DATAFLOW MODELS

1. Dataflow Models: an Example
2. Kahn Process Networks: a Deterministic Model
3. Synchronous Dataflow: Statically Schedulable Dataflow Models
4. Deriving a static Schedule for Synchronous Dataflow Models

Dataflow Models

- Systems are specified as directed graphs where:
  - *nodes* represent computations (processes);
  - *arcs* represent totally ordered sequences (streams) of data (tokens).

Dataflow Models

- Systems are specified as directed graphs where:
  - *nodes* represent computations (processes);
  - *arcs* represent totally ordered sequences (streams) of data (tokens).

- Depending on their particular semantics, several models of computation based on dataflow have been defined:
  - Kahn process networks
  - Dataflow process networks
  - Synchronous dataflow
  - - - - - - -

Dataflow Models

- Systems are specified as directed graphs where:
  - *nodes* represent computations (processes);
  - *arcs* represent totally ordered sequences (streams) of data (tokens).

- Depending on their particular semantics, several models of computation based on dataflow have been defined:
  - Kahn process networks
  - Dataflow process networks
  - Synchronous dataflow
  - - - - - - -

- Dataflow models are suitable for signal-processing algorithms:
  - Code/decode, filter, compression, etc.
  - Streams of periodic and regular data samples

Dataflow Models

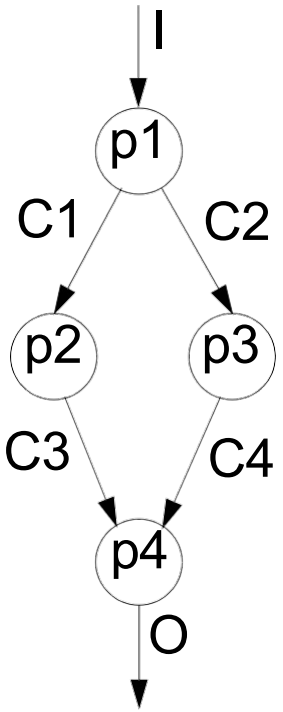
```
Process p1( in int a, out int x, out int y) {
.....
}

Process p2( in int a, out int x) {
.....
}

Process p3( in int a, out int x) {
.....
}

Process p4( in int a, in int b, out int x) {
.....
}

channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p3(C2, C4);
p4(C3, C4, O);
```



- The internal computation of a process can be specified in any programming language (e.g. C).

  This is called the *host language*.

Kahn Process Networks (KPN)

- Processes communicate by passing data tokens through unidirectional FIFO channels.
- Writes to the channel are non-blocking.
- Reads are blocking:
  - the process is blocked until there is sufficient data in the channel

Kahn Process Networks (KPN)

- Processes communicate by passing data tokens through unidirectional FIFO channels.
- Writes to the channel are non-blocking.
- Reads are blocking:
  - the process is blocked until there is sufficient data in the channel

A process that tries to read from an empty channel waits until data is available. It cannot ask whether data is available *before* reading and, for example, if there is no data, decide not to read that channel.

DETERMINISM

Kahn Process Networks

- Kahn process networks are deterministic:
  - For a given sequence of inputs, there is only one possible sequence of outputs (regardless, for example, how long time it takes for a certain computation or communication to finish).

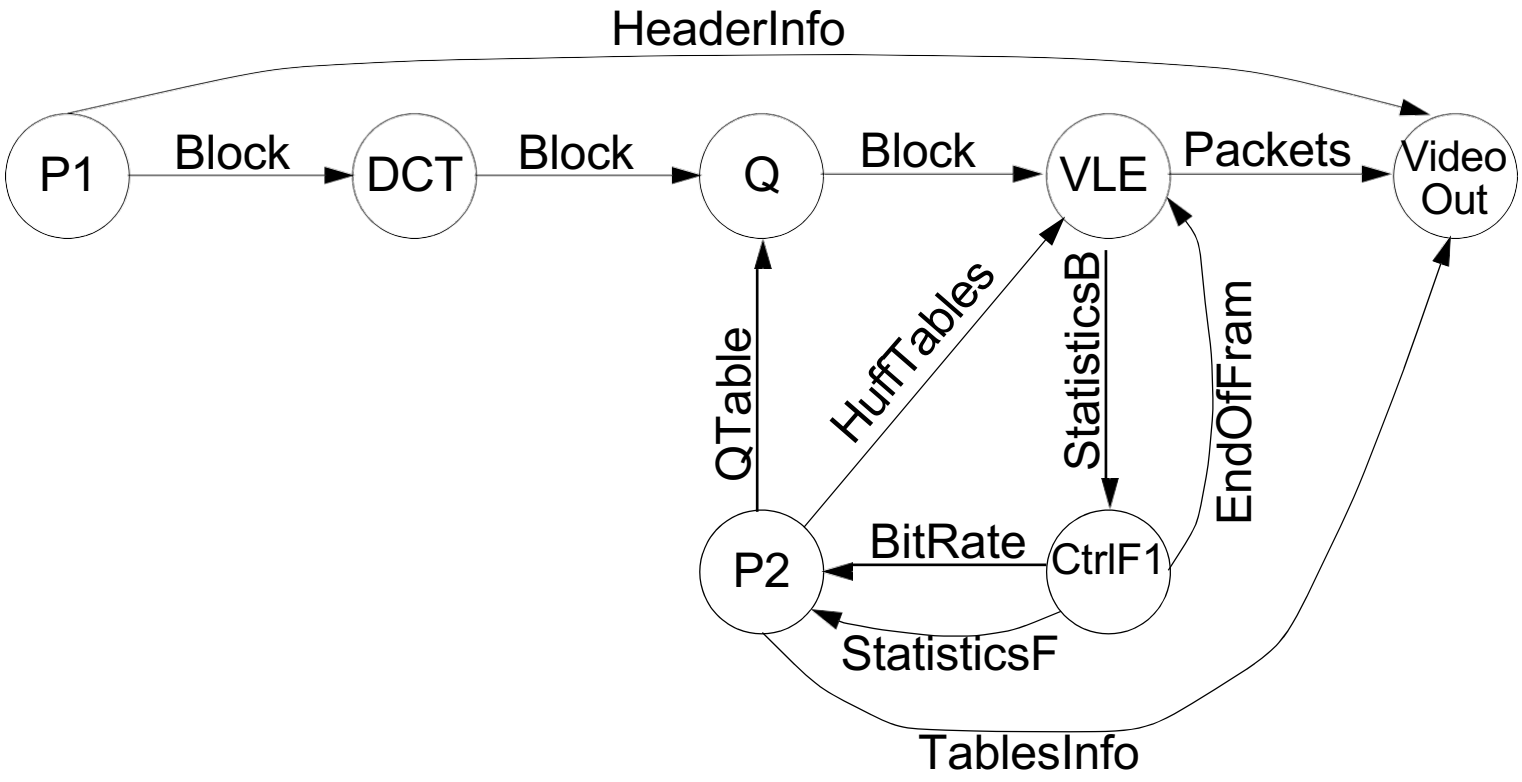
    Looking only at the specification (and not knowing anything about implementation) you can exactly derive the output sequence corresponding to a given input sequence.

Kahn Process Networks

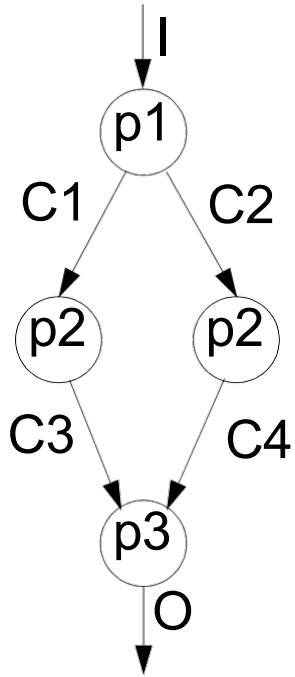
- More on read and write limitations
  - A process cannot wait for data on more than one channel at a time
  - Only a single process is allowed to read from a certain channel
- What if the output data has to be sent to more than one process?
  - Data must be duplicated inside processes
- This limited model of computation implies:
  - More modeling effort for complex systems
  - Retained determinism!

Kahn Process Networks: an Example

KPN model of encoder for Motion JPEG (M-JPEG) video compression format:



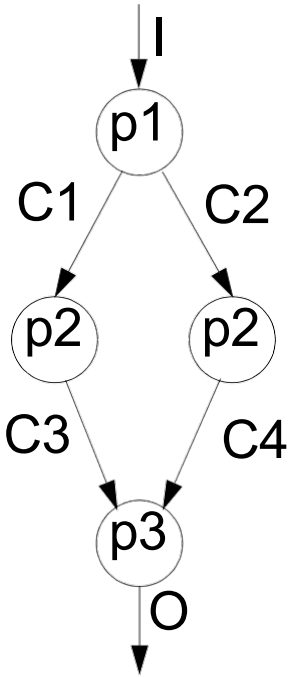
Kahn Process Networks: a Simpler Example



```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
        x.send(k);
    else
        y.send(k);
    end if;
end loop; }

Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }

Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
        k = a.receive();
    else
        k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

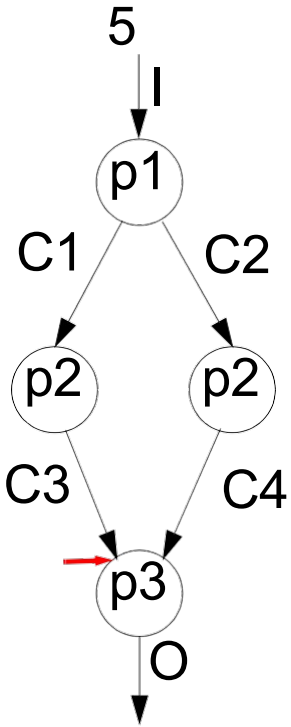



```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
        x.send(k);
    else
        y.send(k);
    end if;
end loop; }

Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }

Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
        k = a.receive();
    else
        k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

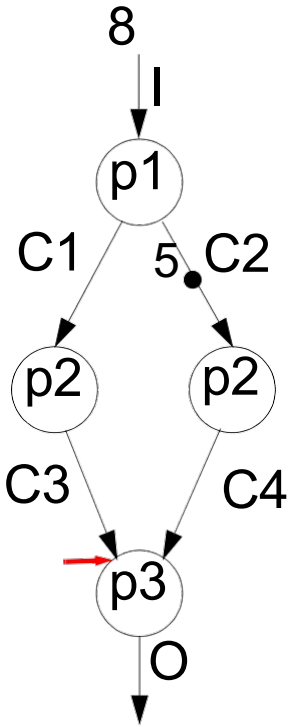



```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
            x.send(k);
    else
            y.send(k);
    end if;
end loop; }
```

```
Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }
```

```
Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
            k = a.receive();
    else
            k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

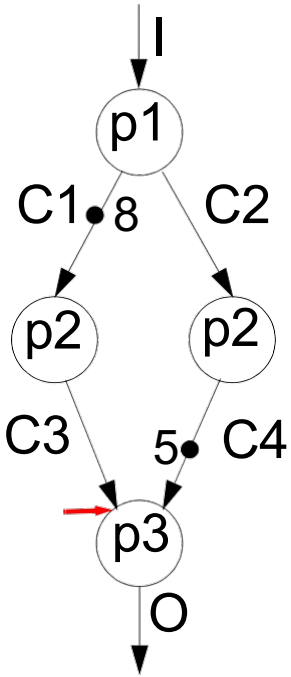



```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
            x.send(k);
    else
            y.send(k);
    end if;
end loop; }

Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }

Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
            k = a.receive();
    else
            k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

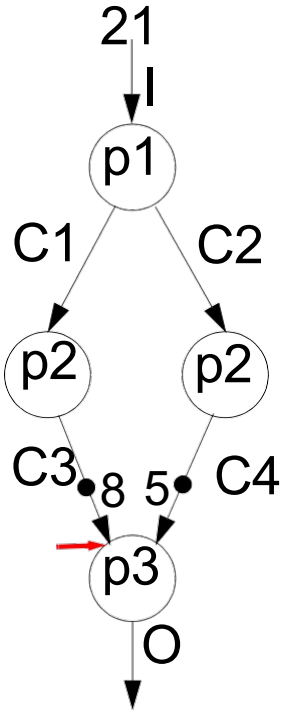



```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
            x.send(k);
    else
            y.send(k);
    end if;
end loop; }
```

```
Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }
```

```
Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
            k = a.receive();
    else
            k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

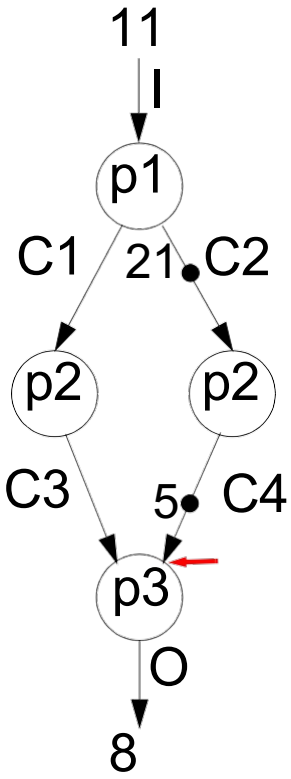



```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
            x.send(k);
    else
            y.send(k);
    end if;
end loop; }

Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }

Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
            k = a.receive();
    else
            k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```



