Department of Computer Science
University of Bristol

COMS20001 - Concurrent Computing

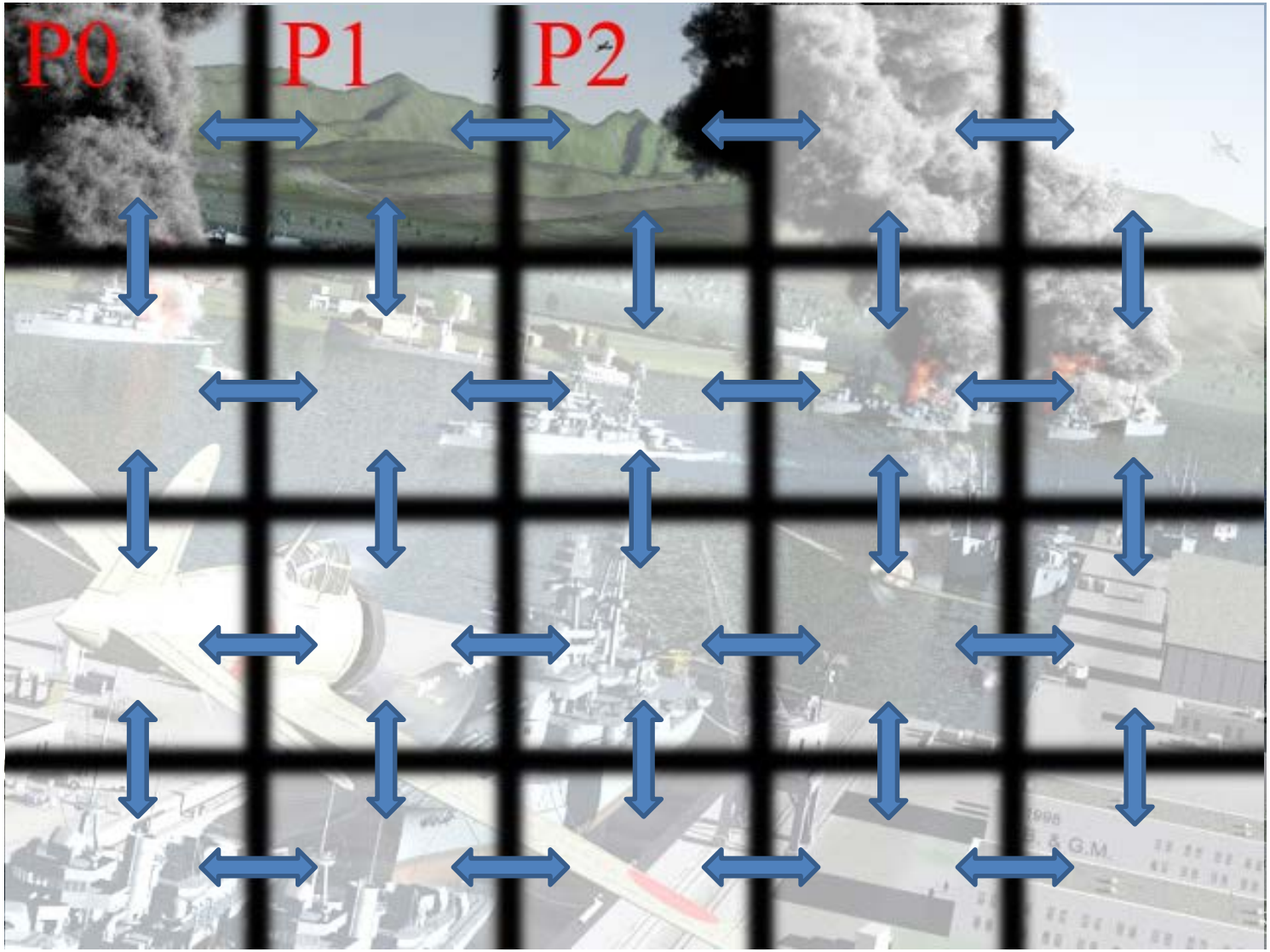
www.cs.bris.ac.uk/Teaching/Resources/COMS20001

Lecture 03

Deadlock & Channel Communication

Sion Hannuna | sh1670@bristol.ac.uk
Dan Page | daniel.page@bristol.ac.uk

Key Concept: Explicit Process Communication



Key Concept: Explicit Process Communication

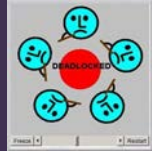
- Dividing computing tasks amongst several processes often requires these processes to **exchange information** during runtime
  - if not, tasks are known as *'embarrassingly parallelizable'*

There are two fundamental ways for explicit information exchange:

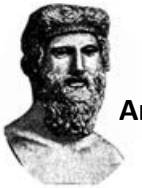
- **1) Shared Memory Model** *(…later in the course…)*
  - communication by altering the contents of shared memory locations
  - requires the application of some form of locking for synchronization

- **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

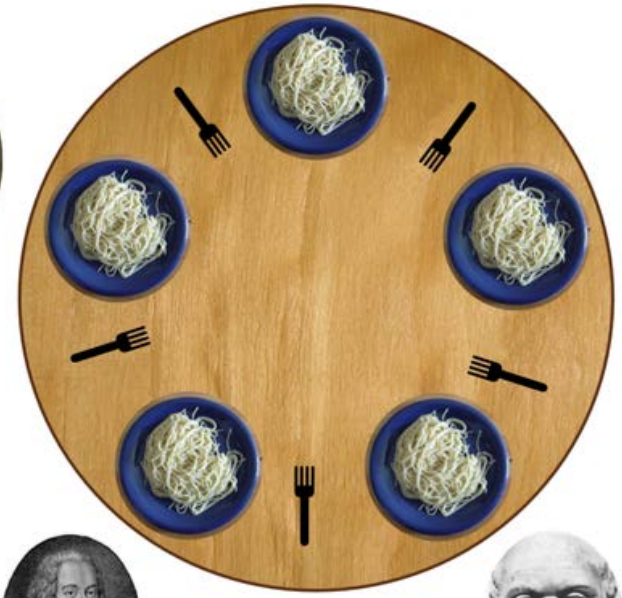


Five philosophers (with free will ☺) are eating or thinking.
While eating, they are not thinking.
While thinking, they are not eating.
Each must pick up 2 forks one after the other to eat.


Aristotle

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


Descartes




Confucius

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

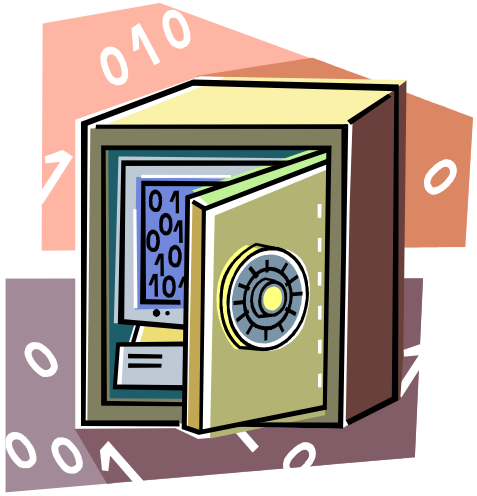

Voltaire

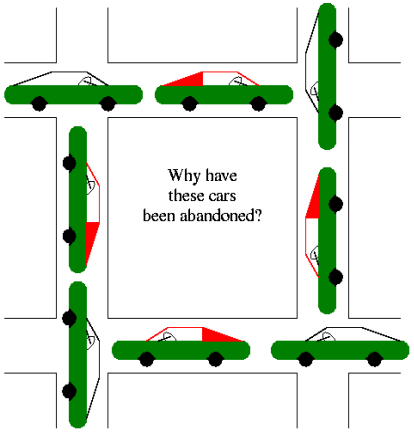

Socrates


WARNING!
The system is either busy or has become unstable. You can wait and see if the system becomes available again and continue working or you can restart your computer.
* Press any key to return to Windows and wait.
* Press CTRL+ALT+DEL again to restart your computer. You will lose any unsaved information in all applications.
Press any key to continue _

Deadlock – Remember it!

https://youtu.be/NeC1Ueqzfes?t=18s



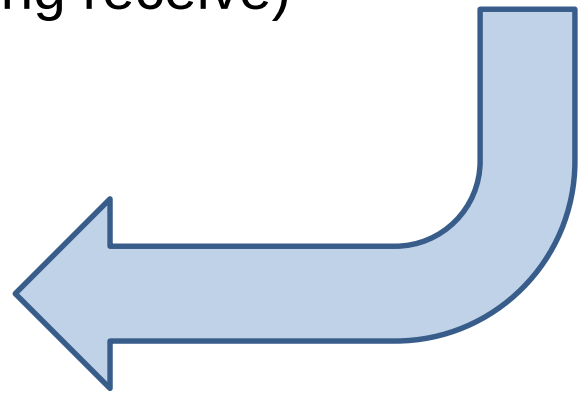


Message Passing

- **Message Passing Model between Concurrent Processes**
  - Our set of processes may have only local memory
    → shared memory not explicitly supported!
  - 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)

