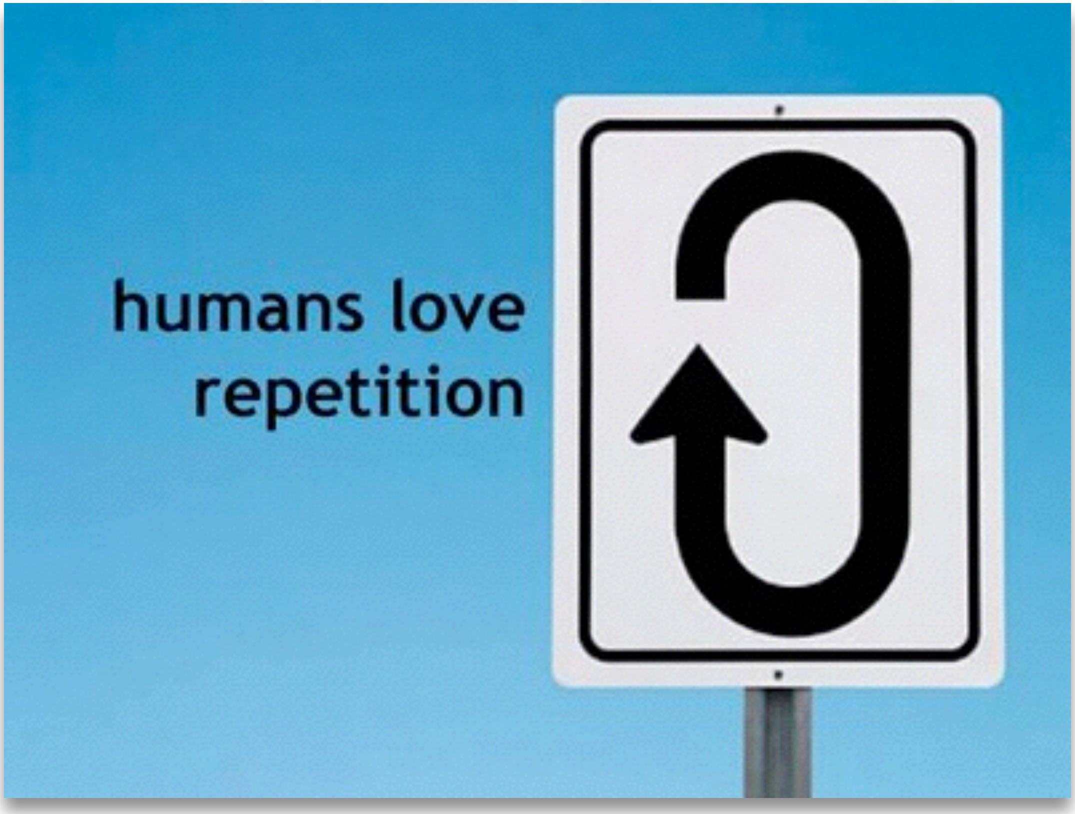
UNIVERSITY OF SOUTHERN DENMARK

Deadlock

But First: Repetition

Monitors and Condition Synchronisation



Monitors & Condition Synchronisation

UNIVERSITY OF SOUTHERN DENMARK

Concepts: monitors:

encapsulated data + access procedures +
mutual exclusion + condition synchronisation +
single access procedure active in the monitor

nested monitors

Models: guarded actions

Practice: private data and synchronized methods (exclusion).

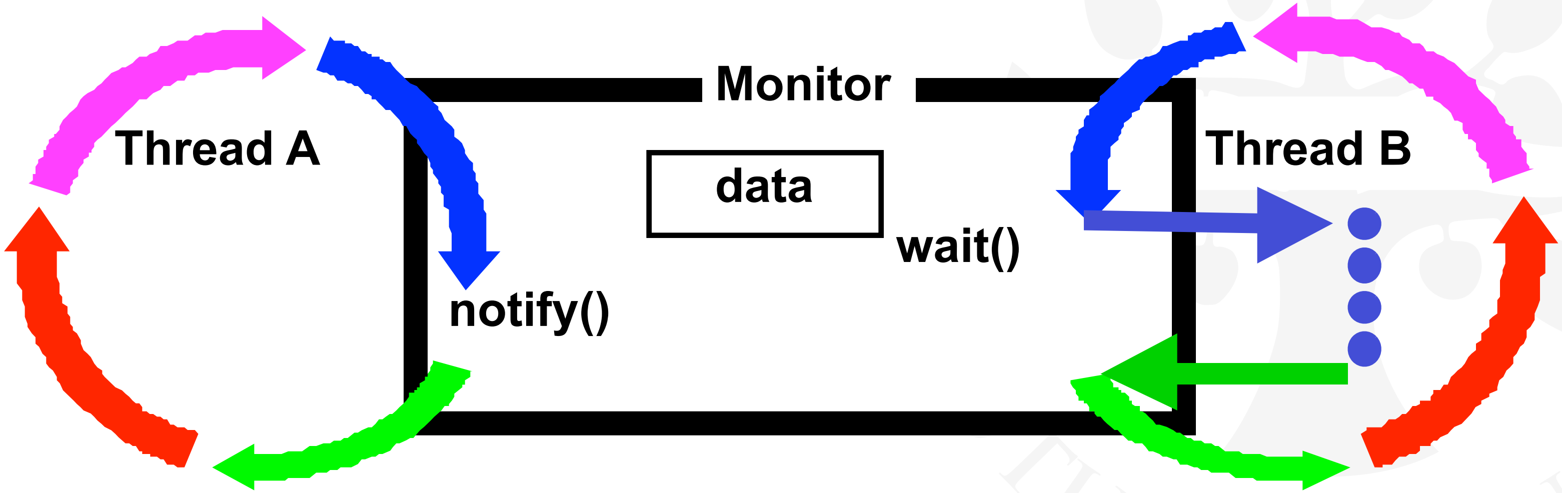
wait(), notify() and notifyAll() for condition synch.

single thread active in the monitor at a time

Wait(), Notify(), And Notifyall()


```
public final void wait() throws InterruptedException;
```

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




```
public final void notify();
```

```
public final void notifyAll();
```

Condition Synchronisation (In Java)

UNIVERSITY OF SOUTHERN DENMARK

```
CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
            (when(spaces>0)          arrive -> SPACES[spaces-1]
            |when(spaces<CAPACITY)   depart -> SPACES[spaces+1]).
```

```java
class CarParkControl {
    protected int spaces, capacity;

    synchronized void arrive()
                throws Int'Exc' {
        while (!(spaces>0)) wait();
        --spaces;
        notifyAll();
    }

    synchronized void depart()
                throws Int'Exc' {
        while (!(spaces<capacity)) wait();
        ++spaces;
        notifyAll();
}   }
```

notify() instead of notifyAll() ?
1. Uniform waiters - everybody waits on the same condition
2. One-in, one-out

What goes wrong with notify and 8xDepartures, 5xArrivals?

