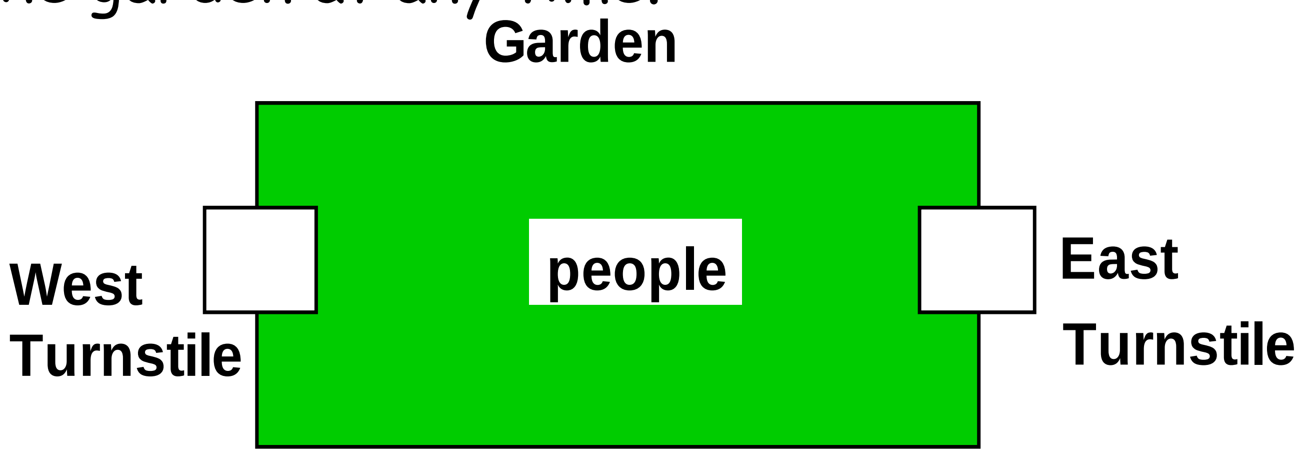
Lecture 6

- Administration

4.1 Interference

Ornamental garden problem:

People enter an ornamental garden through either of two turnstiles. Management wish to know how many are in the garden at any time.



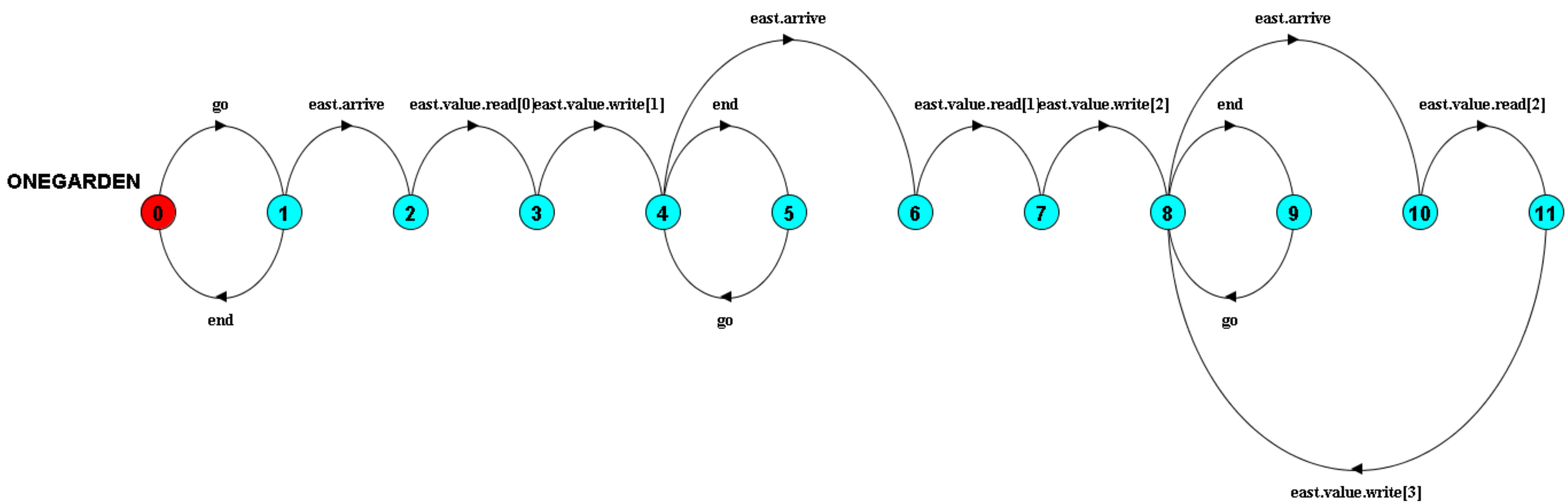
The concurrent program consists of two concurrent threads and a shared counter object.

