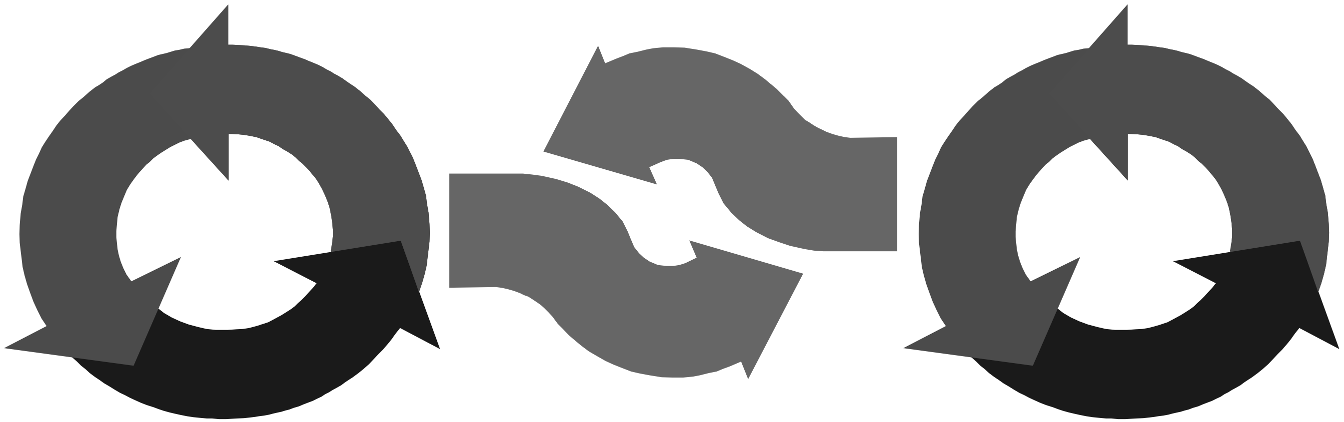
6 - Deadlock



Credits for the slides:
Claus Braband
Jeff Magee & Jeff Kramer

Alexandre David
adavid@cs.aau.dk

©Magee/Kramer

Monitors & Condition Synchronization - Repetition

Concepts: monitors:

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

nested monitors

Models: guarded actions

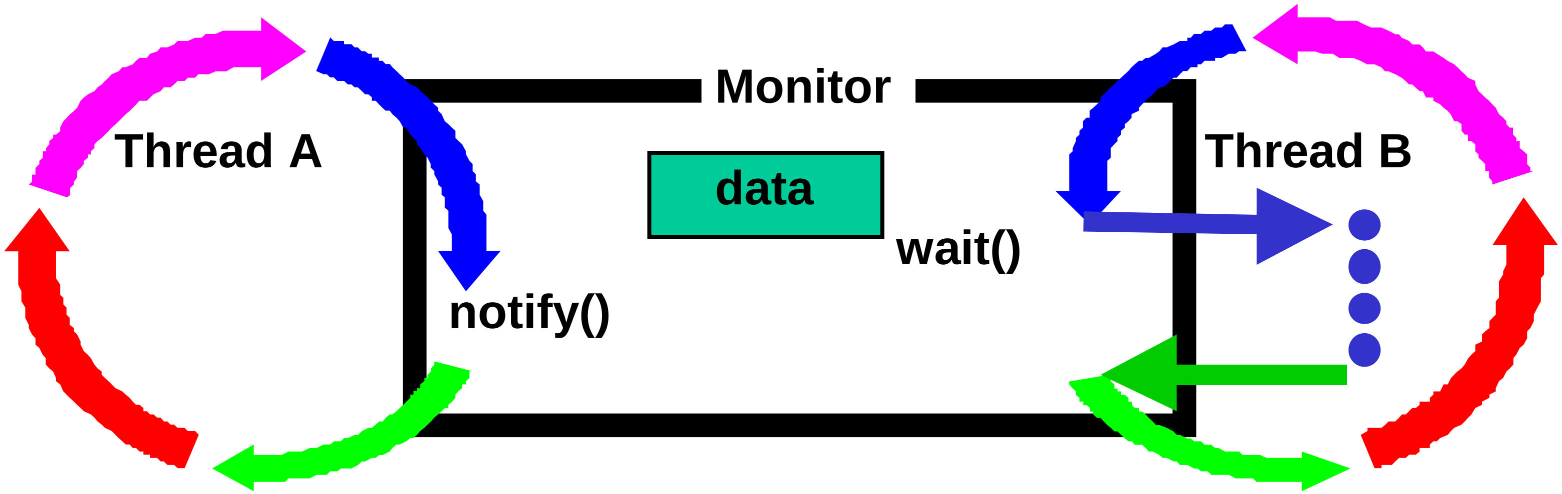
Practice: private data and synchronized methods (exclusion).
wait(), notify() and notifyAll() for condition synch.
single thread active in the monitor at a time