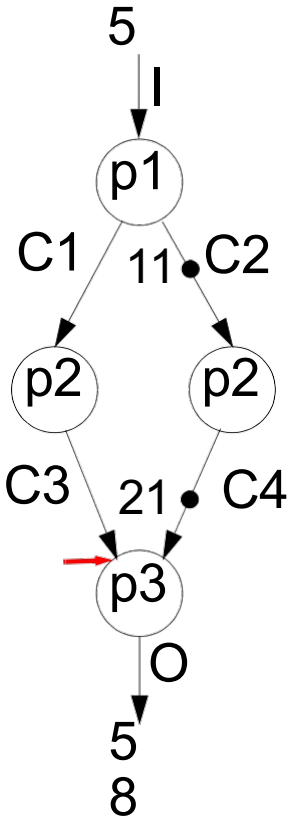

```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
            x.send(k);
    else
            y.send(k);
    end if;
end loop; }

Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }

Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
            k = a.receive();
    else
            k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

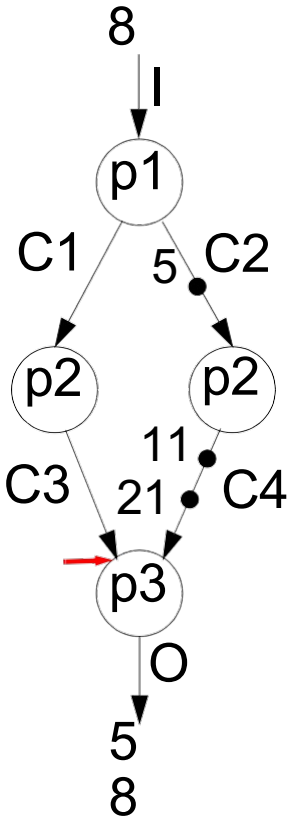


```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
            x.send(k);
    else
            y.send(k);
    end if;
end loop; }

Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }

Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
            k = a.receive();
    else
            k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

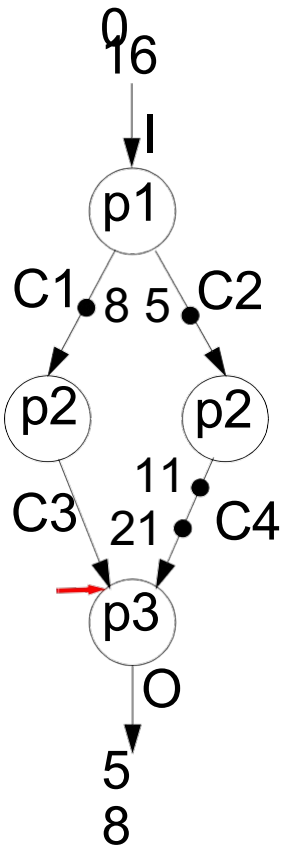


```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
            x.send(k);
    else
            y.send(k);
    end if;
end loop; }

Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }

Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
            k = a.receive();
    else
            k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

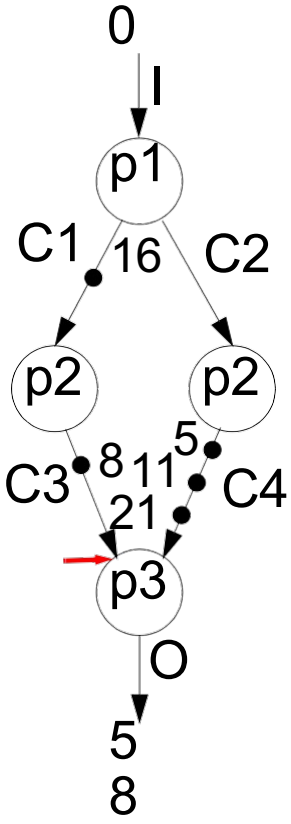


```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
            x.send(k);
    else
            y.send(k);
    end if;
end loop; }

Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }

Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
            k = a.receive();
    else
            k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

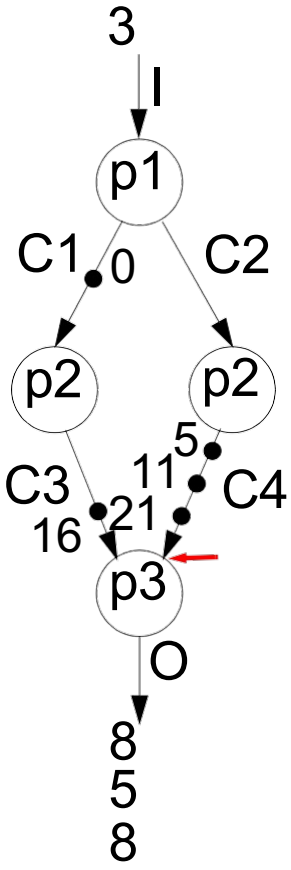


```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
            x.send(k);
    else
            y.send(k);
    end if;
end loop; }
```

```
Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }
```

```
Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
            k = a.receive();
    else
            k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

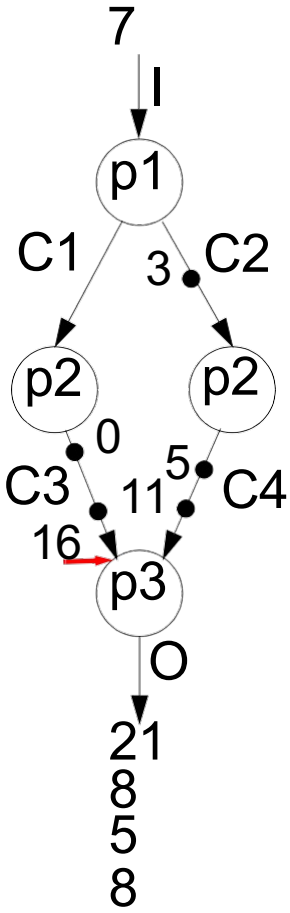


```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
            x.send(k);
    else
            y.send(k);
    end if;
end loop; }

Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }

Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
            k = a.receive();
    else
            k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

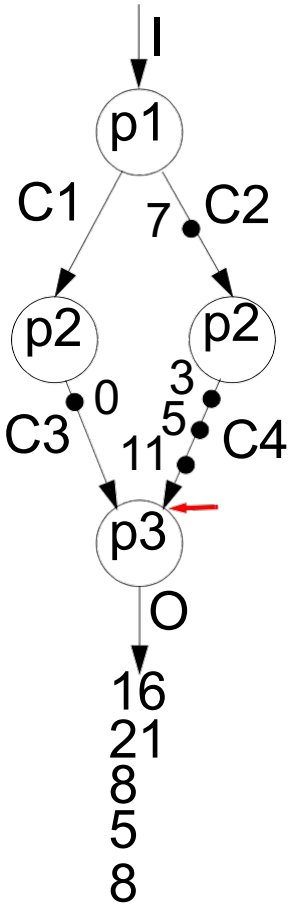


```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
            x.send(k);
    else
            y.send(k);
    end if;
end loop; }

Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }

Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
            k = a.receive();
    else
            k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

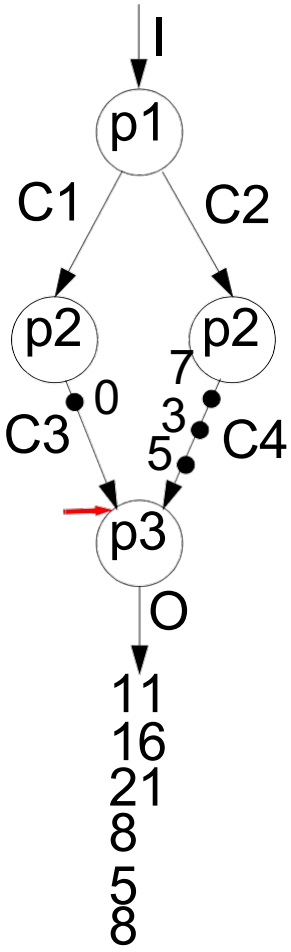


```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
            x.send(k);
    else
            y.send(k);
    end if;
end loop; }

Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }

Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
            k = a.receive();
    else
            k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

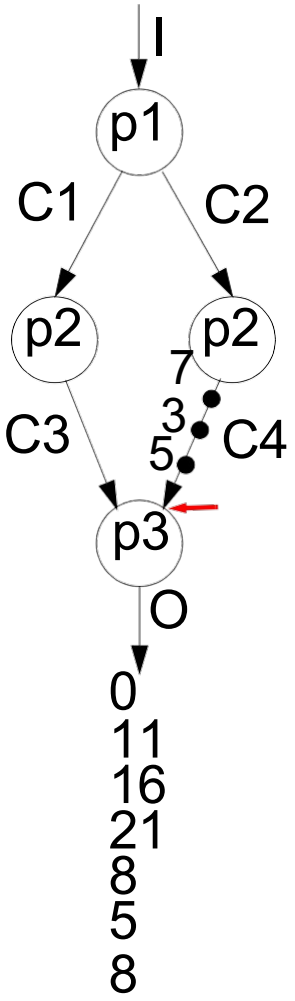


```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
            x.send(k);
    else
            y.send(k);
    end if;
end loop; }

Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }

Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
            k = a.receive();
    else
            k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

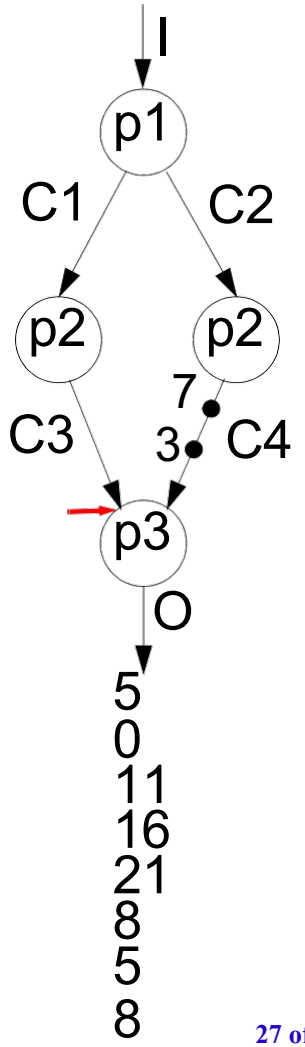


```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
            x.send(k);
    else
            y.send(k);
    end if;
end loop; }

Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }

Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
            k = a.receive();
    else
            k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

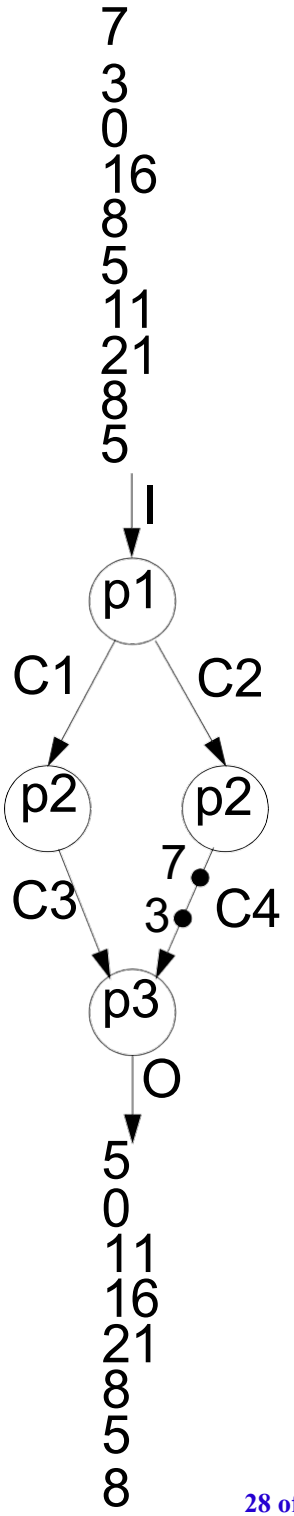


```
Process p1( in int a, out int x, out int y) {
int k;
loop
    k = a.receive();
    if k mod 2 = 0 then
            x.send(k);
    else
            y.send(k);
    end if;
end loop; }

Process p2( in int a, out int x) {
int k;
loop
    k = a.receive();
    x.send(k);
end loop; }

Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
            k = a.receive();
    else
            k = b.receive();
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

```
channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```



Kahn Process Networks: Determinism

- For the same input sequence, the produced output sequence is always the same
- These factors entirely determine the outputs of the system:
  - Processes
  - The network
  - Initial tokens
- Timing of the processes and channels do not affect the outputs of the system

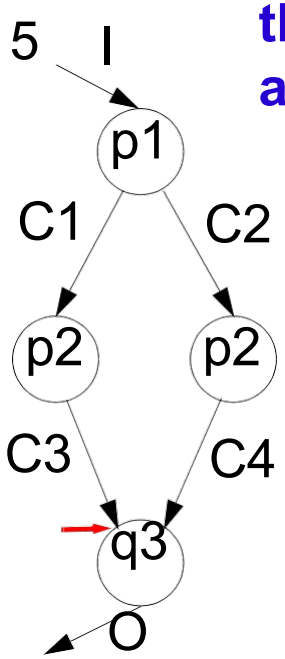
The Modified Network

```
Process q3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
    if sw then
        k = a.receive() on timeout(d) do
            sw = !sw;
            continue;
    else
        k = b.receive() on timeout(d) do
            sw = !sw;
            continue;
    end if;
    x.send(k);
    sw = !sw;
end loop; }
```

- Consider q3 instead of p3:
  - Process q3 first tries channel *a* or *b*, depending on sw, like in the previous version.
  - But, instead of blocking, if nothing comes after a timeout *d*, q3 will switch to read a token from the other channel.

- With q3 we do not have a Kahn process network.
- The system is not deterministic.

