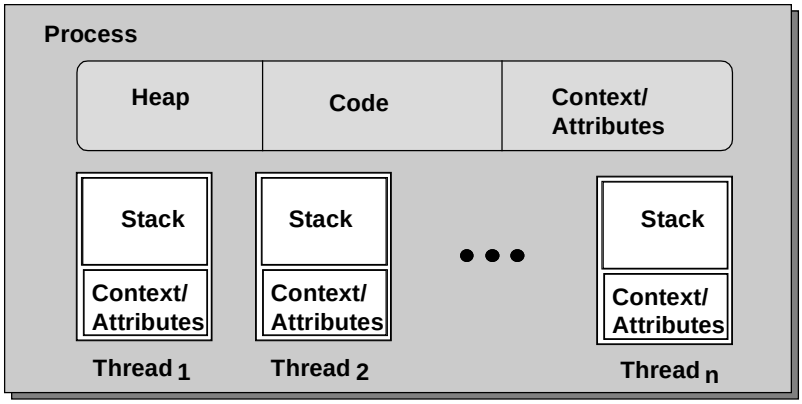
Communications Software Engineering Condition Synchronization and Liveness

Wolfgang Emmerich

Lecture Overview

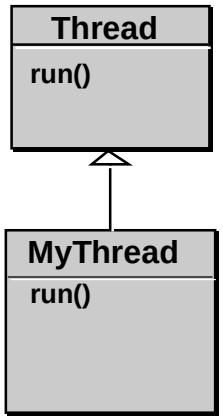
- *Concurrency in Programming Languages*
- *Mutual Exclusion*
- *Condition Synchronization*
- *Deadlocks*
- *Liveness*

Threads and OS Processes



- *OS process provides protected address space.*
- *Many threads may execute within space.*
- *Each thread: stack & context (saved registers).*

Threads using Inheritance

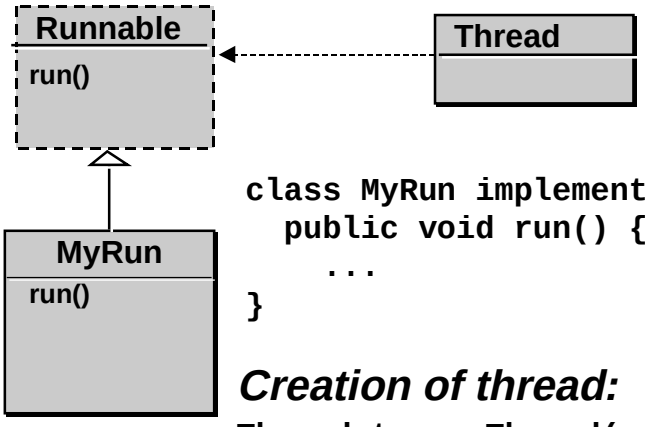



```
class MyThread extends Thread {
  public void run() {
    ...
}
```

Creation of thread:

```
MyThread t=new MyThread();
```

Threads implementing Interfaces




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

Creation of thread:

```
Thread t=new Thread(new MyRun);
```

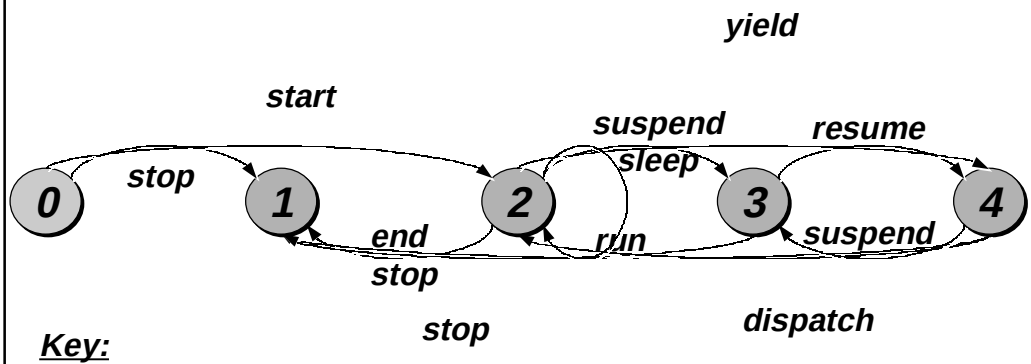

Thread Lifecycle

- ***Started by start() which invokes run()***
- ***Terminated when***
  - ***run() returns or***
  - ***explicitly terminated by stop().***
- ***A started thread may be***
  - ***running or***
  - ***runnable (waiting to be scheduled)***
- ***Thread gives up processor using yield().***
- ***A thread may be suspended by suspend()***
- ***If Suspended gets runnable by resume().***
- ***sleep() suspends for a given time and then resumes***