Garden

```
const N = 2
range T = 0..N
set VarAlpha = { value.{read[T], write[T] }}

VAR = VAR[0],
VAR[u:T] = ( read[u] -> VAR[u] | write[v:T] -> VAR[v] ).

TURNSTILE = (go->RUN),
RUN = (arrive->INCREMENT | end->TURNSTILE),
INCREMENT = (value.read[x:T]->value.write[x+1]->RUN)+VarAlpha.

||GARDEN = (east:TURNSTILE || west:TURNSTILE || {east,west,display}::value:VAR)
                          / {go/{east,west}.go, end/{east,west}.end}.
```

One-Turnstile

Which states can appear infinitely often?



Are there a minimal set of states where eventually all every state is a member of this set?

Actions in terminal set:

{east.{arrive, value.{read[2], write[3]}}, {end, go}}

ornamental garden model

```
const N = 4
range T = 0..N
set VarAlpha = { value.{read[T],write[T]} }

VAR        = VAR[0],
VAR[u:T] = (read[u]    ->VAR[u]
           |write[v:T]->VAR[v]).

TURNSTILE = (go     -> RUN),
RUN        = (arrive-> INCREMENT
            |end    -> TURNSTILE),
INCREMENT = (value.read[x:T]
             -> value.write[x+1]->RUN
            )+VarAlpha.

||GARDEN = (east:TURNSTILE || west:TURNSTILE
           || { east,west,display}::value:VAR)
            /{ go /{ east,west} .go,
              end/{ east,west} .end} .
```

The alphabet of shared process **VAR** is declared explicitly as a **set** constant, **VarAlpha**.

The **TURNSTILE** alphabet is extended with **VarAlpha** to ensure no unintended **free (autonomous) actions** in **VAR** eg. **value.write[0]**. All actions in the shared **VAR** must be controlled (shared) by a **TURNSTILE**.

Checking for errors

```
go
east.arrive
east.value.read.0
west.arrive
west.value.read.0
east.value.write.1
west.value.write.1
end
display.value.read.1
```

checking for errors - exhaustive analysis

Exhaustive checking - compose the model with a TEST process which sums the arrivals and checks against the display value:

```
TEST        = TEST[0],
TEST[v:T]   =
      (when (v<N){east.arrive,west.arrive}->TEST[v+1]
      |end->CHECK[v]
      ),
CHECK[v:T] =
    (display.value.read[u:T] ->
       (when (u==v) right -> TEST[v]
       |when (u!=v) wrong -> ERROR
       )
    )+{display.VarAlpha}.
```

Like **STOP**, **ERROR** is a predefined FSP local process (state), numbered -1 in the equivalent LTS.

Checking for errors

```
||TESTGARDEN = (GARDEN || TEST).
```

Use *LTSA* to perform an exhaustive search for **ERROR**.

```
Trace to property violation in TEST:
    go
    east.arrive
    east.value.read.0
    west.arrive
    west.value.read.0
    east.value.write.1
    west.value.write.1
    end
    display.value.read.1
    wrong
```

LTSA produces the shortest path to reach **ERROR**.

