Concurrent Computing

Lecturers: Prof. Majid Mirmehdi majid@cs.bris.ac.uk
Dr. Tilo Burghardt tilo@cs.bris.ac.uk
Dr. Daniel Page page@cs.bris.ac.uk

Web: http://www.cs.bris.ac.uk/Teaching/Resources/COMS20001

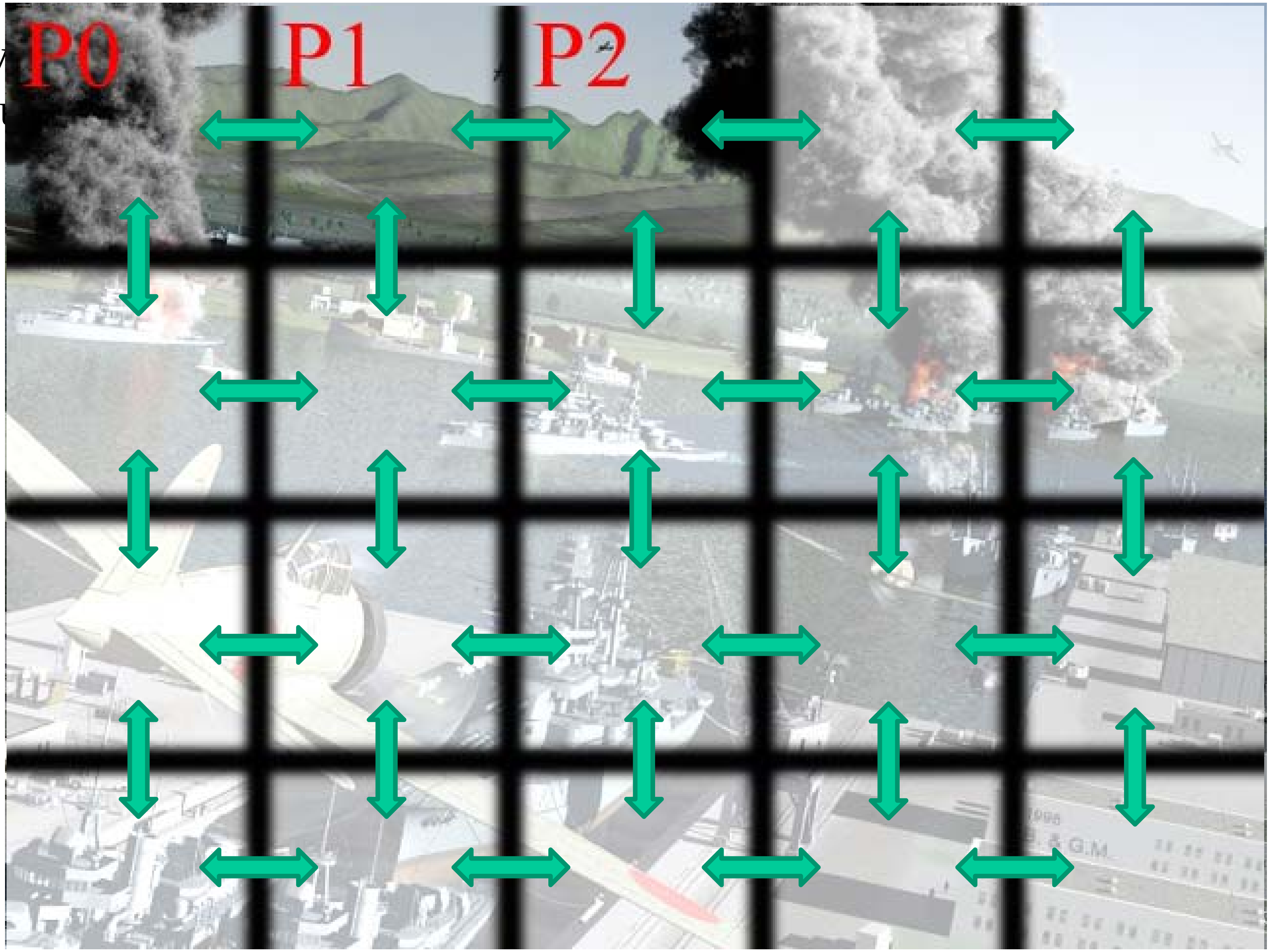
LECTURE MM2

DEADLOCK

CHANNEL COMMUNICATION

Key Concept: Explicit Process Communication

- Div
  req
  –



Key Concept: Explicit Process Communication

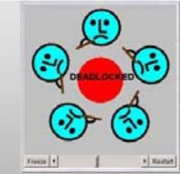
- Dividing computing tasks amongst several processes often requires these processes to **exchange information** during runtime
  - if not, tasks are known as *'embarrassingly parallelizable'*

There are two fundamental ways for explicit information exchange:

- **1) Shared Memory Model** *(…later in the course…)*
  - communication by altering the contents of shared memory locations
  - requires the application of some form of locking for synchronization

- **2) Message Passing Model** *(…our focus today…)*
  - communication by exchange of messages
  - tends to be easier to reason about than shared-memory concurrency