FSP Model of Java Thread Lifecycle

```
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.
```


LTS of Java Thread Lifecycle



Key:

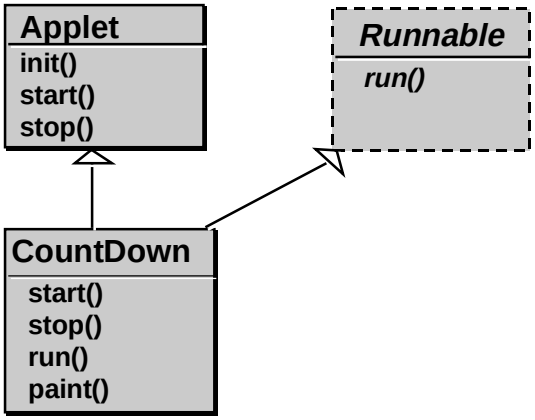
0: CREATED
1: TERMINATED
2: RUNNING
3: NON_RUNNABLE
4: RUNNABLE

Example: CountDown Timer

- ***Demo: CountDown***
- ***FSP of CountDown:***

```
COUNTDOWN (N=60) = COUNTDOWN[N],
COUNTDOWN[i:0..N] =
 ( when(i>0) tick->COUNTDOWN[i-1]
 | when(i==0) beep->STOP
 ).
```


CountDown Timer - Class diagram



`Runnable` *is an interface*

CountDown Timer - Java class

```
import java.awt.*;            //windows toolkit
import java.applet.*;         //applet support
public class CountDown extends Applet implements Runnable{
  int counter; Thread cd;
  public void start() {     // create thread
   counter = 60;  cd = new Thread(this); cd.start();
  }
  public void stop() { cd = null;}
  public void run() {      // executed by Thread
   while (counter>0 && cd!=null) {
    try{Thread.sleep(1000);}
    catch (InterruptedException e){}
    --counter;  repaint(); //update screen
   }
  }
  public void paint(Graphics g) {
    if (counter>0)
      g.drawString(String.valueOf(counter),25,75);
    else g.drawstring("Bang", 10, 50);
  }
}
```

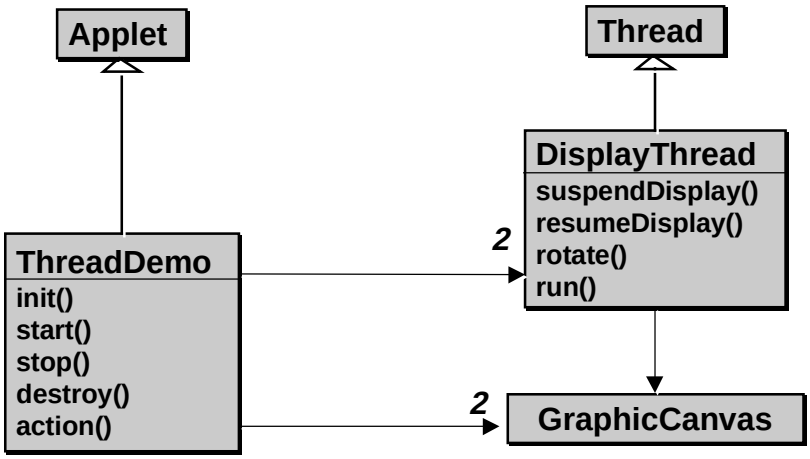

Concurrent Threads

- *Parallel composition operator | |*
- *Implemented by creation of several new thread objects*
- *Example: ThreadDemo*
- *Creates two thread objects that execute concurrently*

FSP Spec of Thread Demo

```
DISPLAY_THREAD = SUSPENDED,
SUSPENDED = ( resume->RUNNING ),
RUNNING =   ( rotate->RUNNING
            | suspend->SUSPENDED
            ).
||THREAD_DEMO =
   (a:DISPLAY_THREAD||b:DISPLAY_THREAD).
```


