Concurrency, Parallelism and Distribution (CPD)

Concurrency: Correctness of Concurrent Programs

Joaquim Gabarro
(gabarro@cs.upc.edu)

Computer Science
BarcelonaTech, Universitat Politècnica de Catalunya

Study material

Jeff Magee and Jeff Cramer
Concurrency, State Models & Java Programs (Chapter 7)
John Wiley & and Sons, 2006.
`http://www.doc.ic.ac.uk/~jnm/book/`
Most of the slides are extracted from:
`http://www.doc.ic.ac.uk/~jnm/book/slides.html`

Correctness

Safety

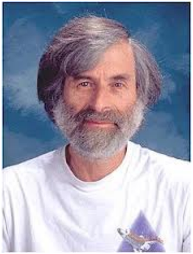
Liveness & Progress

Stress & Priority

Correctness

Correctness properties

- Safety: nothing bad happens.
- Liveness: something good eventually happens.



Leslie Lamport, 2013 Turing Award
Proving the Correctness of Multiprocess Programs
IEEE Transactions on Software Engineering,
Vol. Se-3, March 1977, 125-143.

Safety

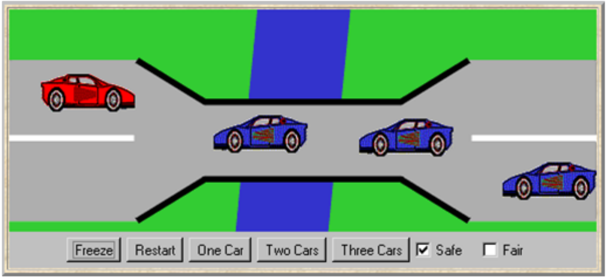
Safety

Safety property: Asserts that nothing bad happens.

- Deterministic process: Safety property P defines a deterministic process that asserts that any trace including actions in the alphabet of P, is accepted by P.
- Transparency of safety properties: Composing a property with a set of processes does not affect their correct behavior. If a behavior violates the safety property, then ERROR is reachable.
- Properties must be deterministic to be transparent.

ERROR process (-1) to detect erroneous behaviour.

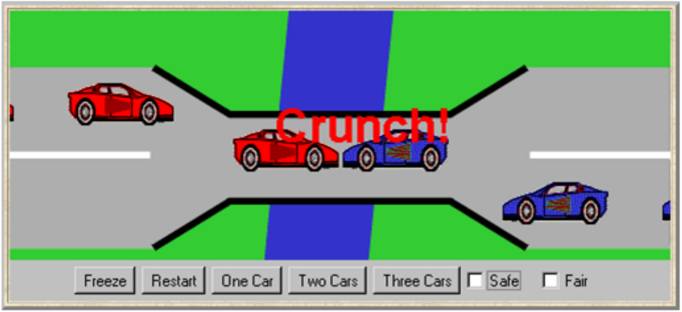
Single Lane Bridge problem



- A bridge is only wide to permit a single lane of traffic.
- Cars can only move concurrently in the same direction.
- A safety violation occurs if two cars moving in different directions enter the bridge at the same time.

Safety violation

A safety violation occurs if two cars moving in different directions enter the bridge at the same time.



Single Lane Bridge Model

||CARS = (red:CONVOY || blue:CONVOY).

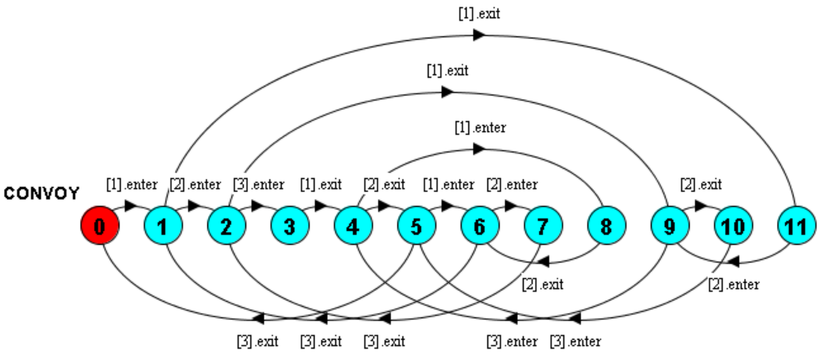
||SingleLaneBridge =
(CARS|| BRIDGE|| $\underbrace{\text{ONEWAY}}_{\textit{Safety property}}$).