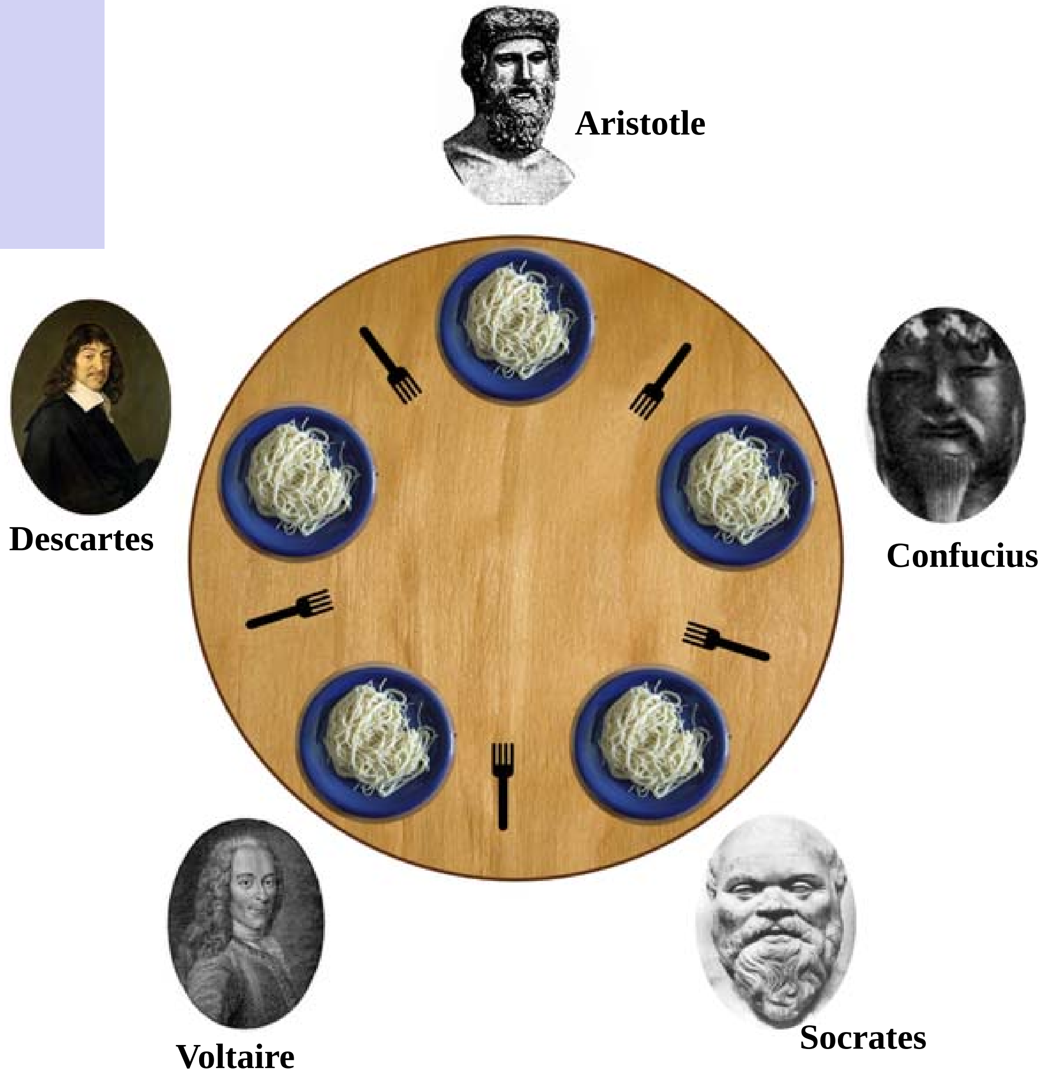
The Dining Philosophers



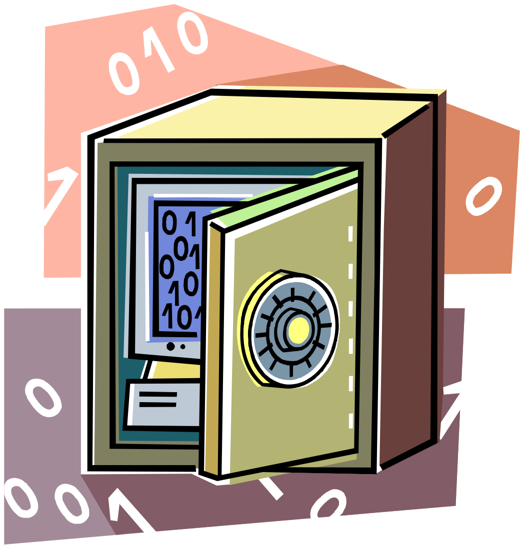
Five philosophers (with free will ☺)
are eating or thinking.
While eating, they are not thinking.
While thinking, they are not eating.
Each must have 2 forks to eat.

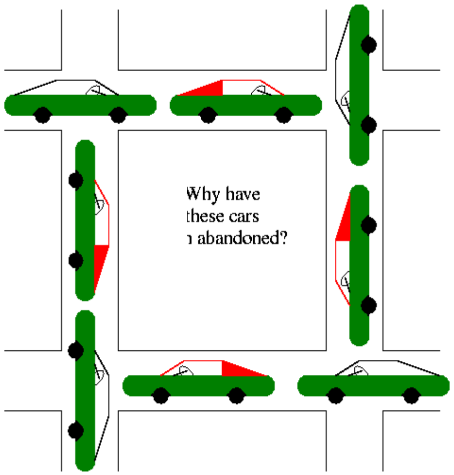
Possibility of **deadlock** in which every philosopher holds a left fork and waits perpetually for a right fork (or vice versa)

The lack of available forks is an analogy to the locking of shared resources in sequential computer programming
→ the system hangs…



Deadlock – Remember it!





