

# Discrete Control

## Real-Time Systems, Lecture 14

---

Karl-Erik Årzén

1 March 2018

Lund University, Department of Automatic Control

[Real-Time Control System: Chapter 12]

1. Discrete Event Systems
2. State Machine Formalisms
3. Statecharts
4. Grafcet
5. Petri Nets
6. Implementation

# Discrete Event Systems

---

# Discrete Event Systems

A *Discrete Event System (DES)* is a *discrete-state, event-driven* system, that is its state evolution depends entirely on the occurrence of asynchronous discrete events over time.

Sometimes the name *Discrete Event Dynamic Systems (DEDS)* is used to emphasize the dynamic nature of a DES.

# Discrete Event Systems

Discrete Event Systems:

- The state space is a discrete set.
- The state transition mechanism is event-driven.
- The events need **not** to be synchronized by, e.g., a clock.

Continuous Systems:

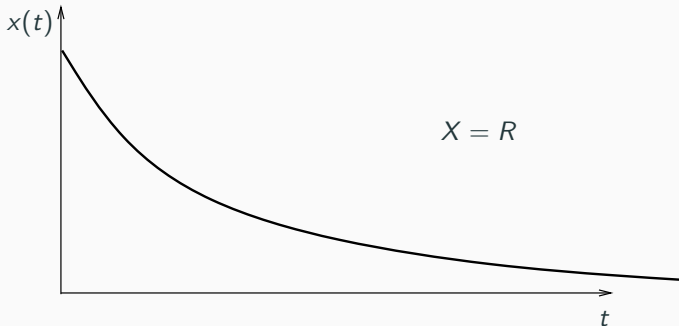
- Continuous-state systems (real-valued variables)
- The state-transition mechanism is time-driven.

Continuous discrete-time systems  $x(k+1) = Ax(k) + Bu(k)$  can be viewed as an event-driven system synchronized by clock events.

# Continuous Systems

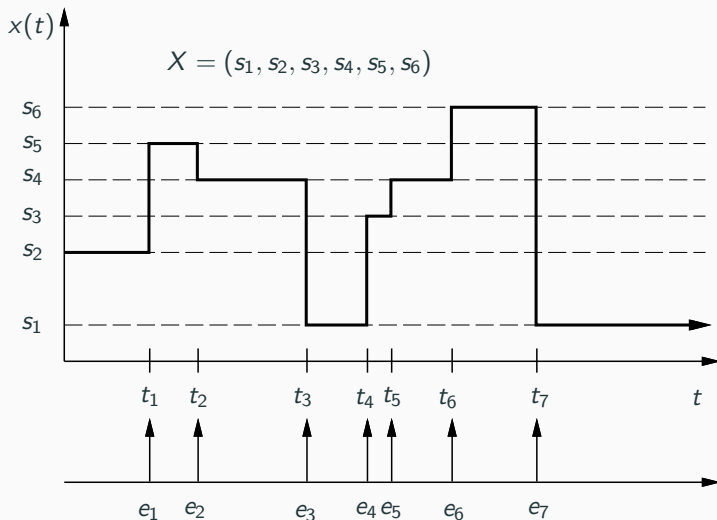
State trajectory is the solution of a differential equation

$$\dot{x}(t) = f(x(t), u(t), t)$$



# Discrete Event System

State trajectory (sample path) is piecewise constant function that jumps from one value to another when an event occurs.



All processes contain discrete elements:

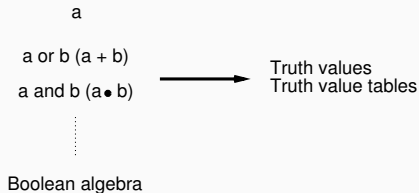
- continuous processes with discrete sensors and/or actuators;
- discrete processes:
  - manufacturing lines, elevators, traffic systems;
- mode changes:
  - manual/auto, startup/shutdown,
  - production (grade) changes;
- alarm and event handling.



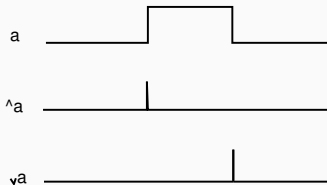
- Larger in volume than continuous control;
- Limited theoretical support:
  - verification, synthesis;
  - formal methods beginning to emerge;
  - still not widespread in industry
  - active research area

# Basic Elements

- Boolean (binary) signals – 0, 1,  
*false, true, a,  $\bar{a}$*
- Expressions



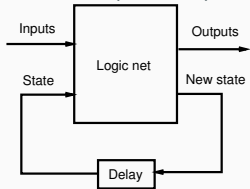
- Events



# Logic Nets

- Combinatorial nets
  - $\text{outputs} = f(\text{inputs})$
  - interlocks, "förreglingar"
- Sequence nets
  - $\text{newstate} = f(\text{state}, \text{inputs})$
  - $\text{outputs} = g(\text{state}, \text{inputs})$
  - state machines
  - automata

Asynchronous nets or synchronous (clocked) nets



# State Machine Formalisms

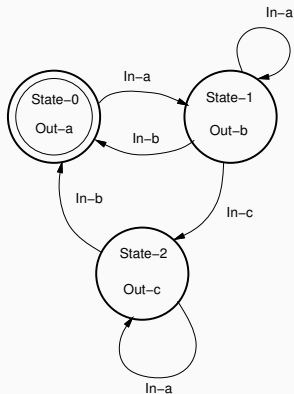
---

Formal properties  $\Rightarrow$  analysis possible in certain cases

Using state machines is often a good way to structure code.

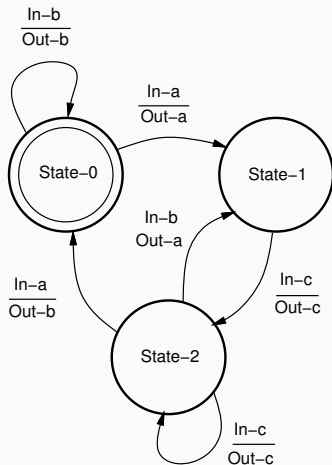
Systematic ways to write automata code often not taught in programming courses.

# Moore Machine



State transitions in response to input events  
Output events (actions) associated with states

# Mealy Machine



Output events (actions) associated with input events

# State Machine Extensions

Ordinary state machines lack structure so extensions are needed to make them practically useful:

- hierarchy;
- concurrency;
- history (memory).



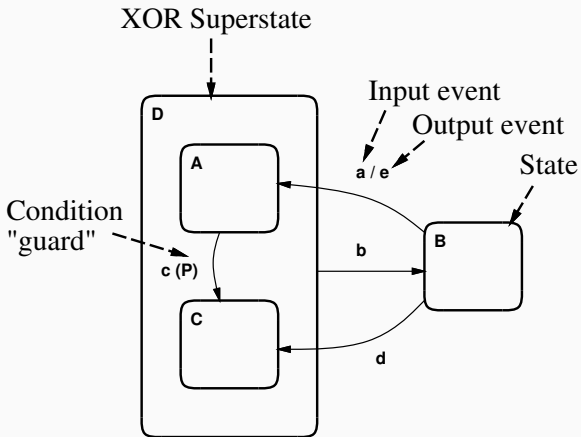
# Statecharts

---

# Statecharts

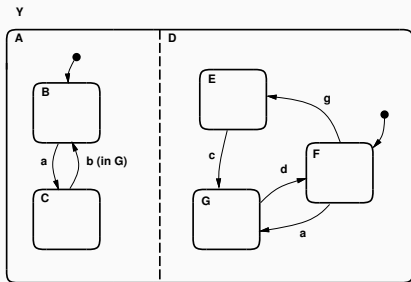
D. Harel, 1987: Statecharts are state machines with hierarchy, concurrency and history.

XOR superstates



# Statecharts

## AND superstates

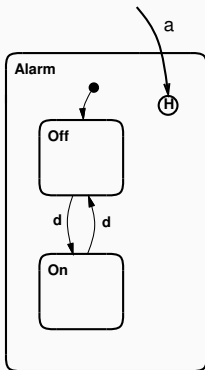


Y is the *orthogonal product* of A and D

When in state (B,F) and event **a** occurs, the system transfers *simultaneously* to (C,G).

# Statecharts

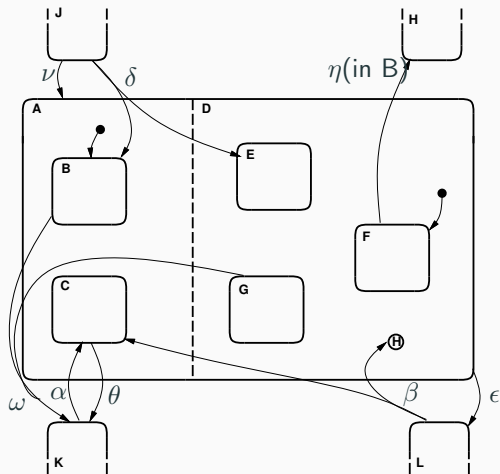
History states



On event 'a' the last visited state within D becomes active.

# Statecharts

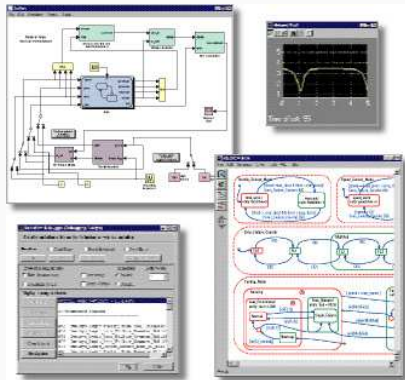
## Interfaces for AND and superstates



- $\delta$  exit from  $J \Rightarrow (B, E)$
- $\alpha$  exit from  $K \Rightarrow (C, F)$
- $\nu$  exit from  $J \Rightarrow (B, F)$
- $\beta$  exit from  $L \Rightarrow (C, \text{most recently visited state in } D)$
- $\omega$  exit from  $(B, G) \Rightarrow K$
- $\eta$  exit from  $(B, F) \Rightarrow H$
- $\theta$  exit from  $(C, D) \Rightarrow K$
- $\epsilon$  exit from  $(A, D) \Rightarrow L$

# Statecharts tools

Statecharts popular for modeling, simulation, and code generation. Used to represent state-transition diagrams in UML tools (Rational/Rose, Rhapsody). Stateflow for Matlab/Simulink.



Unfortunately, Harel only gave an informal definition of the semantics. As a result, a number of competing semantics were defined.

In 1996, Harel presented his semantics (the StateMate semantics) of Statechart and compared it with 11 other semantics.

The lack of a single semantics is still the major problem with Statecharts. Each tool vendor defines his own.



# Grafcet

---

Extended state machine formalism for implementation of sequence control.

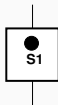
Industrial name: Sequential Function Charts (SFC).

Defined in France in 1977 as a formal specification and realization method for logical controllers.

Part of IEC 61131-3 (industry standard for PLC controllers).