Let us model the different processes:
CONVOY, BRIDGE and ONEWAY

CONVOY and CARS model

```
const N = 3 // number of each type of car
range T = 0..N // type of car count
range ID= 1..N // car identities

CAR = (enter->exit->CAR).

/* cars may not overtake each other */
NOPASS1    = C[1],
C[i:ID]    = ([i].enter -> C[i%N+1]).

NOPASS2    = C[1],
C[i:ID]    = ([i].exit -> C[i%N+1]).

||CONVOY = ([ID]:CAR || NOPASS1 || NOPASS2).

||CARS = (red:CONVOY || blue:CONVOY).
```

CONVOY, LTS



CONVOY
[1].exit
[1].exit
[1].enter
[1].enter
[2].enter
[3].enter
[1].exit
[2].exit
[1].enter
[2].enter
[2].exit
0
1
2
3
4
5
6
7
8
9
10
11
[2].exit
[2].enter
[3].exit
[3].exit
[3].exit
[3].enter
[3].enter

CONVOY, intuitive explanation

- In the initial state the bridge is empty and the convoy 1 2 3 is ready to enter, we model $\big([- \mid - \mid -]\ 1\ 2\ 3\big)$.
- Car 1 enters the bridge

$$\big([- \mid - \mid -]\ 1\ 2\ 3\big) \xrightarrow{[1].\mathit{enter}} \big([- \mid - \mid 1\ ]\ 2\ 3\big)$$

- In $\big([- \mid - \mid -]\ 1\ 2\ 3\big)$ there are two possibilitites:
  - car 1 exit de bridge

$$\big([- \mid - \mid 1\ ]\ 2\ 3\big) \xrightarrow{[1].\mathit{exit}} \big([- \mid - \mid -]\ 2\ 3\ 1\big)$$

  - car 2 enter the bridge

$$\big([- \mid - \mid 1\ ]\ 2\ 3\big) \xrightarrow{[2].\mathit{enter}} \big([- \mid 1\ \mid 2\ ]\ 3\big)$$

- Other transitions are similar.

BRIDGE model

- Cars can move concurrently only if in the same direction.
- The bridge counts the blue and red cars on the bridge.
- Red cars can enter if the blue count is zero, vice-versa.

```
BRIDGE = BRIDGE[0][0],          // initially empty
BRIDGE[nr:T][nb:T] =             //nr and nb are counters
   (when(nb==0)
      red[ID].enter -> BRIDGE[nr+1][nb]        //nb==0
     | red[ID].exit -> BRIDGE[nr-1][nb]
     | when (nr==0)
      blue[ID].enter-> BRIDGE[nr][nb+1]        // nr==0
     | blue[ID].exit -> BRIDGE[nr][nb-1]
).
```

safety property ONEWAY

We specify a safety property to check that cars do not collide!

- While red cars are on the bridge only red cars can enter.
- Similarly for blue cars.
- When the bridge is empty, either a red or a blue car may enter.

property ONEWAY

```
property ONEWAY =
    (red[ID].enter ->RED[1] | blue[ID].enter ->BLUE[1] ),

RED[i:ID] =
    // i counts the red cars
    (red[ID].enter -> RED[i+1]
     |when(i==1)red[ID].exit -> ONEWAY
     |when(i>1) red[ID].exit ->RED[i-1]),

BLUE[i:ID]=
    // i counts the blue cars
    (blue[ID].enter-> BLUE[i+1]
     |when(i==1)blue[ID].exit -> ONEWAY
     |when( i>1)blue[ID].exit ->BLUE[i-1]).
```

Single Lane Bridge - model analysis

||CARS = (red:CONVOY || blue:CONVOY).

||SingleLaneBridge = (CARS|| BRIDGE||ONEWAY).

Safety analysis verifies that ONEWAY property is not violated.

