

Course Web Page

New Students?

1. From Department Home Page:

<http://www.sce.carleton.ca>

2. Pick **Course Materials** (on left)

3. Pick SYSC 5701

<http://www.sce.carleton.ca/dept/sce.php/courses/sysc-5701>

protected content: user: **sysc-5701**

password: **rtos**

SYSC 5701

Operating System Methods for Real-Time Applications

Motivation

Winter 2014

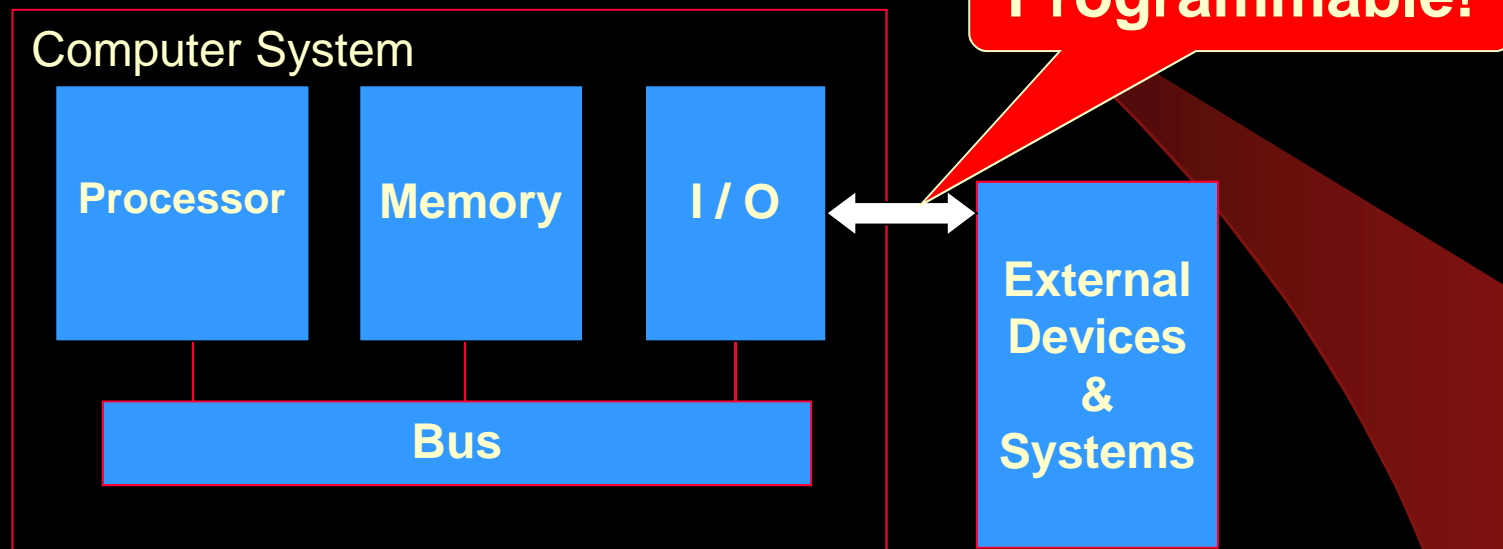
Broad Background

- systems concepts, computer systems
- time
- software engineering: development, design
- concurrency
- interrupts

System

- a set of components that interact to accomplish an objective
- can be applied to just about anything! 😊