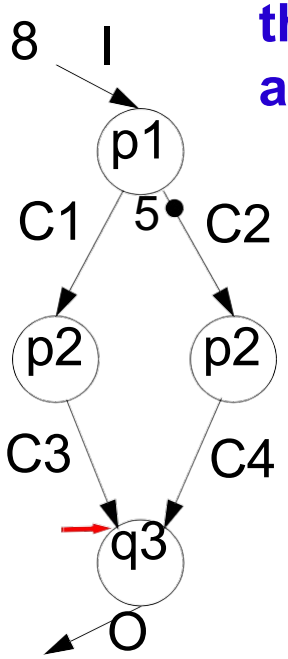
The Modified Network



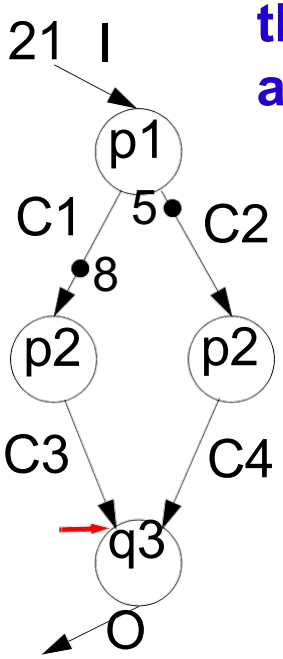
With an implementation such that channel C1 is very fast and C2 is very slow.

The Modified Network

With an implementation such that channel C1 is very fast and C2 is very slow.

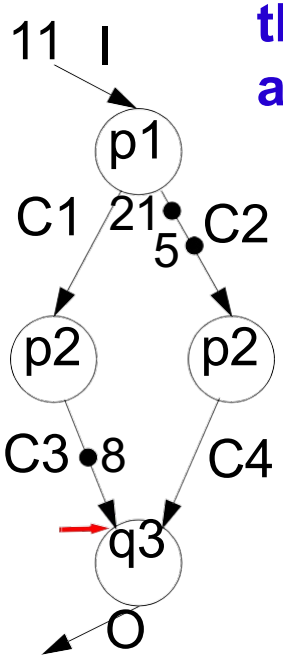


The Modified Network



With an implementation such that channel C1 is very fast and C2 is very slow.

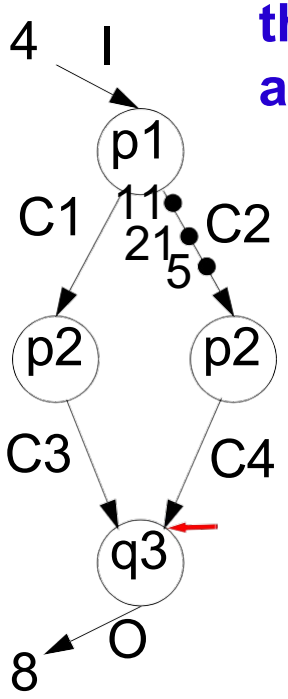
The Modified Network



With an implementation such that channel C1 is very fast and C2 is very slow.

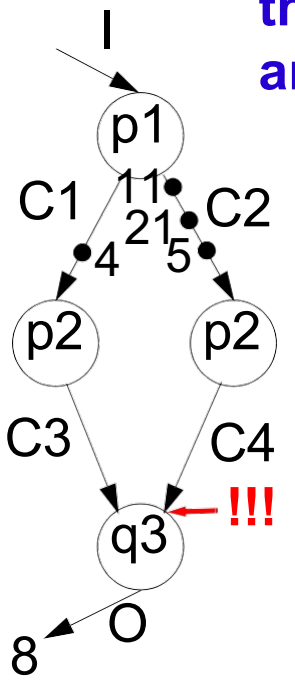
The Modified Network

With an implementation such that channel C1 is very fast and C2 is very slow.



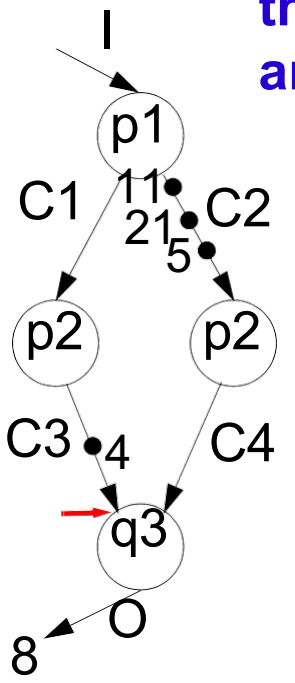
The Modified Network

With an implementation such that channel C1 is very fast and C2 is very slow.



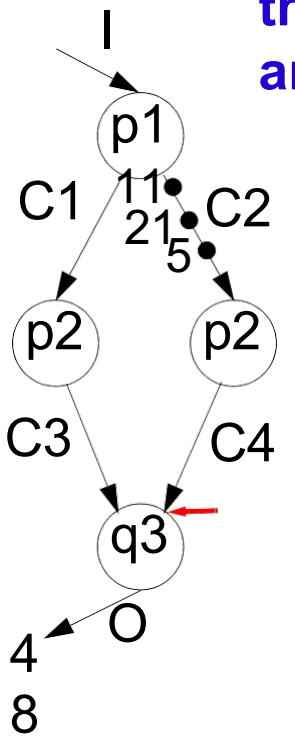
The Modified Network

With an implementation such that channel C1 is very fast and C2 is very slow.



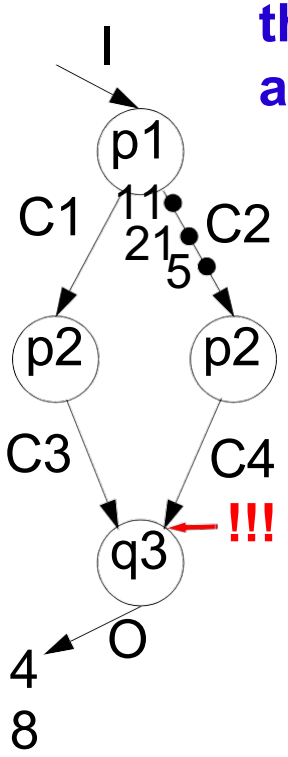
The Modified Network

With an implementation such that channel C1 is very fast and C2 is very slow.



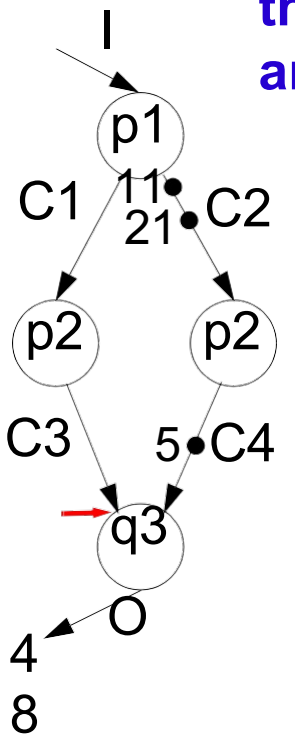
The Modified Network

With an implementation such that channel C1 is very fast and C2 is very slow.



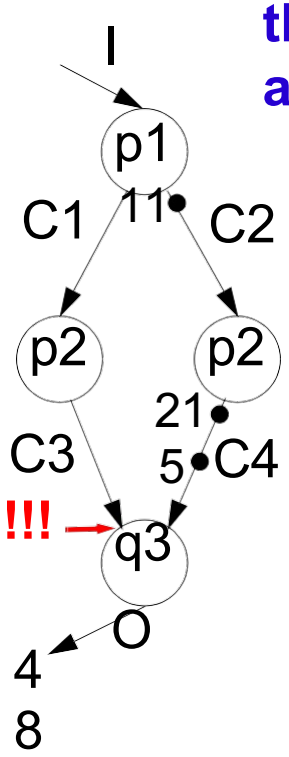
The Modified Network

With an implementation such that channel C1 is very fast and C2 is very slow.



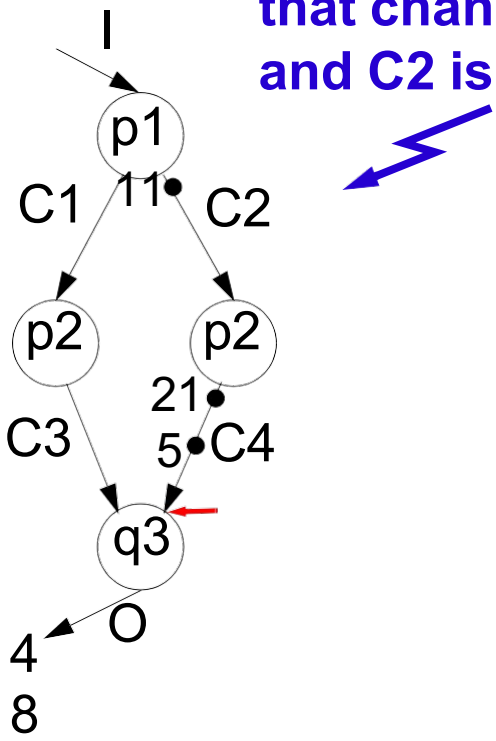
The Modified Network

With an implementation such that channel C1 is very fast and C2 is very slow.



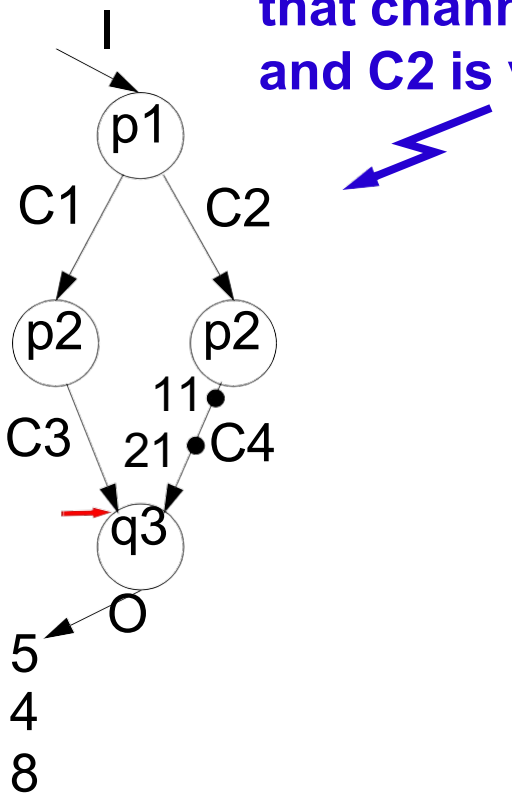
The Modified Network

With an implementation such that channel C1 is very fast and C2 is very slow.



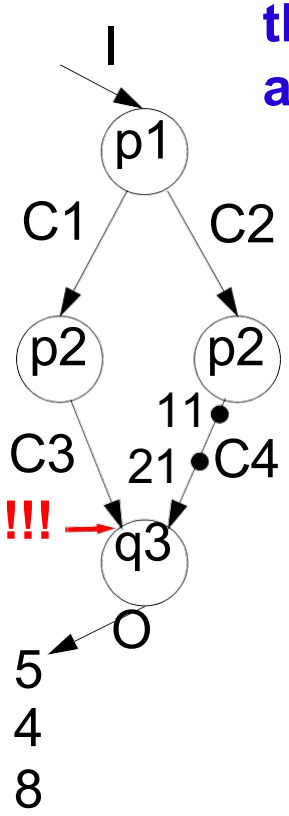
The Modified Network

With an implementation such that channel C1 is very fast and C2 is very slow.



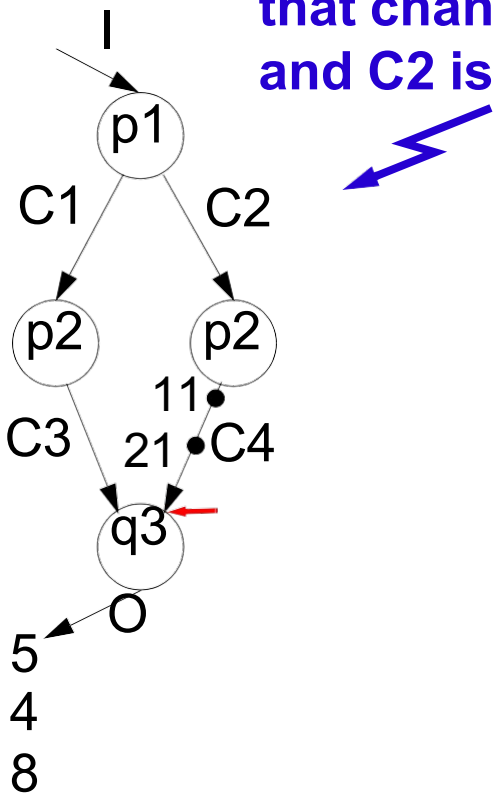
The Modified Network

With an implementation such that channel C1 is very fast and C2 is very slow.



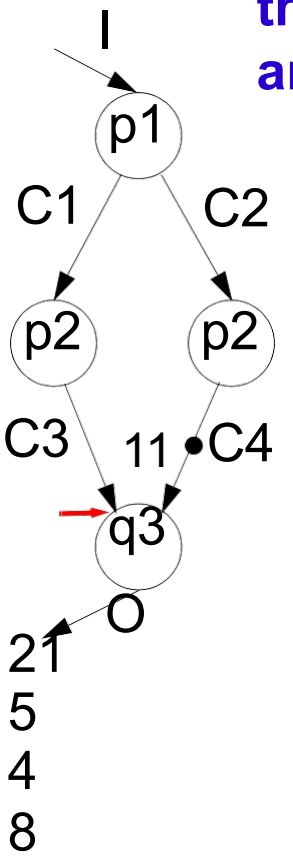
The Modified Network

With an implementation such that channel C1 is very fast and C2 is very slow.