Garden

```
const N = 2
range T = 0..N
set VarAlpha = { value.{read[T], write[T] }}

VAR = VAR[0],
VAR[u:T] = ( read[u] -> VAR[u] | write[v:T] -> VAR[v] ).

TURNSTILE = (go->RUN),
RUN = (arrive->INCREMENT | end->TURNSTILE),
INCREMENT = (value.read[x:T]->value.write[x+1]->RUN)+VarAlpha.

||GARDEN = (east:TURNSTILE || west:TURNSTILE || {east,west,display}::value:VAR)
                          / {go/{east,west}.go, end/{east,west}.end}.

TEST = TEST[0],
TEST[v:T] = (when (v<N) {east.arrive,west.arrive}->TEST[v+1] | end->CHECK[v] ),
CHECK[v:T] = ( display.value.read[u:T]->
                 (when (u==v) right->TEST[v] | when (u!=v) wrong->ERROR )
             )+{display.VarAlpha}.

||TESTGARDEN = (GARDEN || TEST ).
```

Interference and Mutual Exclusion

Destructive update, caused by the arbitrary interleaving of read and write actions, is termed ***interference***.

Interference bugs are extremely difficult to locate. The general solution is to give methods *mutually exclusive* access to shared objects. Mutual exclusion can be modeled as atomic actions.

Mutual Exclusion

4.3 Modeling mutual exclusion

To add locking to our model, define a **LOCK**, compose it with the shared **VAR** in the garden, and modify the alphabet set :

```
LOCK = (acquire->release->LOCK).
||LOCKVAR = (LOCK || VAR).

set VarAlpha = {value.{read[T],write[T],
                      acquire, release}}
```

Modify **TURNSTILE** to acquire and release the lock:

```
TURNSTILE = (go    -> RUN),
RUN       = (arrive-> INCREMENT
            |end   -> TURNSTILE),
INCREMENT = (value.acquire
             -> value.read[x:T]->value.write[x+1]
             -> value.release->RUN
            )+VarAlpha.
```

Garden with Locks

```
const N = 2
range T = 0..N
set VarAlpha = { value.{read[T], write[T], acquire, release }}

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

VAR = VAR[0],
VAR[u:T] = ( read[u] -> VAR[u] | write[v:T] -> VAR[v] ).

||LOCKVAR = (LOCK || VAR).

TURNSTILE = (go->RUN),
RUN = (arrive->INCREMENT | end->TURNSTILE),
INCREMENT = (value.acquire->value.read[x:T]->value.write[x+1]->value.release->RUN)+VarAlpha.

||GARDEN = (east:TURNSTILE || west:TURNSTILE || {east,west,display}::value:LOCKVAR) /
{go/{east,west}.go, end/{east,west}.end}.

TEST = TEST[0],
TEST[v:T] = (when (v<N) {east.arrive,west.arrive}->TEST[v+1] | end->CHECK[v] ),
CHECK[v:T] = (display.value.read[u:T]->
                                            (when (u==v) right->TEST[v] | when (u!=v) wrong->ERROR )
                                   )+{display.VarAlpha}.

||TESTGARDEN = (GARDEN || TEST ).
```

Revised ornamental garden model - checking for errors

A sample animation execution trace

```
go
east.arrive
east.value.acquire
east.value.read.0
east.value.write.1
east.value.release
west.arrive
west.value.acquire
west.value.read.1
west.value.write.2
west.value.release
end
display.value.read.2
right
```

Use TEST and ***LTSA*** to perform an exhaustive check.

Is TEST satisfied?