Uniprocessor Computer System



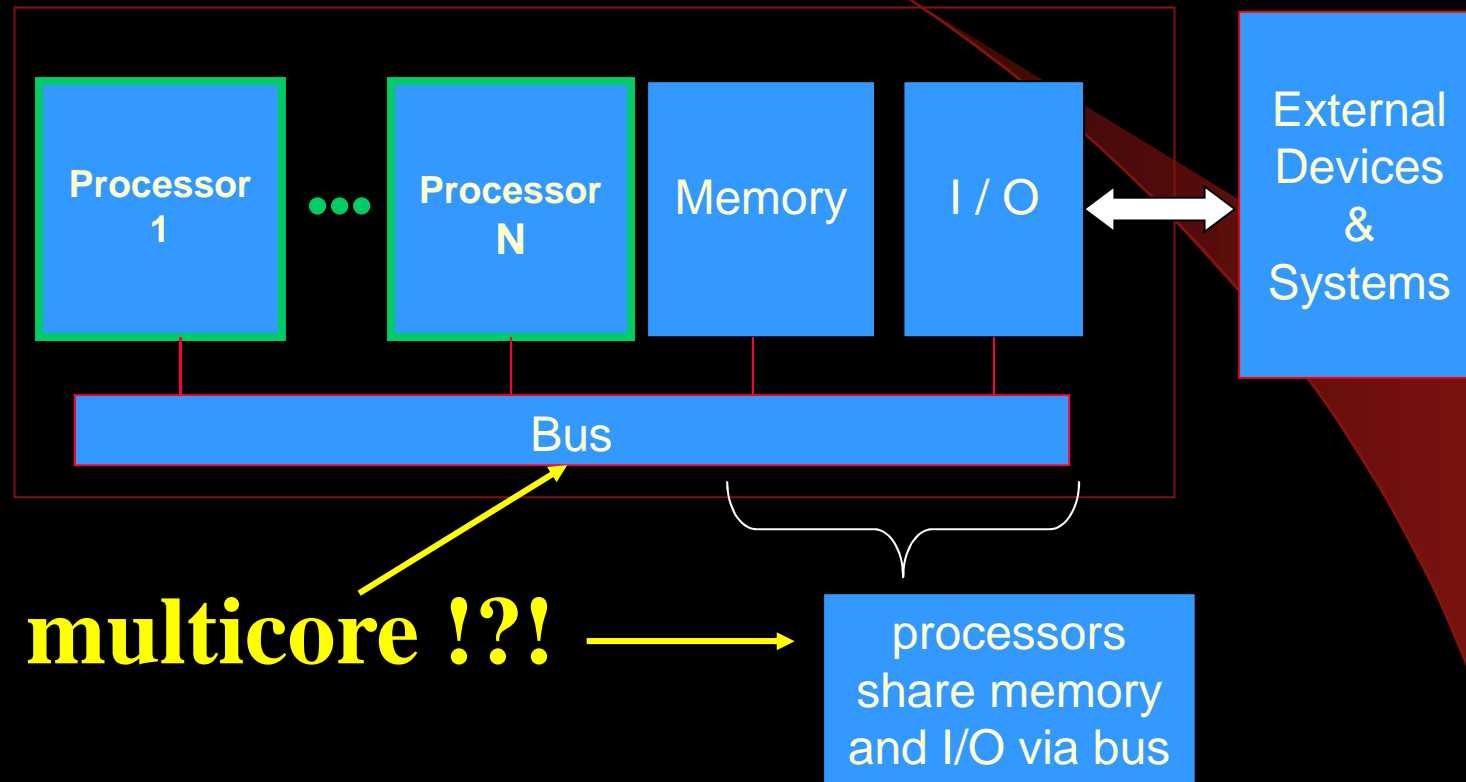
- **Objective:** involves maintaining input/output relationships at the I/O / External interface

Variations: Multiprocessor

- more than one processor → shared bus
- processors share global resources
- a processor may also have private local resources connected via a secondary (private) bus structure
(not shown below)

multicore !!

Multiprocessor (con't)



multicore !?!