©Magee/Kramer

wait(), notify(), and notifyAll() - Repetition

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

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

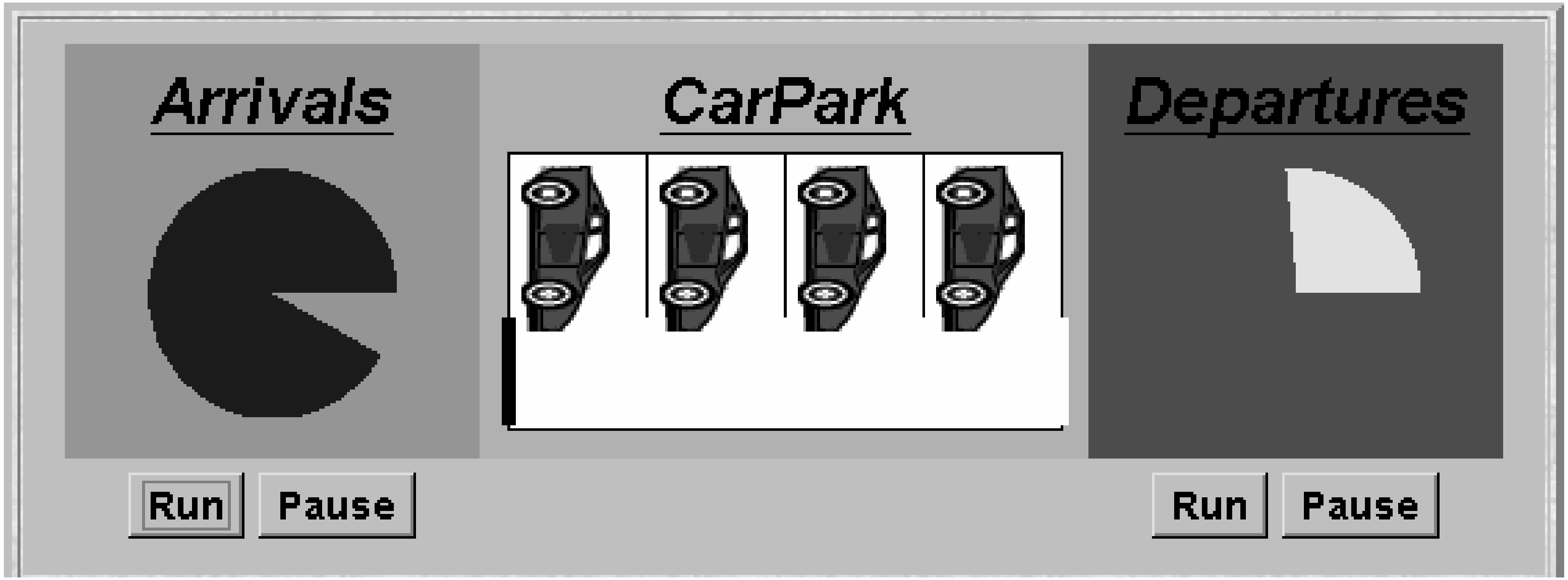



```
public final void notify();
```

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


©Magee/Kramer

The Car Park Example - Repetition



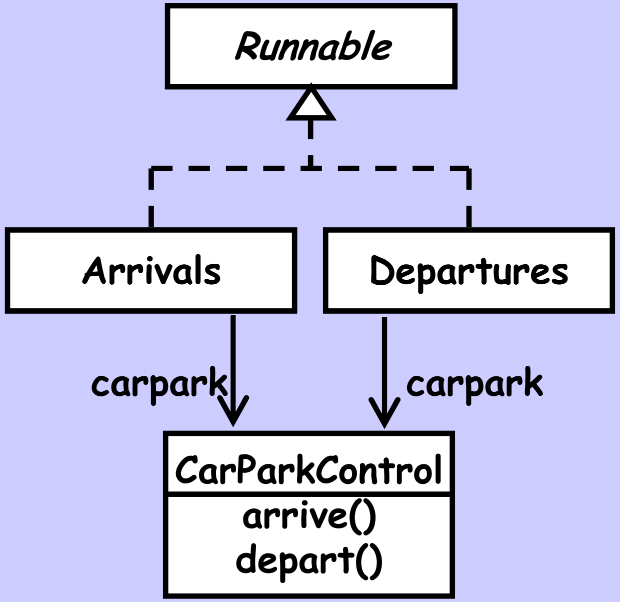
A controller is required to ensure:

- cars can only enter when not full
- cars can only leave when not empty (duh!)

©Magee/Kramer

Condition Synchronization (in Java) - Repetition

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

    synchronized void arrive()
                throws Int'Exc' {
        while (spaces==0) wait();
        --spaces;
        notify();
    }