COUNTER: Abstraction using action hiding

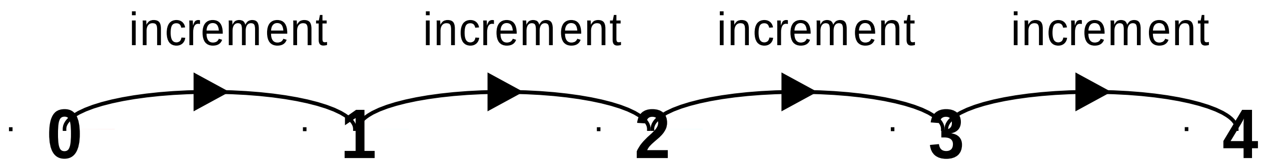
```
const N = 4
range T = 0..N

VAR = VAR[0],
VAR[u:T] = ( read[u]->VAR[u]
           | write[v:T]->VAR[v]).

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

INCREMENT = (acquire->read[x:T]
             -> (when (x<N) write[x+1]
                 ->release->increment->INCREMENT
                )
             )+{read[T],write[T]}.

||COUNTER = (INCREMENT||LOCK||VAR)@{increment}.
```

To model shared objects directly in terms of their synchronized methods, we can abstract the details by hiding.

For **SynchronizedCounter** we hide **read**, **write**, **acquire**, **release** actions.

COUNTER: Abstraction using action hiding

Minimized LTS:



We can give a more abstract, simpler description of a **COUNTER** which generates the same LTS:

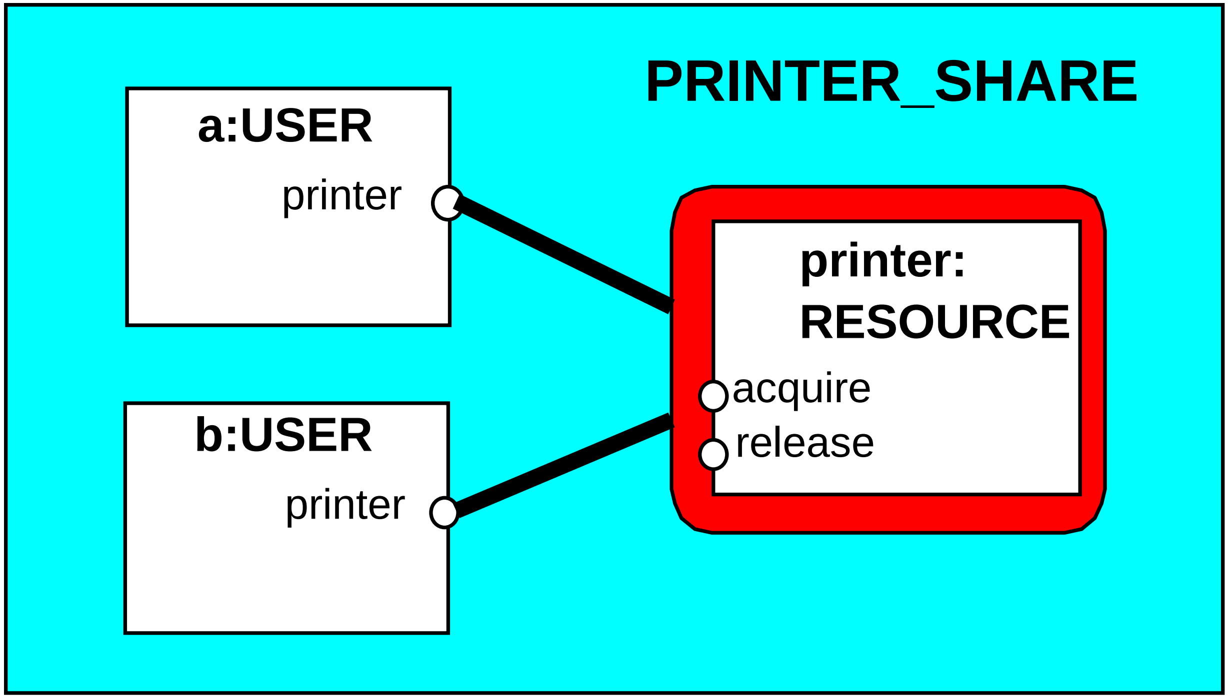
```
COUNTER = COUNTER[0]
COUNTER[v:T] = (when (v<N) increment -> COUNTER[v+1]).
```

This therefore exhibits "equivalent" behavior i.e. has the same observable behavior.