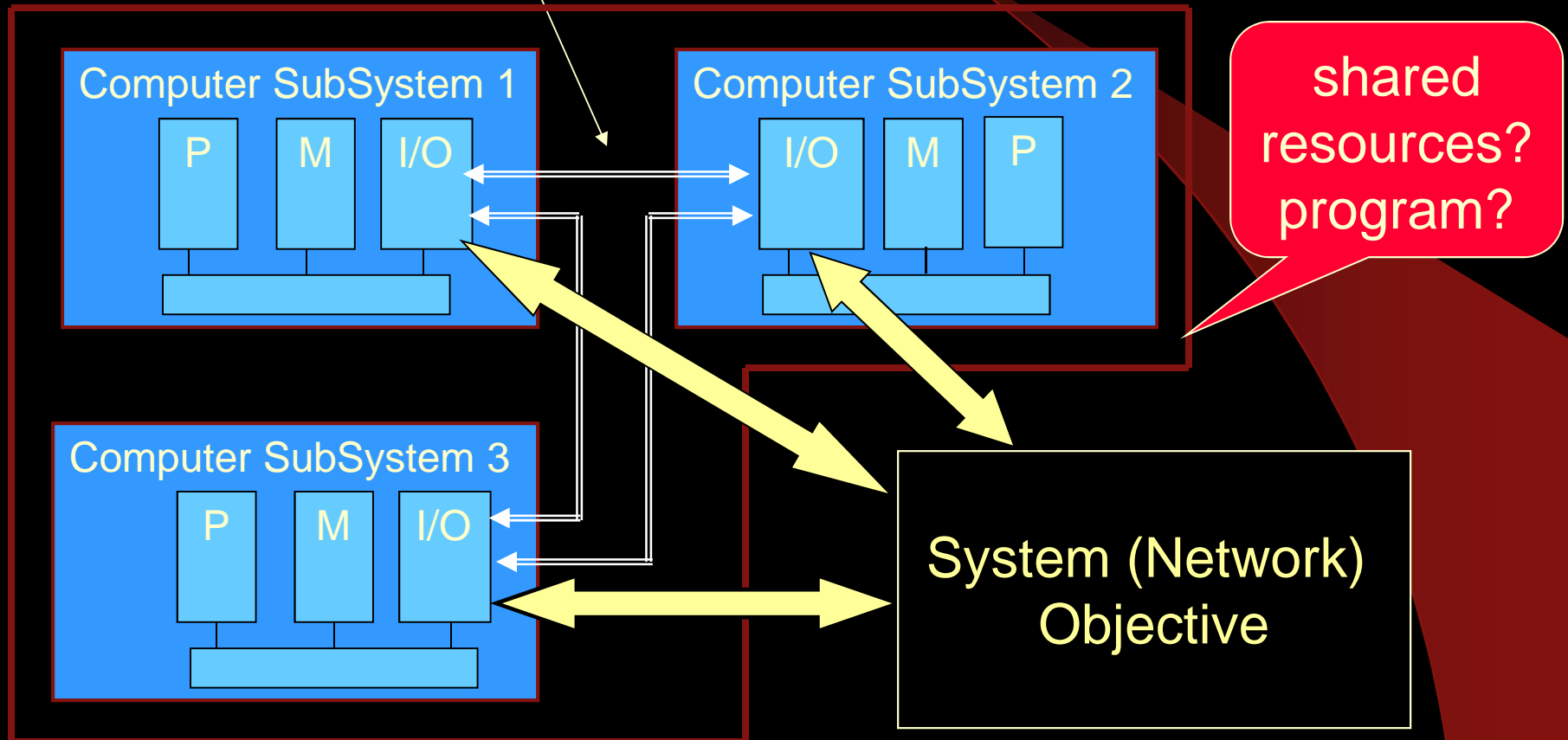
processors
share memory
and I/O via bus

Variations: Network

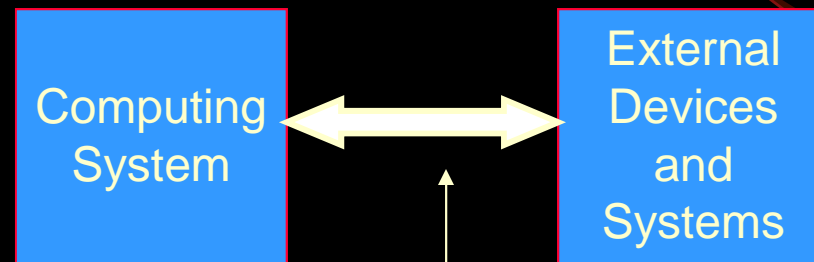
- computer subsystems interconnected **via I/O components**
- subsystems do not share resources via shared bus
- sharing a resource is more complicated!
 - requires co-operation of subsystems
- subsystems co-operate to accomplish network-wide objective

Network (con't)

subsystems must communicate to co-operate



Real-Time Systems



Objective:

- maintain time-constrained input/output relationships between computing system and external devices/systems
- How should these be **described**?