LTS Analyzer→Check→Safety→`No deadlocks/errors`

Java class Bridge

```
class Bridge{
  private int nred  = 0; private int nblue = 0;

  synchronized void redEnter()
                    throws InterruptedException {
    while (nblue>0) wait();
     ++nred;}
  synchronized void redExit(){
    --nred;
    if (nred==0) notifyAll();}
  synchronized void blueEnter()
                    throws InterruptedException {
    while (nred>0) wait();
    ++nblue;}
  synchronized void blueExit(){
     --nblue;
     if (nblue==0) notifyAll();}
}
```

Safety violation

Without the BRIDGE there is a safety violation

||Crunch = (CARS||ONEWAY).

```
Trace to property violation in ONEWAY:
       red.1.enter
       blue.1.enter
```

Class Exercise: `SUPERMARKET`

The recent launch of the drink *Sugarola* has been a success.

- `SUPERMARKET` below models a supermarket where the number of Sugarola bottles that a customer can buy is limited by the number of available bottles on the shelf.
- For instance, action `get[2]` means buying 2 bottles of Sugarola in a purchase.
- The process `WORKER` refills the shelf when Sugarola bottles are scarce (smaller or equal than `Min`).
- At any time, the maximum numbers of bottles in the shelf is `Max`.

```
const Min = 1 //defines the threshold for filling
const Max = 3 //shelf capacity

SHELF = BOT[0],
BOT[i:0..Max]
   = (when (i > 0) get[k:1..i] -> BOT[i - k]
      |when (i<= Min) fill -> BOT[Max]
      ).

WORKER = (fill->WORKER).

CLIENT = (get[1..Max]->CLIENT).

||SUPERMARKET = (SHELF || WORKER || CLIENT).
```

- Draw the `SUPERMARKET` for `Max = 3` and `Min = 1`.
- Define a safety property `NOFILLCHAINED` to show that there are no traces of `SUPERMARKET` issuing two chained fill actions.

```
property NOFILLCHAINED
  = (get[...] -> ...
     |...
     ).
```

- Check the corrrectness with:

```
||CHECK = (SUPERMARKET || NOFILLCHAINED).
```

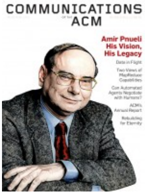
Liveness & Progress

Liveness

Liveness Property: A liveness property asserts that something good eventually happens.

A general treatment of liveness is rather involved and requieres temporal logic. A readable introduction:

Pnueli, A. Specification and Development of Reactive Systems IFIP Congress, 1986.



Amir Pnueli (1941-2009) received in 1996 the Turing Award for seminal work introducing temporal logic into computing science.

Progress

We deal with a restricted class of liveness properties called progress properties.

- A progress property asserts that it is always the case that an action is eventually executed.
- Progress is the opposite of starvation, the name given to a concurrent programming situation in which an action is never executed.

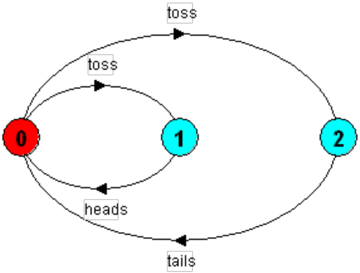
Progress property depends on the scheduling policy used to decide wich transition will be executed.

Single Lane Bridge: Every car eventually crosses the bridge.

Scheduling policy: fair choice

Fair Choice: If a choice over a set of transitions is executed infinitely often, then every transition in the set will be executed infinitely often.

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



Fair Choice: If the coin is tossed infinitely, heads is chosen infinitely and tails is chosen infinitely.

Fairness allow us to deal with likelihood without probabilities

Progress properties

Progress properties: A set P = {a1,a2..an} defines a Pprogress property P which asserts that in an infinite execution of a target system, at least one of the actions a1,a2..an will be executed infinitely often.

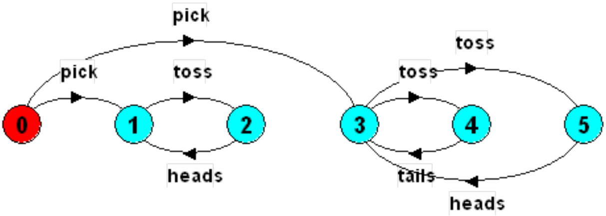
```
COIN=(toss->heads->COIN|toss->tails->COIN).
progress HEADS = {heads}
progress TAILS = {tails}
```

LTS Analyzer→ Check→Progress

```
Progress Check...
-- States: 3 Transitions: 4 Memory used: 5999K
No progress violations detected.
Progress Check in: 16ms
```