structure diagrams - resource sharing

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

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



monitors & condition synchronization

Concepts: monitors:

encapsulated data + access procedures
mutual exclusion + condition synchronization
single access procedure active in the monitor
nested monitors

Models: guarded actions

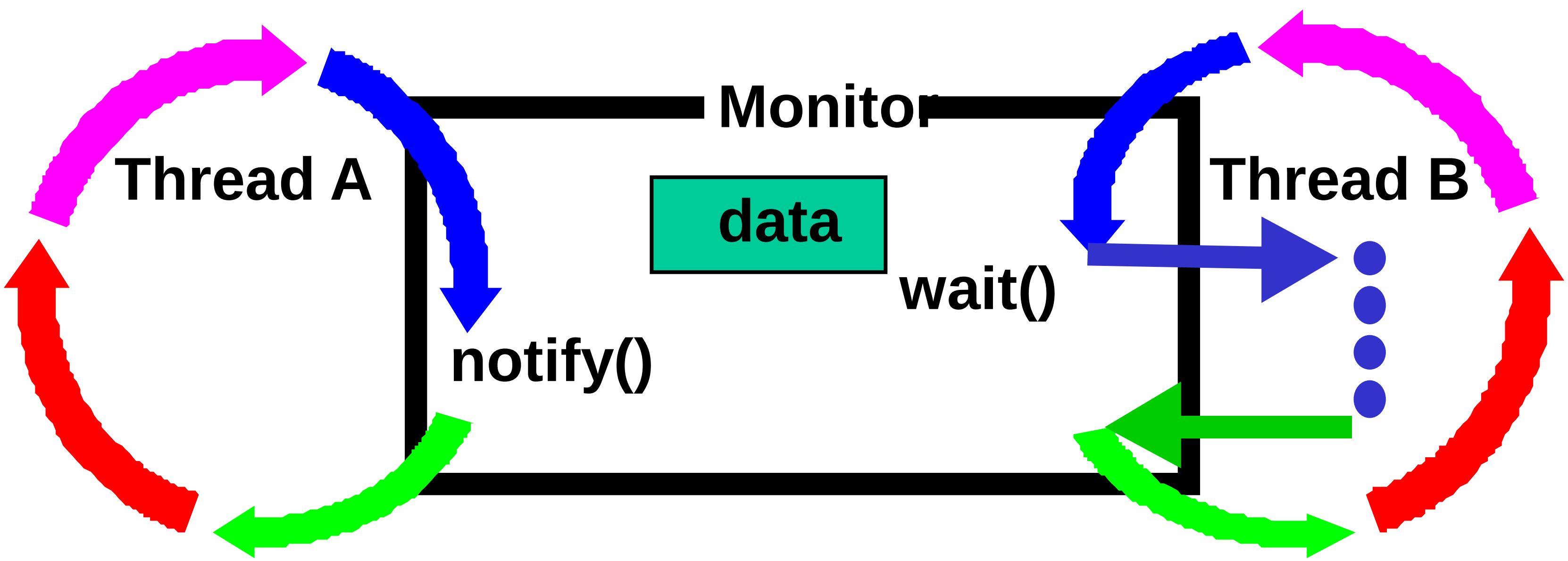
Practice: private data and synchronized methods (exclusion).
wait(), notify() and notifyAll() for condition synch.
single thread active in the monitor at a time