(Typical) Hard vs. Soft Real-Time

- **Hard Real-Time**

- failure to meet time constraints is catastrophic
- recovery may be difficult, or futile
- e.g. reactor melt-down, plane crash, loss of life



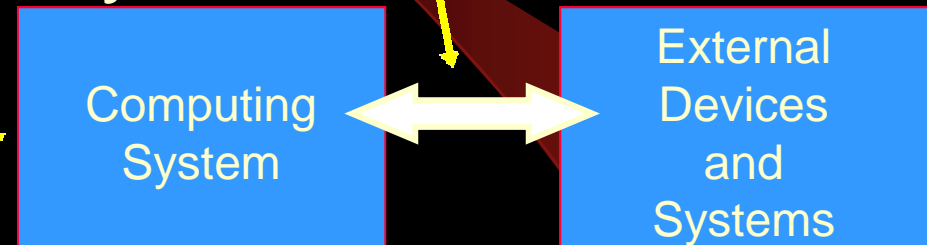
Safety
Critical

- **Soft Real-Time**

- occasional failure to meet time constraints is inconvenient but not catastrophic
- try again, or be patient
- e.g. no dial tone, lost voice packet

Describing Systems

- **Requirements:** specify the objectives in terms of behaviour at the interface to the external devices/systems

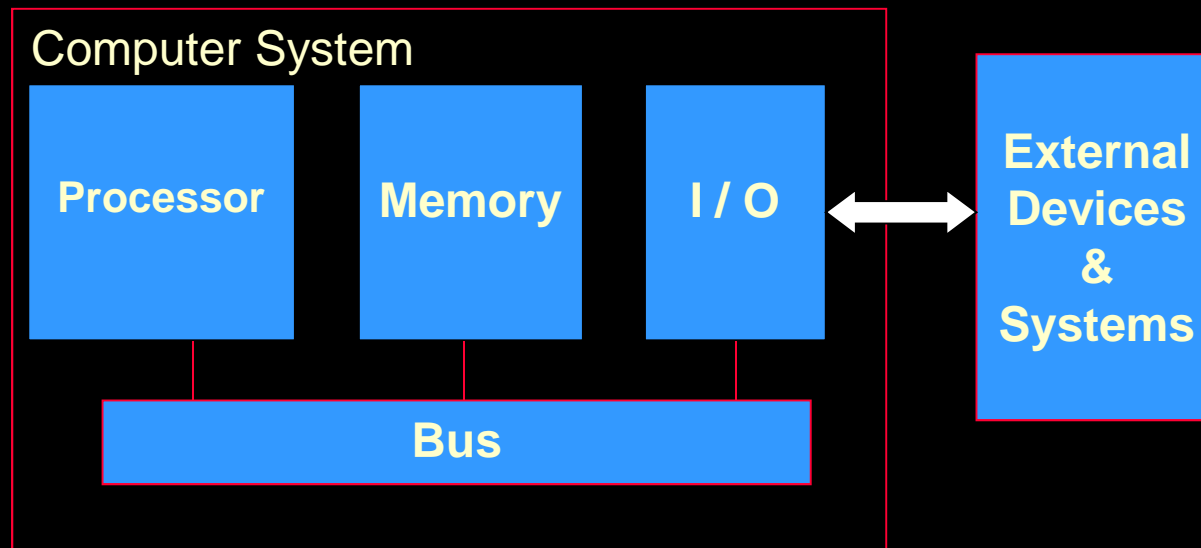


- **Implementation:** describes how the computing system is utilized to meet the requirements
- Why is it useful to describe both? What is a system “**design**”?

Requirements vs. Implementation

- “**Ideally**”: the requirements are independent of the implementation

→ **Abstraction** ← engineering

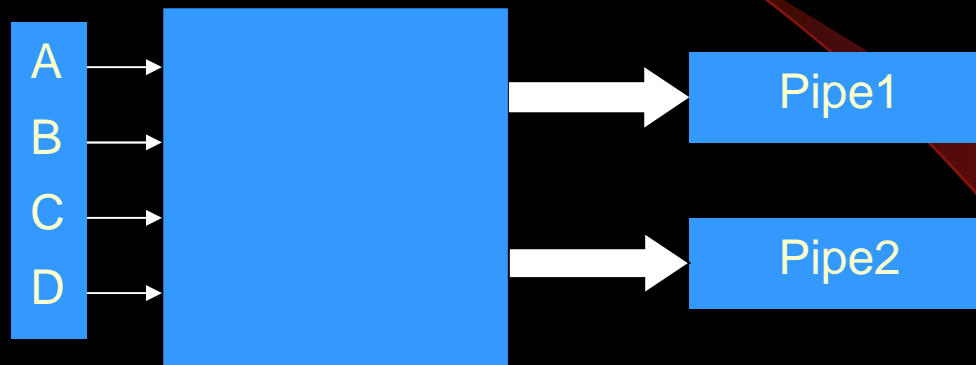


Concurrent Activities

- are in progress at the same time
- **dependent activities**: interact to complete a higher objective
- **independent activities**: do not interact