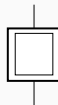
# Basic Elements

- Steps: active or inactive



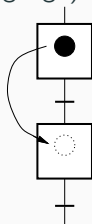
$S1.x = 1$  when active

$S1.T$  = number of time units since the step last became active



Initial step

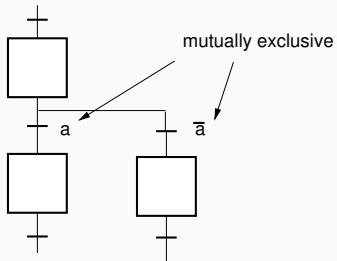
- Transitions ("övergång")



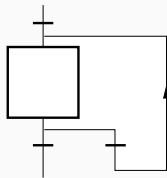
condition true and/or event occurred +  
previous step active

# Basic Elements

- Alternative paths: branch

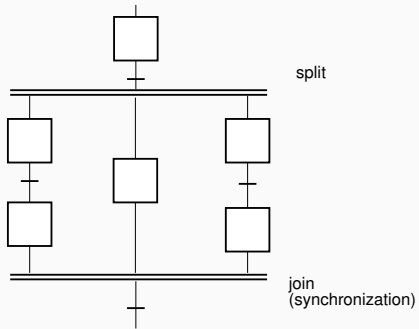


- Alternative paths: repetition



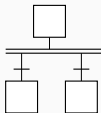
# Basic Elements

- Parallel paths:

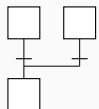
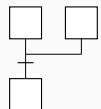
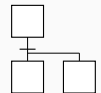
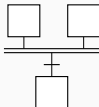
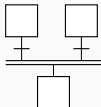
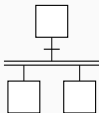


# Basic Elements

Illegal Grafcet



Legal Grafcet

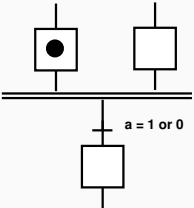


1. The initial step(s) is active when the function chart is initiated.
2. A transition is fireable if:
  - all steps preceding the the transition are active (enabled);
  - the receptivity (transition condition and/or event) of the transition is true.

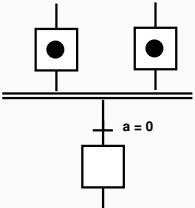
A fireable transition must be fired.

3. All the steps preceding the transition are deactivated and all the steps following the transition are activated when a transition is fired.
4. All fireable transitions are fired simultaneously.
5. When a step must be both deactivated and activated it remains activated without interrupt.

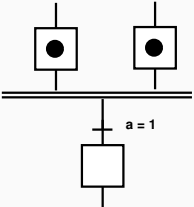
# Semantics



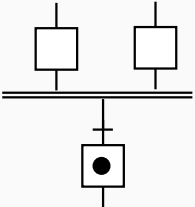
a) Not enabled



b) Enabled but not firable



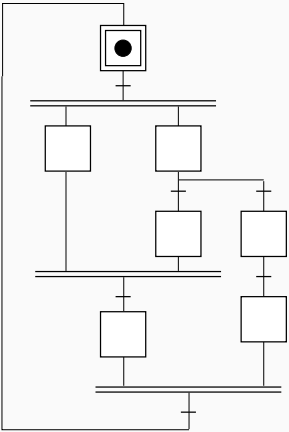
c) Firable



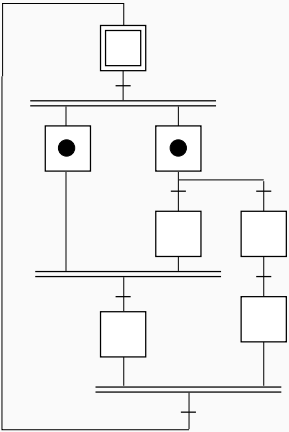
d) After the change from c)



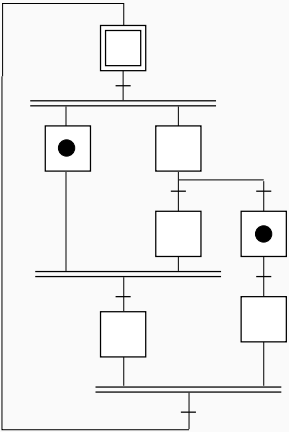
Unreachable grafcet



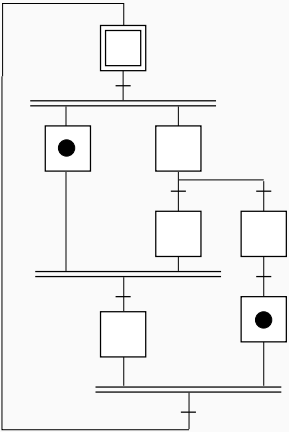
Unreachable grafcet



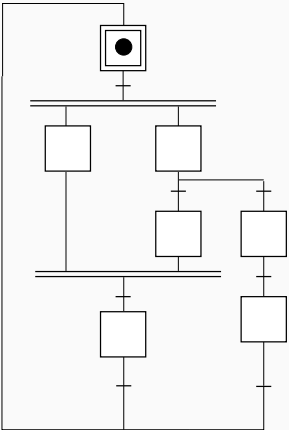
Unreachable grafcet



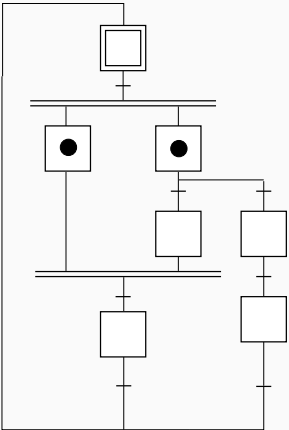
Unreachable grafcet



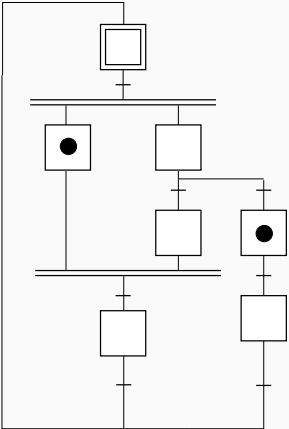
Unsafe grafcet



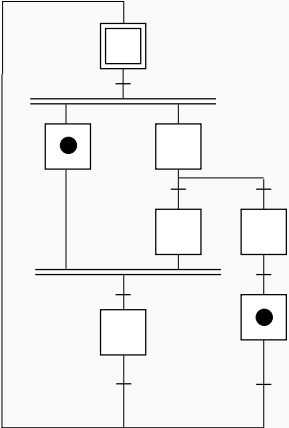
Unsafe grafcet



Unsafe grafcet

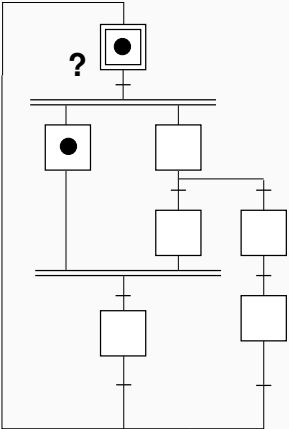


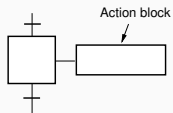
Unsafe grafcet





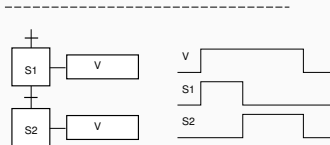
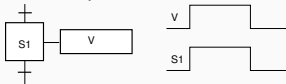
Unsafe grafcet





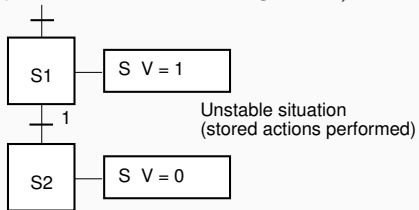
Action types:

- Standard Action (Level Action)

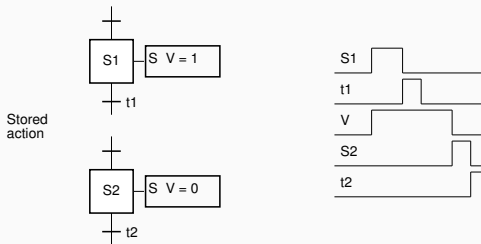
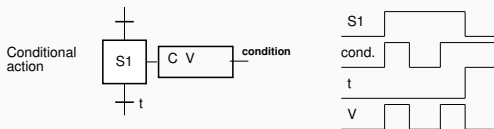


Action types:

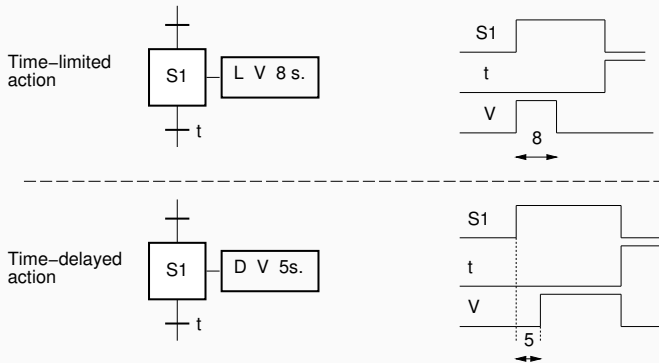
- Stored Action (Impulse Action, local assignment)



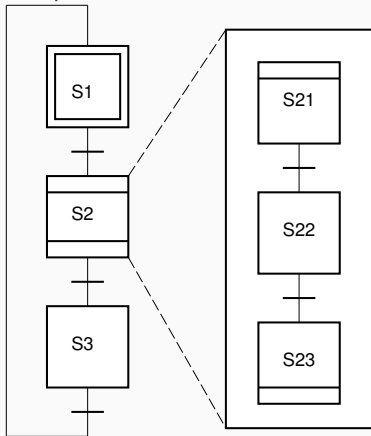
# Summary



# Summary

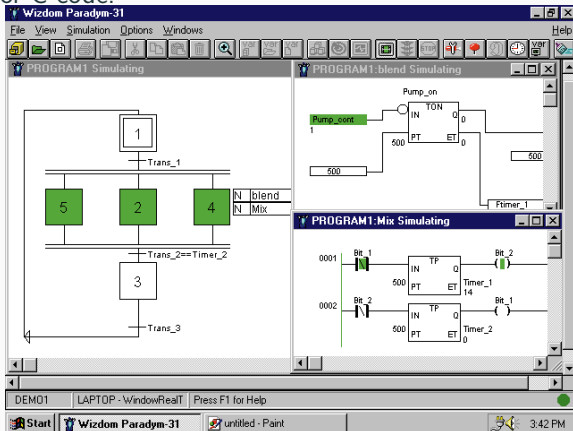


Hierarchy and macro steps



# Grafcet: editors

A large number of graphical IEC 1131-3 editors are available. Generates PLC code or C-code.



