# Lecture 8

- Administration

# Nested Monitors

Suppose that, in place of using the ***count*** variable and condition synchronization directly, we instead use two semaphores ***full*** and ***empty*** to reflect the state of the buffer.

# bounded buffer model

```
const Max = 5
range Int = 0..Max

SEMAPHORE ...as before...

BUFFER =  (put -> empty.down ->full.up ->BUFFER
          |get -> full.down ->empty.up ->BUFFER
          ).

PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).

||BOUNDEDBUFFER = (PRODUCER|| BUFFER || CONSUMER
                  ||empty:SEMAPHORE(5)
                  ||full:SEMAPHORE(0)

                      )@{put,get}.
```

***Does this behave as desired?***

# bounded buffer model

*LTSA* analysis predicts a possible **DEADLOCK**:

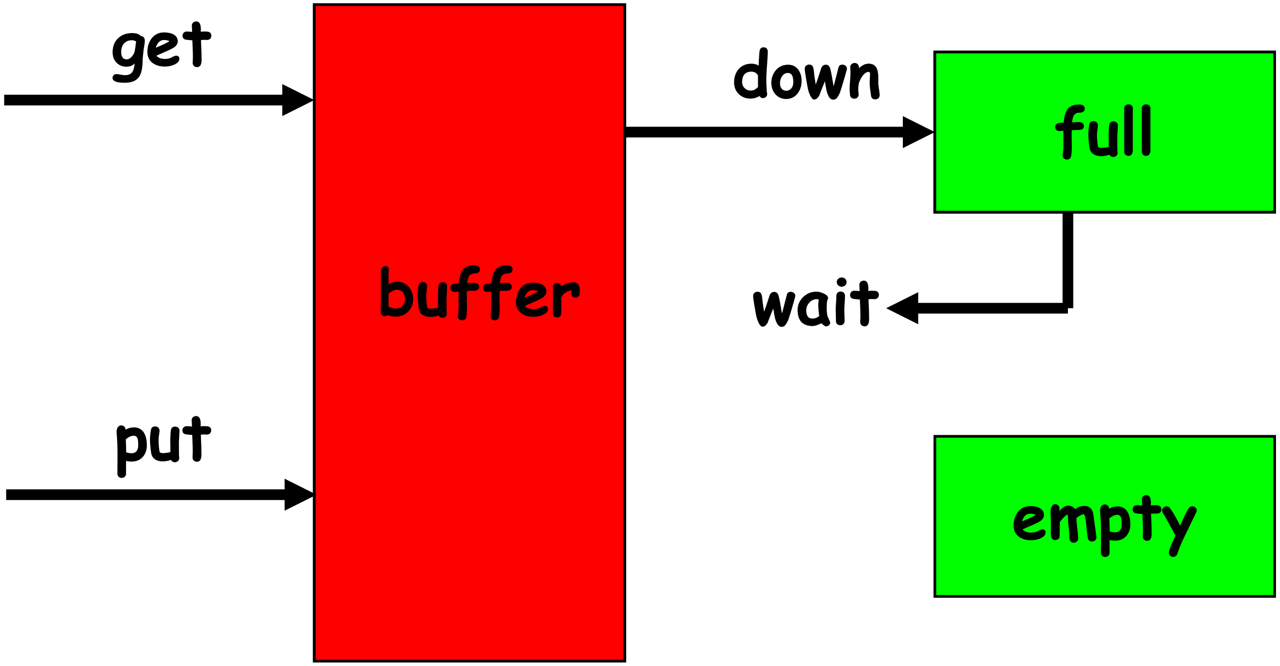
```
Composing
 potential DEADLOCK
States Composed: 28 Transitions: 32 in
60ms
Trace to DEADLOCK:
   get
```

The **`Consumer`** tries to **`get`** a character, but the buffer is empty. It blocks and releases the lock on the semaphore **`full`**. The **`Producer`** tries to **`put`** a character into the buffer, but also blocks. ***Why?***

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

# bounded buffer model

```
synchronized public Object get()
                   throws InterruptedException{
    full.down(); // if no items, block!
    ...
  }
```



# revised bounded buffer program

The only way to avoid it in Java is by careful design. In this example, the deadlock can be removed by ensuring that the monitor lock for the buffer is not acquired until ***after*** semaphores are decremented.

# revised bounded buffer model

```
BUFFER =  (put -> BUFFER
          |get -> BUFFER
          ).

PRODUCER =(empty.down->put->full.up->PRODUCER).
CONSUMER =(full.down->get->empty.up->CONSUMER).
```

The semaphore actions have been moved to the producer and consumer. This is exactly as in the implementation where the semaphore actions are *outside* the monitor .

*Does this behave as desired?*

*Minimized LTS?*

# Monitor invariants

An **invariant** for a monitor is an assertion concerning the variables it encapsulates. This assertion must hold whenever there is no thread executing inside the monitor i.e. on thread **entry** to and **exit** from a monitor .