May have concurrency in the requirements behaviour and in the implementation

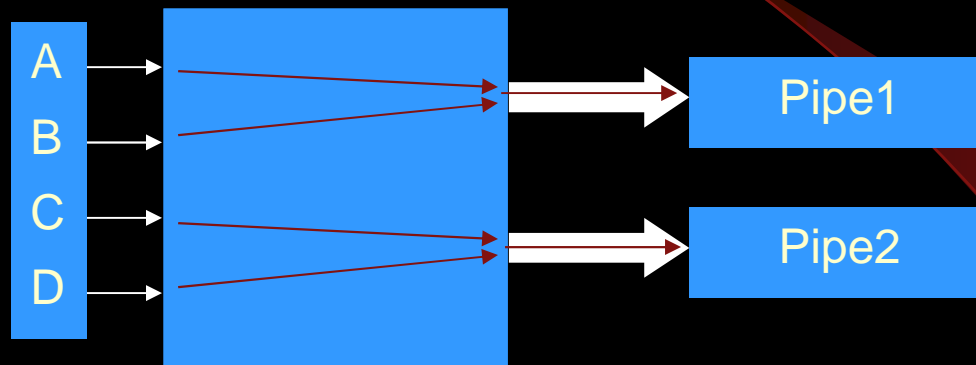
Stream-2-Pipe Example



concurrent activities at interface:

- input (slow) data streams: A, B, C, D
- output (fast) data pipes: Pipe1, Pipe2

Example (con't)



- streams A and B are compressed/multiplexed into stream Pipe1
- streams C and D are compressed/multiplexed into stream Pipe2

Example (con't)

Concurrency at **requirements** level:

- A , B & Pipe1 are **dependent** activities
- C , D & Pipe2 are **dependent** activities
- { A , B , Pipe1 } activities are **independent** of { C , D , Pipe2 } activities

Concurrency in **implementation**?

How might the system be implemented?

Concurrency in Physical Implementations

- **real** concurrency: active h/w components that operate in parallel to support concurrent activities
 - e.g. processors, active I/O components
- **apparent** concurrency: active devices are shared to give the impression (over time) that external activities are being carried out concurrently

Important Distinction!