Sequential Control: bead sorter process

JGrafchart - Lund University

- Grafcet/SFC graphical editor
- Grafcet/SFC run-time system



# Jgraphchart

The screenshot displays the JGraphChart software interface. The main window, titled "JGraphchart1", contains a state machine diagram with the following components:

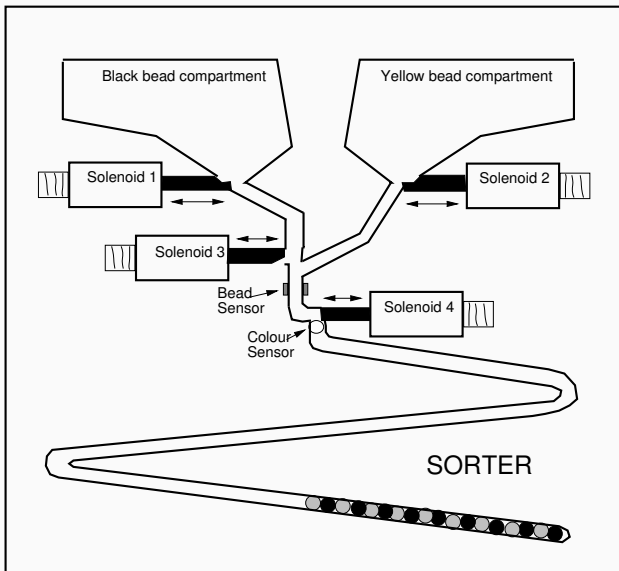
- States:** S1 (Initial Step), S2, S3, and S5.
- Transitions:**
  - S1 to S2: Triggered by event X1.
  - S2 to S3: Triggered by event !X1.
  - S2 to S5: Triggered by event /X1 & In1.
  - S3 to S2: Triggered by event X1.
  - S3 to S5: Triggered by event S5.t > 10.
- Internal Actions:**
  - From S2 to S3: Action A: X2 = X2 + 1;
  - From S2 to S5: Action S: Out1 = 1;
- Diagram Elements:** A transition arrow from S2 to S3 passes through a transition box labeled M2. A transition arrow from S3 to S5 passes through a transition box labeled M1.

Below the diagram, the software lists the variables used:

| Input Variables | Internal Variable | Output Variables |
|-----------------|-------------------|------------------|
| Chan:3<br>0     | Bool 1<br>X1      | Chan:3<br>1      |
| In1             | Int: 0<br>X2      | Out:1<br>0       |
| Chan:4<br>0     |                   | Chan:4<br>0      |
| In2             |                   | Out:2            |

A secondary window titled "M2" is open on the right, showing a detailed view of the transition box M2 from the main diagram, which contains a state machine diagram with two states, S1 and S2, and a transition between them labeled with event 1.

# Process



# Petri Nets

---

C.A Petri, TU Darmstadt, 1962.

A mathematical and graphical modeling method.

Describe systems that are:

- concurrent,
- asynchronous or synchronous,
- distributed,
- nondeterministic or deterministic.

Can be used at all stages of system development:

- modeling,
- analysis,
- simulation/visualization (“playing the token game”),
- synthesis,
- implementation (Grafcet).

Application areas:

- communication protocols,
- distributed systems,
- distributed database systems,
- flexible manufacturing systems,
- logical controller design,
- multiprocessor memory systems,
- dataflow computing systems,
- fault tolerant systems.

A Petri net is a directed bipartite graph consisting of places  $P$  and transitions  $T$ .

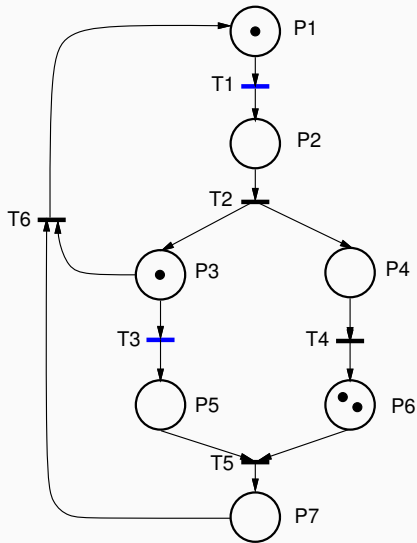
Places are represented by circles.

Transitions are represented by bars (or rectangles).

Places and transitions are connected by arcs.

In a marked Petri net each place contains a cardinal (zero or positive integer) number of tokens or marks.

# Petri Nets





## Petri Nets: Firing Rules

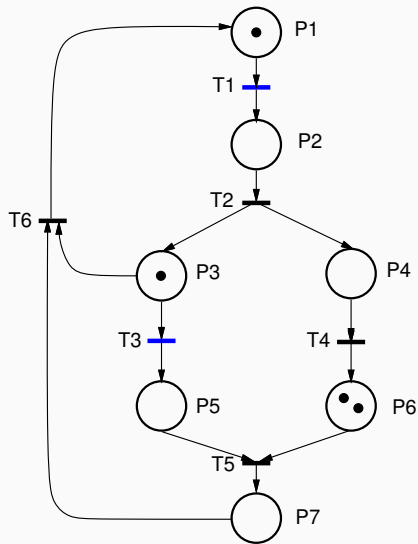
1. A transition  $t$  is enabled if each input place contains at least one token.
2. An enabled transition may or may not fire.
3. Firing an enabled transition  $t$  means removing one token from each input place of  $t$  and adding one token to each output place of  $t$ .

The firing of a transition has zero duration.

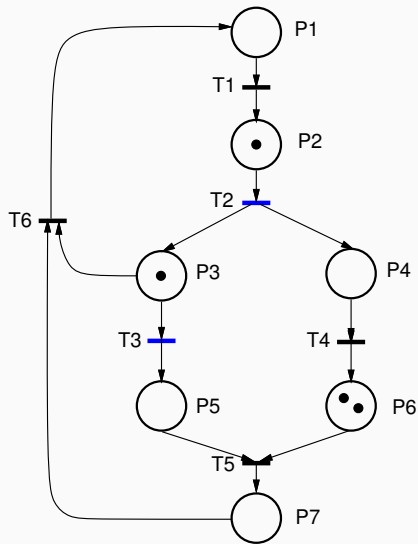
The firing of a sink transition (only input places) only consumes tokens.

The firing of a source transition (only output places) only produces tokens.

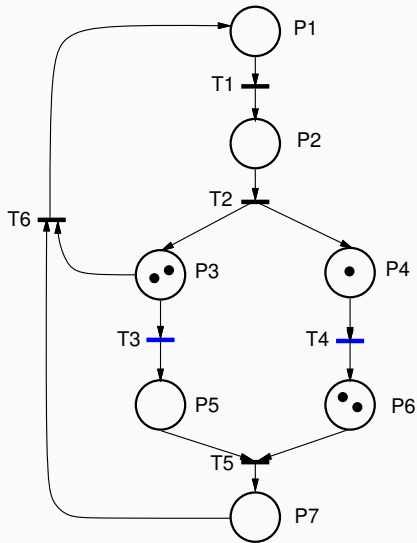
# Petri Nets



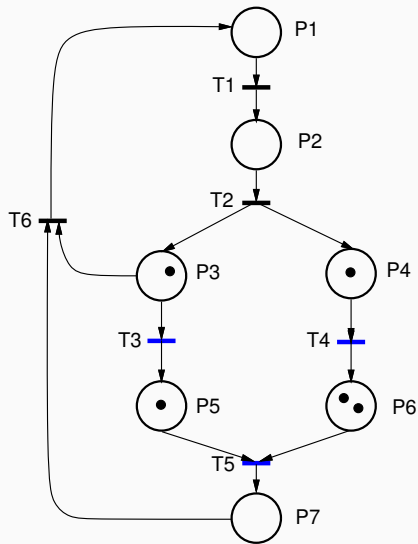
# Petri Nets



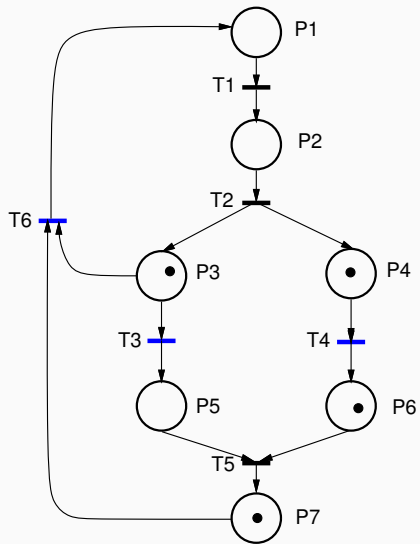
# Petri Nets



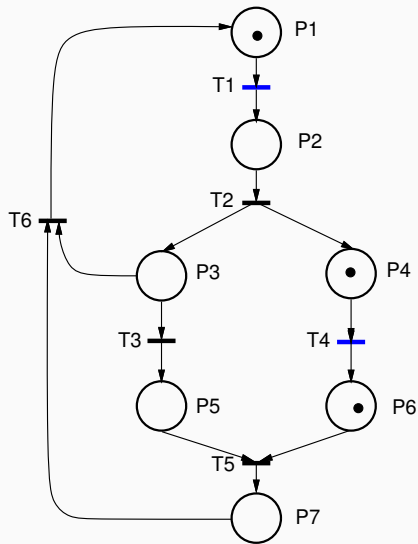
# Petri Nets



# Petri Nets



# Petri Nets

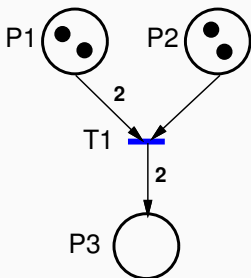


Typical interpretations of places and transitions:

| <b>Input Places</b> | <b>Transition</b> | <b>Output Places</b> |
|---------------------|-------------------|----------------------|
| Preconditions       | Event             | Postconditions       |
| Input data          | Computation step  | Output data          |
| Input signals       | Signal processor  | Output signals       |
| Resources needed    | Task or job       | Resources needed     |
| Conditions          | Clause in logic   | Conclusions          |
| Buffers             | Processor         | Buffers              |



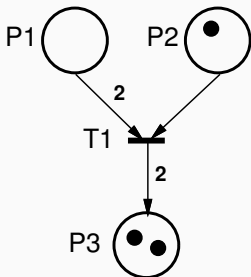
# Generalized Petri Nets



Firing rules:

1. A transition  $t$  is enabled if each input place  $p$  of  $t$  contains at least  $w(p, t)$  tokens
2. Firing a transition  $t$  means removing  $w(p, t)$  tokens from each input place  $p$  and adding  $w(t, q)$  tokens to each output place  $q$ .

# Generalized Petri Nets



Firing rules:

1. A transition  $t$  is enabled if each input place  $p$  of  $t$  contains at least  $w(p, t)$  tokens
2. Firing a transition  $t$  means removing  $w(p, t)$  tokens from each input place  $p$  and adding  $w(t, q)$  tokens to each output place  $q$ .

## **Timed Petri Nets:**

Times associated with transitions or places.

## **High-Level Petri Nets:**

Tokens are structured data types (objects).

## **Continuous & Hybrid Petri Nets:**

The markings are real numbers instead of integers.

Mixed continuous/discrete systems.

- **Live:** No transitions can become unfireable.
- **Deadlock-free:** Transitions can always be fired.
- **Bounded:** Finite number of tokens.

Analysis methods:

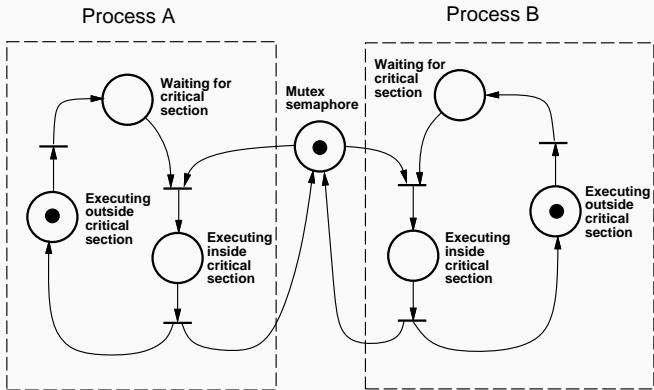
- **Reachability methods:**
  - exhaustive enumeration of all possible markings.
- **Linear algebra methods:**
  - describe the dynamic behaviour as matrix equations.
- **Reduction methods:**
  - transformation rules that reduce the net to a simpler net while preserving the properties of interest.