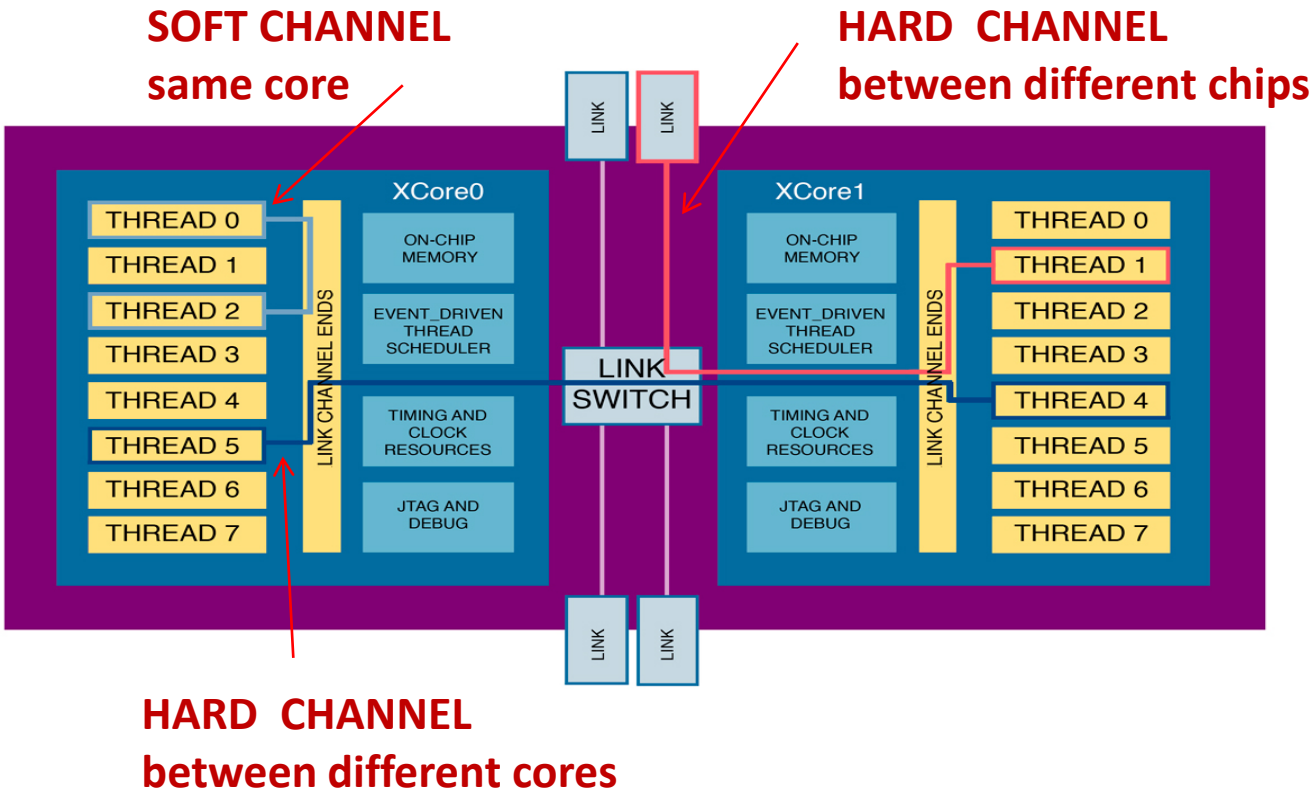
Synchronous

Asynchronous



Soft Channels vs. Hard Channels

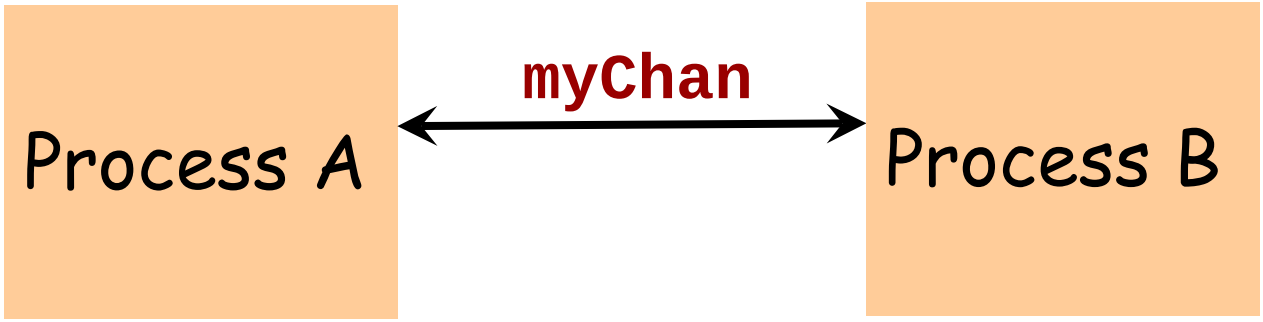
- **Channels** are an abstraction: they provide a **uniform representation** of communication irrespective of physical implementation of communication medium