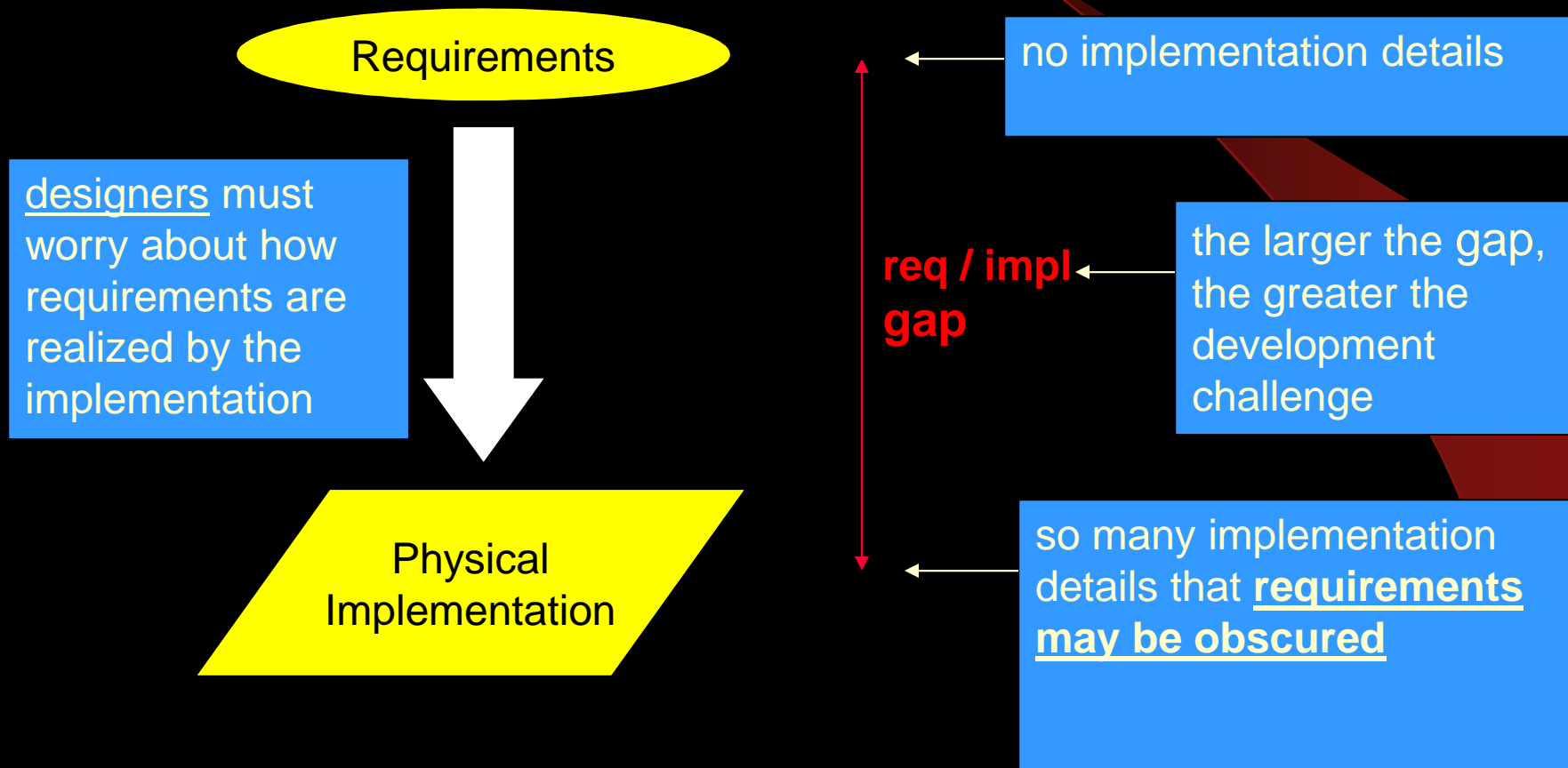
- concurrency in requirements is part of the objective
 - **cannot** be altered by design decisions
- concurrency in implementation is a design decision
 - **not imposed** by requirements

As a result: Concurrent activities in requirements are often at a different granularity than concurrent activities in implementation.

Design for Concurrency

- mapping concurrency in requirements onto implementation resources is a design decision
 - **goal**: allocation of system (implementation) resources to achieve concurrency in requirements
- many tough design issues here!
(more later!)