Class Diagram of ThreadDemo



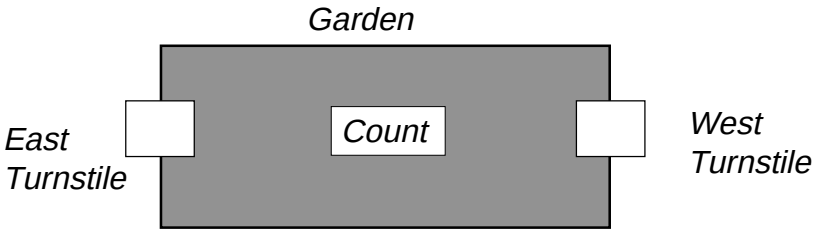
Summary

- *Threads vs. operating system processes*
- *Threads through class inheritance / interface implementation*
- *Thread lifecycle*
- *Concurrent threads by creating new thread objects*
- *Class diagrams*
- *Next: Java Thread Programming Lab*

Mutual Exclusion

Ornamental Garden Problem

- *Garden open to the public*
- *Enter through either one of two turnstiles*
- *Computer to count number of visitors*



- *Each turnstile implemented by a thread*

Ornamental Garden: Counter class

```
class Counter {
    int value_=0;
    public void increment() {
        int temp = value_; //read
        Simulate.interrupt();
       ++temp;             //add1
        value_=temp;       //write
    }
}
```

- *Simulated interrupt calls* `yield()` *to force thread switch.*

Ornamental Garden: Turnstile class

```
class Turnstile extends Thread {
    Counter people_;
    Turnstile(Counter c) {
        people_ = c;
    }
    public void run() {
        while(true)
            people_.increment();
    }
}
```

- *For full implementation see online version*

Ornamental Garden: Program

```
Counter people_ = new Counter();
Turnstile west_ = new Turnstile(people_);
Turnstile east_ = new Turnstile(people_);
west_.start();
east_.start();
```

- *What will happen?*

Demo: Ornamental Garden

FSP Spec of Ornamental Garden

```
const N = 3 range T = 0..N
VAR = VAR[0],
VAR[u:T] = (read[u] -> VAR[u]
             | write[v:T]-> VAR[v]).
TURNSTILE = ( arrive -> INCREMENT
            | suspend-> resume-> TURNSTILE),
INCREMENT = (val.read[x:T] -> val.write[x+1]->
       TURNSTILE)+{val.read[T],val.write[T]}.
||GARDEN = (east:TURNSTILE || west: TURNSTILE
           || {east,west}::val:VAR
          )/{stop/east.suspend,
             stop/west.suspend,
             start/east.start,
             start/west.start}.
```

LTSA

Interference

- ***FSP spec supports the following trace:***

  east.arrive→east.val.read.0→west.arrive→

  west.val.read.0→east.val.write.1→west.val.write.1

- ***This is an example of a destructive update***
- ***Destructive updates caused by arbitrary interleaving of read and write actions on shared variables is called <u>interference</u>***
- ***Avoid interference by making access to <u>critical sections</u> <u>mutually exclusive</u>***

Critical Section

- *A critical section is a sequence of actions that must be executed by at most one process or thread at a time*
- *Can be found by searching for sections of code that access or update variables or objects that are shared by concurrent processes.*

Modelling Mutual Exclusion

- *A lock can be modelled by:*

```
LOCK = (acquire->release->LOCK).
```

- *Attaching lock to shared resource (VAR):*

```
||LOCKVAR = (LOCK || VAR ).
```

- *Critical section acquires/releases lock:*

```
INCREMENT = (value.acquire ->
   val.read[x:T] -> val.write[x+1]->
   value.release -> TURNSTILE)
   +{val.read[T],val.write[T]}.
```

Critical Sections in Java

- *Synchronised methods implement mutual exclusion*
- *Implicitly locking objects*

```
class Counter {
    int value_=0;
    public synchronized void increment() {
        int temp = value_; //read
        Simulate.interrupt();
        ++temp;            //add1
        value_=temp;       //write
    }
}
```

Demo: Correct Ornamental Garden

Synchronised Statements in Java

- *Locks on individual objects:*

```
public void run() {
    while(true)
   synchronized(people_){
            people_.increment();
      }
}
```

- *Less elegant than synchronized methods*
- *More efficient than synchronized methods*

Semaphores and Monitors

Semaphores

- *Introduced by Dijkstra' in 1968*
- *ADT with counter and waiting list*

P/Wait/Down:

```
if (counter > 0)
  counter--
else
  add caller to
  waiting list
```

S/Signal/Up:

```
if (threads wait)
  activate waiting
  thread
else
  counter++
```