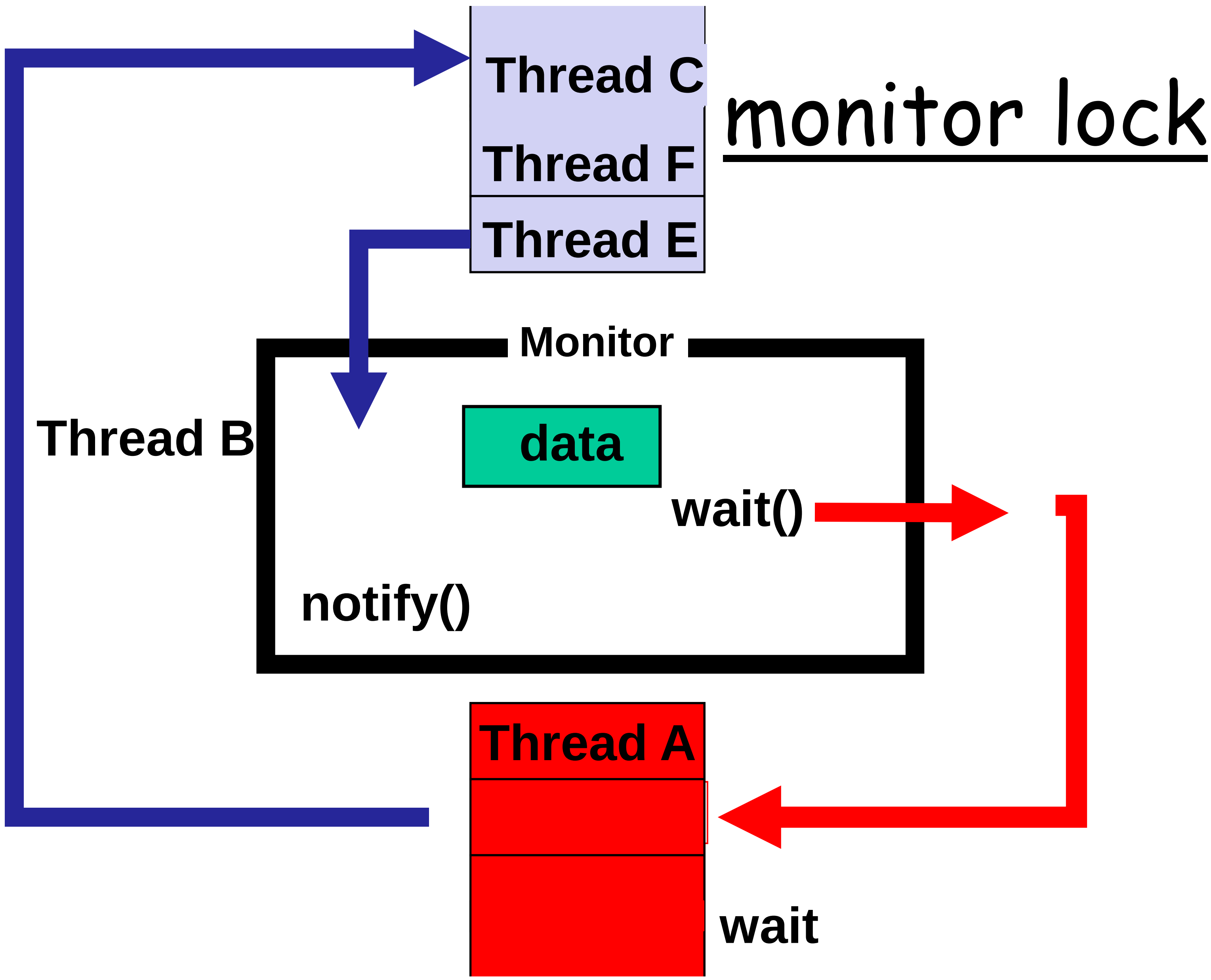
http://stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again

condition synchronization in Java

We refer to a thread ***entering*** a monitor when it acquires the mutual exclusion lock associated with the monitor and ***exiting*** the monitor when it releases the lock.

Wait() - causes the thread to exit the monitor, permitting other threads to enter the monitor.





Thread C
Thread F
Thread E
monitor lock
Monitor
Thread B
data
wait()
notify()
Thread A
wait

condition synchronization in Java

```
FSP:    when cond act -> NEWSTAT
```

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

The **`while`** loop is necessary to retest the condition *cond* to ensure that *cond* is indeed satisfied when it re-enters the monitor.

`notifyall()` is necessary to awaken other thread(s) that may be waiting to enter the monitor now that the monitor data has been changed.