Dijkstra's classical problems:

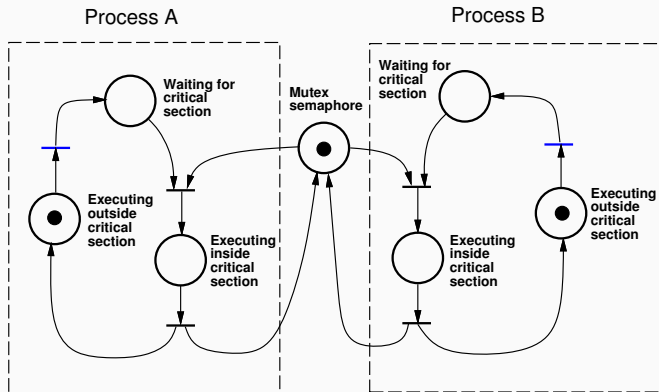
- mutual exclusion problem,
- producer-consumer problem,
- readers-writers problem,
- dining philosophers problem.

All can be modeled by Petri Nets.

# Petri Nets: Mutual Exclusion

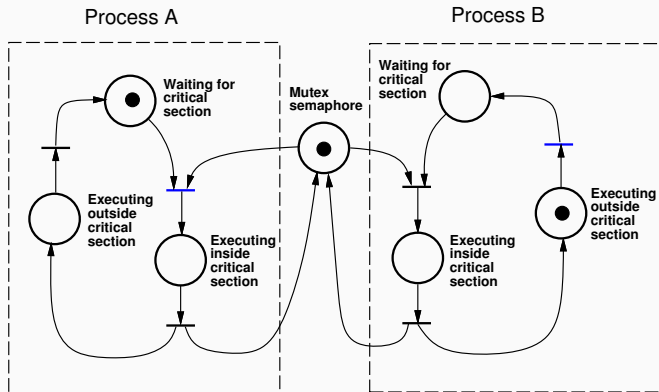


# Petri Nets: Mutual Exclusion

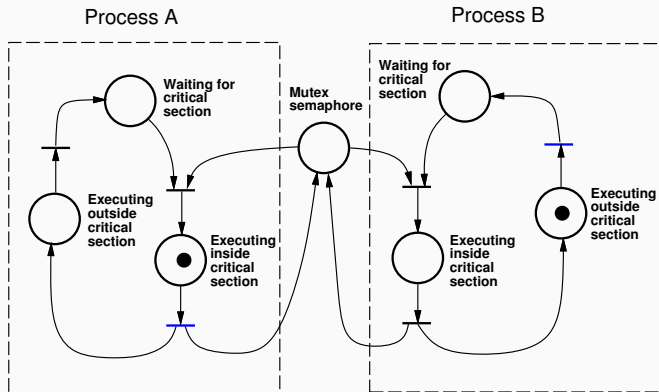




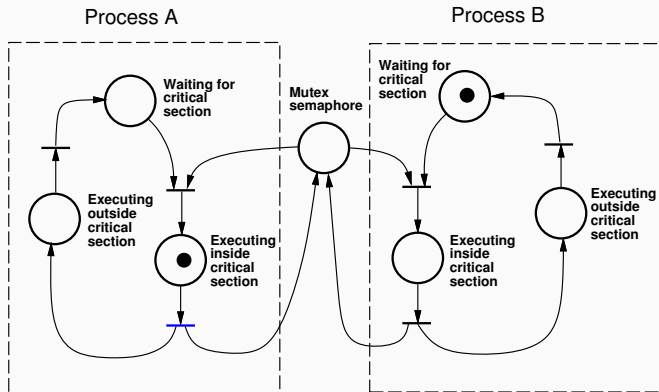
# Petri Nets: Mutual Exclusion



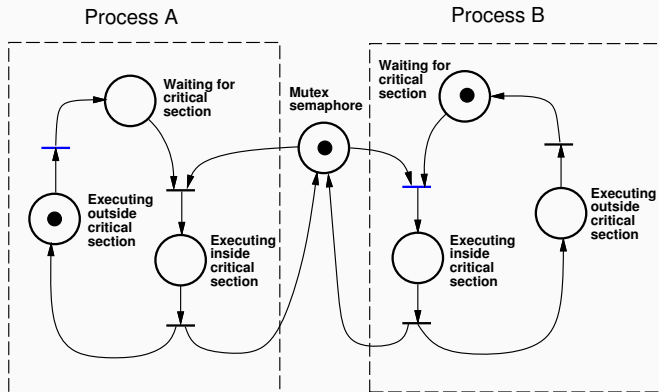
# Petri Nets: Mutual Exclusion



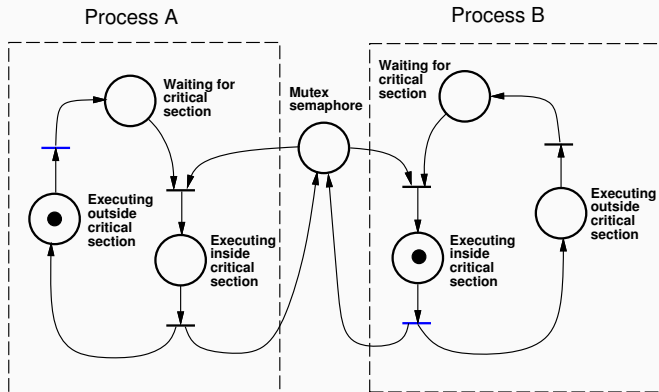
# Petri Nets: Mutual Exclusion



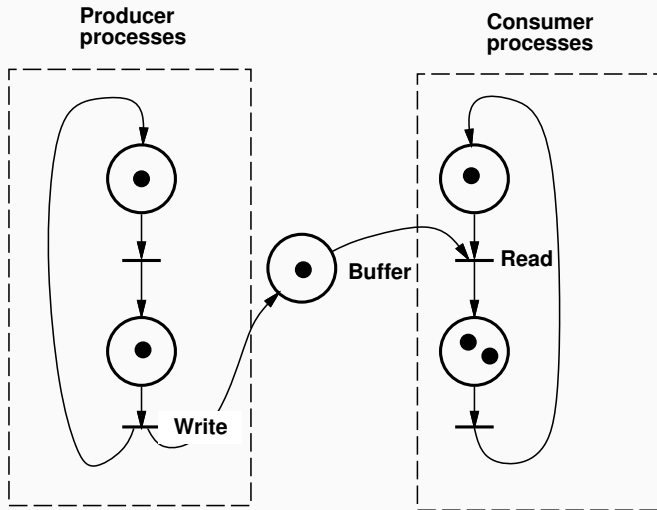
# Petri Nets: Mutual Exclusion



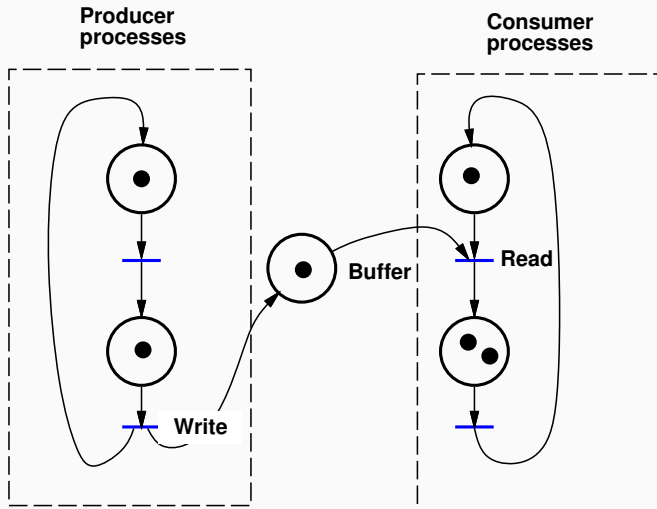
# Petri Nets: Mutual Exclusion



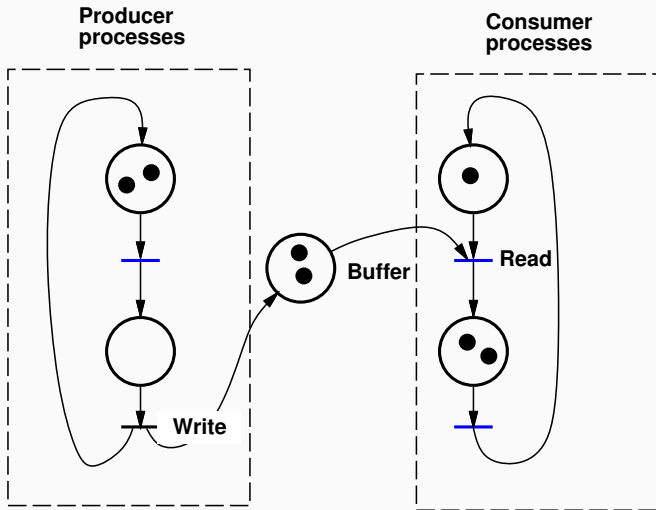
# Petri Nets: Producer Consumer



# Petri Nets: Producer Consumer

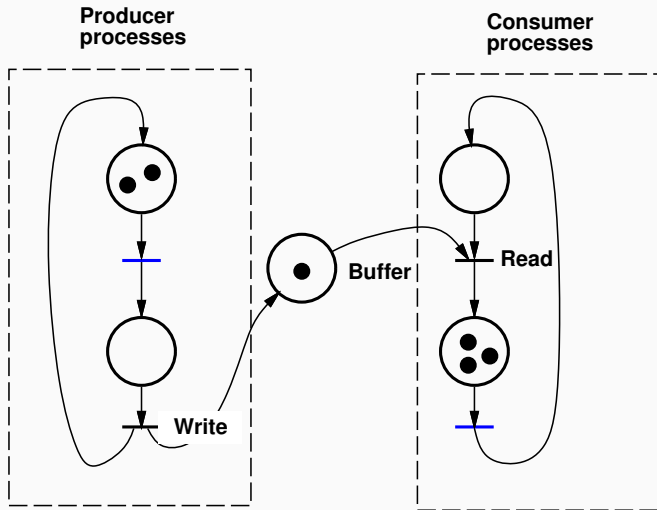