Progress violation example

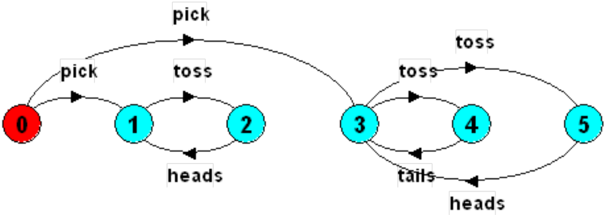
```
TWOCOIN = (pick->COIN|pick->TRICK),
TRICK   = (toss->heads->TRICK),
COIN    = (toss->heads->COIN|toss->tails->COIN).
progress HEADS = {heads}
progress TAILS = {tails}
```




```
Progress Check...
Progress violation: TAILS
Trace to terminal set of states:
pick
Actions in terminal set:
{heads, toss}
```

Progress analysis: terminal set of states

Terminal set of states: A terminal set of states is one in which every state is reachable from every other state in the set via one or more transitions, and there is no transition from within the set to any state outside the set.

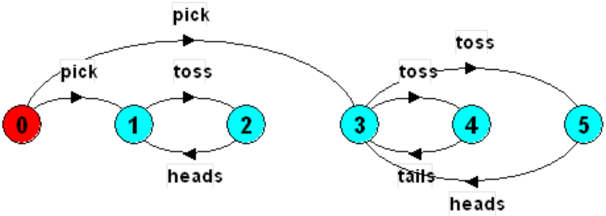


Terminal sets for TWOCOIN are $\{1, 2\}$ and $\{3, 4, 5\}$

Given fair choice, each terminal set represents an execution in the set is executed infinitely often.

Progress violation

Progress violation: A progress property is violated if analysis finds a terminal set of states in which none of the progress set actions appear.




```
Progress Check...
Progress violation: TAILS
Trace to terminal set of states:
pick
Actions in terminal set:
{heads, toss}
```

Problem with fairness (1)

```
||SingleLaneBridge = (CARS||BRIDGE||ONEWAY ).
...
progress BLUECROSS = {blue[ID].enter}
progress REDCROSS =  {red[ID].enter}
```

LTA Analyser→Check→Progress

```
No progress violations detected.
```

However (aparently) with 2 cars, red cars we can prevent the progress of blue cars as follows

Problem with fairness (2)

We have the following looping behavior with two cars

$$
\begin{aligned}
&(2\ 1\ [\ -\ |\ -\ ]\ 1\ 2) \\
&\xrightarrow{[1].enter} (2\ 1\ [\ -\ |\ 1\ ]\ 2) \\
&\xrightarrow{[2].enter} (2\ 1\ [\ 1\ |\ 2\ ]\ ) \\
&\xrightarrow{[1].exit} (2\ 1\ [\ -\ |\ 2\ ]\ 1) \\
&\xrightarrow{[1].enter} (2\ 1\ [\ 2\ |\ 1\ ]\ ) \\
&\xrightarrow{[2].exit} (2\ 1\ [\ -\ |\ 1\ ]\ 2)
\end{aligned}
$$

Looping infinitely across $(2\ 1\ [\ -\ |\ 1\ ]\ 2)$ prevents red cars to enter the bridge.

Why such a behavior is a fairness violations?

Problem with fairness (3)

Why the preceding behavior is a fairness violation?

- In state $\left(2\ 1\ [\ -\ |\ \ 1\ ]\ 2\right)$ there are two transitions

$$\left(2\ 1\ [\ -\ |\ \ 1\ ]\ 2\right) \xrightarrow{[2].enter} \left(2\ 1\ [\ 1\ |\ 2\ ]\ \right)$$
$$\left(2\ 1\ [\ -\ |\ \ 1\ ]\ 2\right) \xrightarrow{[1].exit} \left(2\ 1\ [\ -\ |\ -\ ]\ 2\ 1\right)$$

- Looping infinitely across $\left(2\ 1\ [\ -\ |\ \ 1\ ]\ 2\right)$ and never taking [1].exit is a fairness violation
- In order to avoid fairness violation [1].exit should be taken infitinely often going to $\left(2\ 1\ [\ -\ |\ -\ ]\ 2\ 1\right)$ infinitely often.
- As $\left(2\ 1\ [\ -\ |\ -\ ]\ 2\ 1\right)$ is taken infinely often [2].enter and [1].enter should be taken infinitelly often (each one).
- Eventually red cars enter the bridge.

