabeling and hiding.
  - Structure diagrams
- Practice
  - Multiple threads in Java.

Question asked after Lecture 3 of CS 3211

Abhik Roychoudhury
National University of Singapore

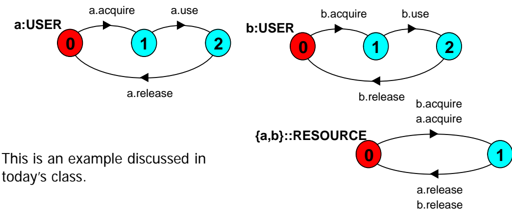
Q. From post-it note

- If JVM is running inside one OS process, are the threads of a Java program truly parallel?
- Answer
  - The threads will be time-shared. At any point, you can assume that one thread is scheduled to run. There might be other threads which are schedulable – but only one is running. This corresponds to the sub-states of the **Alive** state in the thread life cycle discussed in today's lecture.

Q. Concurrency and parallelism

- Why interleaved execution simulates the parallel behavior irrespective of processor speeds?
- Answer: Consider two processes
  - A = a -> A
  - B = b -> B
  - If they are running in two processors a could finish before b, and vice-versa.
  - This captured by two interleavings (global traces)
    - a -> b -> …
    - b -> a -> …

Q. Reasoning from per-process code



This is an example discussed in today's class.

1. From the RESOURCE process, we can see that after an acquire action, a release action must be executed, before another acquire action can happen.
2. We can also see that *a.acquire* cannot be immediately followed by *b.release* in the RESOURCE process --- since *b.release* is a shared action and for it to take place in the b::USER --- there should have been *b.acquire* prior to it

Important note

- While doing multi-threaded programming
  - We are often resorting to such per-process reasoning.
  - It would be nice to have a global state machine and perform verification – but realistically this is often not done.
  - For this reason, it is important for the programmer to at least do such per-process reasoning (or some limited reasoning about communication) while writing the code.

Q. Atomic actions

- You said
  - RADIO = on -> off -> RADIO
  - Here on, off are atomic actions. How do you know?
- Answer:
  - It is the other way round. Whatever is atomic, I show as an action in the process equations.
  - Now in reality, the **on** action could be a method call in Java, say **on()**, which is atomically executed because it is written as a synchronized method.
    - **synchronized on(){ …**
  - This also shows some of the linkage between process equations and multi-threaded Java code.