Message Passing

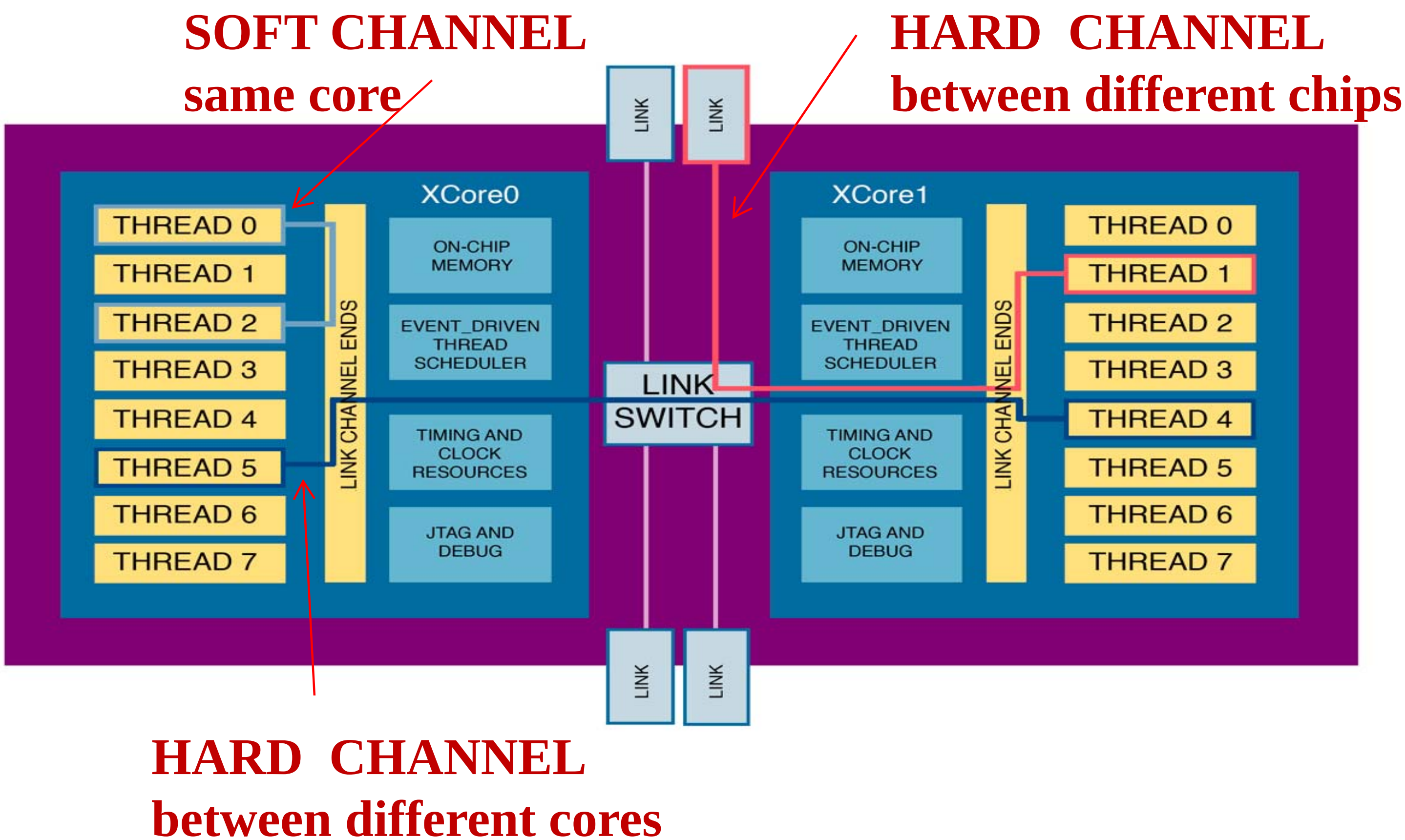
- **Message Passing Model between Concurrent Processes**
  - often set of processes have only local memory
    → shared memory unlikely!
  - processes communicate by **sending and receiving messages**
  - transfer of data between processes needs **cooperative operations** to be performed by each process (e.g. every send operation must have a matching receive)