Semaphores

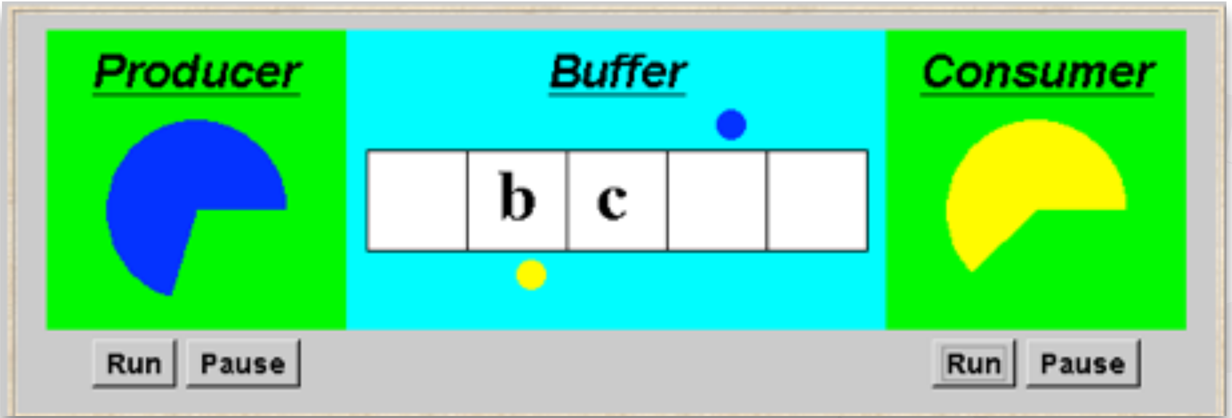
Semaphores are widely used for dealing with inter-process synchronisation in operating systems.

Semaphore s : integer var that can take only non-neg. values.

```
sem.down(); // decrement (block if counter = 0)
```

```
sem.up(); // increment counter (allowing one blocked thread to pass)
```

Nested Monitors - Bounded Buffer Model



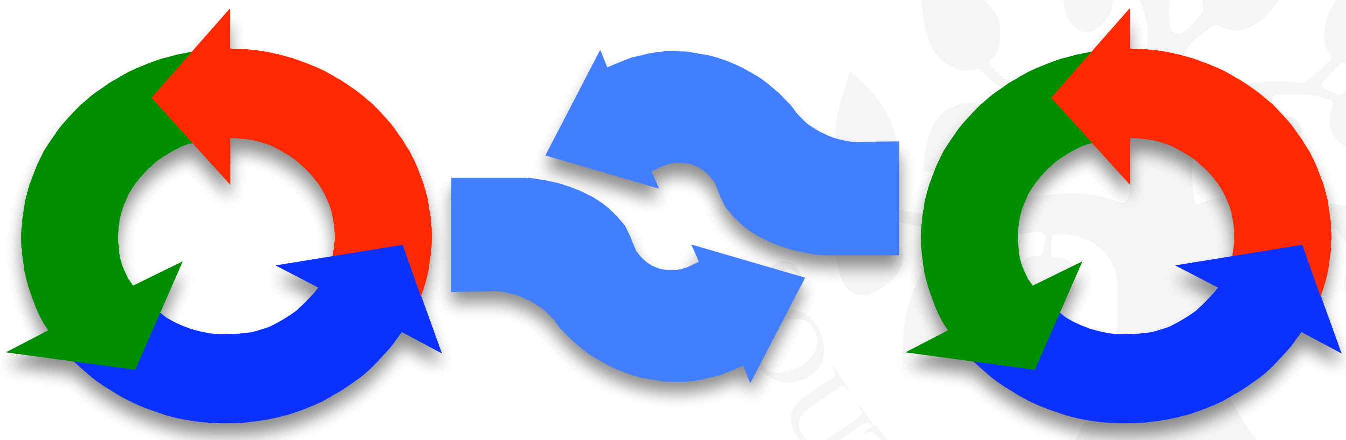
LTSA's (analyse safety) predicts a possible DEADLOCK:

```
Composing
  potential DEADLOCK
  States Composed: 28 Transitions: 32 in 60ms
  Trace to DEADLOCK:
    get
```

This situation is known as the **nested monitor problem**.

UNIVERSITY OF SOUTHERN DENMARK

Deadlock



Deadlock

University of Southern Denmark

Concepts:	system deadlock (no further progress) 4 necessary & sufficient conditions
Models:	deadlock - no eligible actions
Practice:	blocked threads

Aim: **deadlock avoidance** - to design systems where deadlock cannot occur.

Necessary & Sufficient Conditions

Necessary condition:

P *necessary* for Q:
$P \Leftarrow Q$

Sufficient condition:

P *sufficient* for Q:
$P \Rightarrow Q$

Necessary & sufficient condition:

P *necessary* & *sufficient* for Q:
$(P \Leftarrow Q) \wedge (P \Rightarrow Q) \quad \equiv \quad P \Leftrightarrow Q$

P: The sun is shining
Q: I get sunlight on my beer

$P \Leftarrow Q$ only.

Deadlock: 4 Necessary And Sufficient Conditions

1. **Mutual exclusion condition** (aka. "Serially reusable resources"):

   the processes involved share resources which they use under mutual exclusion.