Semaphores and Mutual Exclusion

- *One semaphore for each critical section*
- *Initialize semaphore to 1.*
- *Embed critical sections in wait/signal pair*
- *Example in Java:*

```
Semaphore S=new Semaphore(1);
S.down();
<critical section>
S.up();
```

Demo: Semaphores

Evaluation of Semaphores

+ *Nice and simple mechanism*
+ *Can be efficiently implemented*
+ *Available in every programming language*
– *Too low level of abstraction*
– *Unstructured use of signal and wait leads to spaghetti synchronisation*
– *Error prone and errors are dangerous*
  – *Omitting signal leads to deadlocks*
  – *Omitting wait leads to safety violations*

Semaphore in Java

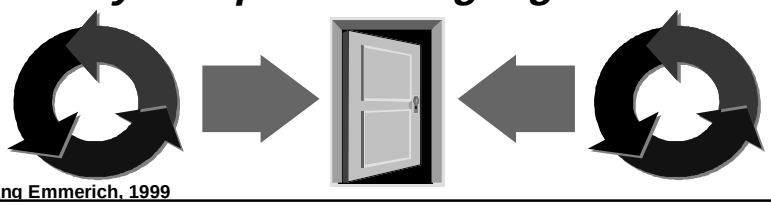
```
class Semaphore {
    private int value_;
    Semaphore (int initial) {
        value_ = initial;
    }
    synchronized public void up() {
        ++value_;
        notify();
    }
    synchronized public void down() {
        while (value_== 0) {
            try {wait();}
            catch (InterruptedException e){}
        }
        --value_;
    }
}
```


Critical Regions

- Guarantee mutual exclusion by definition
- Note subtle difference to critical sections
- language features implement critical regions
- Example: Java synchronised method

Monitors

- *Hoare's response to Dijkstra's semaphores*
  - *Higher-level*
  - *Structured*
- *Monitors encapsulate data structures that are not externally accessible*
- *Mutual exclusive access to data structure enforced by compiler or language run-time*



Monitors in Java

- *All instance and class variables need to be* `private` *or* `protected`
- *All methods need to be* `synchronised`
- *Example: semaphore implementation*
- *Use of Monitors: Carpark Problem*

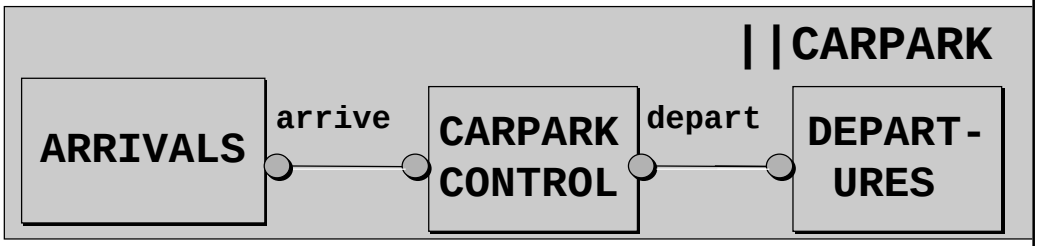
Carpark Problem

- *Only admit cars if carpark is not full*
- *Cars can only leave if carpark is not empty*
- *Car arrival and departure are independent threads*

Demo: CarPark

Carpark Model

- *Events or actions of interest:*
  - *Arrive and depart*
- *Processes:*
  - *Arrivals, departures and carpark control*
- *Process and Interaction structure:*



Carpark FSP Specification

```
CARPARKCONTROL(N=4) = SPACES[N],
SPACES[i:0..N] =
       (when(i>0) arrive-> SPACES[i-1]
       |when(i<N) depart-> SPACES[i+1]
       ).
ARRIVALS = (arrive-> ARRIVALS).
DEPARTURES = (depart-> DEPARTURES).
||CARPARK =
 (ARRIVALS||CARPARKCONTROL||DEPARTURES).
```

LTSA

Java Class Carpark

```
public class Carpark extends Applet  {
 final static int N=4;
 public void init() {
  CarParkControl cpk = new CarParkControl(N);
  Thread arrival,departures;
  arrivals=new Thread(new Arrivals(cpk));
  departures=new Thread(new Departures(cpk));
  arrivals.start();
  departures.start();
 }
}
```

Java Classes Arrivals & Departures

```
public class Arrivals implements Runnable {
    CarParkControl carpark;
    Arrivals(CarParkControl c) {carpark = c;}
    public void run() {
     while (true) carpark.arrive();
   }
}
class Departures implements Runnable {
 ...
 public void run() {
    while (true) carpark.depart();
}
```