Synchronous

Asynchronous

Soft Channels vs. Hard Channels

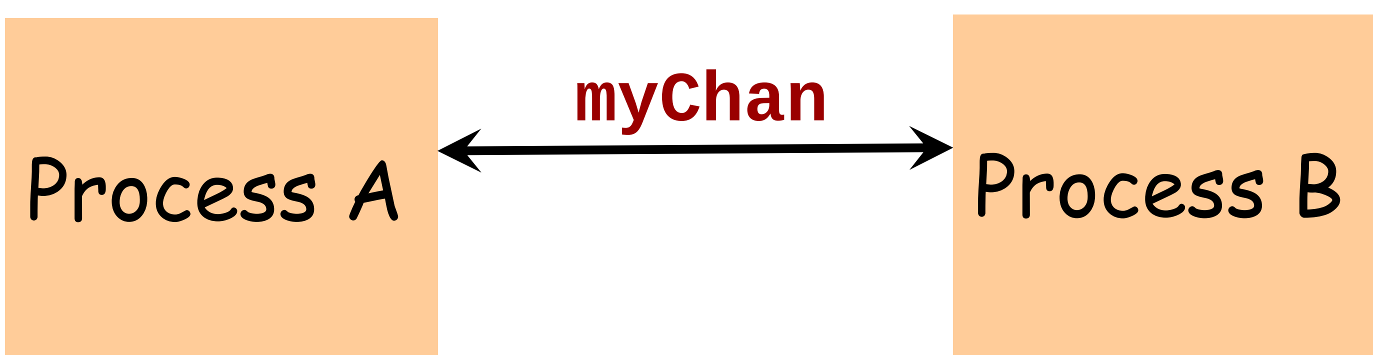
- **Channels** are an abstraction: they provide a **uniform representation** of communication irrespective of physical implementation of communication medium



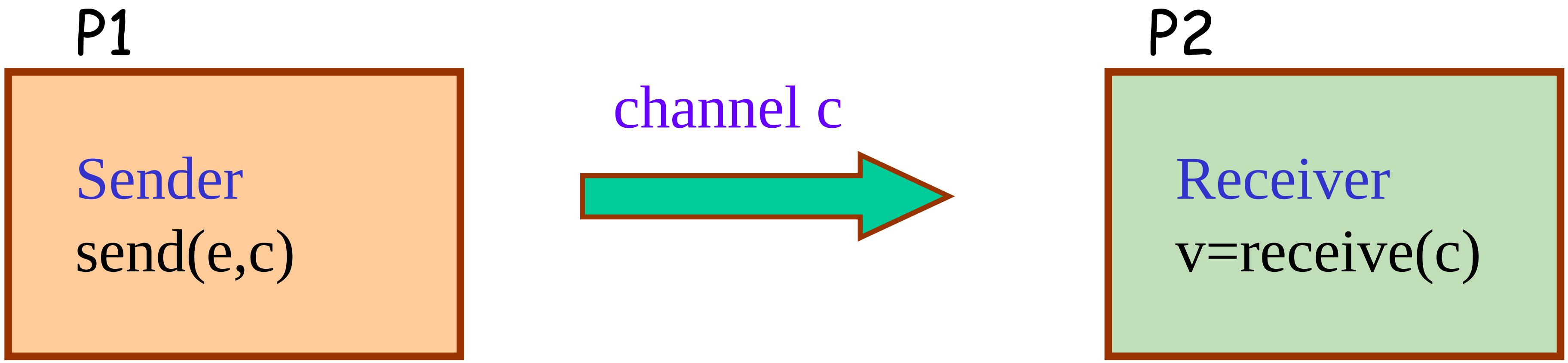
Dedicated Channels: *synchronised*

Channels are dedicated communication links between two processes.

- A ***synchronised*** operation:
  - an input process proceeds when the corresponding output process on the same channel is ready
  - an output process can only proceed when the corresponding input process on the same channel is ready
- The value communicated can be anything that can be assigned to a variable.



Concept: Synchronous Message Passing



send(e,c) - send the value of the expression e to channel c. The process calling the send operation is **blocked** until the message is received from the channel.

v = receive(c) - receive a value into local variable v from channel c. The process calling the receive operation is **blocked** waiting until a message is sent to the channel.