    synchronized void depart()
                throws Int'Exc' {
        while (spaces==capacity) wait();
        ++spaces;
        notify();
    }
}
```

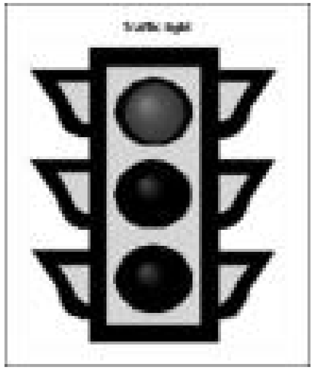


©Magee/Kramer

Semaphores - Repetition

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

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

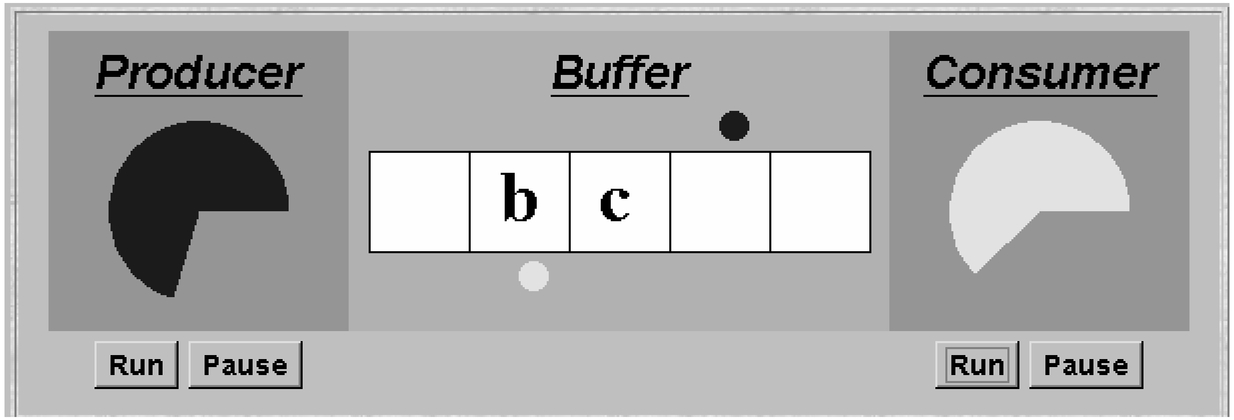


down(s): **if** (s>0) **then** decrement(s);
else block execution of calling process
Aka. "P" ~ Passern

up(s): **if** (processes blocked on s) **then** awake one of them
else increment(s);
Aka. "V" ~ Vrijgeven

©Magee/Kramer

Nested Monitors - Bounded Buffer Model - Repetition



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*.

©Magee/Kramer

Deadlock

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.

©Magee/Kramer

Deadlock: 4 Necessary and Sufficient Conditions

1. **Serially reusable resources:**

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

2. **Incremental acquisition:**

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

3. **No pre-emption:**

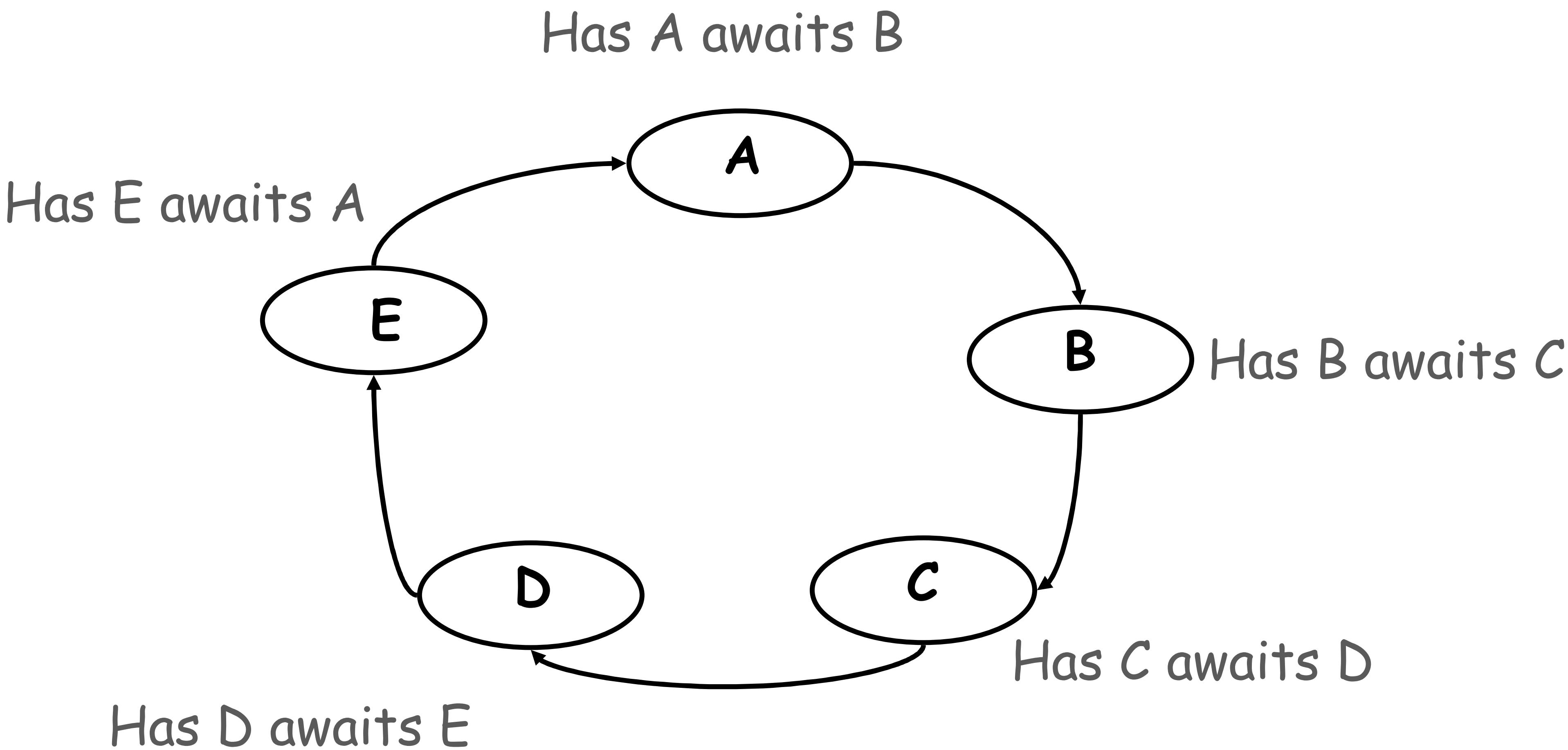
once acquired by a process, resources cannot be "pre-empted" (forcibly withdrawn) but are only released voluntarily.

4. **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.

©Magee/Kramer

Wait-For Cycle

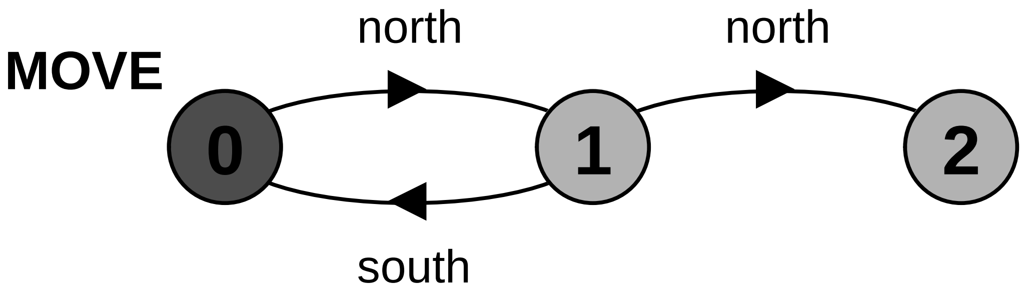


©Magee/Kramer

6.1 Deadlock Analysis - Primitive Processes

- **Deadlocked state** has no outgoing transition
- In FSP: (modelled by) the **STOP** state