Stress & Priority

Adverse Conditions

Fair choice means that eventually every possible execution occurs, including those in which cars do not starve.

- To detect progress problems we must check under adverse conditions.
- We superimpose some scheduling policy for actions, which models the situation in which the bridge is congested (or stressed)t.

Stress model: action priority

Action priority expressions descriAbe scheduling properties

High Priority (<<): The process ||C = (P||Q)<<{a1,...,an} specifies that actions a1,..,an have higher priority than any other action including tau.

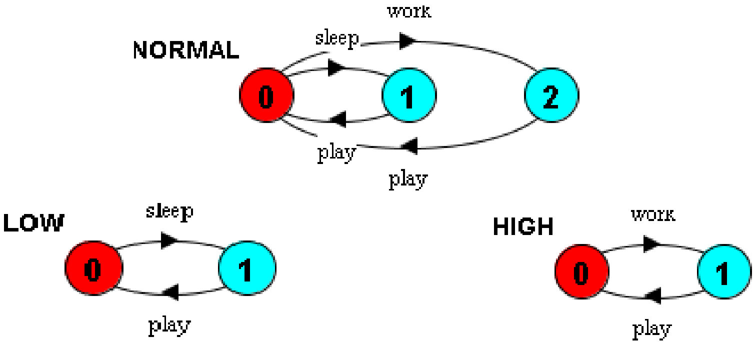
- In any choice in this system which has one or more of the actions a1,..,an labeling a transition, the transitions labeled with lower priority actions are discarded.

Low Priority (>>): A process ||C = (P||Q)>>{a1,...,an} specifies that actions a1,..,an have lower priority than any other action including tau.

- In any choice in this system which has one or more transitions not labeled by a1,..,an, the transitions labeled by a1,..,an are discarded.

Easy example

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



Class Exercise

A new definition of SUPERMARKET models two types of client, one of them greedy

```
||NEW_MARKET = (SHELF || WORKER || {a,b}:CLIENT)
               /{{a,b}.get/get}<<{a.get}.
```

- Draw NEW_MARKET when Max = 3 and Min = 1.
- Think about the following progress properties concerning NEW_MARKET. Are they true?
  1. progress FILL = {fill}
  2. progress B_GET = {b.get[1..Max]}

Model of a congested Bridge

```
/*
bridge becomes congested when we give
less priority to exit that entry
*/

||CongestedBridge
    = SingleLaneBridge
            >>{red[ID].exit,blue[ID].exit}.

progress BLUECROSS = {blue[ID].enter}
progress REDCROSS =  {red[ID].enter}
```

LTS Progress analysis

LTS Analyzer→Check→Progress

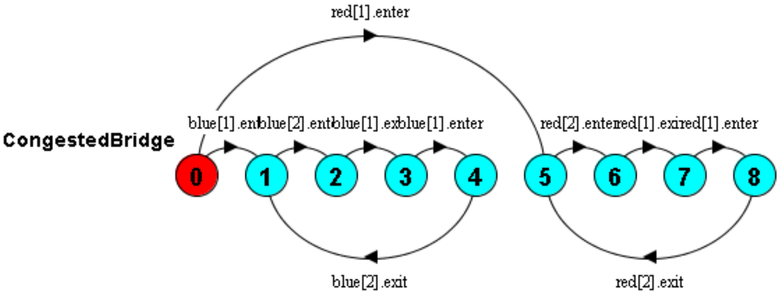
```
Progress Check...
...
Progress violation: REDCROSS
Trace to terminal set of states:
       blue.1.enter
Cycle in terminal set:
       blue.2.enter
       blue.1.exit
       blue.1.enter
       blue.2.exit
Actions in terminal set:
       blue[1..2].{enter, exit}
Progress Check in:...
```

There is also a progress violation in `BLUECROSS`.

Why?

CongestedBridge (1)

```
||CongestedBridge
   = SingleLaneBridge
               >>{red[ID].exit,blue[ID].exit}.
```



With more than one car, whichever color car enters the bridge first will continuously occupy the bridge preventing the other color from crossing.