# Petri Nets: Producer Consumer

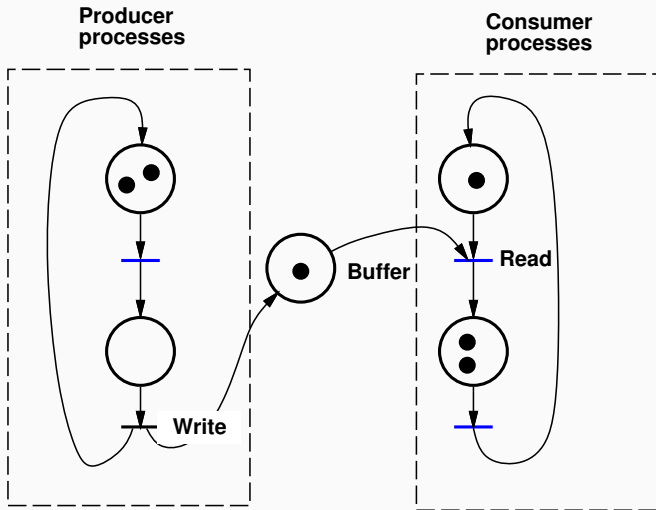




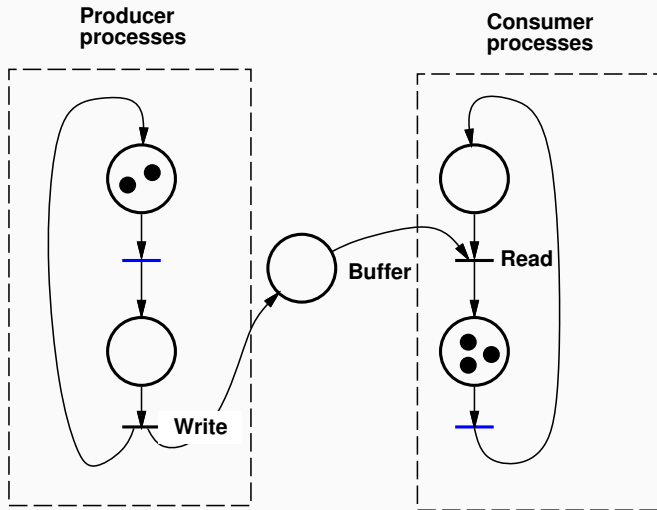
# Petri Nets: Producer Consumer



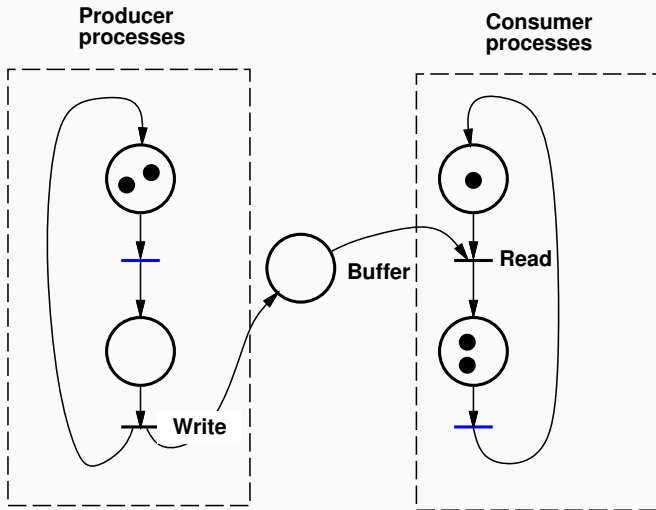
# Petri Nets: Producer Consumer



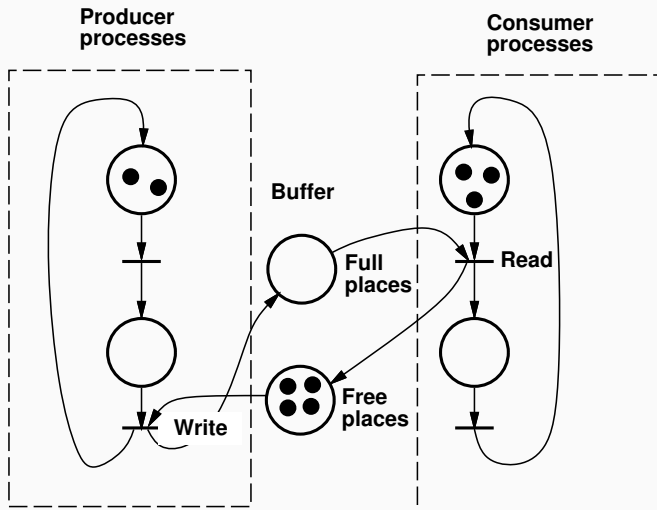
# Petri Nets: Producer Consumer



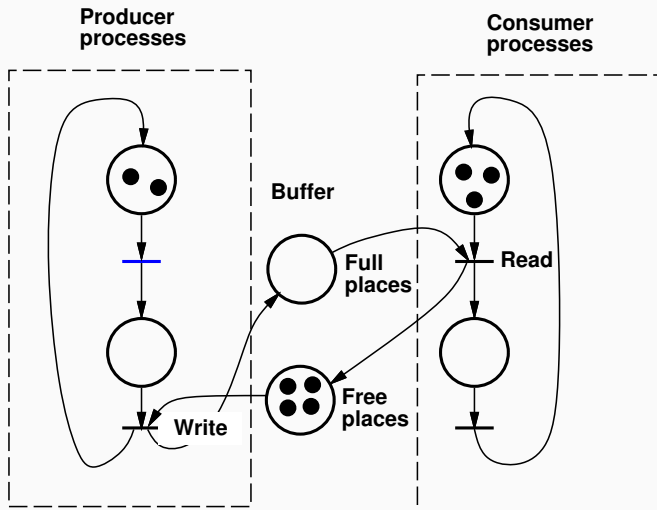
# Petri Nets: Producer Consumer



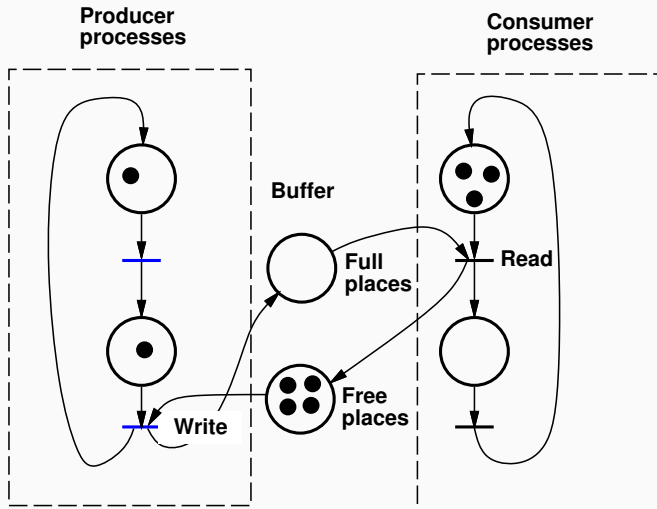
# Petri Nets: Producer Consumer with Bounded Buffer



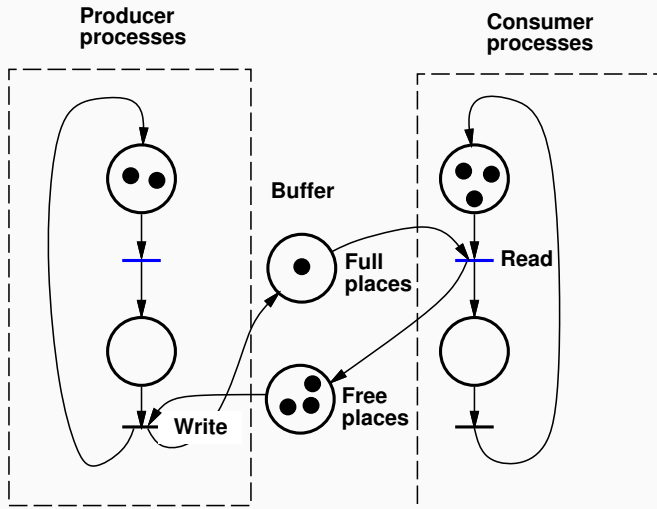
# Petri Nets: Producer Consumer with Bounded Buffer



# Petri Nets: Producer Consumer with Bounded Buffer

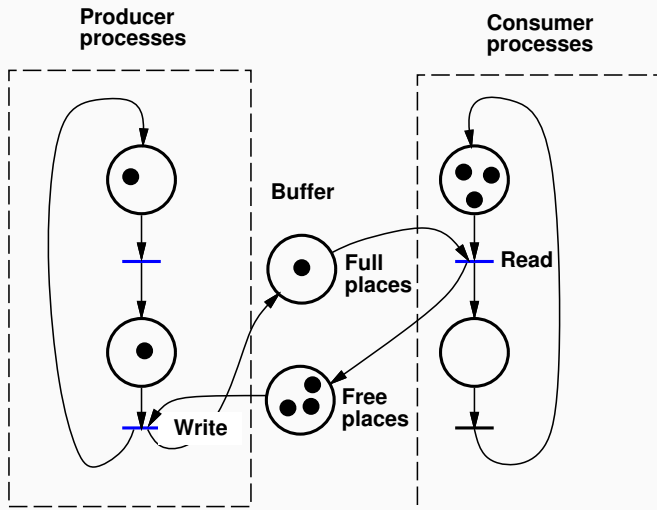


# Petri Nets: Producer Consumer with Bounded Buffer

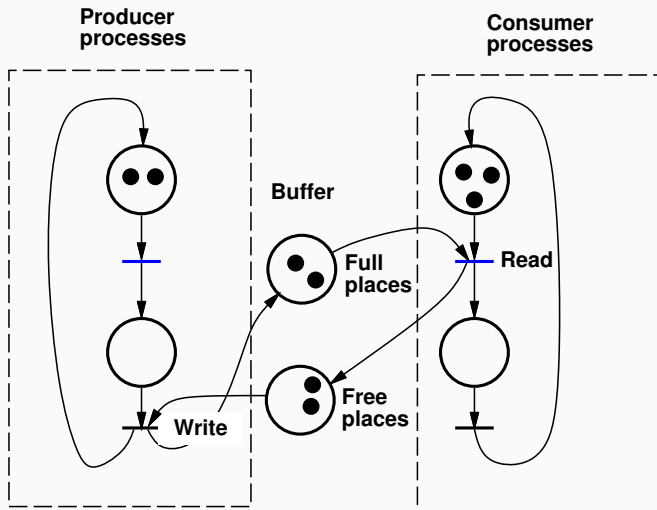




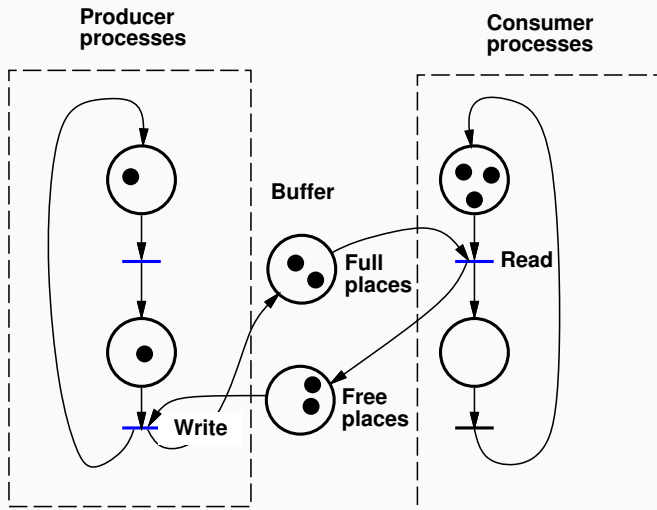
# Petri Nets: Producer Consumer with Bounded Buffer