2. **Hold-and-wait condition** (aka. "Incremental acquisition"):

   processes hold on to resources already allocated to them while waiting to acquire additional resources.

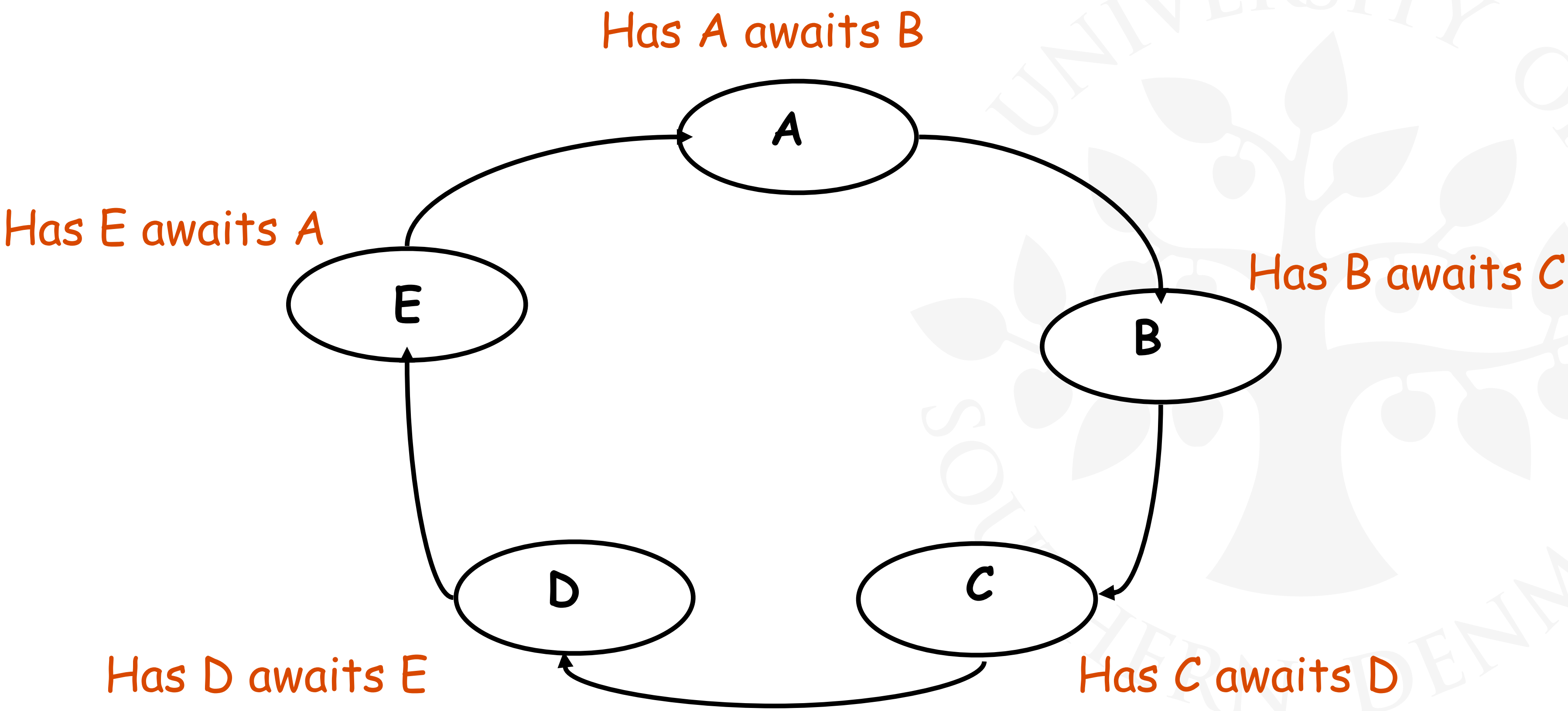
3. **No preemption condition:**

   once acquired by a process, resources cannot be "pre-empted" (forcibly withdrawn) but are only released voluntarily.

4. **Circular-wait condition** (aka. "Wait-for cycle"):

   a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.

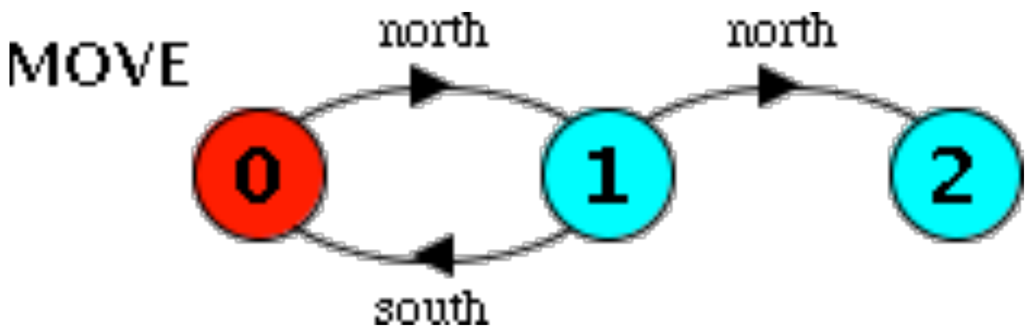
Wait-For Cycle



6.1 Deadlock Analysis - Primitive Processes

- **Deadlocked state** is one with no outgoing transitions
- In FSP: (modelled by) the **STOP** process

```
MOVE = (north->(south->MOVE|north->STOP)).
```



- Analysis using **LTSA**:

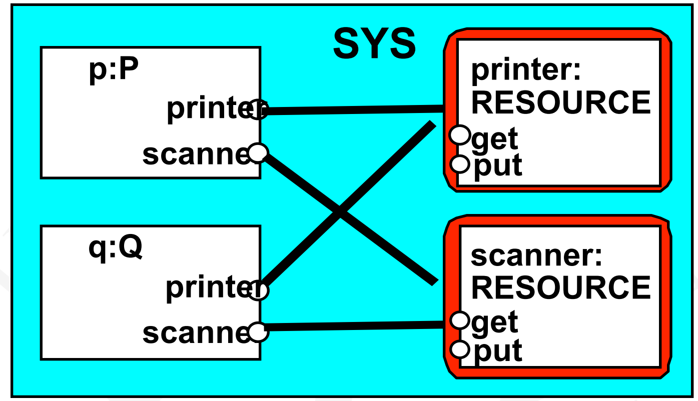
Shortest path to DEADLOCK:

```
Trace to DEADLOCK:
    north
    north
```

Deadlock Analysis - Parallel Composition

- In practice, deadlock arises from parallel composition of interacting processes.

```
P = (x -> y -> P).
Q = (y -> x -> Q).
||D = (P || Q).
```




```
RESOURCE = (get-> put-> RESOURCE).

P = (printer.get-> scanner.get-> copy-> printer.put->

Q = (scanner.get-> printer.get-> copy-> scanner.put-> printer.put-> Q).

||SYS = (p:P || q:Q || {p,q}::printer:RESOURCE ||
{p,q}::scanner:RESOURCE).
```

```
Trace to DEADLOCK:
  p.printer.get
  q.scanner.get
```

b Avoidance...