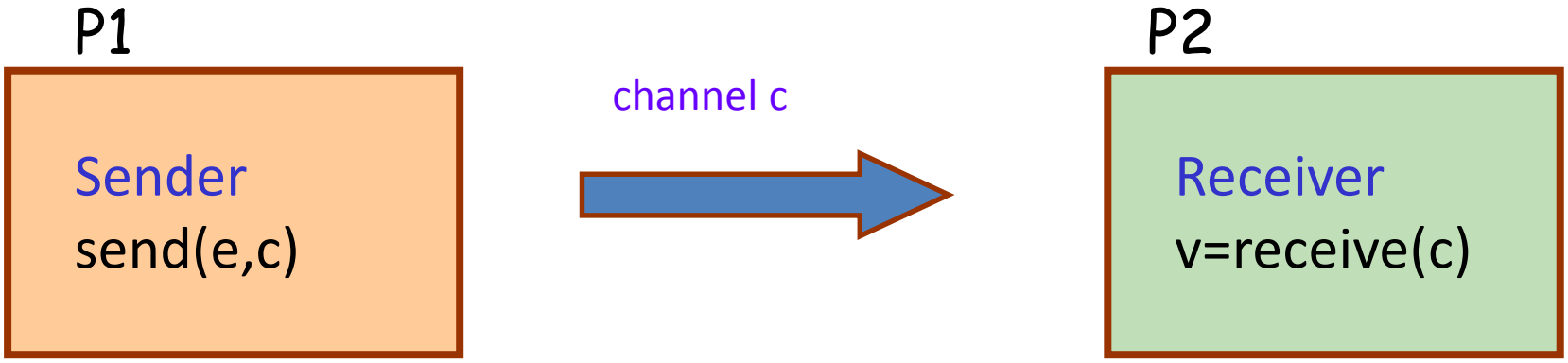
Dedicated Channels: *synchronised*

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



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**)

channel.xc

```
#include <platform.h>

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

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

...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 exactly two threads
- A channel is **lossless** (every data item sent is delivered), **exclusive** (to two threads) and **bidirectional** in XC

XC: Sending <: and Receiving :>

sendreceive.xc

```
#include <stdio.h>
#include <platform.h>

void receive ( chanend dataIncoming ) {
  char data;
  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++ ;
} }

int main ( void ) {
  chan c;
  par {
    on tile[0] : receive (c);     // Thread 1
    on tile[1] : send(1, c);      // Thread 2
  }
  return 0;
}
```

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

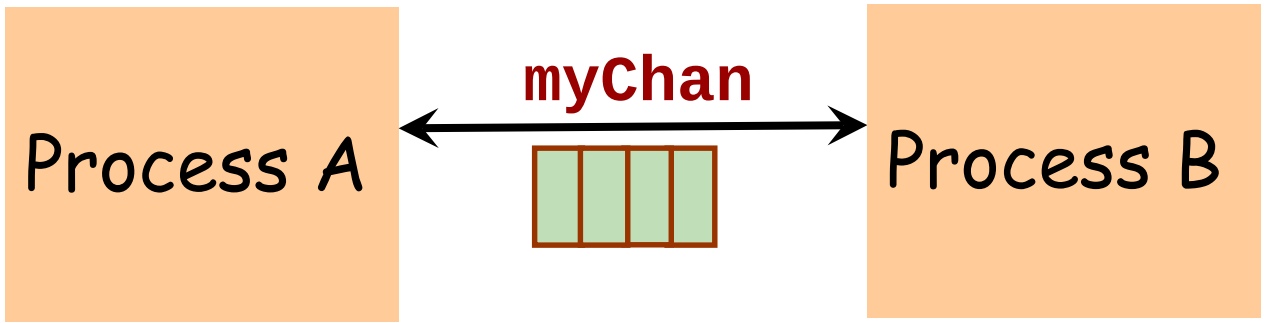
…**wait here** until data has been delivered to the other end of the channel…

Main program

Dedicated Channels: *asynchronous*

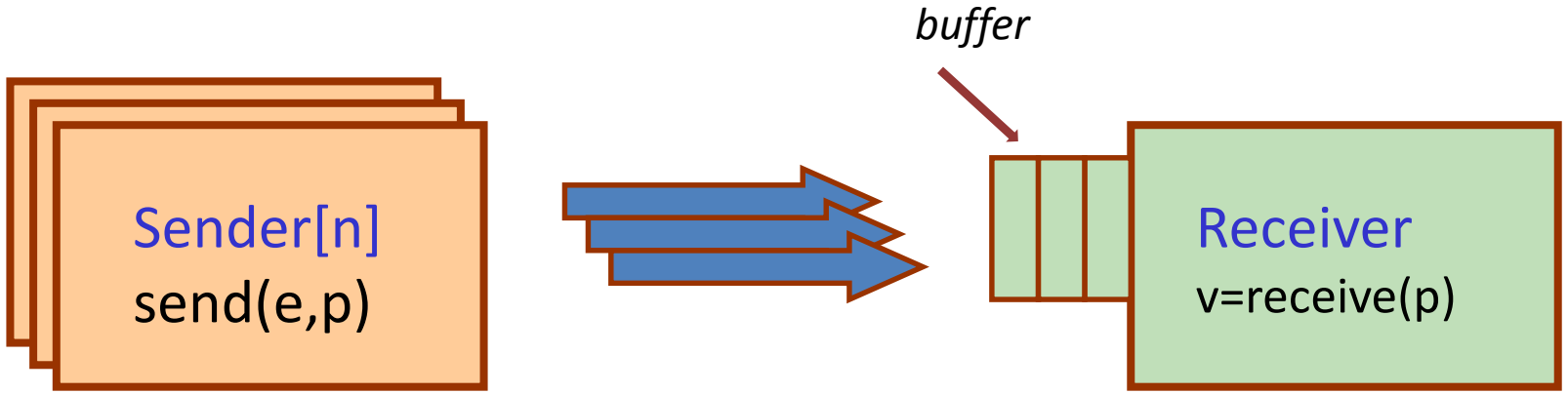
Channels are dedicated communication links between two processes.

- 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!)



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 Communication in XC

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 exactly the total amount input.

XC Streaming (Asynchronous Channel)