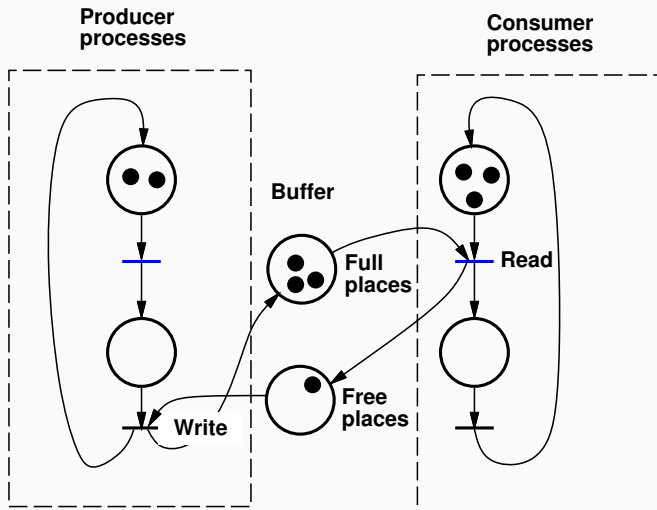
# Petri Nets: Producer Consumer with Bounded Buffer



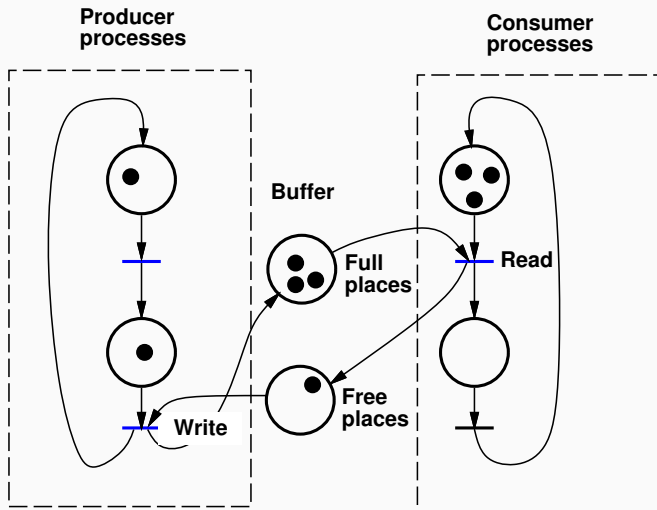
# Petri Nets: Producer Consumer with Bounded Buffer



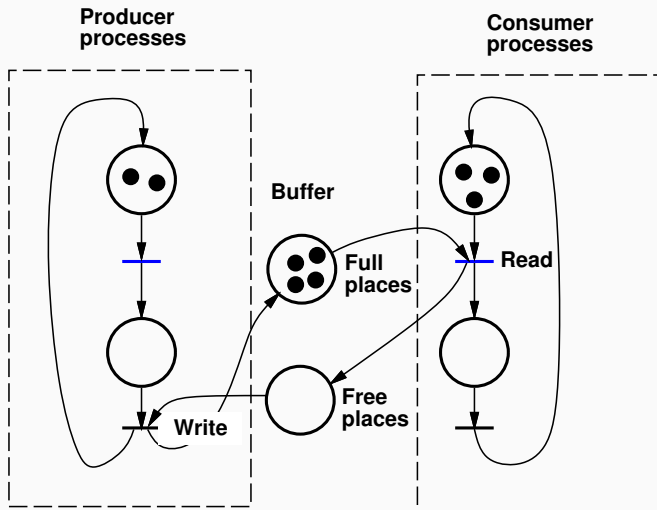
# Petri Nets: Producer Consumer with Bounded Buffer



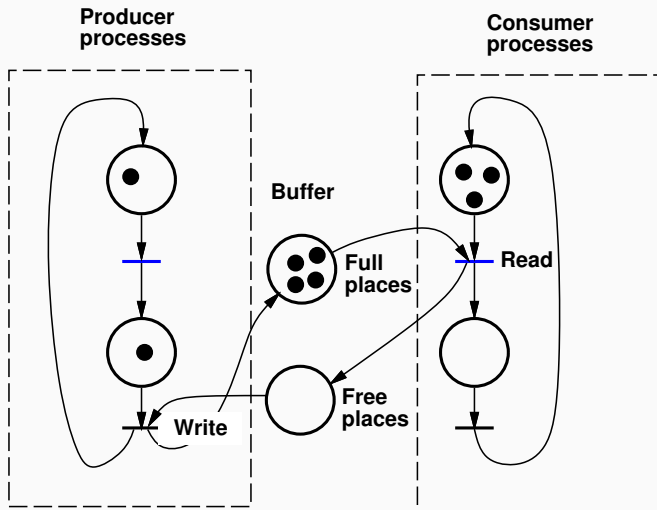
# Petri Nets: Producer Consumer with Bounded Buffer



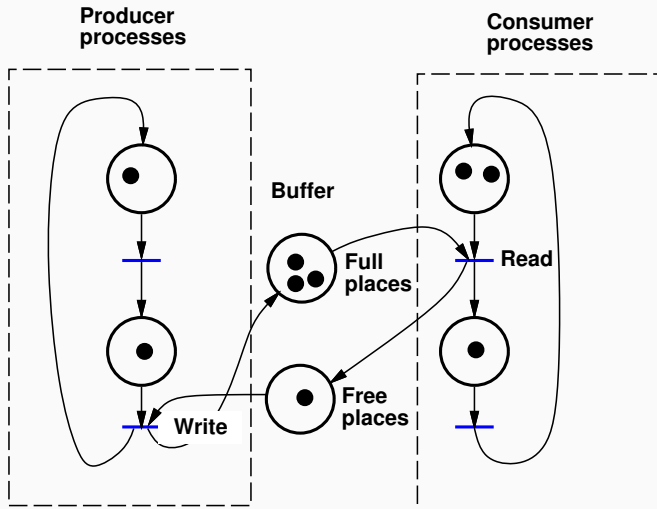
# Petri Nets: Producer Consumer with Bounded Buffer