Recall The 4 Conditions

UNIVERSITY OF SOUTHERN DENMARK

1. **Mutual exclusion condition** (aka. "Serially reusable resources"):

   the processes involved share resources which they use under mutual exclusion.

2. **Hold-and-wait condition** (aka. "Incremental acquisition"):

   processes hold on to resources already allocated to them while waiting to acquire additional resources.

3. **No preemption condition:**

   once acquired by a process, resources cannot be "pre-empted" (forcibly withdrawn) but are only released voluntarily.

4. **Circular-wait condition** (aka. "Wait-for cycle"):

   a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.

Deadlock Analysis – Avoidance (#1 ?)

1. **Mutual exclusion condition** (aka. "Serially reusable resources"):

   the processes involved share resources which they use under mutual exclusion.

- Ideas?
  - ...avoid shared resources (used under mutual exclusion)

- No shared resources (buy **two** printers and **two** scanners)

Deadlock? ☺ **Scalability?** ☹

Deadlock Analysis – Avoidance (#2 ?)

University of Southern Denmark

2. Hold-and-wait condition (aka. "Incremental acquisition"):

processes hold on to resources already allocated to them while waiting to acquire additional resources.

- Only **one** "mutex" lock for **both** scanner and printer:

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

P = (scanner_printer.acquire->
       printer.get->
         scanner.get->
           copy->
         scanner.put->
       printer.put->
     scanner_printer.release-> P).
```

Deadlock? ☺ Efficiency/Scalability? ☹

Deadlock Analysis – Avoidance (#3 ?)

UNIVERSITY OF SOUTHERN DENMARK

3. No pre-emption condition:

once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily.

- Force release (e.g., through timeout or arbiter):

```
P           = (printer.get-> GETSCANNER),
GETSCANNER  = (scanner.get-> copy-> printer.put-> scanner.put-> P
              |timeout -> printer.put-> P).

Q           = (scanner.get-> GETPRINTER),
GETPRINTER  = (printer.get-> copy-> printer.put-> scanner.put-> Q
              |timeout -> scanner.put-> Q).
```

Deadlock? ☺ Progress? ☹

Deadlock Analysis – Avoidance (#4 ?)

University of Southern Denmark

4. Circular-wait condition (aka. "Wait-for cycle"):

a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.

- Acquire resources in the same order:

```
P = (printer.get->
        scanner.get->
          copy-> printer.put-> scanner.put-> P).