Java Class CarParkControl (Monitor)

```
 class CarParkControl {// synchronisation?
   private int spaces;
   private int N;
   CarParkControl(int capacity) {
     N = capacity;
     spaces = capacity;
   }
   synchronized public void arrive() {
     … -- spaces; … } {// Block if full?
   synchronized public void depart() {
     … ++ spaces; …   {// Block if empty?
   }
 }
```

Problems with CarParkControl

- How do we send arrivals to sleep if car park is full?
- How do we awake it if space becomes available?
- Solution: Condition synchronisation

Condition Synchronization

Thread Waiting Queues in Java

- `public final void notify()`

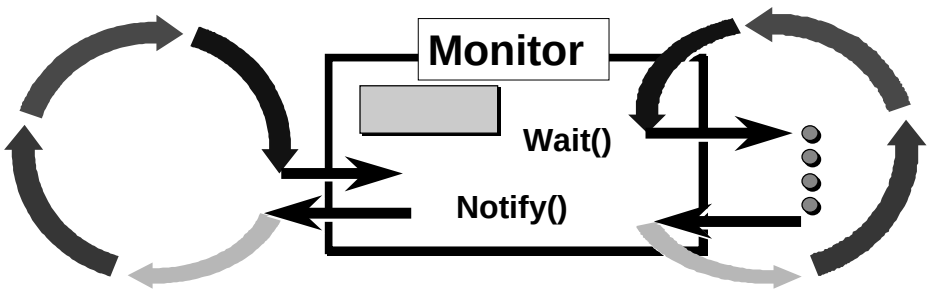
  Wakes up a single thread that is waiting on this object's queue
- `public final void notifyAll()`

  Wakes up all threads that are waiting on this object's queue
- `public final void wait()`
  `throws InterruptedException`

  Waits to be notified by another thread when notified must reacquire monitor

Condition synchronisation in Java

- *Thread enters monitor when it acquires mutual exclusion lock of monitor*
- *Thread exits monitor when releasing lock*
- *Wait causes thread to exit monitor*



Semaphore as a Java Monitor

```
class Semaphore {
  private int value_;
  Semaphore (int initial) {
    value_=initial;
  }
  public synchronised up() {
    ++value_;
    notify();
  }
  public synchronised down() {
    while (value_==0) wait();
    --value;
  }
}
```


Condition Synchronisation in Java

- *FSP Model: when cond act -> NEWSTATE*
- *Java:*

```
public synchronized void act()
throws InterruptedException
{
  while (! cond) wait();
  // modify monitor data
  notifyAll();
}
```

- *Loop re-tests cond to make sure that it is valid when it re-enters the monitor*

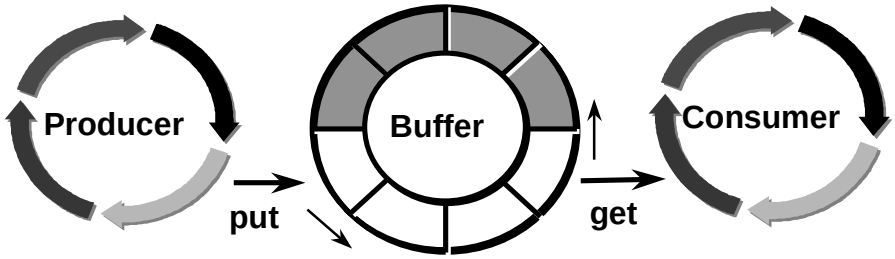
CarParkControl revisited

```
class CarParkControl {
  private int spaces;
  private int N;
  synchronized public void arrive() {
  while (spaces<=0) {
    try {
     wait();
    } catch(InterruptedException e){}
  }
  --spaces;
  notify();
 }
```


FSP and Condition Synchronisation

- *For each guarded action in the FSP model of a monitor*
  - *Implement action as a synchronised method*
  - *That invokes* `wait()` *in a while loop before it begins*
  - *While condition is negation of guard condition*
- *Every change in the monitor are signalled to waiting threads using* `notify()` *or* `notifyAll()`

Example: Producer/Consumer



Demo

Producer Consumer in FSP

```
PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).
BUFFER(SIZE=5) = BUFFER[0],
BUFFER[count:0..SIZE] = (
       when (count<SIZE) put->BUFFER[count+1]
      |when (count>0) get -> BUFFER[count-1]).
||PC=(PRODUCER||BUFFER||CONSUMER).
```