CarParkControl Invariant: $\mathbf{0} \leq \boldsymbol{spaces} \leq \boldsymbol{N}$

Semaphore Invariant: $\mathbf{0} \leq \boldsymbol{value}$

Buffer Invariant: $\mathbf{0} \leq \boldsymbol{count} \leq \boldsymbol{size}$
**and** $\mathbf{0} \leq \boldsymbol{in} < \boldsymbol{size}$
**and** $\mathbf{0} \leq \boldsymbol{out} < \boldsymbol{size}$
**and** $\boldsymbol{in} = (\boldsymbol{out} + \boldsymbol{count})\ \mathbf{modulo}\ \boldsymbol{size}$

Invariants can be helpful in reasoning about correctness of monitors using a logical *proof-based* approach. Generally we prefer to use a *model-based* approach amenable to mechanical checking .

# Deadlock

**Concepts**: system deadlock: no further progress

four necessary & sufficient conditions

**Models**: deadlock - no eligible actions

**Practice**: blocked threads

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

# Deadlock: four necessary and sufficient conditions

- **Serially reusable resources:**

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

- **Incremental acquisition:**

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

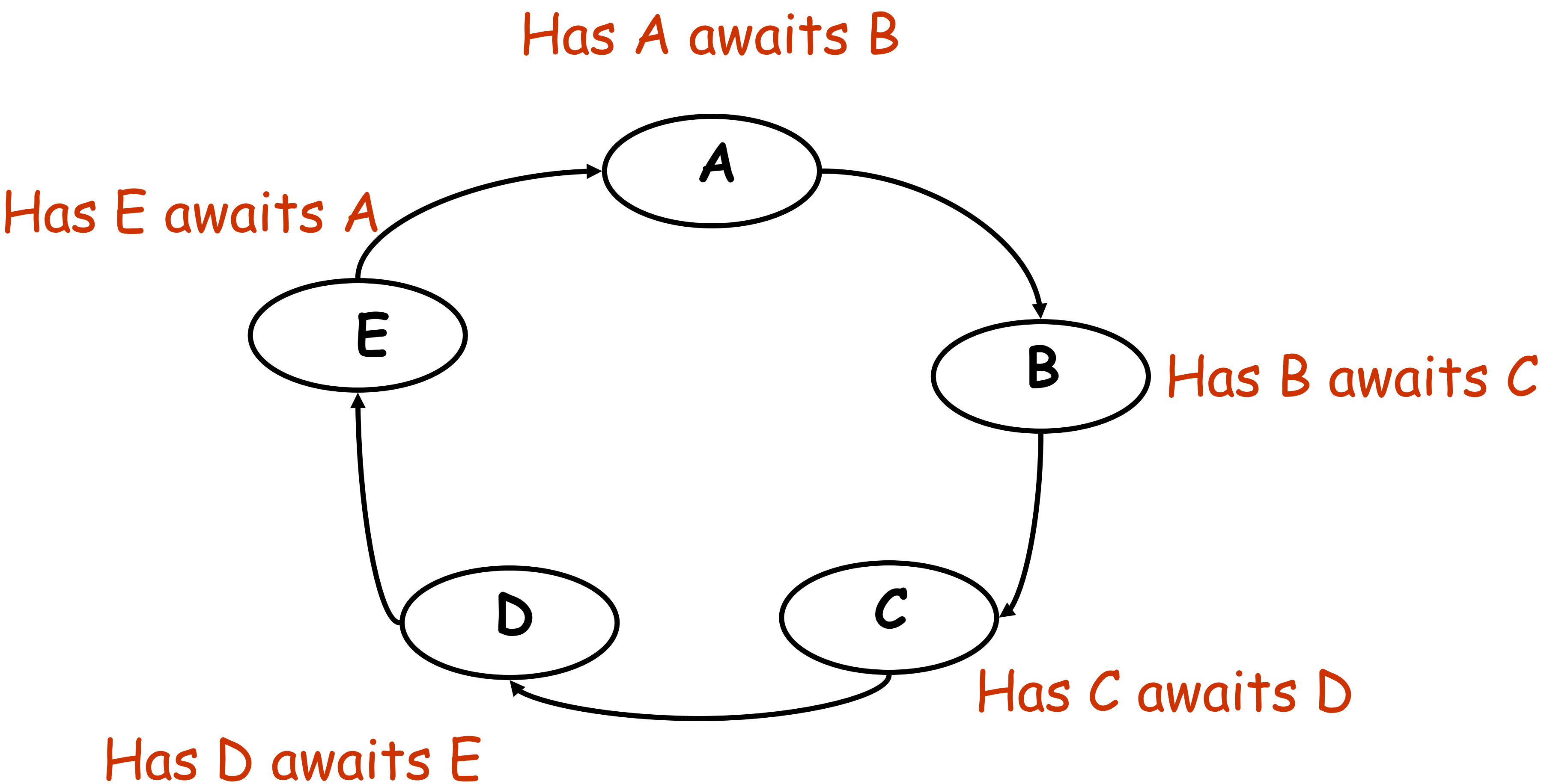
- **No pre-emption:**

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

- **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.*

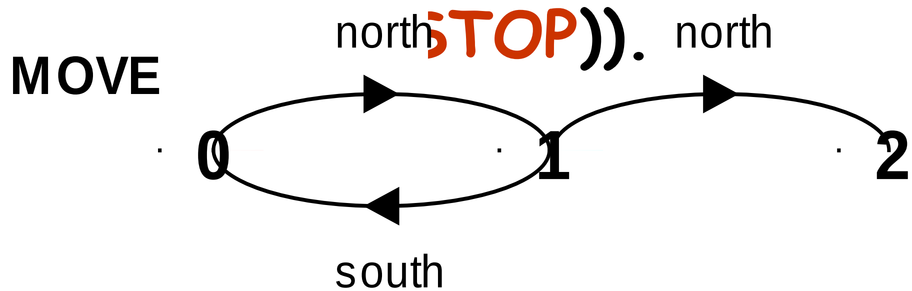
# Wait-for cycle



# Deadlock analysis - primitive processes

- deadlocked state is one with no outgoing transitions
- in FSP: STOP process

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



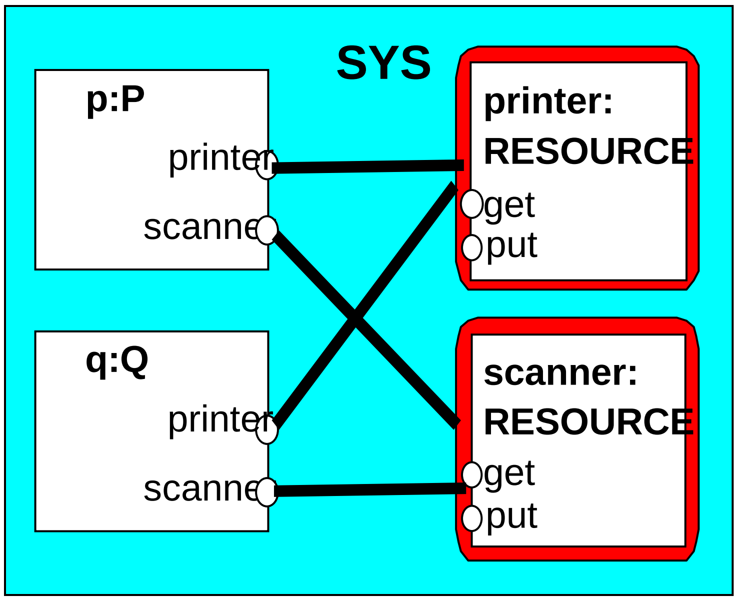

- animation to produce a trace.
- analysis using *LTSA*:

(shortest trace to STOP)

**Trace to DEADLOCK:**
**north**
**north**

# deadlock analysis - parallel composition

- in systems, deadlock may arise from the parallel composition of interacting processes.




```
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?

# deadlock analysis - avoidance

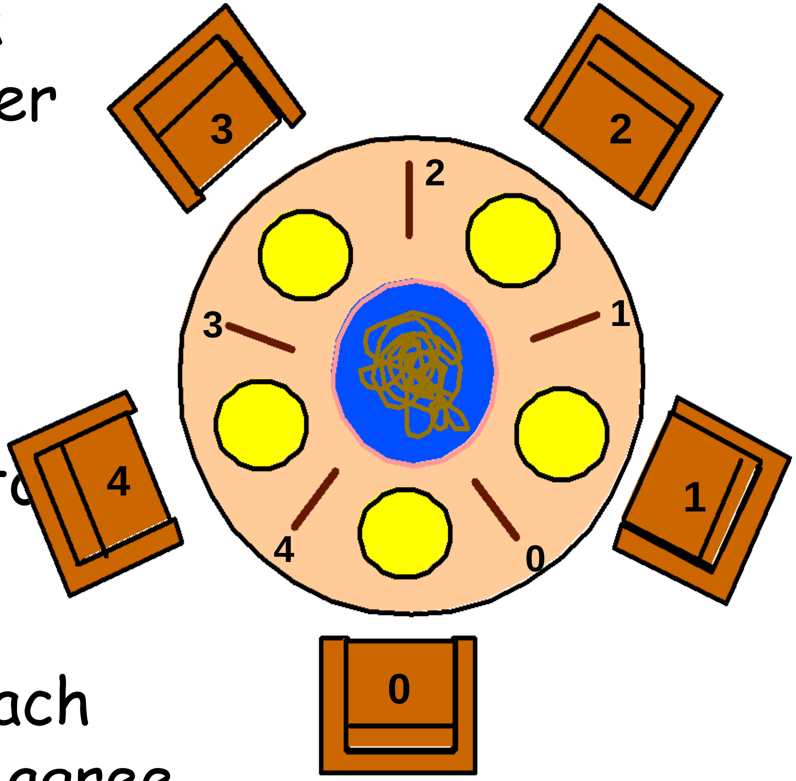
- acquire resources in the same order?
- Timeout:

```
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?***

# 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.