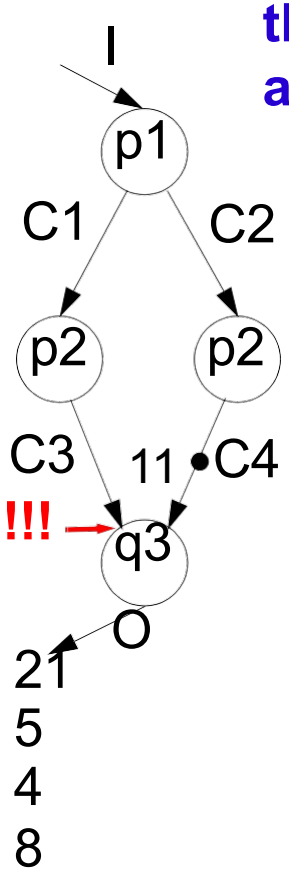
The Modified Network

With an implementation such that channel C1 is very fast and C2 is very slow.



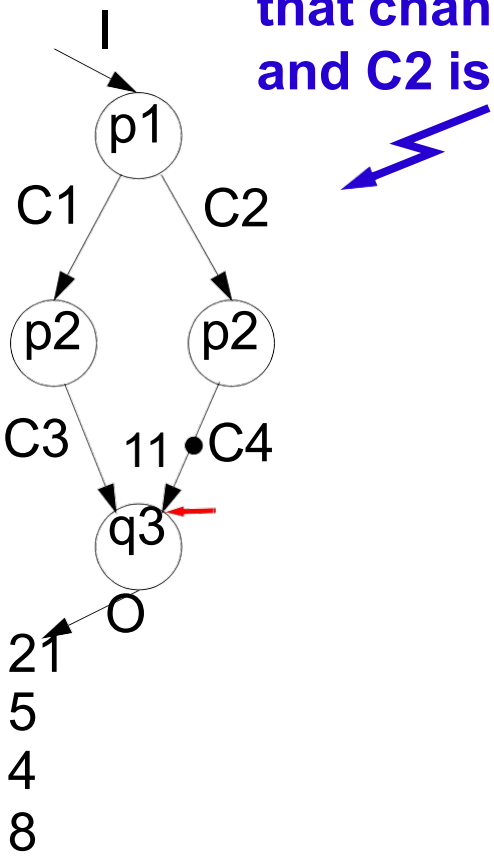
The Modified Network

With an implementation such that channel C1 is very fast and C2 is very slow.



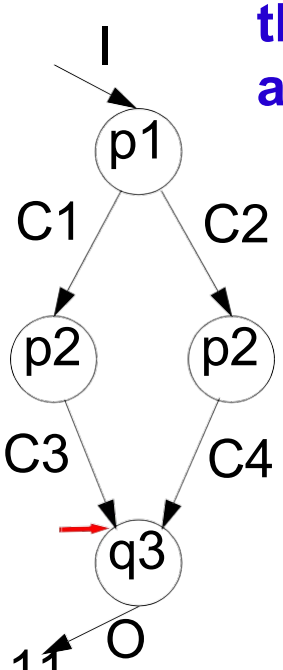
The Modified Network

With an implementation such that channel C1 is very fast and C2 is very slow.



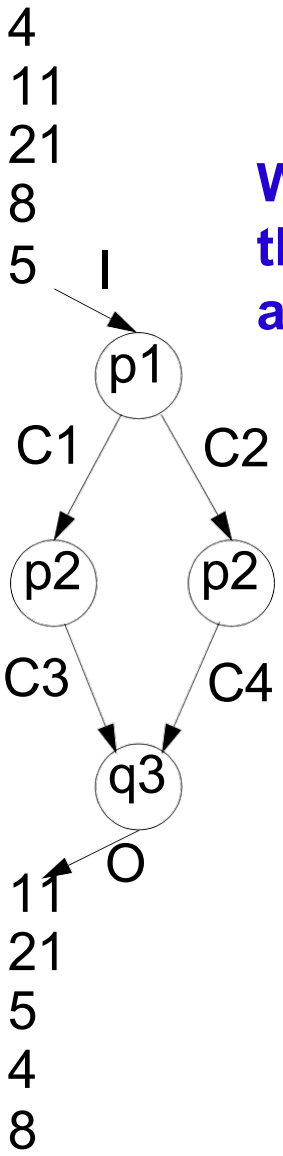
The Modified Network

With an implementation such that channel C1 is very fast and C2 is very slow.



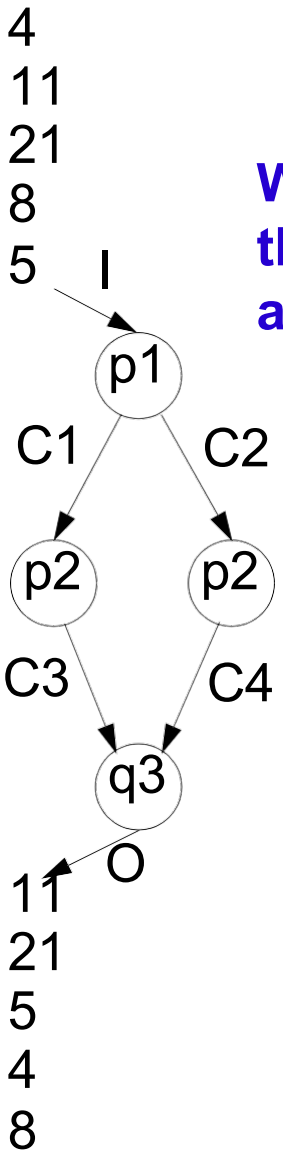
11
21
5
4
8

The Modified Network



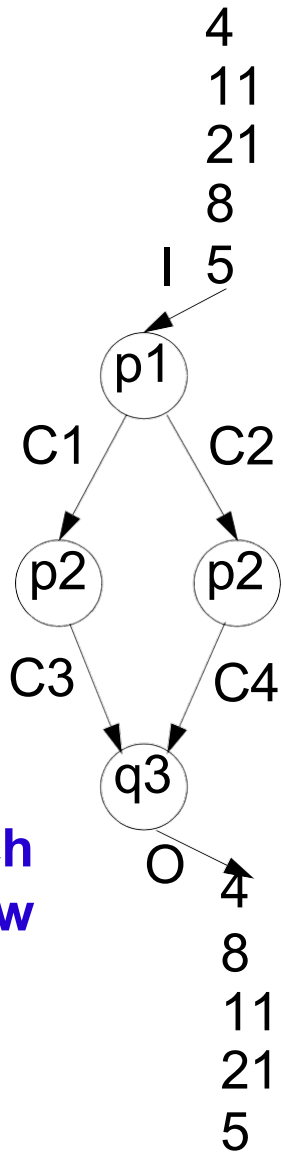
With an implementation such that channel C1 is very fast and C2 is very slow.

The Modified Network

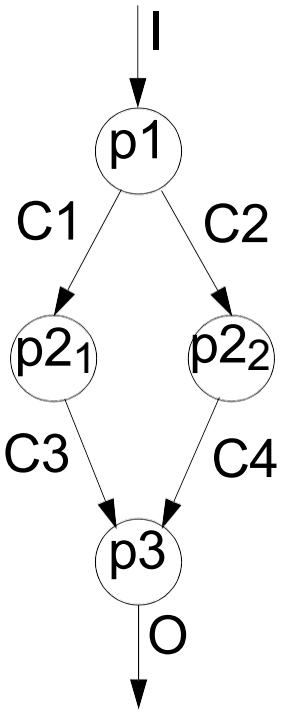


With an implementation such that channel C1 is very fast and C2 is very slow.

With an implementation such that channel C1 is very slow and C2 is very fast.

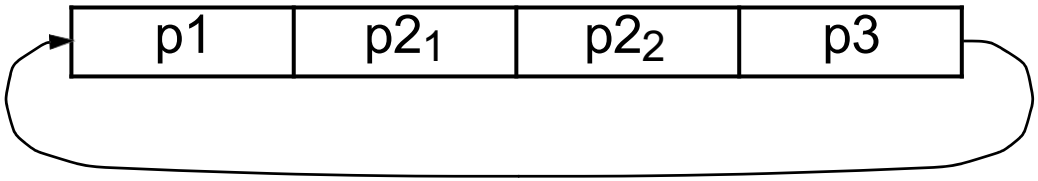


Scheduling of Kahn Process Networks

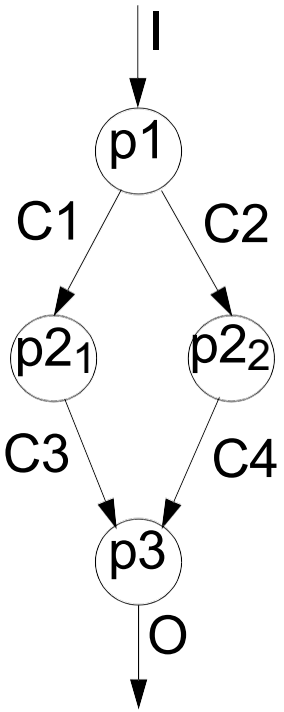


- Let us imagine we have to implement the system on a single processor architecture.

  Let's try the following static schedule:

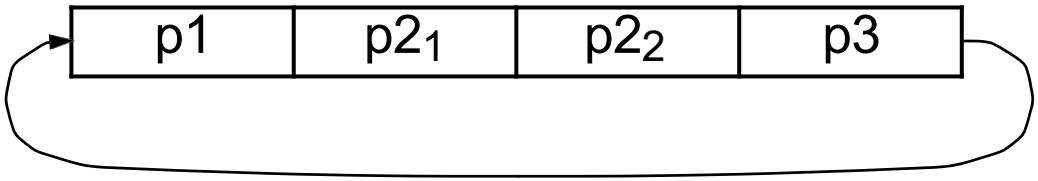


Scheduling of Kahn Process Networks



- Let us imagine we have to implement the system on a single processor architecture.

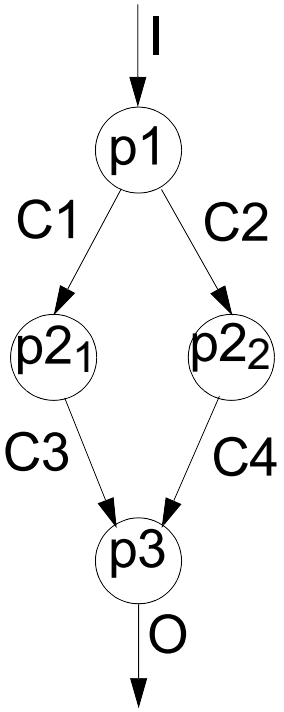
  Let's try the following static schedule:

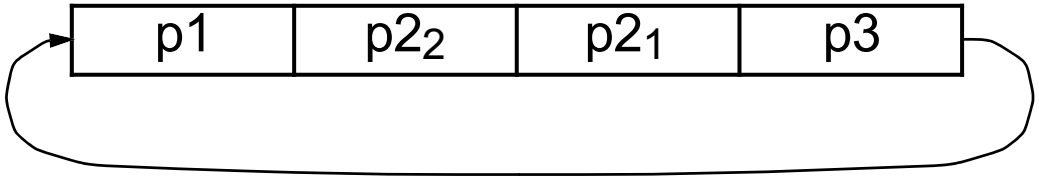


The system will block!

Scheduling of Kahn Process Networks

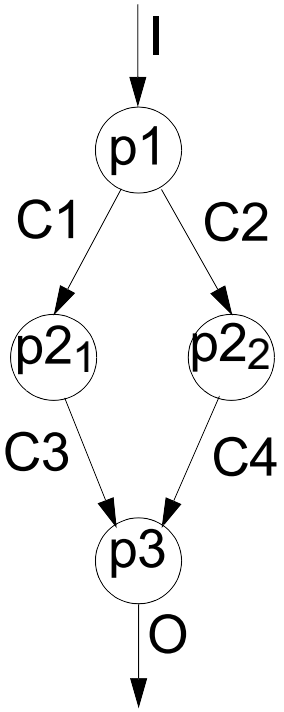
And all other schedules will block:

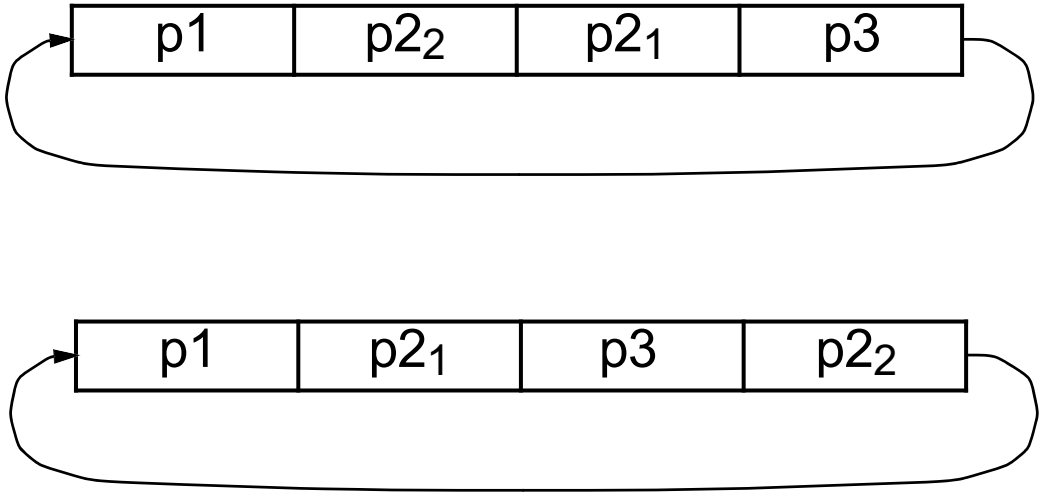




Scheduling of Kahn Process Networks

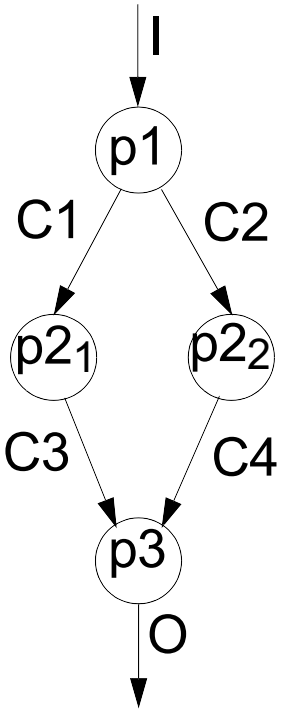
And all other schedules will block:

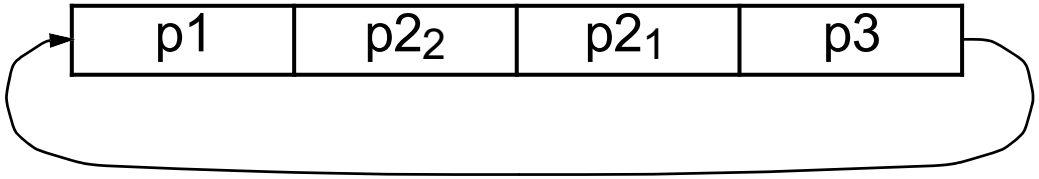


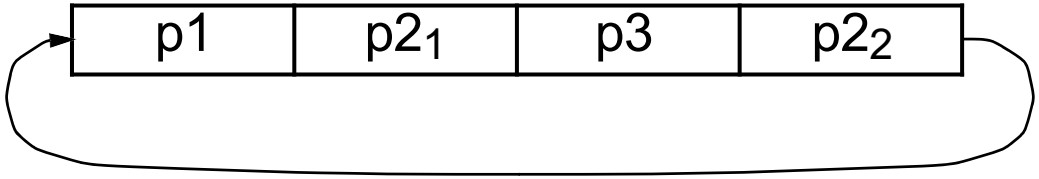


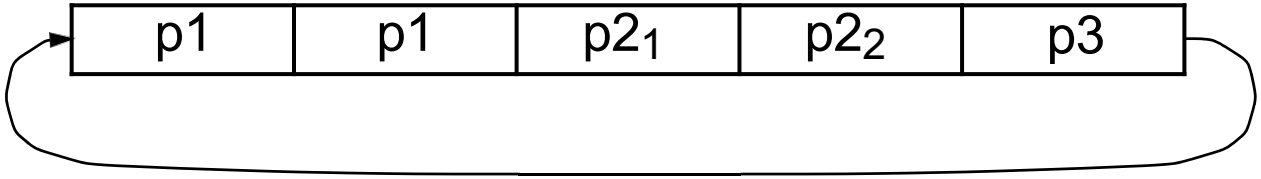
Scheduling of Kahn Process Networks

And all other schedules will block:









Scheduling of Kahn Process Networks

- Kahn process networks are *dynamic* dataflow models: their behavior is data dependent; depending on the input data one or the other process is activated.

- Kahn process networks cannot be scheduled statically $\Rightarrow$ It is not possible to derive, at compile time, a sequence of process activations such that the system does not block under any circumstances.

Kahn process networks have to be scheduled dynamically $\Rightarrow$ which process to activate at a certain moment has to be decided, during execution time, based on the current situation.

There is an overhead in implementing Kahn process networks.

Kahn Process Networks

- Another problem: memory overhead with buffers.
  Potentially, it is possible that the memory need for buffers grows unlimited.

  Possible approaches:
  - For some applications and restrictions on inputs, FIFO bounds can be mathematically derived in design to avoid FIFO overflows
  - FIFO bounds can be grown on demand
  - Blocking writes can be used so that a process blocks if a FIFO is full (this deviates from the KPN semantics and may lead to deadlocks, which add further implementation issues)

- Kahn process networks are relatively strong in their expressive power but sometimes cannot be implemented efficiently.

Introduce more limitations so that you can get efficient implementations.

Synchronous Dataflow Models

- *Dataflow process networks* are a particular case of Kahn process networks.

  A particular kind of dataflow process networks, which can be efficiently implemented, are *synchronous dataflow (SDF) networks*.

- *Synchronous dataflow networks* are Kahn process networks with restriction:
  - At each activation (firing) a process produces and consumes a fixed number of tokens on each of its outgoing and incoming channels.
  - For a process to fire, it must have at least as many tokens on its input channels as it has to consume.

Synchronous Dataflow Models

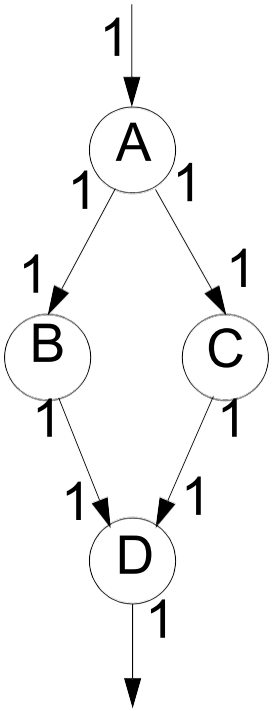
- Synchronous dataflow models are less expressive than Kahn process networks:
  - With SDF models it is impossible to express conditional firing, where a process' firing depends on a certain condition; SDF are *static* dataflow models.

Synchronous Dataflow Models

- Synchronous dataflow models are less expressive than Kahn process networks:
  - With SDF models it is impossible to express conditional firing, where a process' firing depends on a certain condition; SDF are *static* dataflow models.

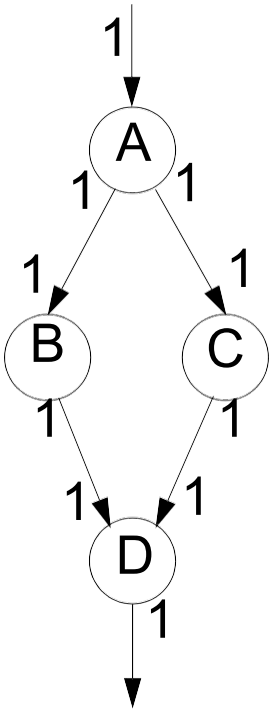
- For the above reduced expressiveness, however, we get two nice features of SDF models:
  1. Possibility to produce static schedules.
  2. Limited and predictable amount of needed buffer space.

Synchronous Dataflow Models



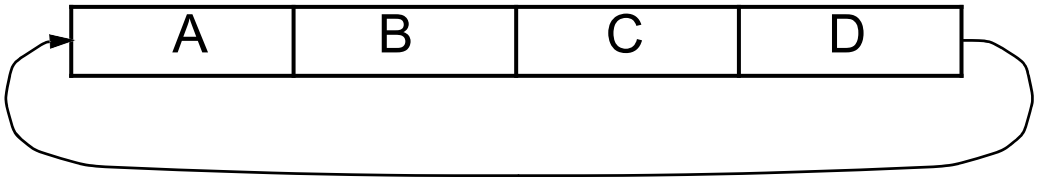
- Arcs are marked with the number of tokens produced or consumed.
- This is a simple “single-rate” system: every process is activated one single time before the system returns to its initial state.

Synchronous Dataflow Models



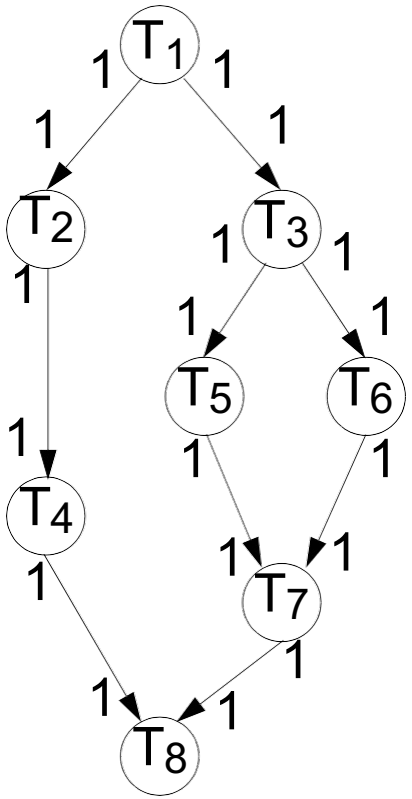
- Arcs are marked with the number of tokens produced or consumed.
- This is a simple “single-rate” system: every process is activated one single time before the system returns to its initial state.

Possible static schedule:



Synchronous Dataflow Models

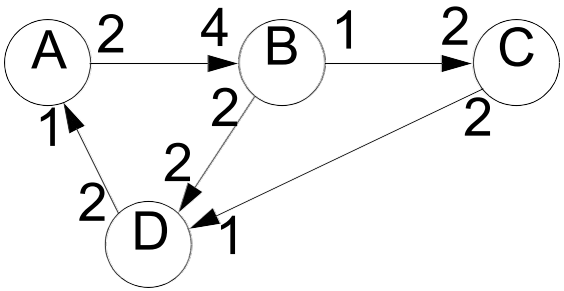
Our example from Lecture 1:



A static schedule:

T_1	T_2	T_4	T_3	T_5	T_6	T_7	T_8

Deriving a static schedule for SDF



- For a correct synchronous dataflow network there exists a sequence of firings which returns the network in its original state.

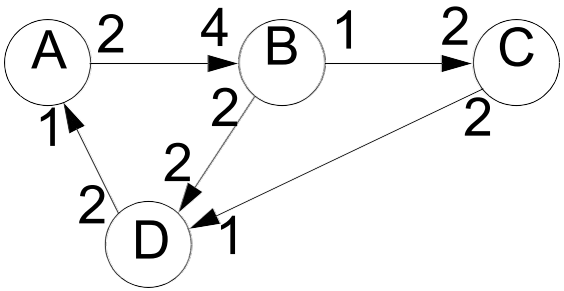
  This sequence represents a static schedule which has to be repeated in a cycle.

- The schedule is such that a finite amount of memory is required (no infinite buffers)

Problem

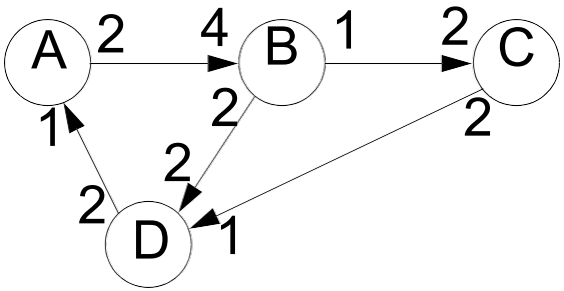
How to derive such a cyclic schedule?

Deriving a static schedule for SDF



- Along the periodic sequence of firing, on each arc the same number of tokens has to be produced and consumed.

Deriving a static schedule for SDF



- Along the periodic sequence of firing, on each arc the same number of tokens has to be produced and consumed.

  a, b, c, d: the number of firings, during a period, for process A, B, C, D.

Balance equations:

$$2a - 4b = 0$$

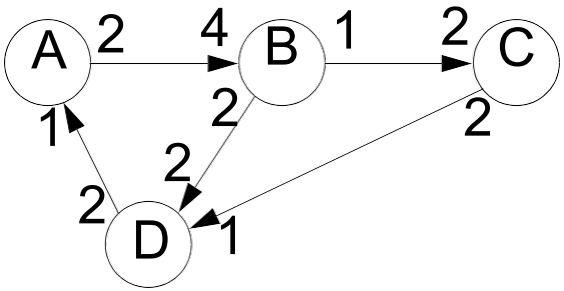
$$b - 2c = 0$$

$$2c - d = 0$$

$$2b - 2d = 0$$

$$2d - a = 0$$

Deriving a static schedule for SDF



- Along the periodic sequence of firing, on each arc the same number of tokens has to be produced and consumed.

  a, b, c, d: the number of firings, during a period, for process A, B, C, D.