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



- Analysis using ***LTSA***:

Shortest path to DEADLOCK:

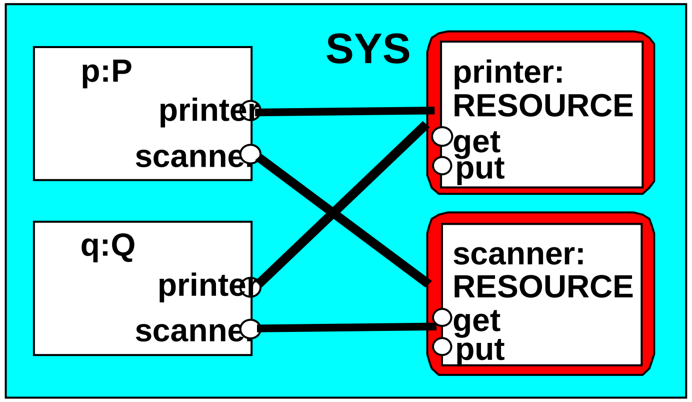
```
Trace to DEADLOCK:
    north
    north
```


©Magee/Kramer

Deadlock Analysis - Parallel Composition

- In practise, 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-> scanner.put-> P).

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

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

Deadlock trace?

Avoidance...

©Magee/Kramer

Recall the 4 Conditions…

1. **Serially reusable resources:**

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

2. **Incremental acquisition:**

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

3. **No pre-emption:**

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

4. **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.

©Magee/Kramer

Deadlock Analysis – Avoidance (#1 ?)

1. Serially reusable resources:
the processes involved share resources which they use under mutual exclusion.

- Inherent in *"copy using shared scanner/printer"* problem.

©Magee/Kramer

Deadlock Analysis – Avoidance (#2 ?)

2. Incremental acquisition:
processes hold on to resources already allocated to them while waiting to acquire additional resources.

- A "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?* ☹

©Magee/Kramer

Deadlock Analysis – Avoidance (#3 ?)

3. No pre-emption:
once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily.

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