LTSA

Bounded Buffer - Outline

```
class Buffer {
  private protected Object[] buf;
  private protected int in = 0;//index put
  private protected int out = 0;//index get
  private protected int count = 0; //no items
  private protected int size;
  Buffer(int size) {
      this.size = size;
      buf = new Object[size];
  }
  synchronized public void put(Object o) {…}
  synchronized public Object get() {…}
}
```


Bounded Buffer - put

```
synchronized public void put(Object o) {
  while (count==size) {
     try {
       wait();
     } catch(InterruptedException e){}
  }
  buf[in] = o;
  ++count;
  in=(in+1) % size;
  notify(); // [count>0]
}
```

Bounded Buffer - get

```
synchronized public Object get() {
  while (count==0) {
     try {
       wait();
     } catch (InterruptedException e){}
  }
  Object o =buf[out];
  buf[out]=null; // for display purposes
  --count;
  out=(out+1) % size;
  notify(); // [count < size]
  return (o);
}
```


Monitor Invariants

- *Monitor invariant is assertion concerning attributes encapsulated by monitor*
- *Assertion must hold when no thread is in monit*
- *Examples:*
  - *CarParkControl: 0<=spaces<=N*
  - *Semaphore: 0<=value*
  - *BoundedBuffer: (0<=count && 0<=in<=size && 0<=out<=size && in=(out+count) % size)*
- *Used to reason about correctness monitors*

Summary

- *Condition synchronization*
- *In Java using* `wait()`, `notify()` *and* `notifyAll()`
- *Used to implement Semaphores in Java*
- *Relation between FSP model and implementation in Java monitor*
- *Monitor invariants*

Deadlocks

Goals

- *Reader/Writer problem*
- *Starvation*
- *Dining Philosophers Problem*
- *Deadlocks*
- *Liveness Analysis using LTS*

Reader / Writer Problem

- *Monitors and Java's synchronize statement guarantee mutual access to objects / methods*
- *Often it is ok for multiple readers to access the object concurrently*
- *Properties required:*

Demo: Reader/Writer

Read/Write Monitor

```
class ReadWrite {
 private protected int readers = 0;
 private protected boolean writing = false;
 // Invariant: (readers>=0 and !writing) or
 // (readers==0 and writing)
 synchronized public void acquireRead() {
  while (writing) {… wait(); …} ++readers;
 }
 synchronized public void releaseRead() {
    --readers; if(readers==0) notify();
 }
 synchronized public void acquireWrite() {
  while (readers>0||writing) {… wait(); …}
  writing = true;
 }
 synchronized public void releaseWrite() {
  writing = false; notifyAll();
 }
}
```

Starvation

Writer Starvation

- *NotifyAll awakes both readers and writers*
- *Program relies on Java having a fair scheduling strategy*
- *When readers continually read resource:*

  Writer never gets chance to write. This is an example of starvation.
- *Solution: Avoid writer starvation by making readers defer if there is a writer waiting*

Read/Write Monitor (Version 2)

```
class ReadWrite {
 … // as before
 private int waitingW = 0;// # waiting Writers
 synchronized public void acquireRead() {
  while (writing || waitingW>0) {… wait(); … }
  ++readers;
 }
 synchronized public void releaseRead() {… }
 synchronized public void acquireWrite() {
  while (readers>0 || writing) {
  ++waitingW; … try{ wait(); … --waitingW; }
  writing = true;
 }
 synchronized public void releaseWrite() {… }
}
```

Demo: Reader/Writer v2

Reader Starvation

- ***If there is always a waiting writer:***

  Readers starve
- ***Solution: Alternating preference between readers and writers***
- ***To do so: Another*** `boolean` ***attribute*** `readersturn` ***in Monitor that indicates whose turn it is***
- `readersturn` ***is set by*** `releaseWrite()` ***and cleared by*** `releaseRead()`

Read/Write Monitor (Version 3)

```
class ReadWrite {
 … // as before
 private boolean readersturn = false;
 synchronized public void acquireRead() {
  while(writing ||(waitingW>0 && !readersturn))
     { … wait(); … }
  ++readers;
 }
 synchronized public void releaseRead() {
  --readers;  readersturn=false;
  if(readers==0) notifyAll();
 }
 synchronized public void acquireWrite() {… }
 synchronized public void releaseWrite() {
  writing=false; readersturn=true; notifyAll();
  }
}
```