Balance equations:

$2a - 4b = 0$

$b - 2c = 0$

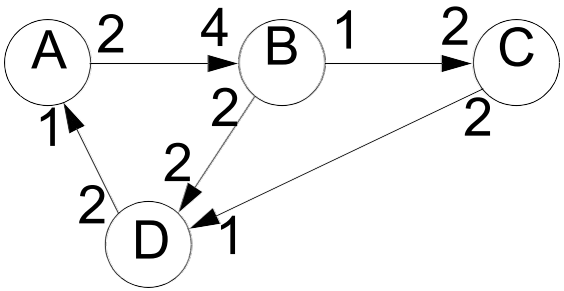
$2c - d = 0$

$2b - 2d = 0$

$2d - a = 0$

$$\begin{bmatrix} 2 & -4 & 0 & 0 \\ 0 & 1 & -2 & 0 \\ 0 & 0 & 2 & -1 \\ 0 & 2 & 0 & -2 \\ -1 & 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = 0$$

Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

$$\Gamma \mathbf{q} = \mathbf{0}$$

Balance equations:

2a - 4b = 0

b - 2c = 0

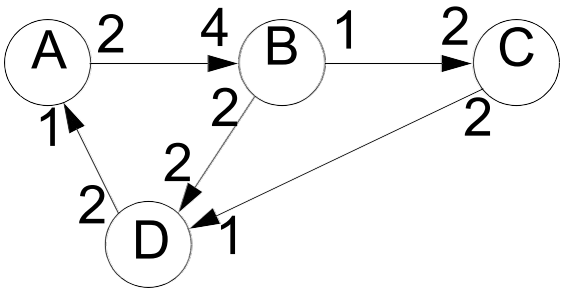
2c - d = 0

2b - 2d = 0

2d - a = 0

$$\begin{bmatrix} 2 & -4 & 0 & 0 \\ 0 & 1 & -2 & 0 \\ 0 & 0 & 2 & -1 \\ 0 & 2 & 0 & -2 \\ -1 & 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = 0$$

Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

$$\Gamma \mathbf{q} = \mathbf{0}$$

topology matrix of the graph

Balance equations:

2a - 4b = 0

b - 2c = 0

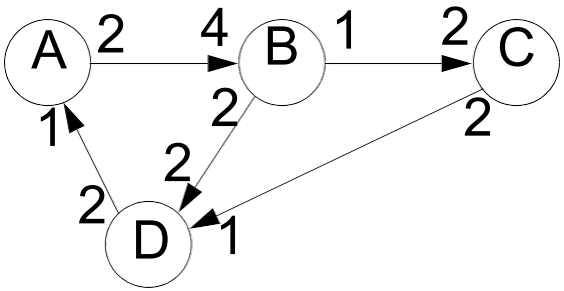
2c - d = 0

2b - 2d = 0

2d - a = 0

$$\begin{bmatrix} 2 & -4 & 0 & 0 \\ 0 & 1 & -2 & 0 \\ 0 & 0 & 2 & -1 \\ 0 & 2 & 0 & -2 \\ -1 & 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = 0$$

Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

$$\Gamma \mathbf{q} = \mathbf{0}$$

topology matrix of the graph

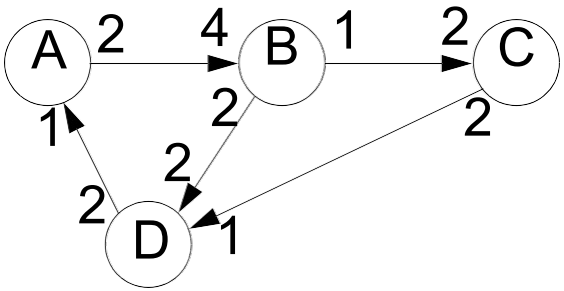
firing vector

Balance equations:

$$2a - 4b = 0$$
$$b - 2c = 0$$
$$2c - d = 0$$
$$2b - 2d = 0$$
$$2d - a = 0$$

$$\begin{bmatrix} 2 & -4 & 0 & 0 \\ 0 & 1 & -2 & 0 \\ 0 & 0 & 2 & -1 \\ 0 & 2 & 0 & -2 \\ -1 & 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = 0$$

Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

$$\Gamma \mathbf{q} = \mathbf{0}$$

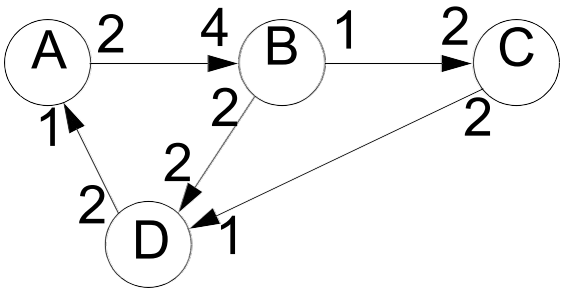
- Γ: topology matrix of the graph
- $\mathbf{q}$: firing vector
- $\mathbf{0}$: vector of zeros

Balance equations:

$$2a - 4b = 0$$
$$b - 2c = 0$$
$$2c - d = 0$$
$$2b - 2d = 0$$
$$2d - a = 0$$

$$\begin{bmatrix} 2 & -4 & 0 & 0 \\ 0 & 1 & -2 & 0 \\ 0 & 0 & 2 & -1 \\ 0 & 2 & 0 & -2 \\ -1 & 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 0 \end{bmatrix}$$

Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

$$\Gamma \mathbf{q} = \mathbf{0}$$

- Γ: topology matrix of the graph
- $\mathbf{q}$: firing vector
- $\mathbf{0}$: vector of zeros

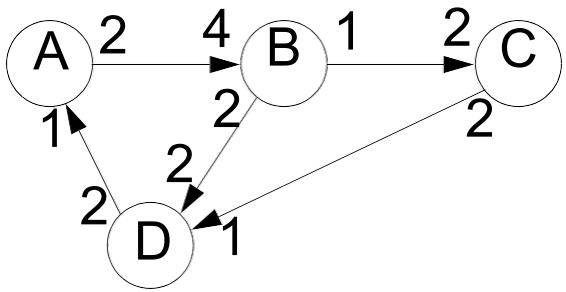
- If there is no $q \neq 0$ which satisfies the equation above $\Rightarrow$ there is no static schedule (there is a *rate inconsistency* between processes).

Balance equations:

$$2a - 4b = 0$$
$$b - 2c = 0$$
$$2c - d = 0$$
$$2b - 2d = 0$$
$$2d - a = 0$$

$$\begin{bmatrix} 2 & -4 & 0 & 0 \\ 0 & 1 & -2 & 0 \\ 0 & 0 & 2 & -1 \\ 0 & 2 & 0 & -2 \\ -1 & 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = 0$$

Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

$$\Gamma \mathbf{q} = \mathbf{0}$$

- **Among possible solutions for vector *q*, we are interested in the smallest positive integer vector (smallest sum of the elements).**

  For our SDF graph, this solution is: a=4, b=2, c=1, d=2.

  a, b, c, d indicate how often each task is activated during one period.

Balance equations:

2a - 4b = 0

b - 2c = 0

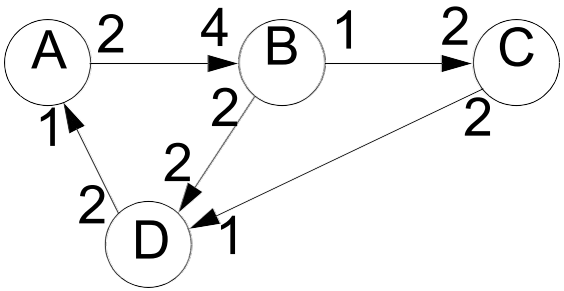
2c - d = 0

2b - 2d = 0

2d - a = 0

$$\begin{bmatrix} 2 & -4 & 0 & 0 \\ 0 & 1 & -2 & 0 \\ 0 & 0 & 2 & -1 \\ 0 & 2 & 0 & -2 \\ -1 & 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = 0$$

Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

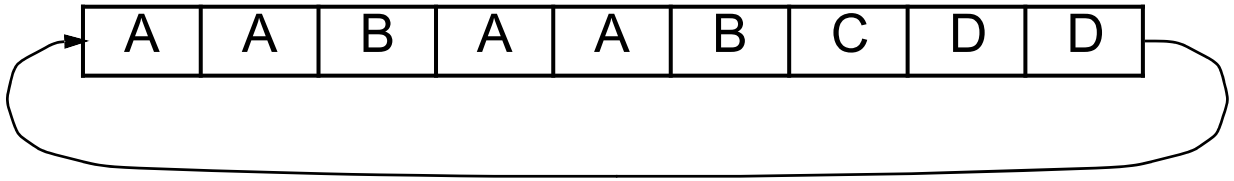
$$\Gamma q = 0$$

- Among possible solutions for vector *q*, we are interested in the smallest positive integer vector (smallest sum of the elements).

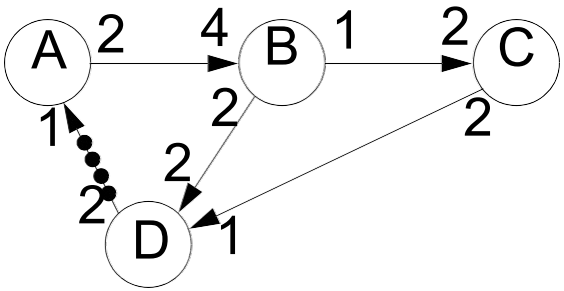
  For our SDF graph, this solution is:
  a=4, b=2, c=1, d=2.

  a, b, c, d indicate how often each task is activated during one period.

A possible schedule:



Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

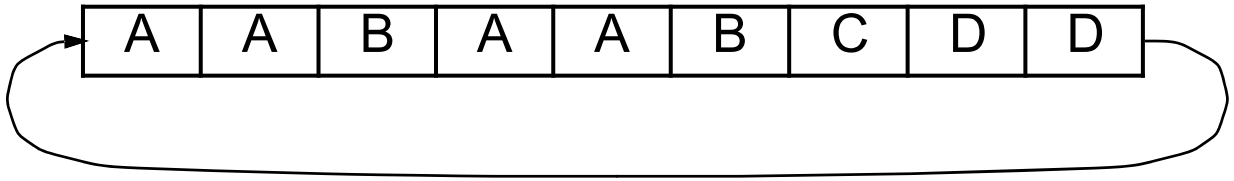
$$\Gamma \mathbf{q} = \mathbf{0}$$

- Among possible solutions for vector *q*, we are interested in the smallest positive integer vector (smallest sum of the elements).

  For our SDF graph, this solution is:
  a=4, b=2, c=1, d=2.

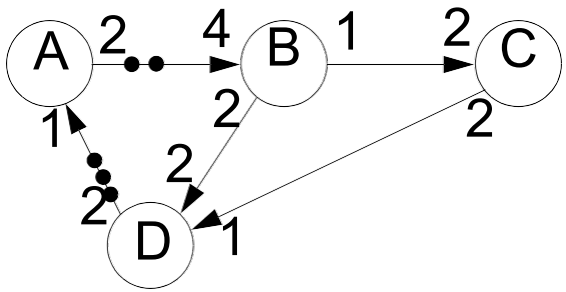
  a, b, c, d indicate how often each task is activated during one period.

A possible schedule:



The schedule is possible, without deadlock, only if 4 initial tokens are provided on the channel D → A.

Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

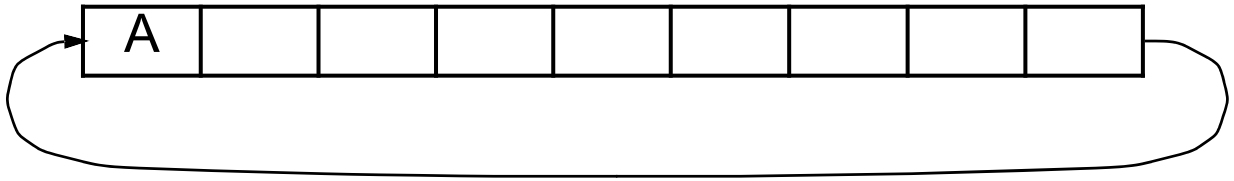
$$\Gamma q = 0$$

- Among possible solutions for vector *q*, we are interested in the smallest positive integer vector (smallest sum of the elements).

  For our SDF graph, this solution is:
  a=4, b=2, c=1, d=2.

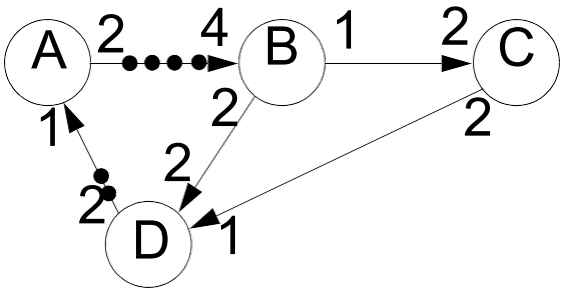
  a, b, c, d indicate how often each task is activated during one period.

A possible schedule:



The schedule is possible, without deadlock, only if 4 initial tokens are provided on the channel D → A.

Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

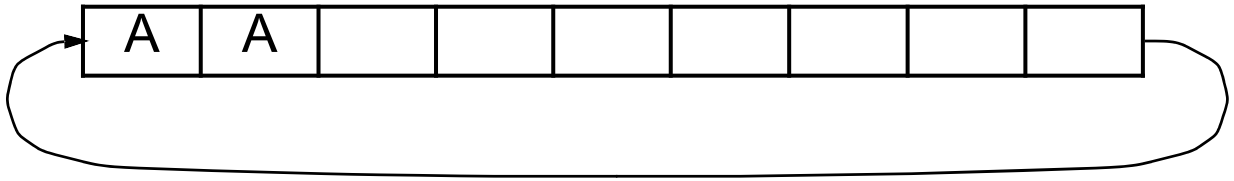
$$\Gamma \mathbf{q} = \mathbf{0}$$

- Among possible solutions for vector *q*, we are interested in the smallest positive integer vector (smallest sum of the elements).

  For our SDF graph, this solution is:
  a=4, b=2, c=1, d=2.

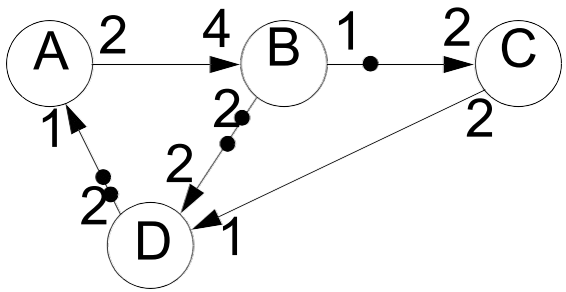
  a, b, c, d indicate how often each task is activated during one period.

A possible schedule:



The schedule is possible, without deadlock, only if 4 initial tokens are provided on the channel D → A.

Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

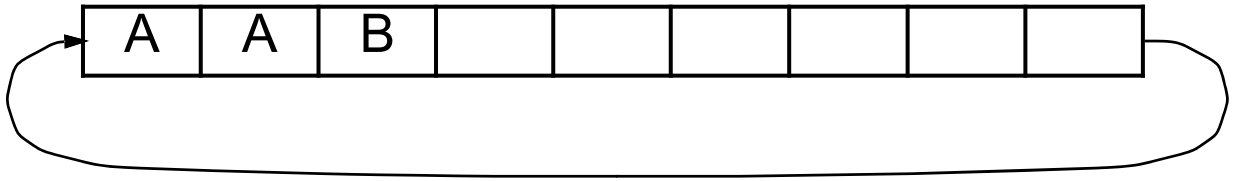
$$\Gamma q = 0$$

- Among possible solutions for vector *q*, we are interested in the smallest positive integer vector (smallest sum of the elements).

  For our SDF graph, this solution is:
  a=4, b=2, c=1, d=2.

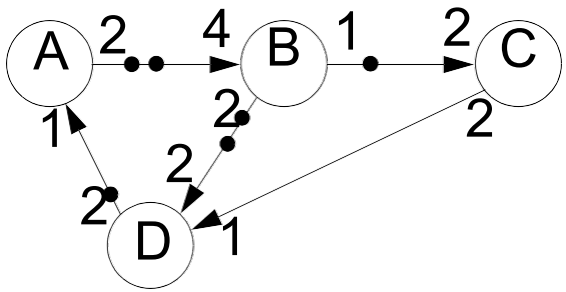
  a, b, c, d indicate how often each task is activated during one period.

A possible schedule:



The schedule is possible, without deadlock, only if 4 initial tokens are provided on the channel D → A.

Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

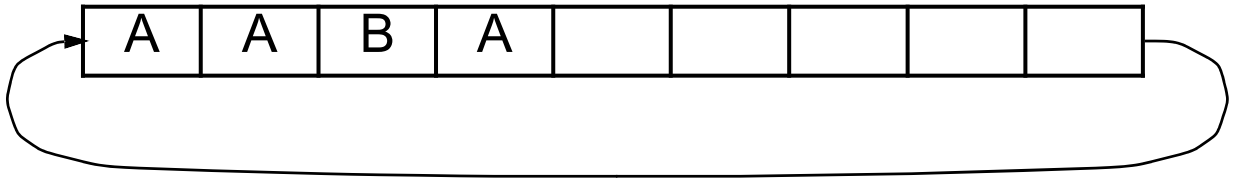
$$\Gamma \mathbf{q} = \mathbf{0}$$

- Among possible solutions for vector *q*, we are interested in the smallest positive integer vector (smallest sum of the elements).

  For our SDF graph, this solution is:
  a=4, b=2, c=1, d=2.

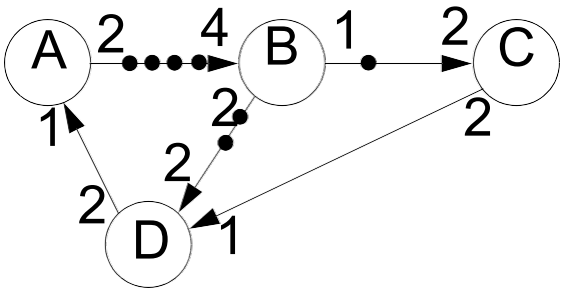
  a, b, c, d indicate how often each task is activated during one period.

A possible schedule:



The schedule is possible, without deadlock, only if 4 initial tokens are provided on the channel D → A.

Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

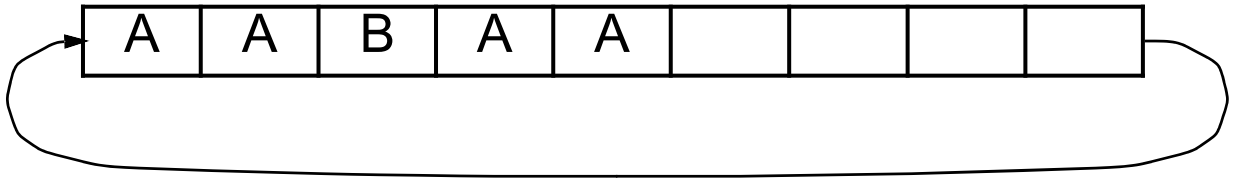
$$\Gamma \mathbf{q} = \mathbf{0}$$

- Among possible solutions for vector *q*, we are interested in the smallest positive integer vector (smallest sum of the elements).

  For our SDF graph, this solution is:
  a=4, b=2, c=1, d=2.

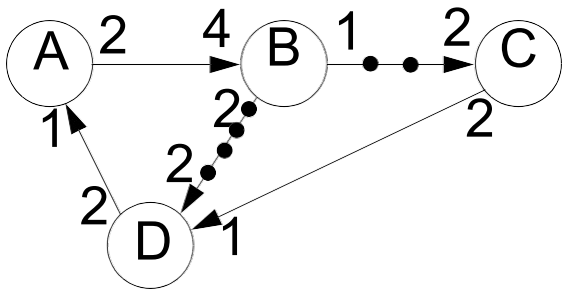
  a, b, c, d indicate how often each task is activated during one period.

A possible schedule:



The schedule is possible, without deadlock, only if 4 initial tokens are provided on the channel D → A.

Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

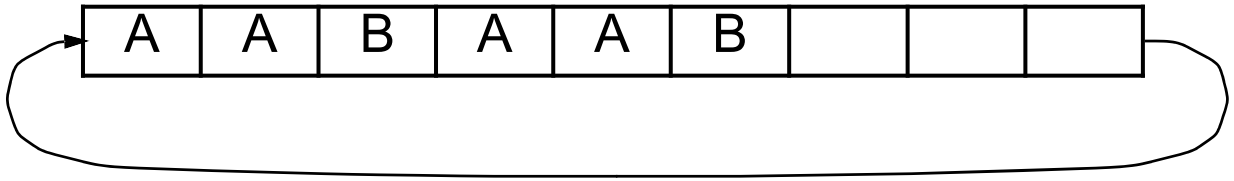
$$\Gamma \mathbf{q} = \mathbf{0}$$

- Among possible solutions for vector *q*, we are interested in the smallest positive integer vector (smallest sum of the elements).

  For our SDF graph, this solution is:
  a=4, b=2, c=1, d=2.

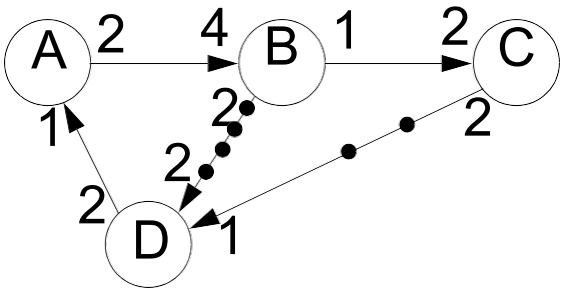
  a, b, c, d indicate how often each task is activated during one period.

A possible schedule:



The schedule is possible, without deadlock, only if 4 initial tokens are provided on the channel D → A.

Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

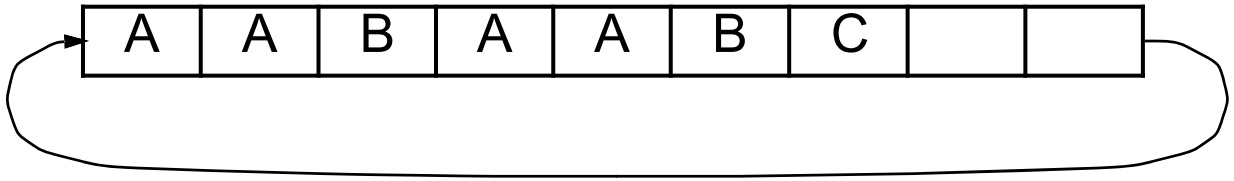
$$\Gamma \mathbf{q} = \mathbf{0}$$

- Among possible solutions for vector *q*, we are interested in the smallest positive integer vector (smallest sum of the elements).

  For our SDF graph, this solution is:
  a=4, b=2, c=1, d=2.

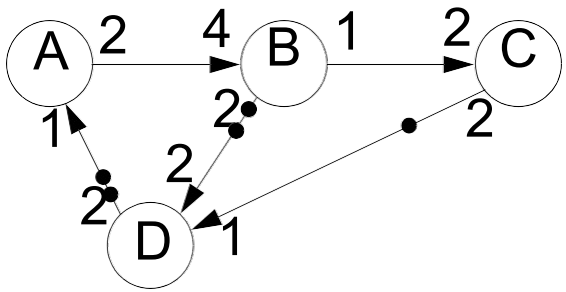
  a, b, c, d indicate how often each task is activated during one period.

A possible schedule:



The schedule is possible, without deadlock, only if 4 initial tokens are provided on the channel D → A.

Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

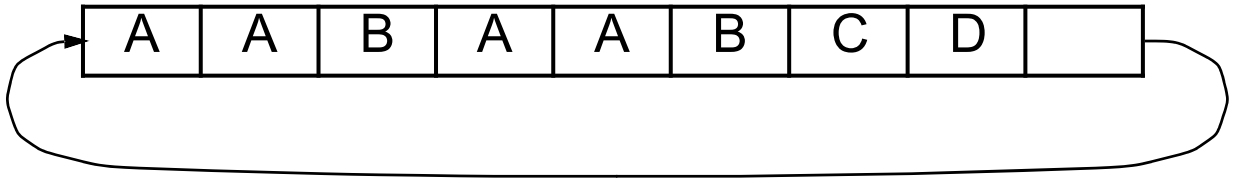
$$\Gamma \mathbf{q} = \mathbf{0}$$

- Among possible solutions for vector *q*, we are interested in the smallest positive integer vector (smallest sum of the elements).

  For our SDF graph, this solution is:
  a=4, b=2, c=1, d=2.

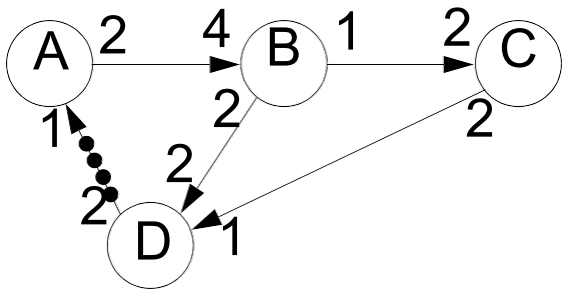
  a, b, c, d indicate how often each task is activated during one period.

A possible schedule:



The schedule is possible, without deadlock, only if 4 initial tokens are provided on the channel D → A.

Deriving a static schedule for SDF



For a given SDF network (graph) we get equation:

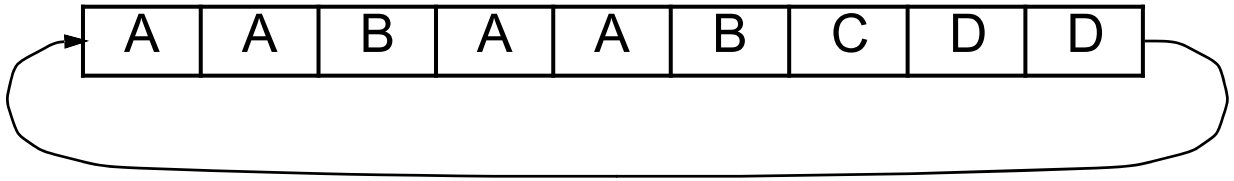
$$\Gamma \mathbf{q} = \mathbf{0}$$

- Among possible solutions for vector *q*, we are interested in the smallest positive integer vector (smallest sum of the elements).

  For our SDF graph, this solution is:
  a=4, b=2, c=1, d=2.

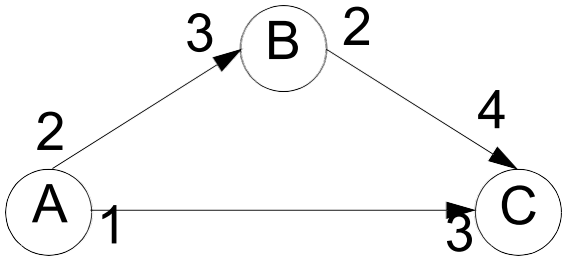
  a, b, c, d indicate how often each task is activated during one period.

A possible schedule:



The schedule is possible, without deadlock, only if 4 initial tokens are provided on the channel D → A.

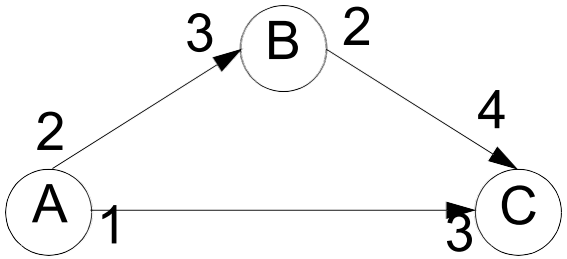
Deriving a static schedule for SDF



$$\begin{bmatrix} 2 & -3 & 0 \\ 0 & 2 & -4 \\ 1 & 0 & -3 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = 0$$

Solution: a=3, b=2, c=1.