```
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?* ☹

©Magee/Kramer

Deadlock Analysis – Avoidance (#4 ?)

4. 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:

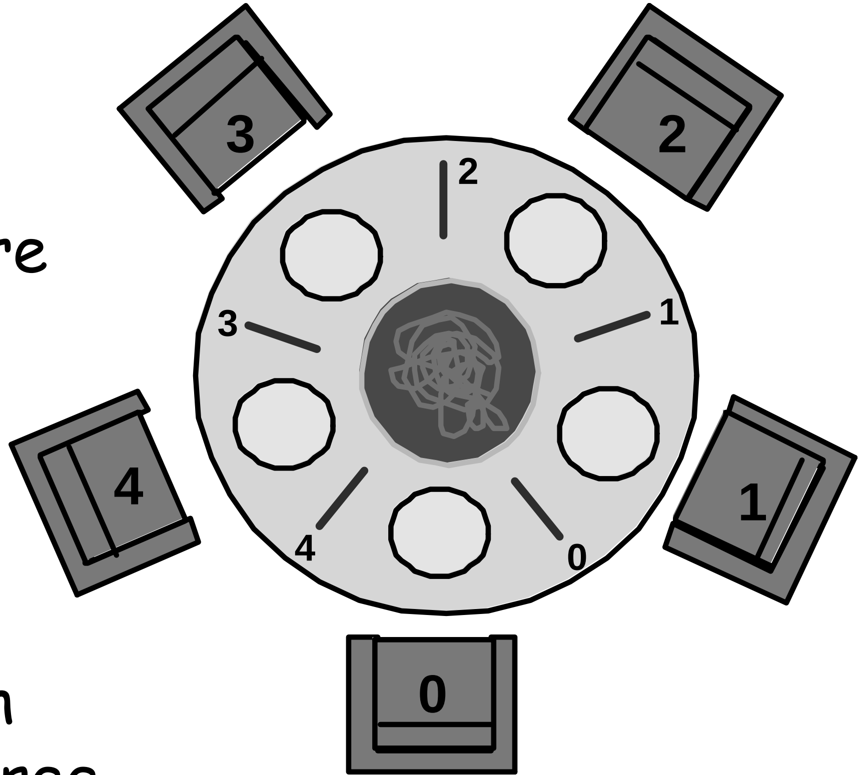
```
P = (printer.get->
       scanner.get->
         copy->
           printer.put-> scanner.put-> P).

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

Deadlock? ☺   Scalability/Progress/...? ☺

©Magee/Kramer

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.



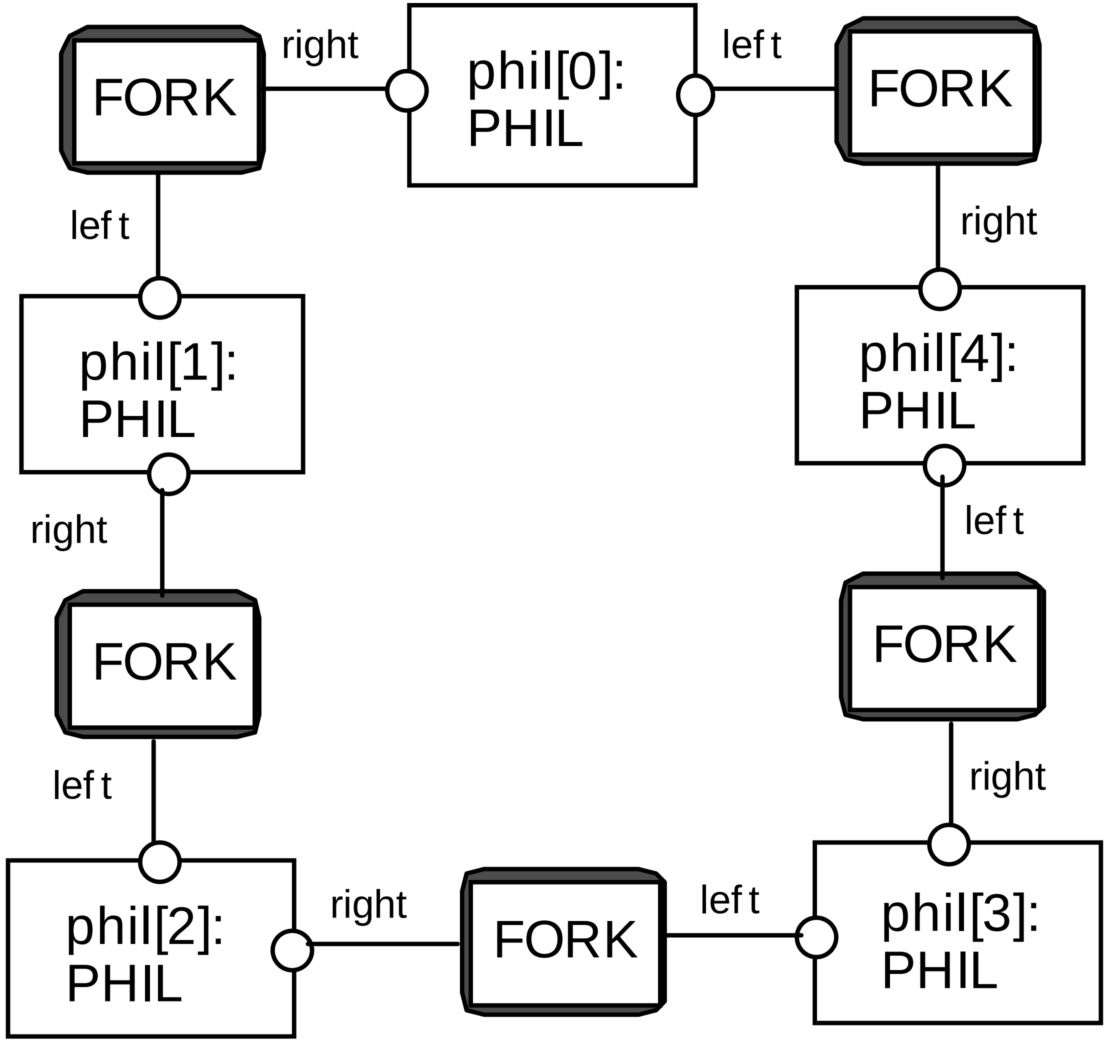
One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.

©Magee/Kramer

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.



©Magee/Kramer

Dining Philosophers - Model