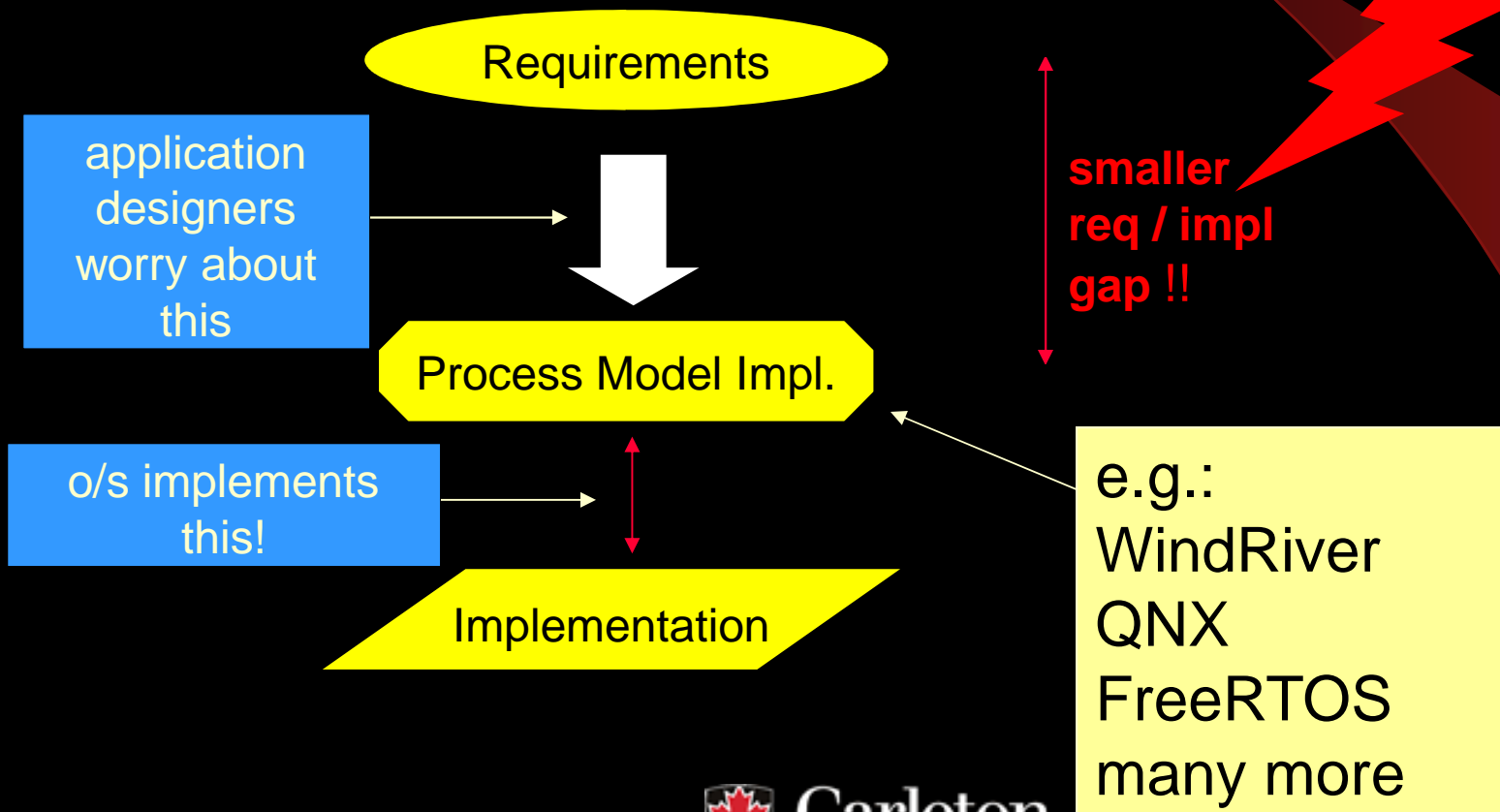
Development Problem: requirements/implementation gap



To reduce/manage the requirements/implementation gap:

- introduce an **intermediate level** between requirements and implementation
 - resides “above” implementation
- virtual machine: deals with **concurrency** explicitly!
- introduce an abstract **process model**
- design implementation in terms of the process model
- **operating system** provides process model support