XC: Declaring a Synchronous Channel (**chan**)

```
#include <platform.h>

void myProcessA( chanend dataIn )  {}
void myProcessB( chanend dataOut ) {}

main ( void ) {
  chan c;
  par {
    on stdcore [0] : myProcessA(c); // Thread 1
    on stdcore [1] : myProcessB(c); // Thread 2
  }
}
```

...basic **channel declaration…**

NOTE:
In XC a channel is realized as a type of variable.

- A channel declaration provides a **synchronous**, **point-to-point** connection between two threads
- A channel is **lossless** (every data item sent is delivered), **exclusive** (to two threads) and **bidirectional** in XC

XC: Sending <: and Receiving :>

```
#include <platform.h>

void receive ( chanend dataIncoming ) {
  ...
  while (1) {
   dataIncoming :> data;
    printf ("Received %i\n", data);
  }
}
void send ( char data, chanend dataOutgoing ) {
while (1) {
   dataOutgoing <: data;
    printf ("Sent %i\n", data);
    data++ ;
  }
}
main ( void ) {
  chan c;
  char e = 1;
  par {
    on stdcore [0] : receive (c);    // Thread 1
    on stdcore [1] : send(e, c);     // Thread 2
  }
}
```

...**wait here** until a data item is available on the channel…

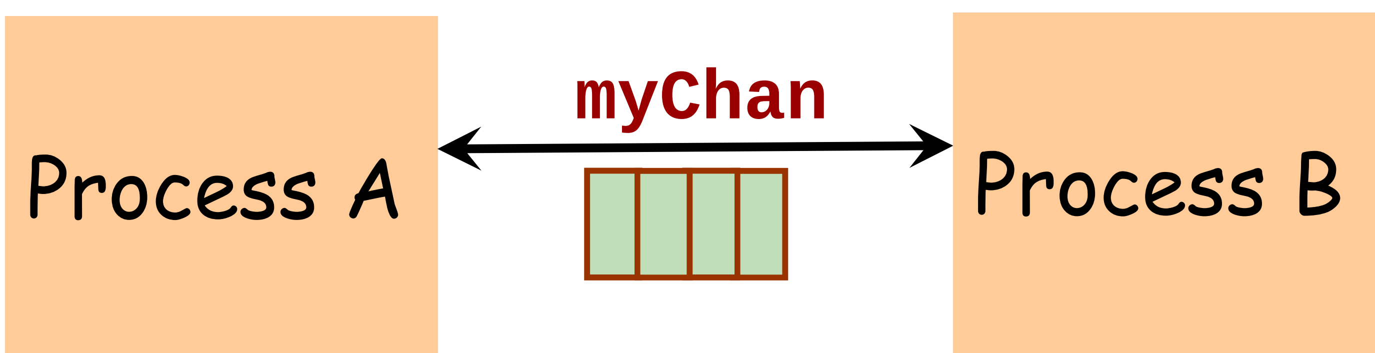
...**wait here** until data has been delivered to the other end of the channel…