```
const N = 5

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

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


©Magee/Kramer

Dining Philosophers - Model Analysis

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

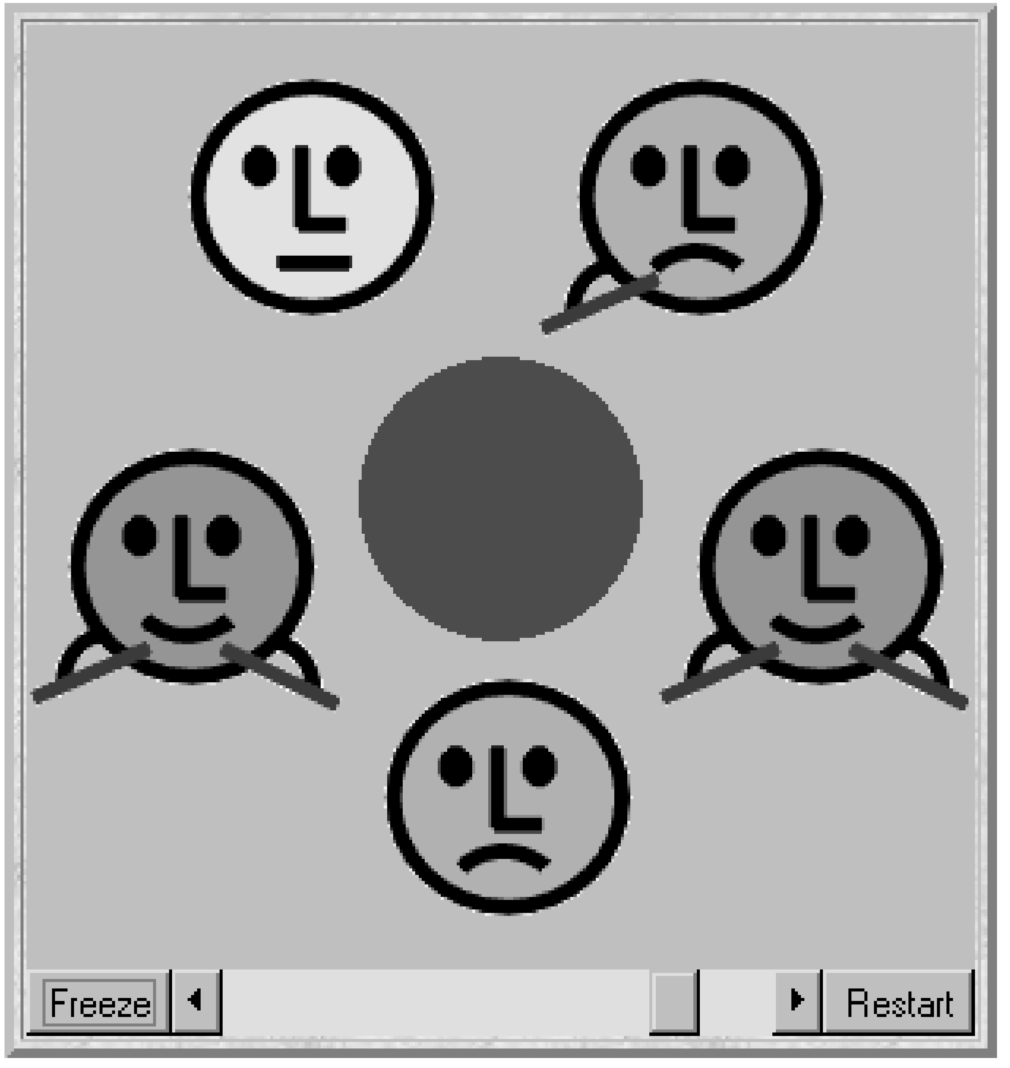
The system can make no further progress since each philosopher is waiting for a left fork held by his neighbour (i.e., a ***wait-for cycle*** exists)!

©Magee/Kramer

Dining Philosophers

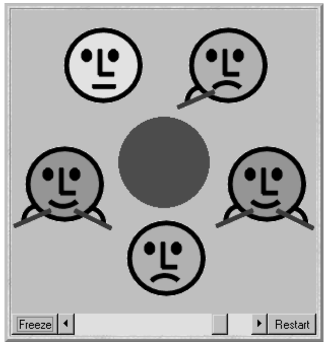
Deadlock is easily detected in our model.