```
#include <platform.h>
...
int main ( void ) {
  streaming chan c;
  par {
    on tile[0] : receive(c);  // Thread A
    on tile[1] : send(1,c);   // Thread B – sends a 1
  }
 return 0;
}
```

…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>
...

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

…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)

transaction.xc

```
#include <platform.h>

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

…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.

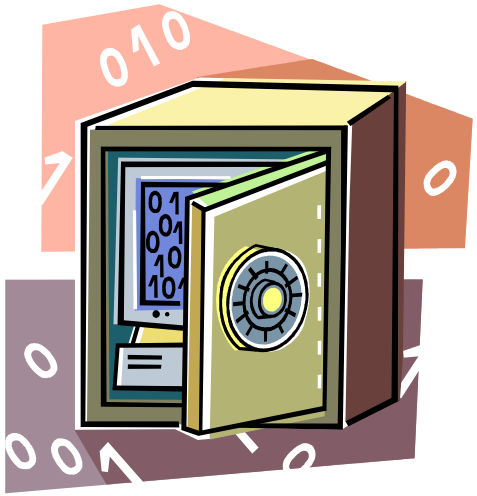
XC: Channel Disjointness Rules

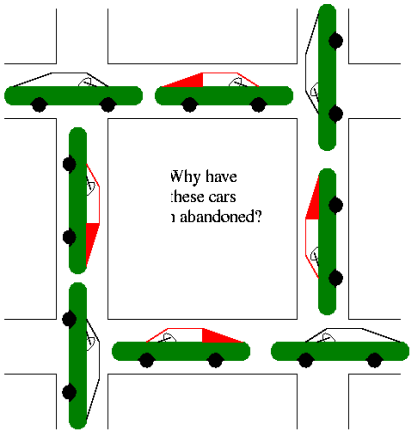
The rules for disjointness on a set of threads $T_0 \ldots T_i$ and a set of channels $C_0 \ldots C_j$:

- 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.

- So 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 or raise an ILLEGAL_RESOURCE exception**
- 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.

Deadlock – Remember it!

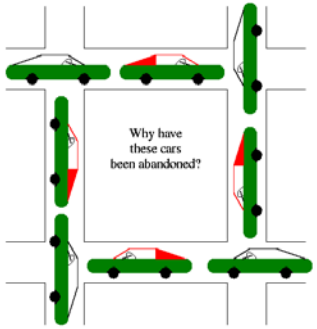




Example: Communication Deadlock

deadlock.xc

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

Let’s practice…

Write an XC fragment of code which has two parallel processes:

- In one, the user process, a button is pressed to request a lift.
- In the other, the lift management process, it receives the signal and sends an instruction to the lift motor to take it to the floor.