# Petri Nets: Producer Consumer with Bounded Buffer

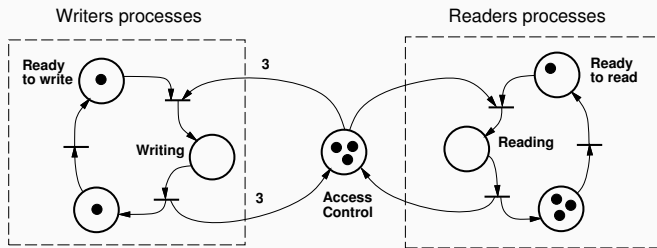


# Petri Nets: Producer Consumer with Bounded Buffer

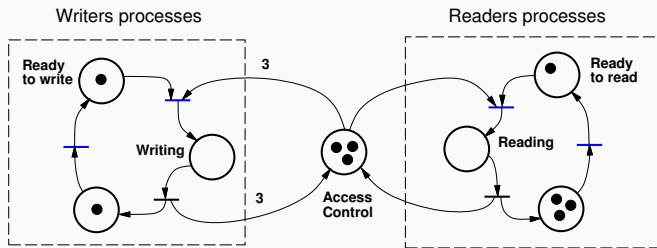




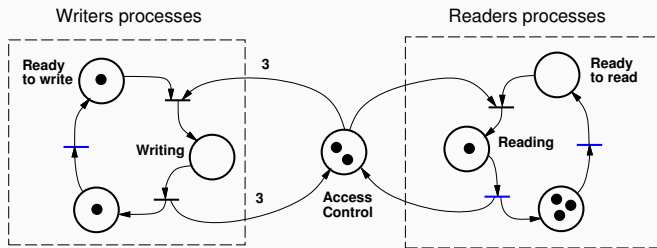
# Petri Nets: Reader and Writer



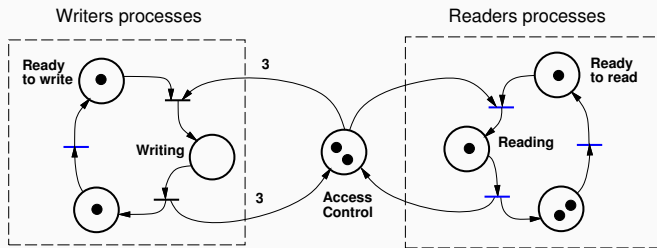
# Petri Nets: Reader and Writer



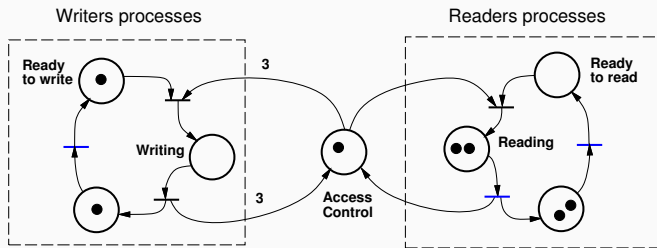
# Petri Nets: Reader and Writer



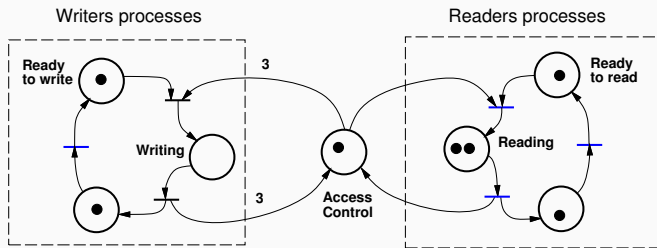
# Petri Nets: Reader and Writer



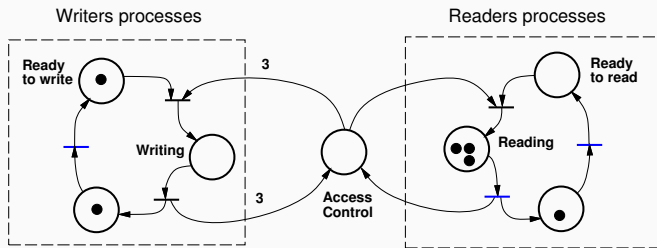
# Petri Nets: Reader and Writer



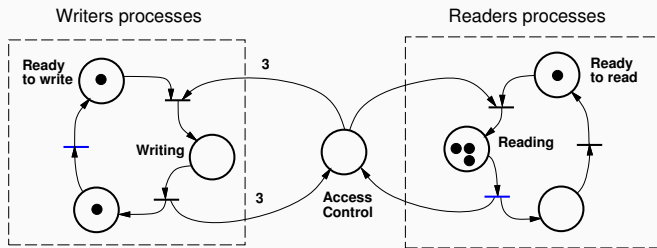
# Petri Nets: Reader and Writer



# Petri Nets: Reader and Writer

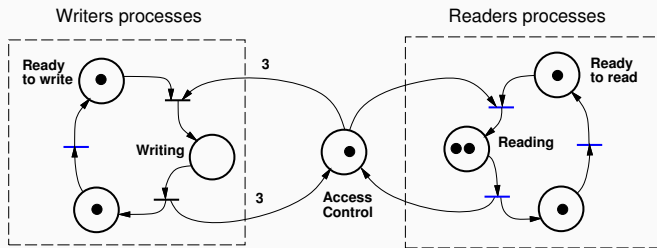


# Petri Nets: Reader and Writer

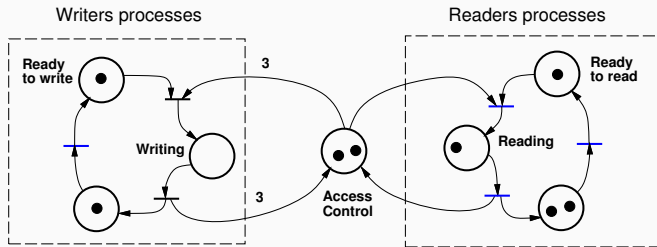




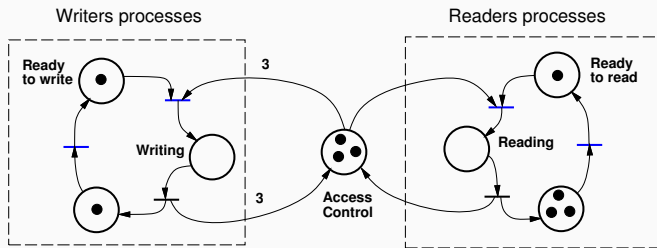
# Petri Nets: Reader and Writer



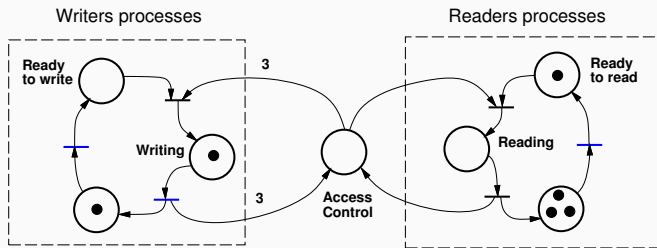
# Petri Nets: Reader and Writer



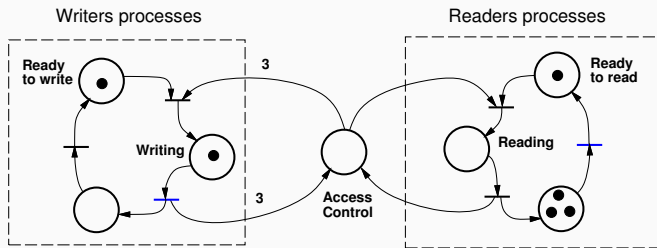
# Petri Nets: Reader and Writer



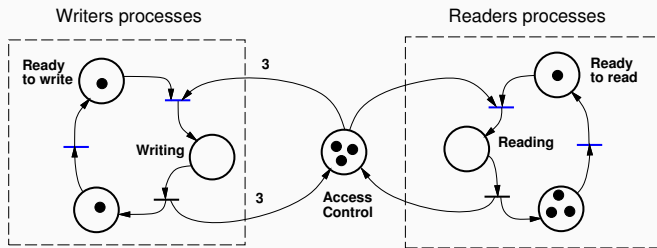
# Petri Nets: Reader and Writer



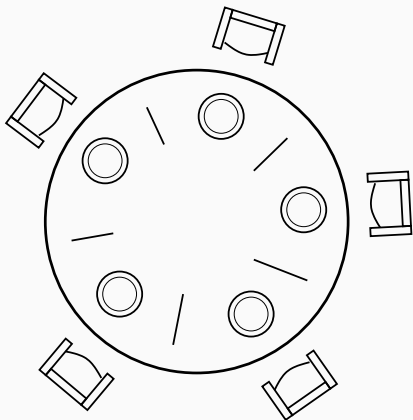
# Petri Nets: Reader and Writer



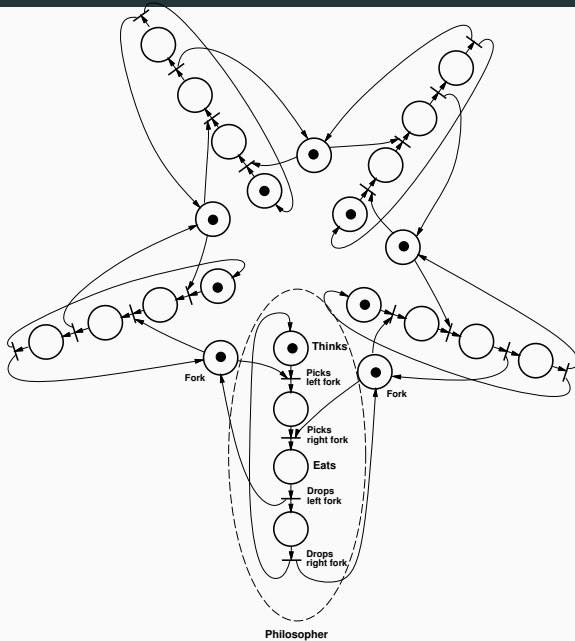
# Petri Nets: Reader and Writer



## Petri Nets: Dining Philosophers

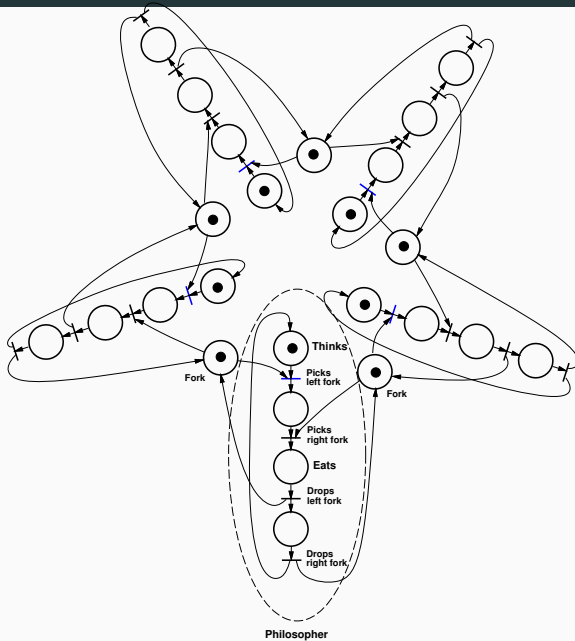


# Petri Nets: Dining Philosophers

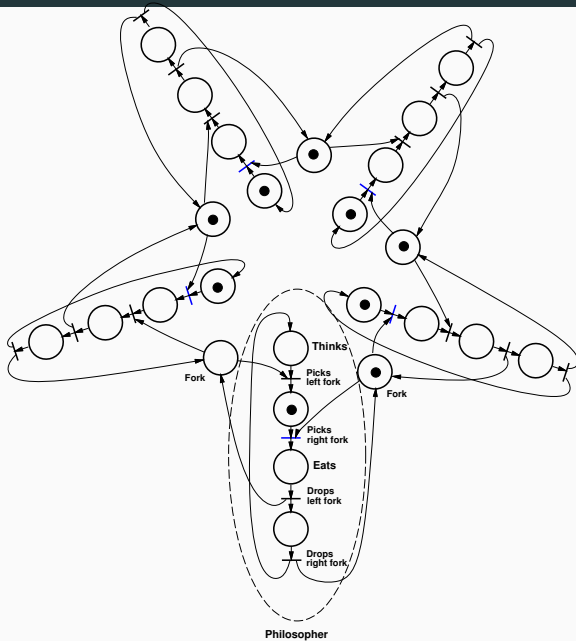




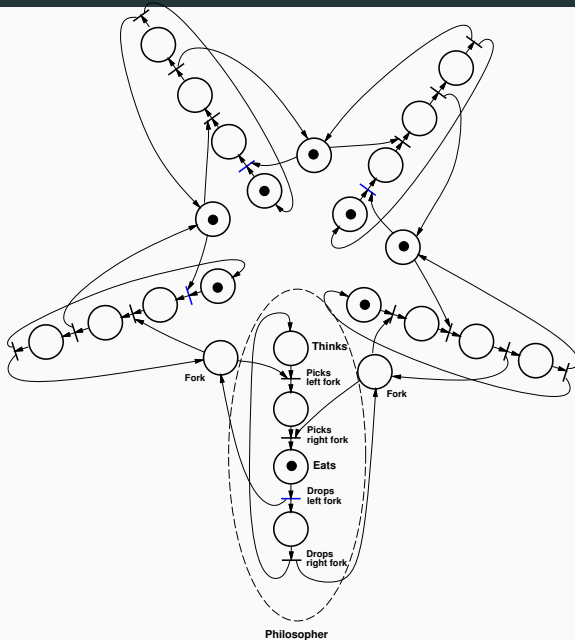
# Petri Nets: Dining Philosophers



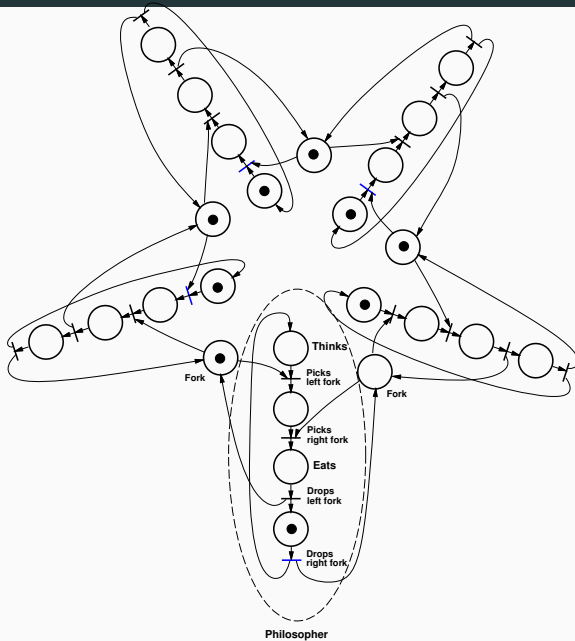
# Petri Nets: Dining Philosophers



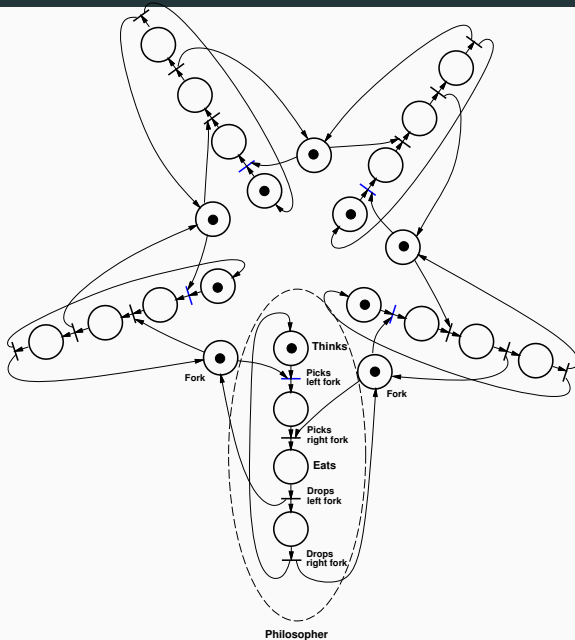
# Petri Nets: Dining Philosophers



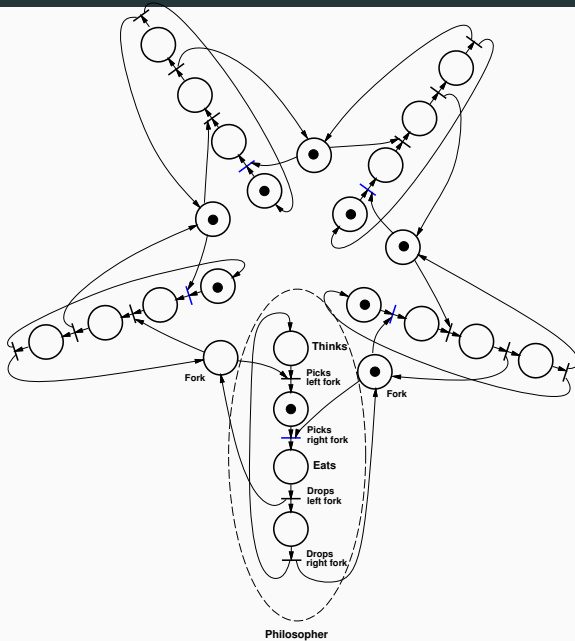
# Petri Nets: Dining Philosophers



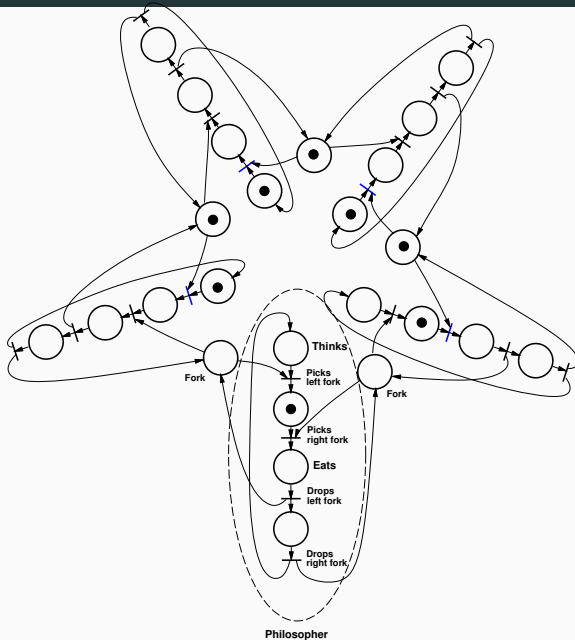
# Petri Nets: Dining Philosophers



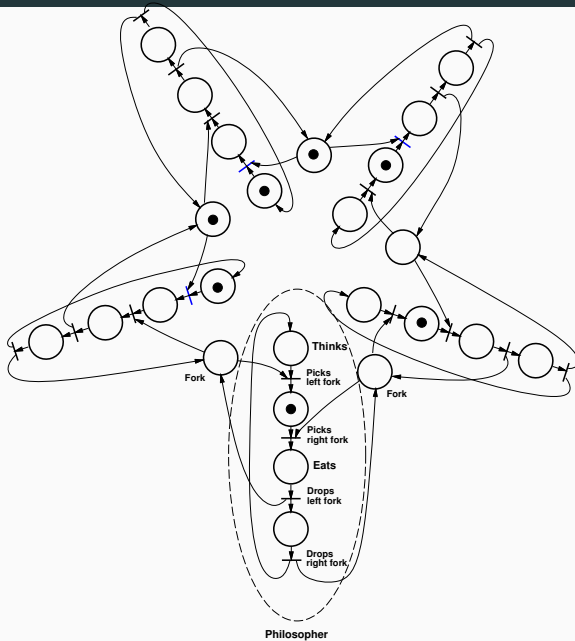
# Petri Nets: Dining Philosophers



# Petri Nets: Dining Philosophers

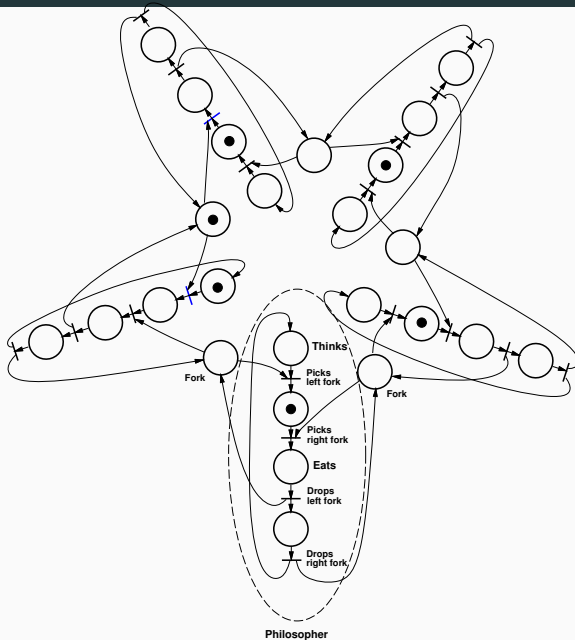


# Petri Nets: Dining Philosophers

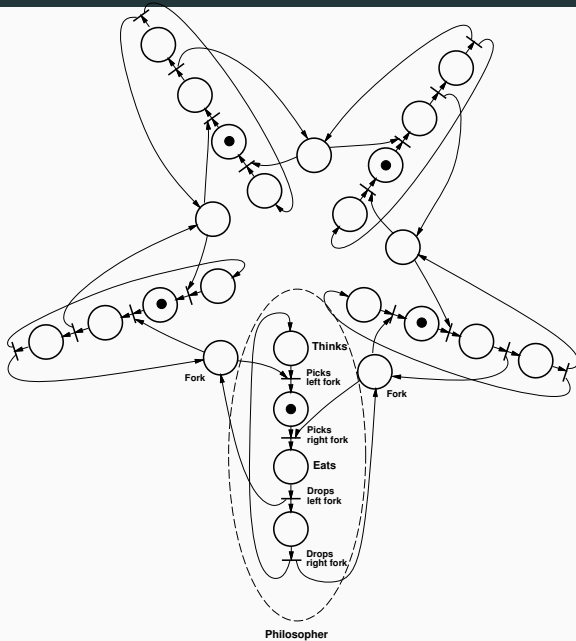




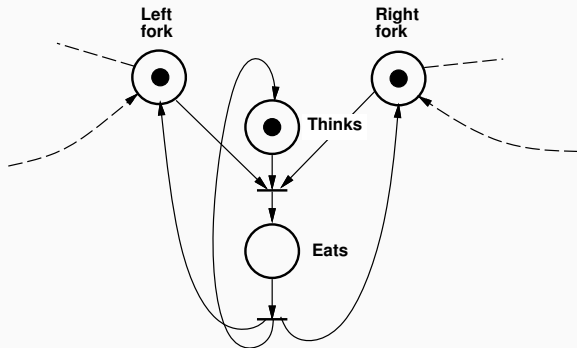
# Petri Nets: Dining Philosophers



# Petri Nets: Dining Philosophers



# Petri Nets: Dining Philosophers



# Implementation

---

## Implementation (Not covered in the lecture – Homework)

Using state machines is often a good way to structure code.

Systematic ways to write automata code often not taught in programming courses.

Issues:

- active or passive object
- Mealy vs Moore machines
- states with timeout events
- states with periodic activities

Often convenient to implement state machines as periodic processes with a period that is determined by the shortest time required when making a state transition.

## Example: Passive State Machine

The state machine is implemented as a synchronized object.

```
1 public class PassiveMealyMachine {
2     private static final int STATE0 = 0;
3     private static final int STATE1 = 1;
4     private static final int STATE2 = 2;
5     private static final int INA = 0;
6     private static final int INB = 1;
7     private static final int INC = 2;
8     private static final int OUTA = 0;
9     private static final int OUTB = 1;
10    private static final int OUTC = 2;
11    private int state;
12    PassiveMealyMachine() {
13        state = STATE0;
14    }
15    private void generateEvent(int outEvent) {
16        // Do something
17    }
```

## Example: Passive State Machine

```
1  public synchronized void inputEvent(int event) {
2      switch (state) {
3          case STATE0 : switch (event) {
4              case INA : generateEvent(OUTA); state = STATE1; break;
5              case INB : generateEvent(OUTB); break;
6              default : break;
7          }; break;
8          case STATE1 : switch (event) {
9              case INC : generateEvent(OUTC); state = STATE2; break;
10             default : break;