Q = (printer.get->
        scanner.get->
          copy-> printer.put-> scanner.put-> Q).
```

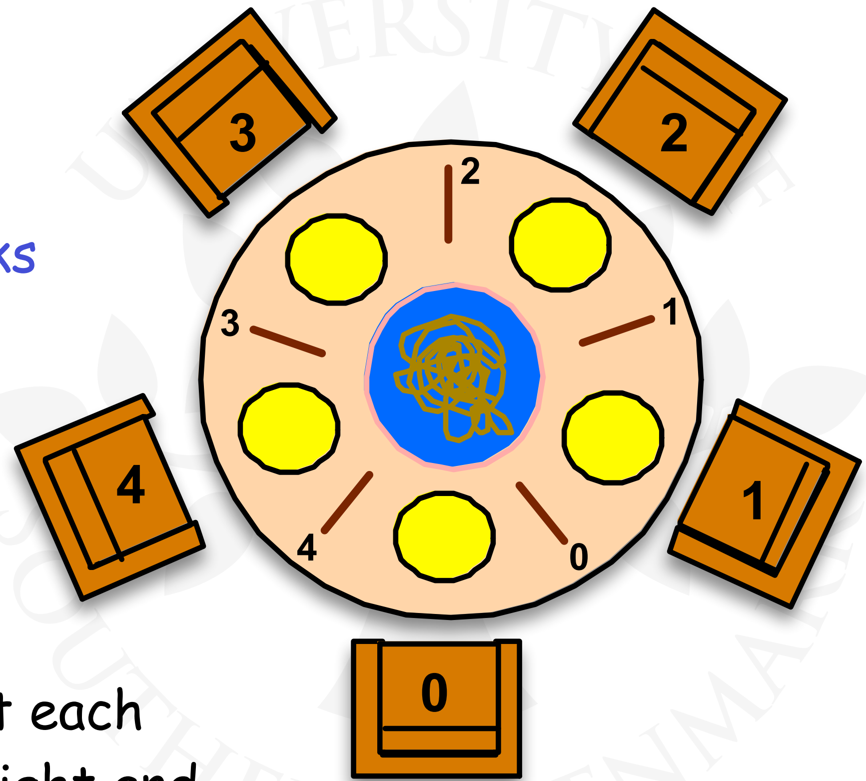
Deadlock? ☺ Scalability/Progress/…? ☺

General solution: "sort" resource acquisitions

BUT Sort by... ...what?

6.2 Dining Philosophers

Five philosophers sit around a circular table. Each philosopher spends his life alternately thinking and eating. In the centre of the table is a large bowl of spaghetti. A philosopher needs two forks to eat a helping of spaghetti.

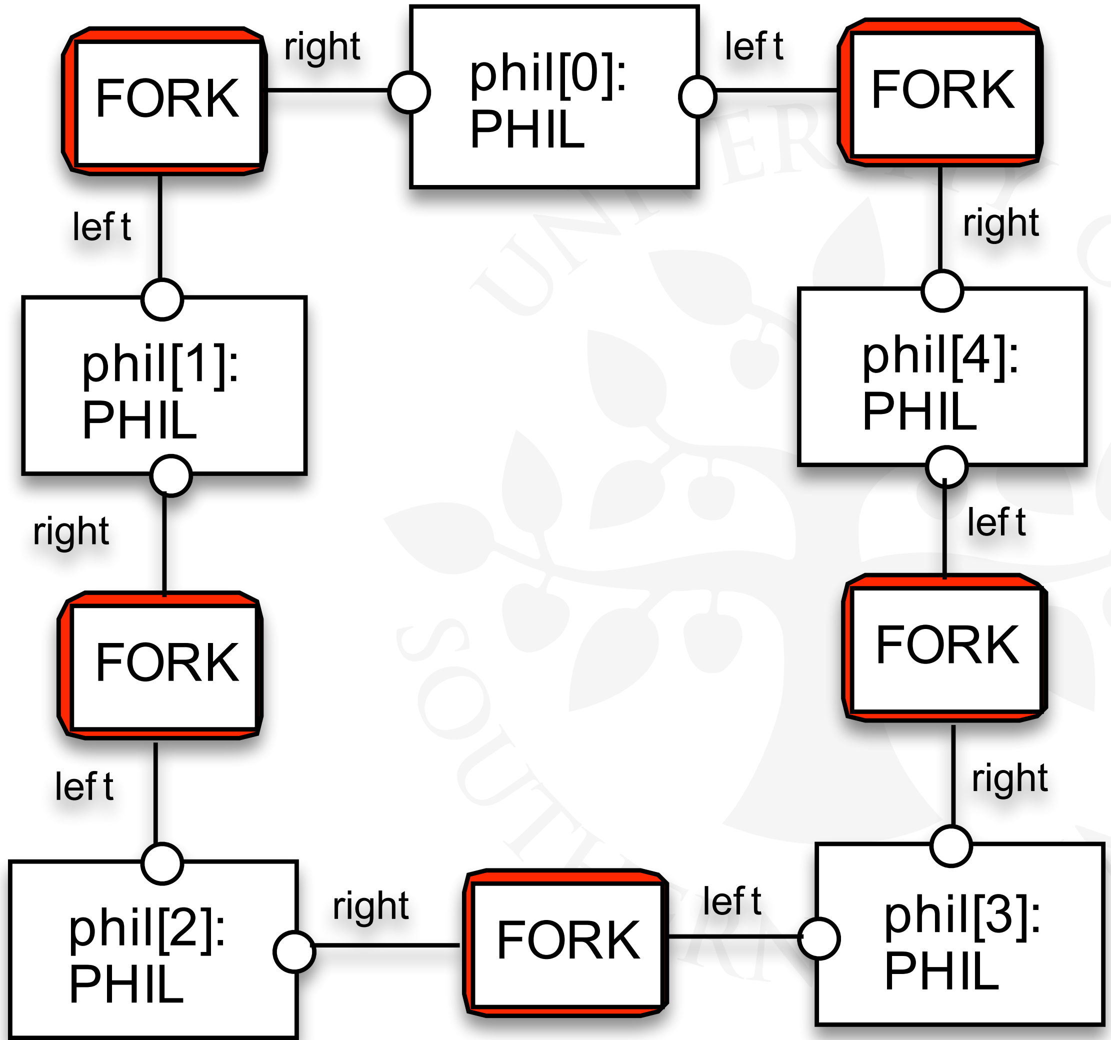


One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.

Dining Philosophers - Model Structure Diagram

Each **FORK** is a shared resource with actions **get** and **put**.

When hungry, each **PHIL** must first get his right and left forks before he can start eating.

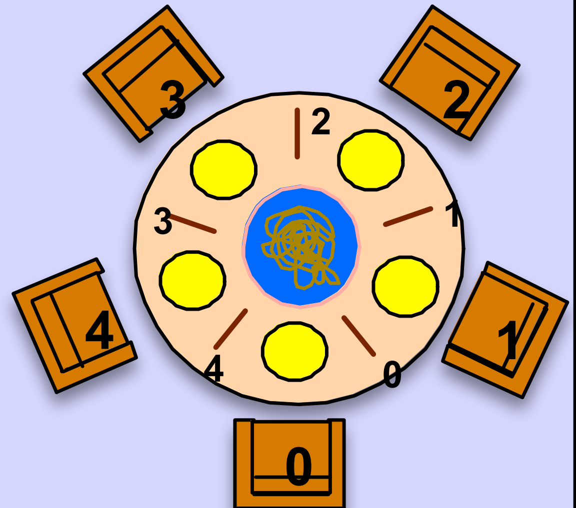


Dining Philosophers - Model


```
const N = 5

FORK = (get-> put-> FORK).

PHIL = (sit          ->
            right.get ->
              left.get  ->
                eat         ->
                  left.put  ->
                    right.put ->
                      arise     -> PHIL).
```



Can this system deadlock?

```
||DINING_PHILOSOPHERS =
   forall [i:0..N-1] (phil[i]:PHIL ||
            { phil[i].left, phil[((i-1)+N)%N].right }::FORK).
```

Dining Philosophers - Model Analysis