Main program

Dedicated Channels: *asynchronised*

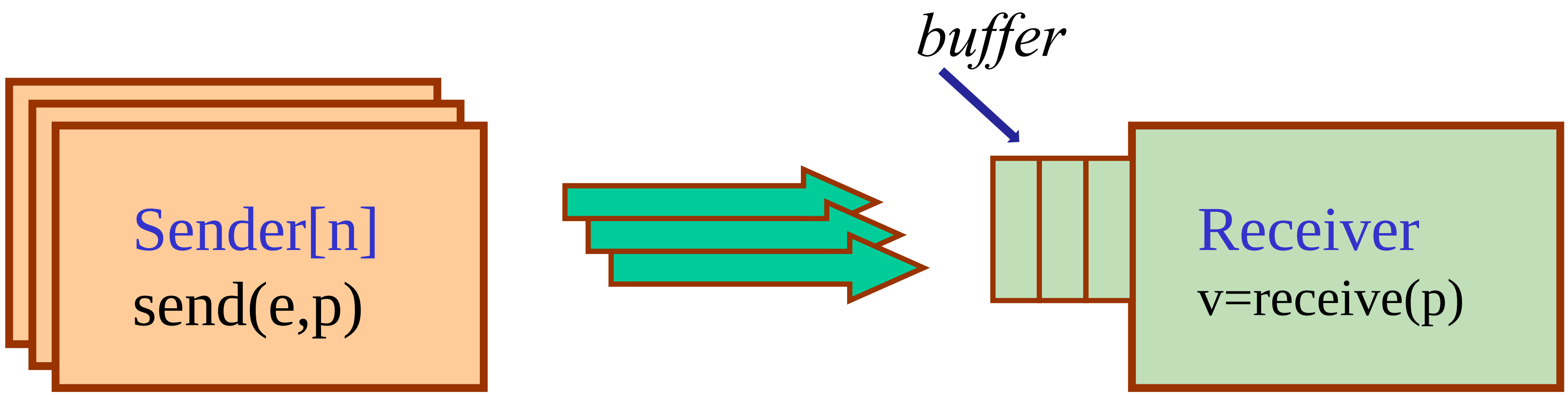
Channels are dedicated communication links between two processes.

- An a***synchronised*** communication:
  - operates via a buffer
  - an input process proceeds only when data is available for it on the channel (via a buffer)
  - an output process proceeds once it has placed data on its channel (unless the buffer has filled up!)
- The value communicated can be anything that can be assigned to a variable.



This buffer is hidden and inconspicuous to the XC programmer ☺

Concept: Asynchronous Message Passing



send(e,p) - send the value of the expression e to channel p. The process calling the send operation is **not** blocked. The message is queued at the port if the receiver is not waiting.

v = receive(p) - receive a value into local variable v from channel p. The process calling the receive operation is blocked if there are no messages queued to the port.

Asynchronous Comms in XC

Time taken to synchronise (including the time spent idle while blocking) can reduce overall performance.

In XC there are two fundamental ways to allow for **asynchronous** message passing:

- **1) Streaming**
  - ... data is channelled using a buffer at the receiver end

- **2) Transactions**
  - …**sequences of matching outputs and inputs** that are communicated over a channel asynchronously, with the entire transaction being synchronised at its beginning and end

The total amount of data output must equal the total amount input.

XC Streaming (Asynchronous Channel)

```
#include <platform.h>

main ( void ) {
  streaming chan c;
  par {
    on stdcore [0] : receive(c);   // Thread A
    on stdcore [1] : send(1,c);    // Thread B – sends a 1
  }
}
```

…declaring an asynchronous channel…

- streaming channels implement **asynchronous**, **point-to-point** connection providing the **fastest possible** data rates
- takes a **single instruction** for input/output statement
- data dispatched **immediately** as long as there is space in the channel's buffer, receiver blocks only if the buffer is empty

XC Example: Several Concurrent Streams

```
#include <platform.h>

on stdcore [0] : port lineIn = XS1_PORT_8A ;
on stdcore [2] : port spkOut = XS1_PORT_8A ;

main ( void ) {
  streaming chan s1 , s2;
  par {
    on stdcore [0] : audioAmp (lineIn , s1 );
    on stdcore [1] : audioFilter (s1 ,s2 );
    on stdcore [2] : audioSpk (spkOut , s2 );
  }
}
```

…two separate channels `s1` and `s2` declared…

…`s1` and `s2` are operating concurrently…

- **multiple streams** can be processed **physically in parallel**
- **limit to how many streams** can be declared together, since hard inter-core or inter-chip streaming channels require **capacity to be reserved in switches**

XC Transactions (Asynchronous Channel)

Two threads can engage in a ***transaction*** in which a sequence of matching outputs and inputs are communicated over a channel asynchronously.

1. The threads **synchronise upon entry** into the transaction.

2. A predefined sequence of values are then communicated asynchronously
   - sender thread blocks only if data can no longer be dispatched (due to the channel buffering being full)
   - receiver thread blocks only if there is no data available.