CongestedBridge (2)

- Remind that in SingleLaneBridge (not Congested Bridge) in state $(2\ 1\ [\ -\ |\ \ 1\ ]\ 2)$ there are two transitions

$$(2\ 1\ [\ -\ |\ \ 1\ ]\ 2) \xrightarrow{[2].enter} (2\ 1\ [\ 1\ |\ 2\ ]\ )$$
$$(2\ 1\ [\ -\ |\ \ 1\ ]\ 2) \xrightarrow{[1].exit} (2\ 1\ [\ -\ |\ -\ ]\ 1\ 2)$$

- In CongestedBridge action [1].exit has low priority and therefore will never be executed. In this case we have only the transition

$$(2\ 1\ [\ -\ |\ \ 1\ ]\ 2) \xrightarrow{[2].enter} (2\ 1\ [\ 1\ |\ 2\ ]\ )$$

- In CongestedBridge we can loop infinitely across $(2\ 1\ [\ -\ |\ \ 1\ ]\ 2)$ with no fairness violation.

Bridge under stress (1)

The bridge need to know whether cars are waiting to access.
A car it request to access.

```
CAR = (request->enter->exit->CAR).
/* nr  - number of red cars on the bridge
   nb  - number of blue cars on the bridge
   wr  - number of red cars waiting to enter
   wb  - number of blue cars waiting to enter
*/
BRIDGE = BRIDGE[0][0][0][0],
BRIDGE[nr:T][nb:T][wr:T][wb:T] =
  (red[ID].request -> BRIDGE[nr][nb][wr+1][wb]
  |when (nb==0 && wb==0)
     red[ID].enter  -> BRIDGE[nr+1][nb][wr-1][wb]
  |red[ID].exit     -> BRIDGE[nr-1][nb][wr][wb]
  |blue[ID].request -> BRIDGE[nr][nb][wr][wb+1]
  |when (nr==0 && wr==0)
     blue[ID].enter -> BRIDGE[nr][nb+1][wr][wb-1]
  |blue[ID].exit    -> BRIDGE[nr][nb-1][wr][wb]
  ).
```

Bridge under stress (2)

Deadlock problem

```
red.1.request,
red.2.request
red.3.request,
blue.1.request,
blue.2.request,
blue.3.request,
```

- Introduce some asymmetry in the problem.
- This takes the form of a boolean variable (bt) which breaks the deadlock by indicating whether it is the turn of blue cars or red cars to enter the bridge.
- Arbitrarily set bt to true initially giving blue initial precedence.

Bridge under stress (3)

```
BRIDGE = BRIDGE[0][0][0][0][True], //initially empty
BRIDGE[nr:T][nb:T][wr:T][wb:T][bt:B] =
   (red[ID].request  -> BRIDGE[nr][nb][wr+1][wb][bt]
   |when (nb==0 && (wb==0 || !bt))
      red[ID].enter  -> BRIDGE[nr+1][nb][wr-1][wb][bt]
   |red[ID].exit     -> BRIDGE[nr-1][nb][wr][wb][True]
   |blue[ID].request->BRIDGE[nr][nb][wr][wb+1][bt]
   |when (nr==0 && (wr==0 || bt))
       blue[ID].enter -> BRIDGE[nr][nb+1][wr][wb-1][bt]
   |blue[ID].exit    ->  BRIDGE[nr][nb-1][wr][wb][False]
).
```

Class BridgeUnderStress (1)

```
class BridgeUnderStress{

  private int nred  = 0;
  private int nblue = 0;
  private int waitblue = 0;
  private int waitred = 0;
  private boolean blueturn = true;

  synchronized void redEnter() throws ... {...}

  synchronized void redExit(){...}

  synchronized void blueEnter() throws ... {...}

  synchronized void blueExit(){...}
}
```

Class BridgeUnderStress (2)

```
class BridgeUnderStress{
  ...
  synchronized void redEnter() throws Int...Exc...{
     ++waitred;
      while(nblue>0||(waitblue>0&&blueturn))wait();
      --waitred; ++nred; }
  synchronized void redExit(){
      --nred; blueturn = true;
      if (nred==0) notifyAll();}
  synchronized void blueEnter() throws Int...Exc...{
     ++waitblue;
      while (nred>0||(waitred>0&&!blueturn))wait();
      --waitblue;++nblue;}
  synchronized void blueExit(){
     --nblue; blueturn = false;
     if (nblue==0) notifyAll();}
}
```

Strongly recommended reading

Chapter 7 of Concurrency book.