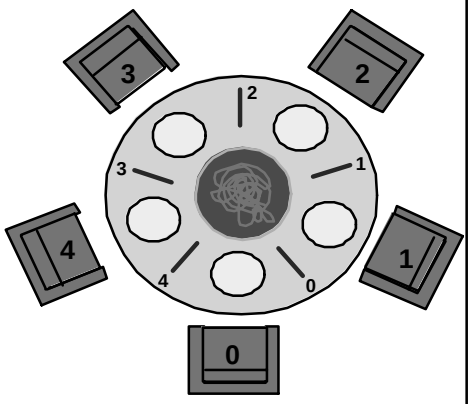

Demo: Reader/Writer v3

Deadlocks

- *Process is in a deadlock if it is blocked waiting for a condition that will never become true*
- *Process is in a livelock if it is spinning while waiting for a condition that will never become true (busy wait deadlock)*
- *Both happen if concurrent processes and threads are mutually waiting for each other*
- *Example: Dining philosophers*

Dining Philosopher Problem

- *5 Philosophers sit around table*
- *They think or eat*
- *Eat with 2 chopsticks*
- *Only 5 chopsticks available*
- *Each philosopher only uses sticks to her left and right*



FSP Model of Dining Philosophers

```
PHIL=(hungry->left.get->right.get->eating->
      left.put->right.put->thinking->PHIL).
FORK = (left.get-> left.put -> FORK
       |right.get->right.put -> FORK).
||COLLEGE(N=5)=
(phil[0..N-1]:PHIL||fork[0..N-1]:FORK)
 /{phil[i:0..N-1].left/fork[i].left,
  phil[i:0..N-1].right/fork[((i-1)+N)%N].right}.
```

LTSA

Dining Philosophers in Java

```
class Philosopher extends Thread {
  int identity;
  Chopstick left; Chopstick right;
  Philosopher(Chopstick left,Chopstick right){
    this.left = left; this.right = right;
  }
  public void run() {
   while (true) {
    try {
     sleep(…);                // thinking
     right.get(); left.get(); // hungry
     sleep(…) ;               // eating
     right.put();  left.put();
    } catch (InterruptedException e) {}
   }
  }
 }
```


Chopstick Monitor

```
 class Chopstick {
  boolean taken=false;
  synchronized void put() {
   taken=false;
   notify();
  }
  synchronized void get() throws
                      InterruptedException {
   while (taken) wait();
      taken=true;
   }
 }
```

Applet for Diners

```
for (int i =0; i<N; ++I)
  // create Chopsticks
  stick[i] = new Chopstick();
for (int i =0; i<N; ++i){
  // create Philosophers
  phil[i]=new Philosopher(
             stick[(i-1+N)%N],stick[i]);
  phil[i].start();
}
```

Demo: Diners

Deadlock in Dining Philosopher

- *If each philosopher has acquired her left chopstick the threads are mutually waiting for each other*
- *Potential for deadlock exists independent of thinking and eating times*
- *Only probability is increased if these times become shorter*

Analysing cause of Deadlock

- *We can use LTS for deadlock analysis*
- *A dead state in the composed LTS is one that does not have outgoing transitions*
- *Are these dead states reachable?*
- *Use of reachability analysis*
- *Traces to dead states helps understanding the causes of a deadlock*

LTSA

Deadlock Avoidance

- *Deadlock in dining philosophers can be avoided if one philosopher picks up sticks in reverse order (right before left).*

Demo: Deadlock free Diners

- *What is the problem with this solution?*
- *Are there other solutions?*
- *Deadlock can also be avoided if there is always one philosopher who thinks*

Summary

- *Reader / Writer Problem*
- *Starvation*
- *Avoidance of Starvation*
- *Dining Philosophers Problem*
- *Deadlocks and Livelocks*
- *Deadlock Avoidance*
- *Next Session: Safety*

Liveness & Progress

Motivation

- *Problem with single lane bridge:*
- *Cars cannot pass from north to south if there is a continuous stream of cars from south to north!*
- *We would like to guarantee that cars will eventually cross the bridge.*
- *In more general terms this is referred to as liveness*

Liveness

- *A liveness property asserts that something good eventually happens.*
- *We want to specify liveness for our FSP models*
- *We want to analyze our FSP models to be certain that the liveness properties hold*
- *General form of liveness requires consideration of temporal precedence relationship between states*
- *We use more restricted form of progress*