3. Finally, the threads **synchronise upon exiting** the transaction.

XC Transactions (Asynchronous Channel)

```
#include <platform.h>

int snd [64], rcv [64];

main ( void ) {
  chan c;
  par {
    on stdcore [0] : master {    // Sender Thread
      for (int i=0; i < 64; i ++)
        c <: snd[i];
    }
    on stdcore [1] : slave {    // Receiver Thread
      for (int i=0; i < 64; i ++)
        c :> rcv[i];
    }
  }
}
```

…master initiates communication process…

- The threads first synchronise upon entry to the master and slave blocks.
- 64 integer values are then communicated asynchronously.
- Finally, the threads synchronise upon exiting the master and slave blocks.

XC Transactions (Asynchronous Channel)

```
#include <platform.h>

int snd [64], rcv [64];

main ( void ) {
  chan c;
  par {
    on stdcore [0] : master {     // Sender Thread
      for (int i=0; i < 64; i ++) {
        c <: snd[i];
        do_some_other_processes(snd[i]);
      }
    }
    on stdcore [1] : slave {      // Receiver Thread
      for (int i=0; i < 64; i ++) {
        c :> rcv[i];
        do_other_processes_too(rcv[i]);
      }
    }
  }
}
```

…master initiates communication process…

XC Transactions

- A transaction consists of a *master thread and a slave thread running concurrently.*

master and ***slave*** are keywords and compulsory!

- The threads first synchronise upon entry to the master and slave blocks.

- A predefined sequence of values are then communicated asynchronously
  - sender thread blocks only if data can no longer be dispatched (due to the channel buffering being full)
  - receiver thread blocks only if there is no data available.

- Finally, the threads synchronise upon exiting the master AND slave blocks.

XC: Channel Disjointness Rules

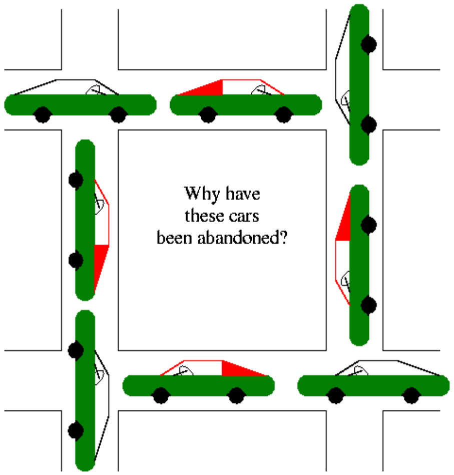
The rules for disjointness on a set of threads $T_0 \ldots T_i$ and a set of channels $C_0 \ldots C_j$:

- ❖ If threads T_x and T_y contain a use of channel C_m then none of the other threads (T_t ; $t \neq x;y$) are allowed to use C_m.
- ❖ If thread T_x contains a use of channel end C_m then none of the other threads (T_t ; $t \neq x$) are allowed to use C_m.

- each channel can be used in at **most two** threads
- if a channel is used in only one thread then attempting to input or output on the channel **will block forever**
- disjointness rules for channels (and variables) guarantee that any two threads can be run concurrently on any two processors, subject to a physical route existing between the processors.

Example: Communication Deadlock

```
...
  chan c1, c2;
  par {
    on stdcore [0] : {
      char x;
      c2 :> x;
      c1 <: 1;
    }
    on stdcore [1] : {
      char y;
      c1 :> y;
      c2 <: 2;
    }
  }
...
```



Why is there a deadlock and how can it be resolved?

Take care to sequence programs so that processes don't wait (too long) for communication with each other.

Message Passing in XC – Summary

Message Passing Models in XC

Synchronous

Asynchronous

- **streaming**: asynchronous, point-to-point connection
- **Transaction**: a *master* thread and a *slave* thread running concurrently. Synchronous at beginning and end, asynchronous in between