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

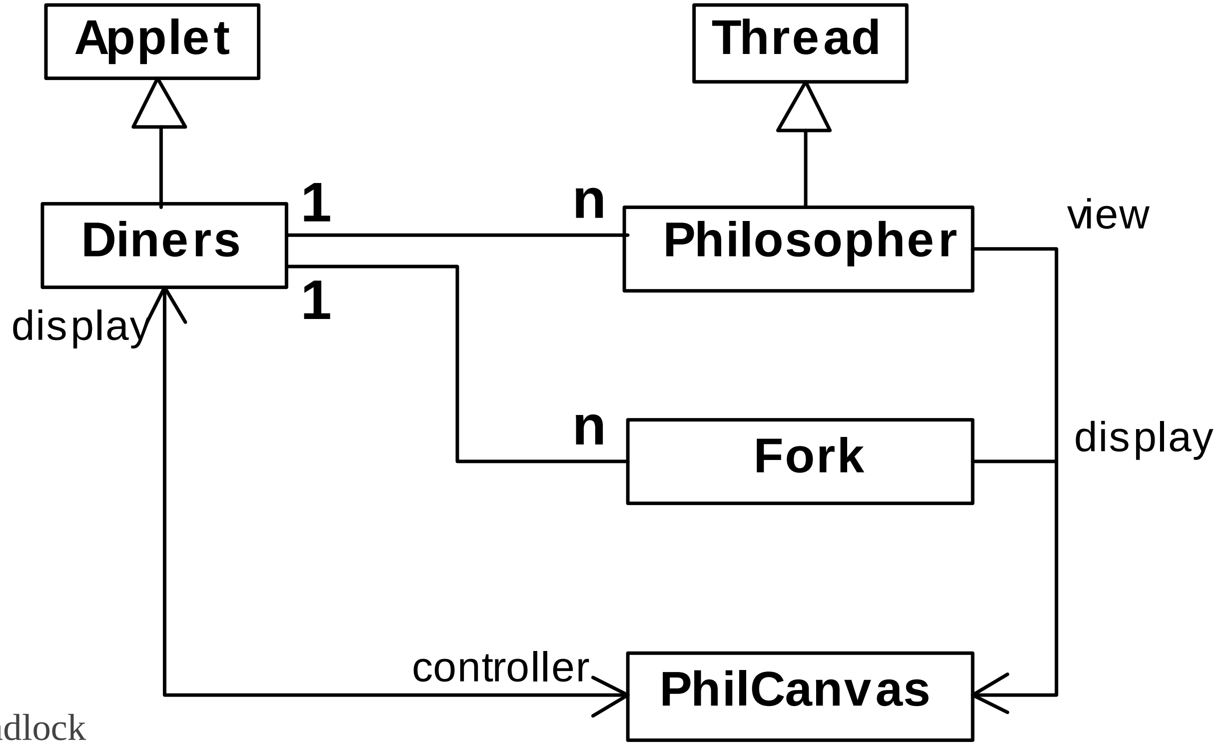


©Magee/Kramer

Dining Philosophers - Implementation in Java



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



©Magee/Kramer

Dining Philosophers – Fork (Monitor)

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

    Fork(PhilCanvas disp, int id)
        { display = disp; identity = id;}

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

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

taken encodes the state of the fork

©Magee/Kramer

Dining Philosophers – Philosopher (Thread)

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

Sitting down and leaving the table has been omitted.

©Magee/Kramer

Dining Philosophers – Main Applet

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 each of which is start()'ed:

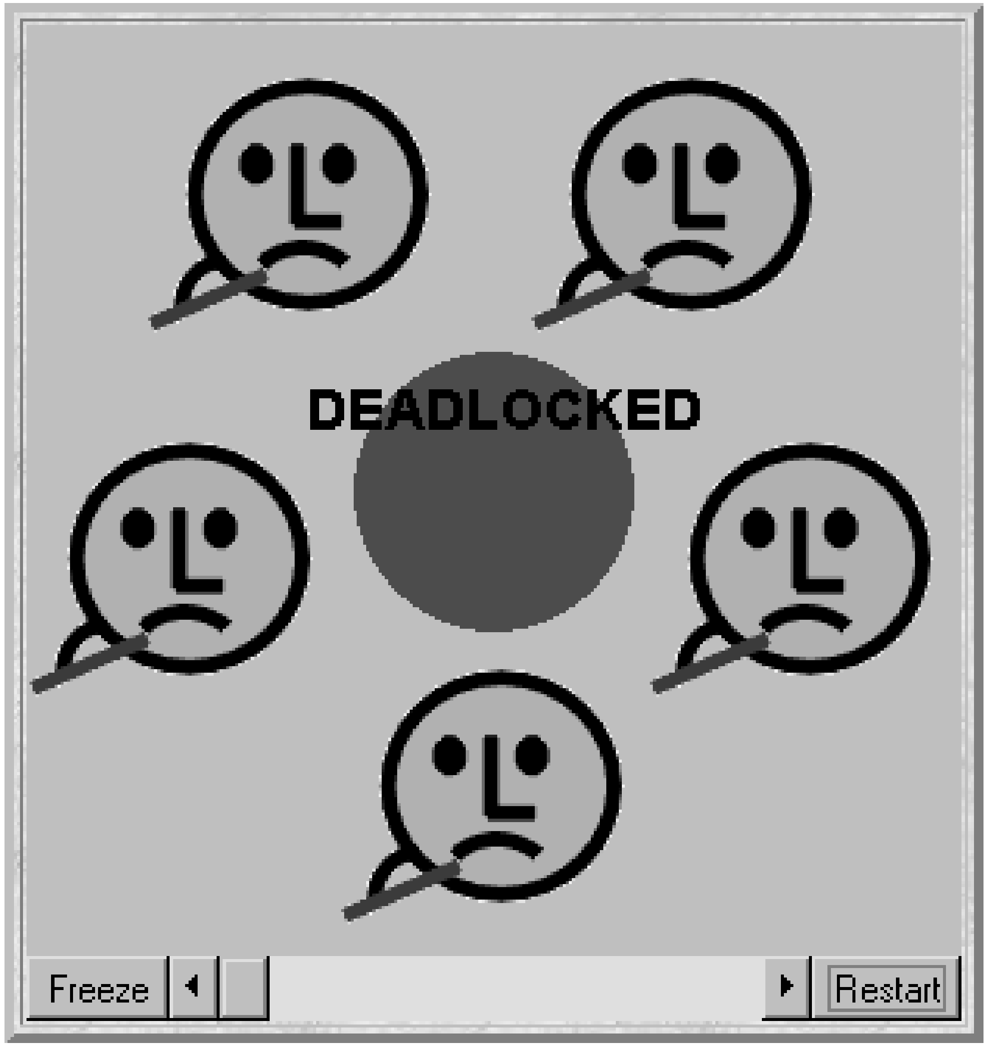
```
for (int i=0; i<N; i++) {
    phil[i] =
                                      left              right
        new Philosopher(this, i, fork[(i-1+N)%N], fork[i]);
    phil[i].start();
}
```


©Magee/Kramer

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.