Progress

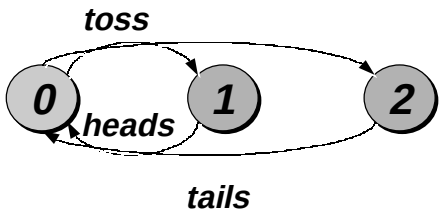
- *A progress property asserts that whatever state a system is in, it is always the case that a specified action will eventually be executed*
- *Progress is the opposite form of starvation*
- *Notion of progress is sufficiently powerful to capture wide range of liveness properties*
- *Progress properties are simple to specify in FSP*

Progress Properties in FSP

- *Specification of progress needs assumption of a fair scheduling policy.*
- *If a transition from a set is chosen infinitely often and every transition in the set will be executed infinitely often, the scheduling policy is said to be fair.*
- `progress P={a1,a2,…,an}` *defines a progress property* `P` *which asserts that in an infinite execution at least one of the actions* `a1, a2, …, an` *will be executed infinitely often.*

Example: Tossing Coins

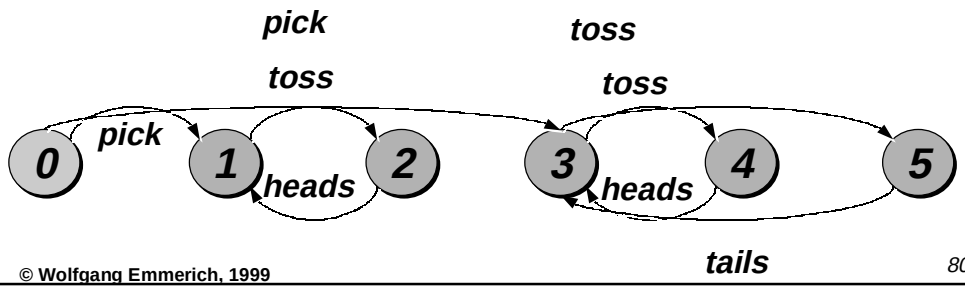
```
COIN = ( toss -> heads -> COIN
       | toss -> tails -> COIN).
```




```
progress HEADS = {heads}
progress TAILS = {tails}
```


Example: Tossing Trick Coins

```
TWOCOIN = (pick->COIN | pick->TRICK),
COIN    = ( toss -> heads -> COIN
          | toss -> tails -> COIN),
TRICK   = (toss->heads->TRICK).
progress HEADS = {heads}
progress TAILS = {tails}
```



Progress Analysis

- *We can automate analysis of progress properties*
- *A set of states where every state is reachable from every other state in the set and no state has transitions to states outside the set is a terminal set of states.*
- *Terminal set of states can be found using a graph algorithm that searches for a strongly connected component.*

LTSA

Default Progress Properties

- *Default progress properties assert in a system with fair choices that every action in the alphabet will be executed infinitely often.*
- *Default progress properties of example:*

```
progress p1 = {pick}
progress p2 = {toss}
progress p3 = {heads}
progress p4 = {tail}
```

- *How many violations?*

LTSA

Priorities

- *Default progress analysis of single lane bridge does not reveal violation.*
- *Problem is scheduling policy. Cars arriving in the south get 'priority' if there are already northbound cars on the bridge*
- *To detect such progress violations we have to reflect such priorities in the FSP model*

LTSA

High Priority in FSP

- `||C = (P||Q)<<{a1,…,an}` *specifies a composition in which the actions* `a1,…,an` *have higher priority than any other action in the alphabet of* `P||Q` *including the silent action* `tau`*. In any choice in this system which has one or more of the actions a1,…,an labelling a transition, the transitions labelled with lower priority actions are discarded.*

Low Priority in FSP

- `||C = (P||Q)>>{a1,…,an}` ***specifies a composition in which the actions*** `a1,…,an` ***have lower priority than any other action in the alphabet of*** `P||Q` ***including the silent action*** `tau`***. In any choice in this system which has one or more transitions not labelled by*** `a1,…,an`***, the transitions labelled by*** `a1,…,an` ***are discarded.***

Simplification of LTS

- ***Priorities simplify the LTS resulting of the composition.***
- ***Example:***

```
NORMAL=(work->play->NORMAL
       |sleep->play->NORMAL).
||HIGH=(NORMAL)<<{work}.
||LOW= (NORMAL)>>{work}.
```

LTSA

- ***Use of priorities lead to more realistic liveness checks.***

Summary

- *Liveness*
- *Progress*
- *Progress Specification in FSP*
- *Progress-Analysis of LTS*
- *Priorities*