```
Trace to DEADLOCK:
  phil.0.sit
  phil.0.right.get
  phil.1.sit
  phil.1.right.get
  phil.2.sit
  phil.2.right.get
  phil.3.sit
  phil.3.right.get
  phil.4.sit
  phil.4.right.get
```

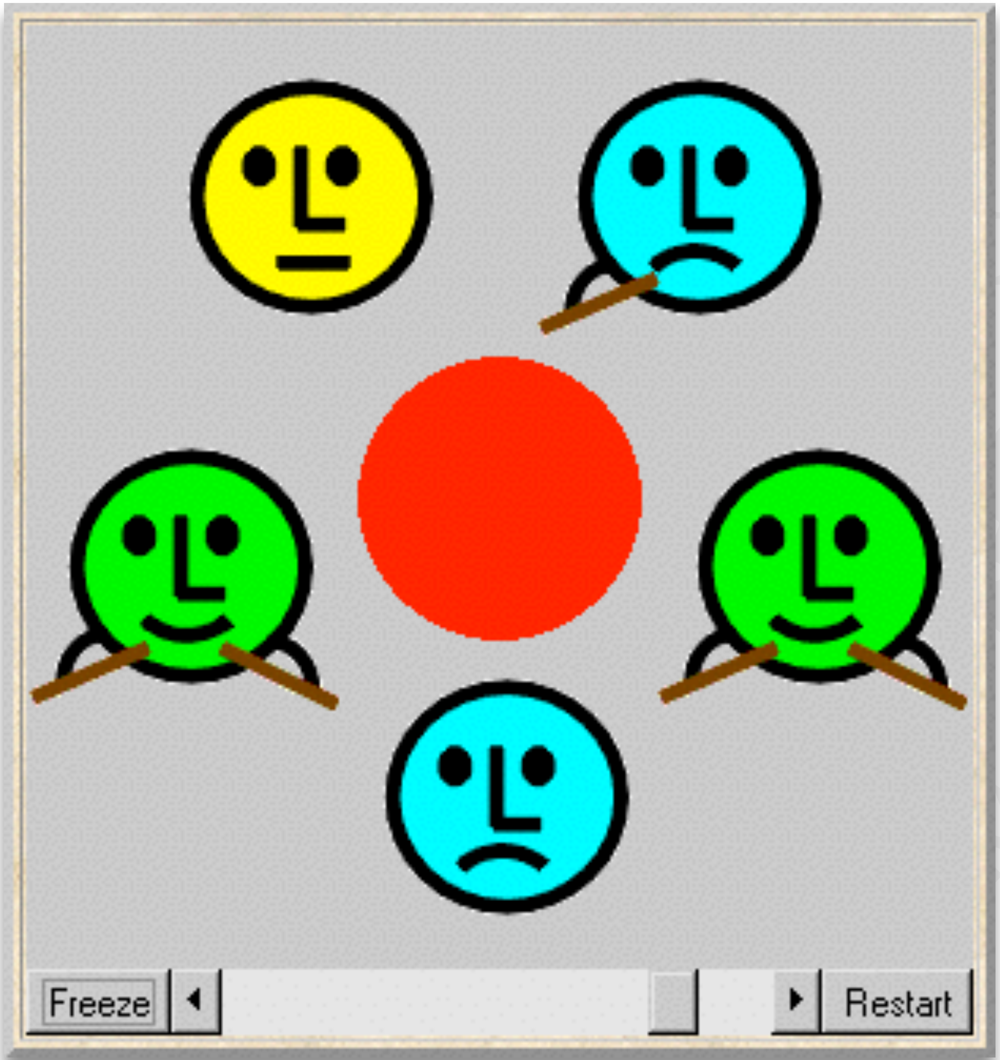
This is the situation where all the philosophers become hungry at the same time, sit down at the table and each philosopher picks up the fork to his right.

The system can make no further progress since each philosopher is waiting for a left fork held by his neighbour (i.e., a **wait-for cycle** exists)!

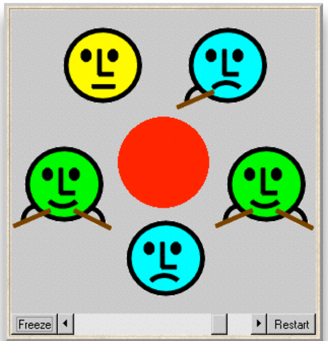
Dining Philosophers

Deadlock is easily detected in our model.

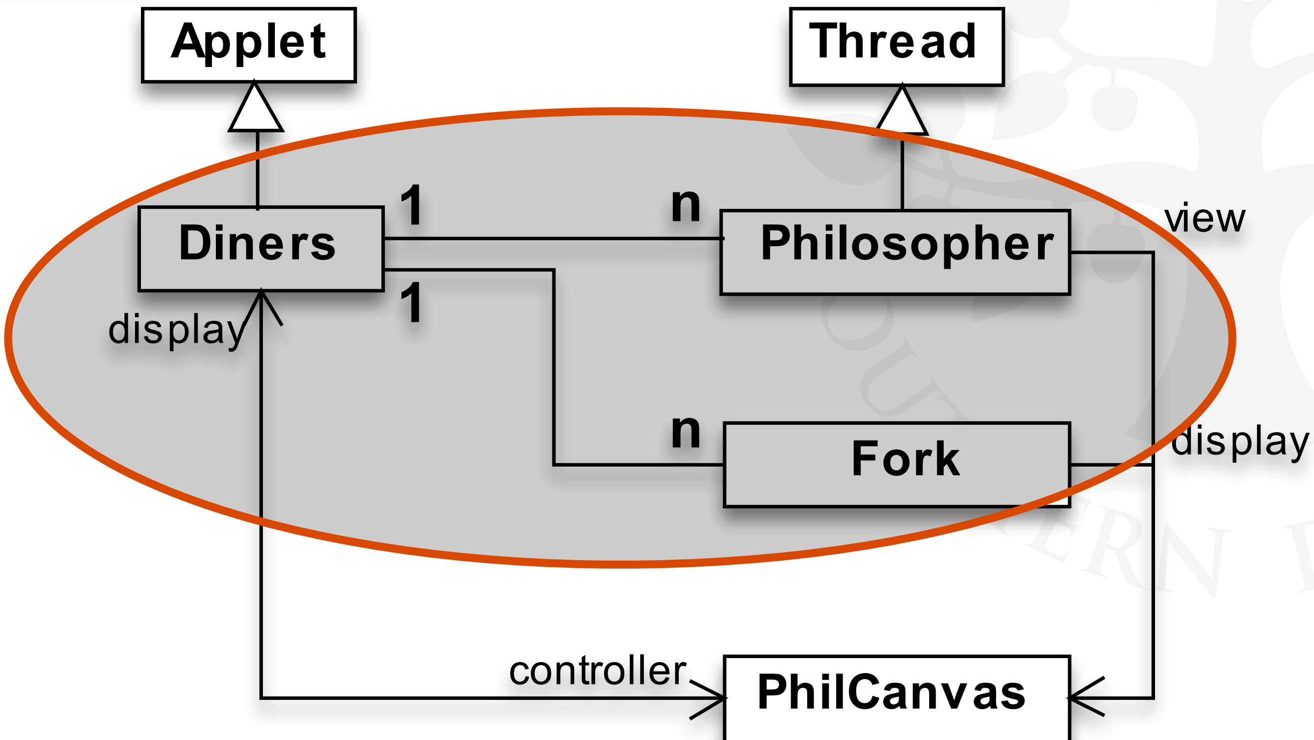
How easy is it to detect a potential deadlock in an implementation?



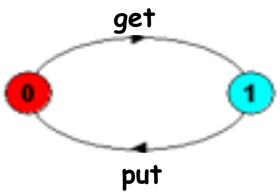
Dining Philosophers - Implementation In Java



- **Philosophers**: active entities (implement as threads)
- **Forks**: shared passive entities (implement as monitors)



Dining Philosophers – Fork (Monitor)