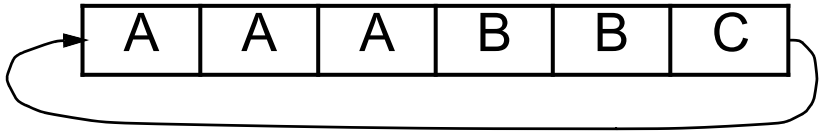
Deriving a static schedule for SDF



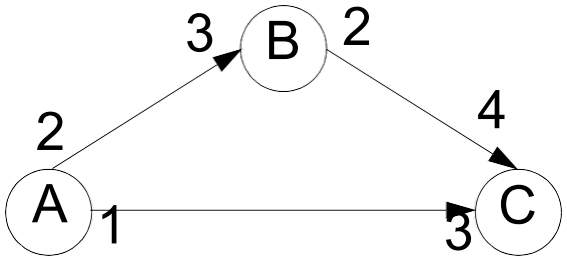
$$\begin{bmatrix} 2 & -3 & 0 \\ 0 & 2 & -4 \\ 1 & 0 & -3 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = 0$$

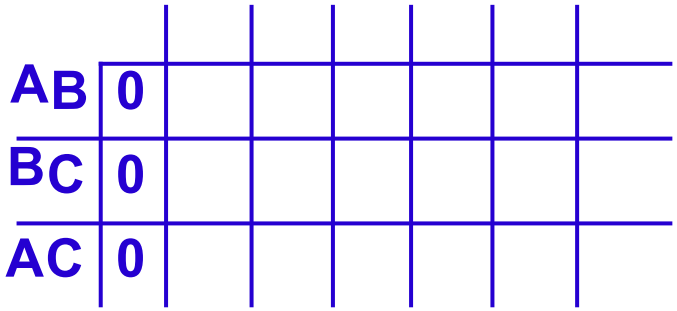
Solution: a=3, b=2, c=1.

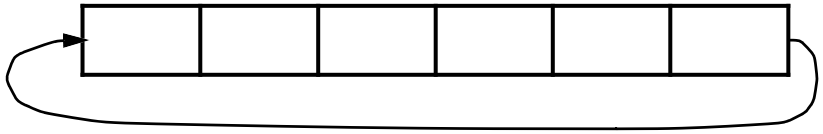
Possible schedule:



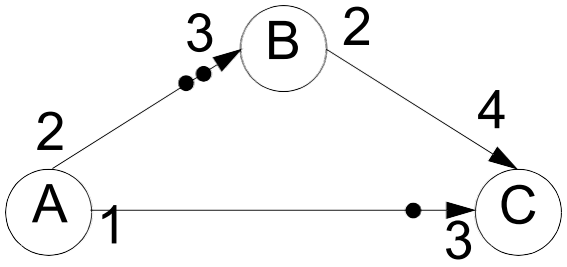
Deriving a static schedule for SDF



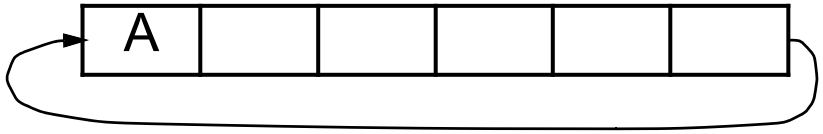




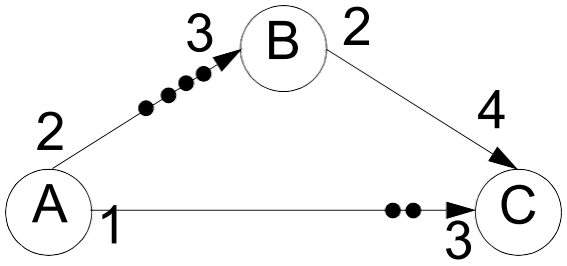
Deriving a static schedule for SDF



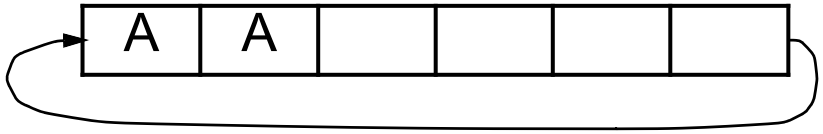
		A
AB	0	2
BC	0	0
AC	0	1



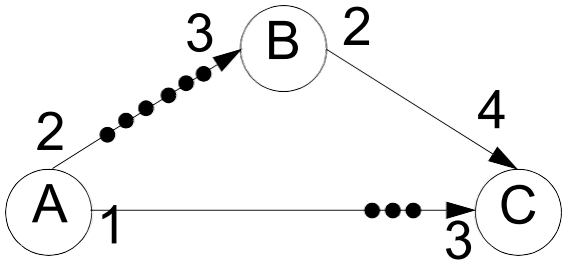
Deriving a static schedule for SDF



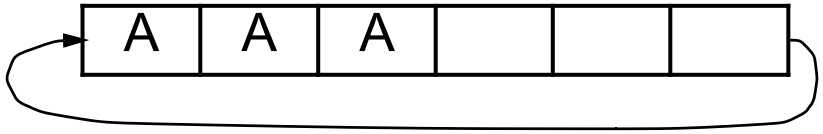
		A	A				
AB	0	2	4				
BC	0	0	0				
AC	0	1	2				



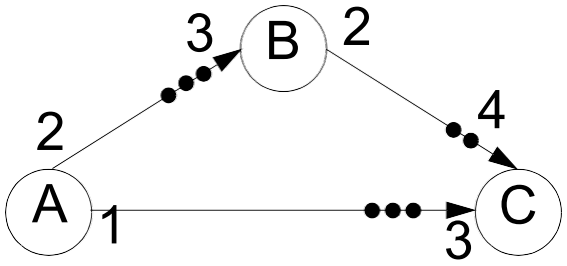
Deriving a static schedule for SDF



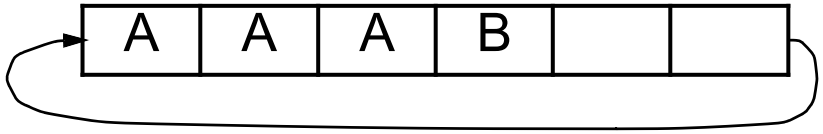
		A	A	A			
AB	0	2	4	6			
BC	0	0	0	0			
AC	0	1	2	3			



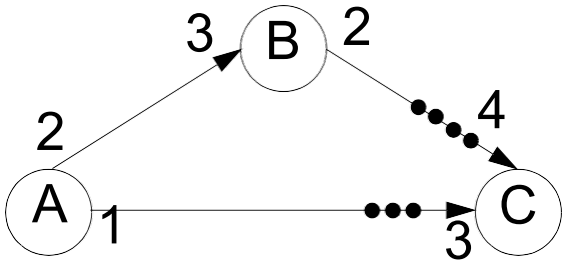
Deriving a static schedule for SDF



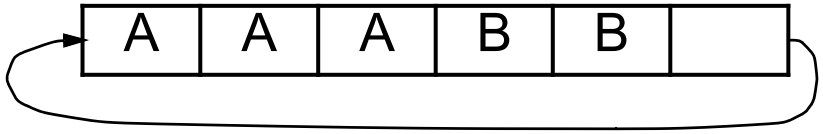
		A	A	A	B		
AB	0	2	4	6	3		
BC	0	0	0	0	2		
AC	0	1	2	3	3		



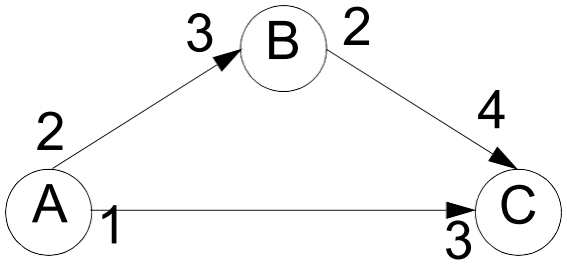
Deriving a static schedule for SDF



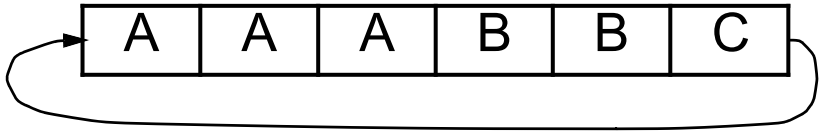
		A	A	A	B	B	
AB	0	2	4	6	3	0	
BC	0	0	0	0	2	4	
AC	0	1	2	3	3	3	



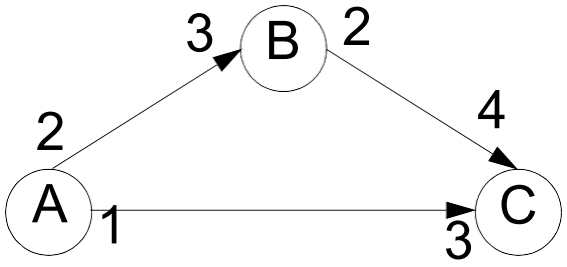
Deriving a static schedule for SDF



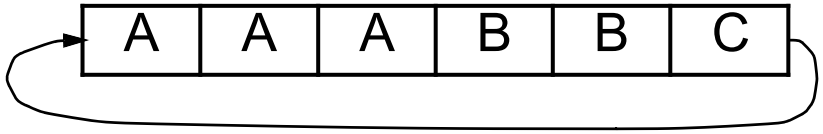
		A	A	A	B	B	C
AB	0	2	4	6	3	0	0
BC	0	0	0	0	2	4	0
AC	0	1	2	3	3	3	0



Deriving a static schedule for SDF



		A	A	A	B	B	C
AB	0	2	4	6	3	0	0
BC	0	0	0	0	2	4	0
AC	0	1	2	3	3	3	0

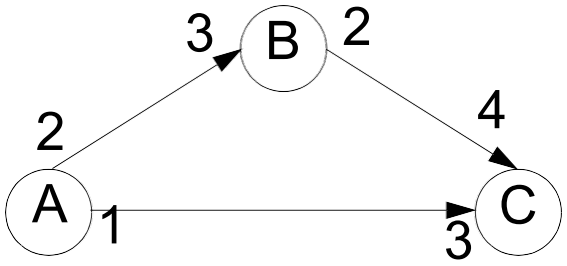


Buffer space needed:

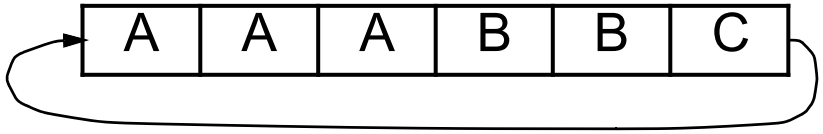
A-B: 6; B-C: 4; A-C: 3;

Total: 13 if buffers not shared

Deriving a static schedule for SDF



		A	A	A	B	B	C
AB	0	2	4	6	3	0	0
BC	0	0	0	0	2	4	0
AC	0	1	2	3	3	3	0
total	0	3	6	9	8	7	0



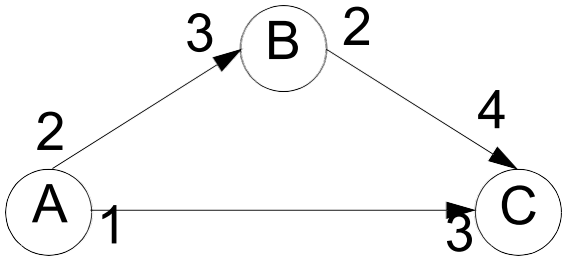
Buffer space needed:

A-B: 6; B-C: 4; A-C: 3;

Total: 13 if buffers not shared

9 if buffers shared

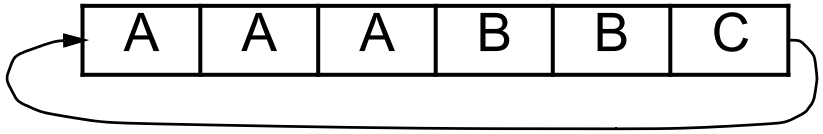
Deriving a static schedule for SDF



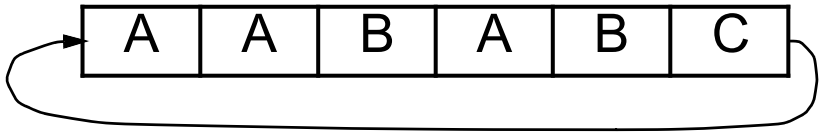
$$\begin{bmatrix} 2 & -3 & 0 \\ 0 & 2 & -4 \\ 1 & 0 & -3 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = 0$$

Solution: a=3, b=2, c=1.

Possible schedule:



Another schedule:



Buffer space needed:

A-B: 6; B-C: 4; A-C: 3;

Total: 13 if buffers not shared

9 if buffers shared

Buffer space needed:

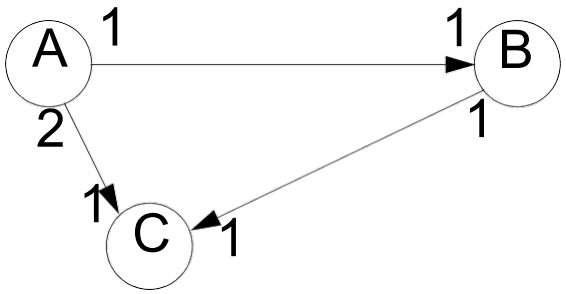
A-B: 4; B-C: 4; A-C: 3;

Total: 11 if buffers not shared

8 if buffers shared

Deriving a static schedule for SDF

- With this example we have a rate inconsistency $\Rightarrow$ No static, periodic schedule with finite buffers is possible.



$$\begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = 0$$

- There is no solution for the equation, different from a=b=c=0.

- It is easy to observe that on the arc A $\rightarrow$ C, tokens continuously accumulate.

Treatment of Time

- Dataflow systems are *asynchronous concurrent*.
  - Events can happen at any time.
  - There exists a a partial order of events:
    - Producing a token by A strictly precedes consuming a token by B and C.
    - There is no order between consuming a token by B and consuming a token by C.