©Magee/Kramer

Deadlock-free Philosophers

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.

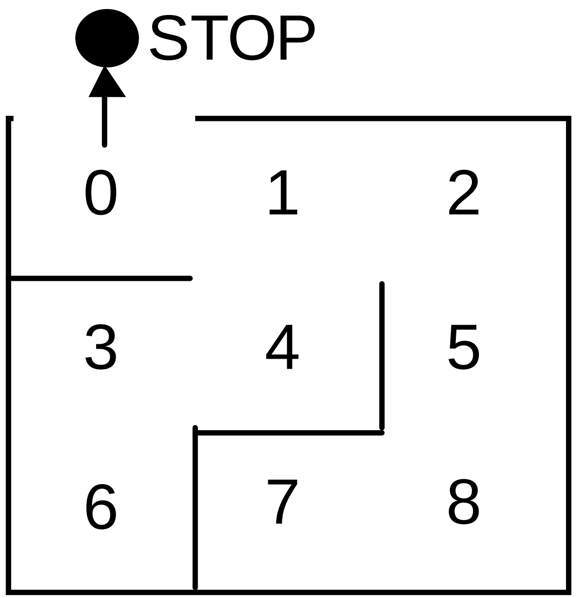
```
PHIL[i:0..N-1] =
  (when (i%2==0) sitdown-> left.get-> ...-> PHIL
  |when (i%2==1) sitdown-> right.get->...-> PHIL).
```

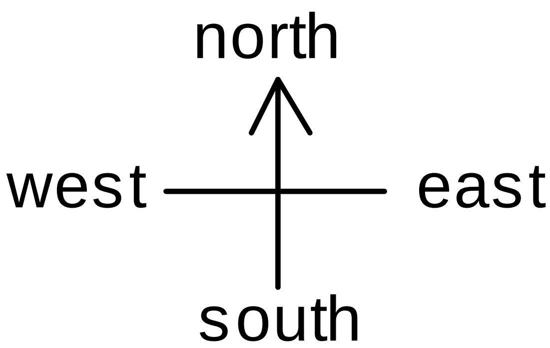
Other strategies?

©Magee/Kramer

Maze Example - Shortest Path to STOP (Goal State)

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 must first model the **MAZE**.

Each position can be modelled by the moves that it permits. The **MAZE** parameter gives the starting position.

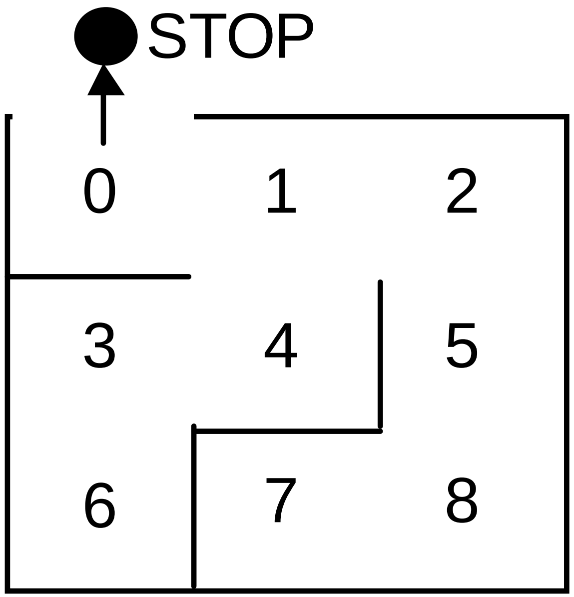
```
eg. MAZE(Start=8) = P[Start],
    P[0] = (north->STOP|east->P[1]),...
```

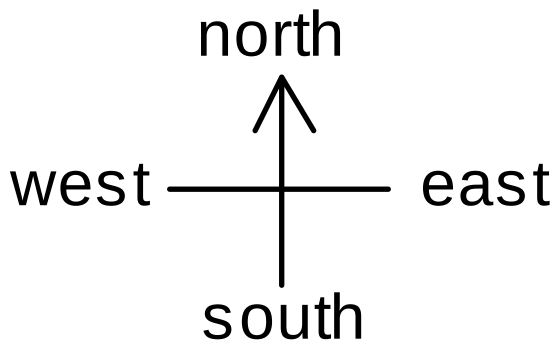

©Magee/Kramer

Maze Example - Shortest Path to STOP (Goal State)

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

Shortest path escape trace from position 7 ?






```
Trace to
DEADLOCK:
    east
    north
    north
    west
    west
    north
```


©Magee/Kramer

Summary

- Concepts
  - deadlock (no further progress)
  - 4x necessary and sufficient conditions:
    1. Serially reusable resources
    2. Incremental acquisition
    3. No preemption
    4. Wait-for cycle

Aim - deadlock avoidance:

"Design systems where deadlock cannot occur".

- Models
  - no eligible actions (analysis gives shortest path trace)
- Practice
  - blocked threads

©Magee/Kramer