```
FORK = (get->
        put->
        FORK).
```

≡

```
FORK = (FORK[FALSE],
FORK[taken:B] (when (!taken) get-> FORK[TRUE]
              |when (taken)  put-> FORK[FALSE]).
```

Not needed (if we always "get before put")

```
class Fork {
    private PhilCanvas display;
    private boolean taken = false;

    synchronized void get() throws Int'Exc' {
        while (taken) wait();              // cond. synch. (!)
        taken = true;
        display.setFork(identity, taken);
    }

    synchronized void put() {
        taken = false;
        display.setFork(identity, taken);
        notify();                          // cond. synch. (!)
}   }
```

taken encodes the state of the fork

Dining Philosophers – Philosopher (Thread)

UNIVERSITY OF SOUTHERN DENMARK

```
PHIL = (sit -> right.get -> left.get -> eat -> left.put ->  right.put -> arise -> PHIL).
```

```
class Philosopher extends Thread {
    Fork left, right;
    public void run() {
        try {
            while (true) {
                view.setPhil(identity,view.SIT);
                sleep(controller.sitTime());
                right.get();
                view.setPhil(identity,view.GOTRIGHT);
                sleep(500); // constant pause!
                left.get();
                view.setPhil(identity,view.EATING);
                sleep(controller.eatTime());
                left.put();
                right.put();
                view.setPhil(identity,view.ARISE);
                sleep(controller.ariseTime());
            }
        } catch (InterruptedException _) {}
}   }
```

Dining Philosophers – Main Applet

University of Southern Denmark

```
||DINING_PHILOSOPHERS =
   forall [i:0..N-1] (phil[i]:PHIL ||
                        { phil[i].left, phil[((i-1)+N)%N].right }::FORK).
```

The applet's start() method creates (an array of) shared **Fork** monitors...:

```
for (int i=0; i<N; i++) fork[i] = new Fork(display, i);
```

...and (an array of) **Philosopher** threads (with refs to forks):

```
for (int i=0; i<N; i++)
    phil[i] =
                                         left              right
        new Philosopher(this, i, fork[(i-1+N)%N], fork[i]);
```

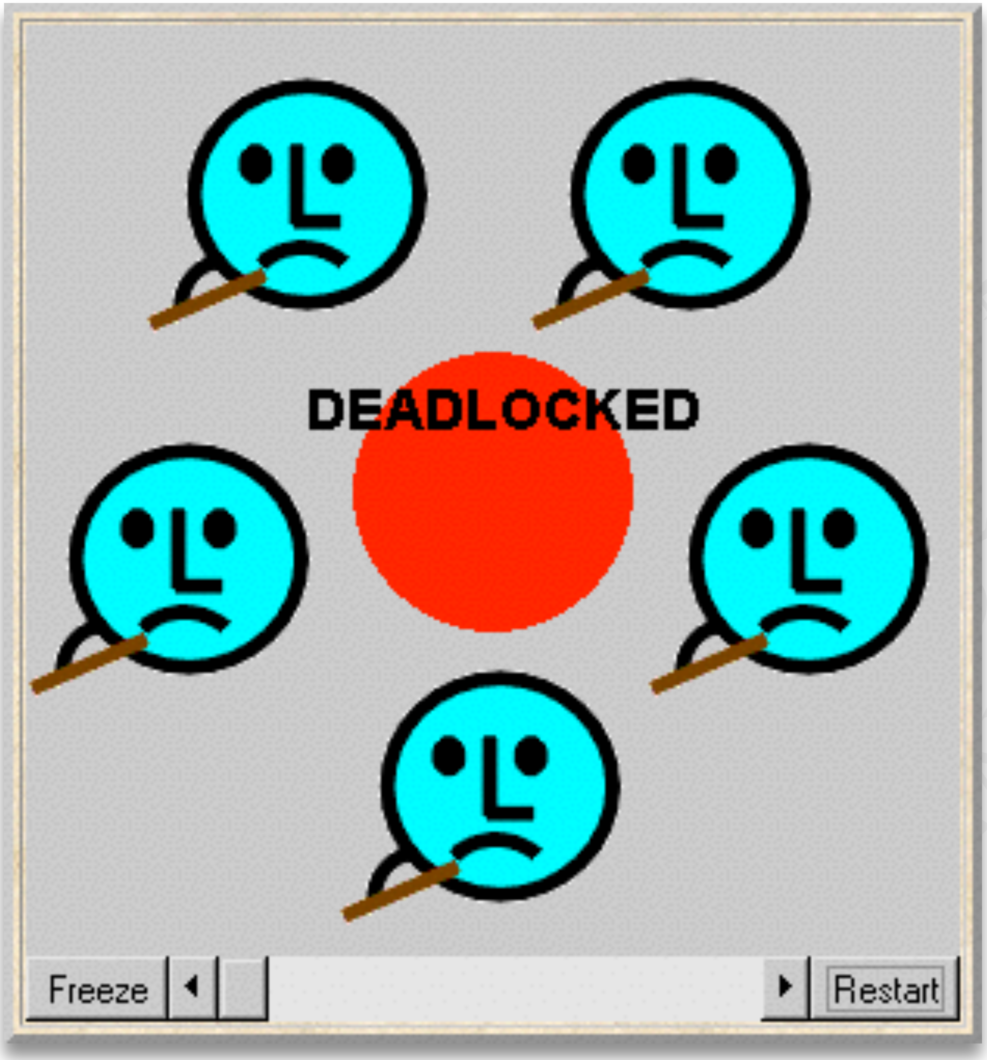
...and start all Philosopher threads:

```
for (int i=0; i<N; i++) phil[i].start();
```

Dining Philosophers

To ensure deadlock occurs eventually, the slider control may be moved to the left. This reduces the time each philosopher spends thinking and eating.

This "speedup" increases the **probability** of deadlock occurring.



Deadlock-Free Philosophers

University of Southern Denmark

Deadlock can be avoided by ensuring that a wait-for cycle cannot exist.

How?

Introduce an **asymmetry** into definition of philosophers.

Use the identity `'i'` of a philosopher to make even numbered philosophers get their left forks first, odd their right first.

```
PHIL[i:0..N-1] =
   (when (i%2==0) sitdown-> left.get ->...-> PHIL
   |when (i%2==1) sitdown-> right.get->...-> PHIL).
```

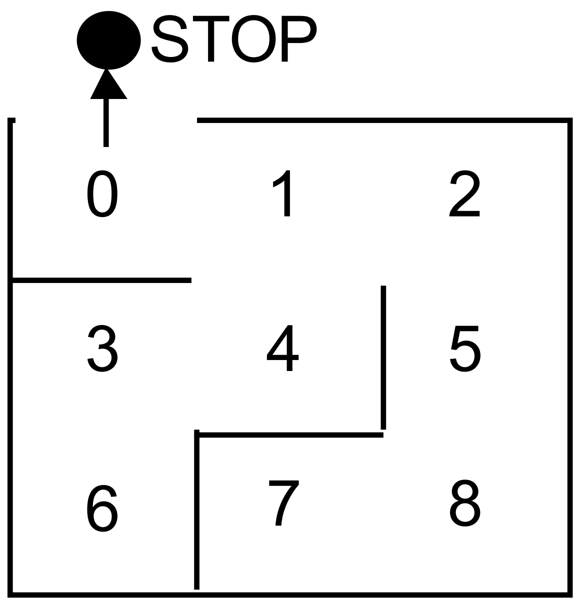
How does this solution compare to the "sort-shared-acquisitions" idea?

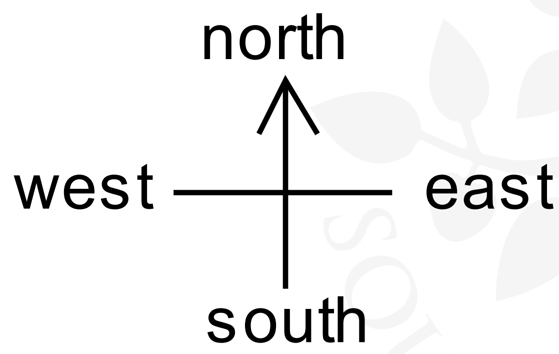
1. Mutual exclusion condition
2. Hold-and-wait condition
3. No pre-emption condition
4. Circular-wait condition

Other strategies?

Maze Example - Shortest Path To "Deadlock"

We can exploit the shortest path trace produced by the deadlock detection mechanism of **LTSA** to find the shortest path out of a maze to the `STOP` process!





We first model the `MAZE`.

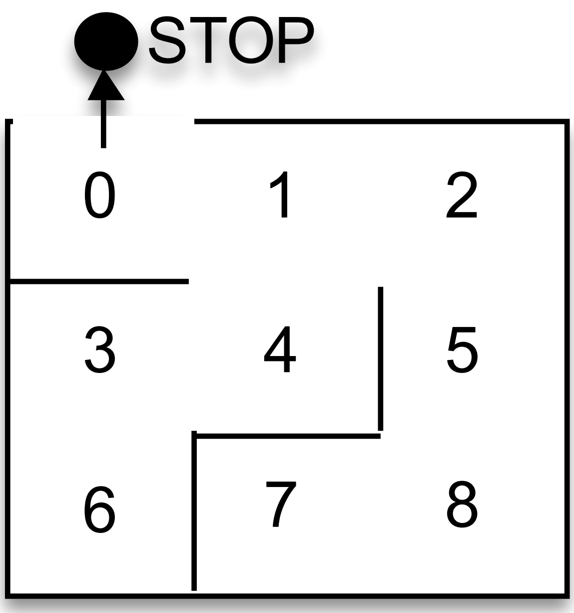
Each position is modelled by the moves that it permits. The `MAZE` parameter gives the starting position.

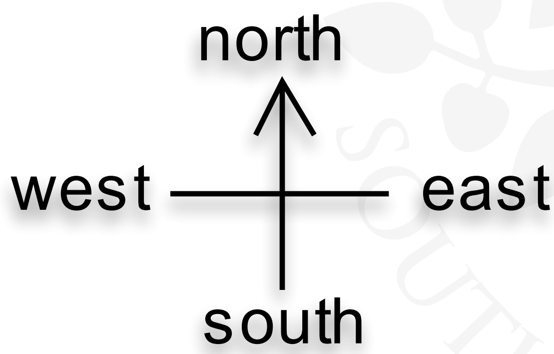
```
eg.  MAZE(Start=8) = P[Start],
     P[0] = (north->STOP|east->P[1]),...
```

Maze Example - Shortest Path To "Deadlock"


```
||GETOUT = MAZE(7).
```

Shortest path escape trace from position 7 ?






```
Trace to
DEADLOCK:

    east
    north
    north
    west
    west
    north
```

Summary

- Concepts
  - deadlock (no further progress)
  - 4x necessary and sufficient conditions:
    1. Mutual exclusion condition
    2. Hold-and-wait condition
    3. No pre-emption condition
    4. Circular-wait condition
- Models
  - no eligible actions (analysis gives shortest path trace)
- Practice
  - blocked threads

Aim - deadlock avoidance:

"Break at least one of the deadlock conditions".