11          }; break;
12          case STATE2 : switch (event) {
13              case INA : generateEvent(OUTB); state = STATE0; break;
14              case INC : generateEvent(OUTC); break;
15              default : break;
16          }; break;
17      }
18  }
19 }
```

## Example: Active State Machine

The state machine could also be implemented as an active object (thread)

The thread object would typically contain an event-buffer (e.g., an `RTEventBuffer`).

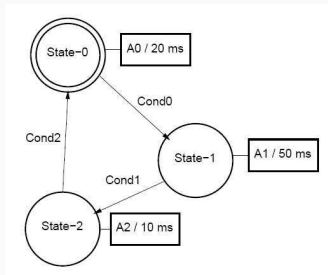
The run method would consist of an infinite loop that waits for an incoming event (`RTEvent`) and switches state depending on the event.



## Example: Active State Machine

An activity is an action that is executed periodically while a state is active.

More natural to implement the state machine as a thread.



# Example: Active State Machine 1

```
1  public class ActiveMachine1 extends Thread {
2      private static final int STATE0 = 0;
3      private static final int STATE1 = 1;
4      private static final int STATE2 = 2;
5      private int state;
6
7      ActiveMachine1() { state = STATE0; }
8
9      private boolean cond0() {
10         // Returns true if condition 0 is true
11     }
12     private boolean cond1() { }
13     private boolean cond2() { }
14
15     private void action0() {
16         // Executes action 0
17     }
18     private void action1() { }
19     private void action2() { }
```

# Example: Active State Machine 1

```
1  public void run() {
2      long t = System.currentTimeMillis();
3      long duration;
4
5      while (true) {
6          switch (state) {
7              case STATE0 : {
8                  action0(); t = t + 20;
9                  duration = t - System.currentTimeMillis();
10                 if (duration > 0) {
11                     try { sleep(duration);
12                     } catch (InterruptedException e) {}
13                 }
14                 if (cond0()) { state = STATE1; }
15             } break;
16             case STATE1 : {
17                 // Similar as for STATE0. Executes action1,
18                 // waits for 50 ms, checks
19                 // cond1 and then changes to STATE2
20             }; break;
```

## Example: Active State Machine 1

```
1     case STATE2 : {
2         // Similar as for STATE0. Executes action2,
3         // waits for 10 ms, checks
4         // cond2 and then changes to STATE0
5     }; break;
6 } } } }
```

- Conditions tested at a frequency determined by the activity frequencies of the different states.
- `sleep()` spread out in the code.

## Example: Active State Machine 2

The thread runs at a constant (high) base frequency. Activity frequencies multiples of the base frequency. Conditions tested at the base frequency.

```
1  public void run() {
2      long t = System.currentTimeMillis();
3      long duration;
4      int counter = 0;
5      while (true) {
6          counter++;
7          switch (state) {
8              case STATE0 : {
9                  if (counter == 4) { counter = 0; action0(); }
10                 if (cond0()) { counter = 0; state = STATE1; }
11             }; break;
12             case STATE1 : {
13                 // Similar as for STATE0. Executes action1
14                 // if counter == 10. Changes to STATE2 if
15                 // cond1() is true
16             }; break;
```

## Example: Active State Machine 2

```
1     case STATE2 : {
2         // Similar as for STATE0. Executes action2
3         // if counter == 12. Changes to STATE0 if
4         // cond2() is true
5     }; break;
6 }
7 t = t + 5; // Base sampling time
8 duration = t - System.currentTimeMillis();
9 if (duration > 0) {
10     try { sleep(duration);
11         } catch (InterruptedException e) {}
12 } } }
```

- Polled time handling.
- Complicated handling of counter.
- Conditions tested at a high rate.