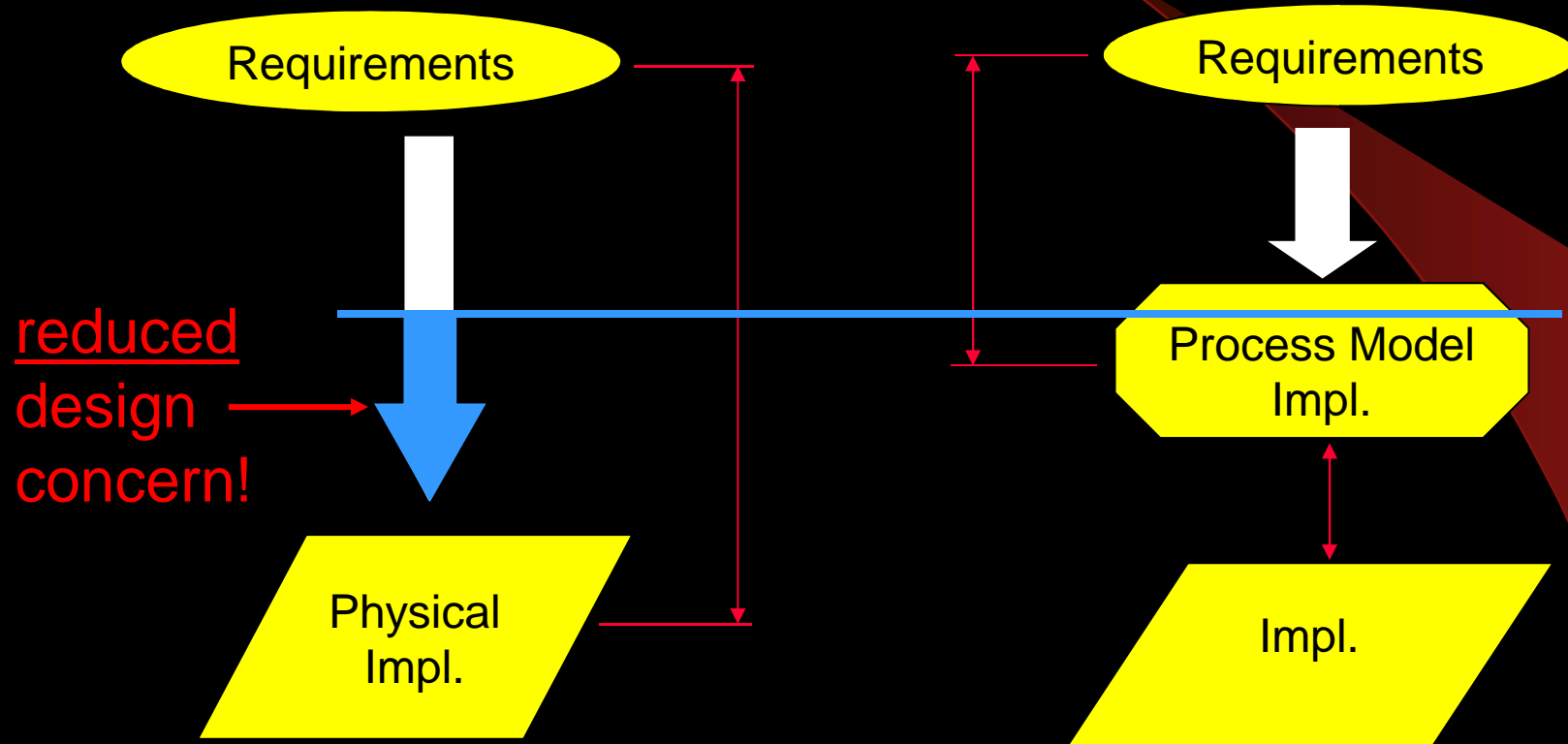
Modified Development Problem: **reduced** **requirements/implementation gap**



Before

&

After



What SYSC 5701 Is

- concerned with using a process model to help reduce the development challenges for real-time applications
- primary concern: **designer's perspective!**
- Goals:
 - simplify the implementation of concurrency
 - hide some machine details
 - use “standard” process model
 - simplify the mapping of concurrency in requirements onto concurrency in implementation

What SYSC 5701 Is Not

- **NOT** concerned with particular real-time applications
- **NOT** about Linux or Windows

So . . . what's so hard about concurrency? 😊

- event-driven vs. sequential mindset
- interference – shared resources
- synchronization – mutual exclusion, coordinate progress
- communication among concurrent activities
 - for application purposes & synchronization

Will elaborate on these in the rest of these slides

Sequential Mindset

- control is managed sequentially
 - **only one thread of control**
- hardware/state is polled to decide when to perform work
- response to events depends on when event sources are polled

Sequential Mindset: Polling

General form of polling-only implementation:

```
loop (forever)
{
    poll for next event/work to do
    process events/work as needed
}
```

Polling & Priority

- for polled events, can often give work relative priorities
- e.g. poll all devices and decide on processing order
- **higher-priority work**: performed a.s.a.p.
 - e.g. service I/O hardware
- **lower-priority work**: after higher-priority work

Timing Example:

- suppose a **h/w timer** is being used to implement a displayed clock
- h/w timer **“tick”** every millisecond
 - can poll for tick
- update **display clock** every second

Polling Approach

```
poll h/w timer
if ( tick )
{ count++;
  if ( count == 1000 )
    { count = 0; }
  update display;
}
```


Priority in Timer Example

- manipulating count is **higher-priority** processing
- failure to sense every tick = lost time !
- must poll "often enough" to sense all ticks

- update clock display is **lower-priority** processing
- could be delayed "a bit" in favour of higher-priority processing

Event-Driven Mindset: H/W Interrupts

- high-priority processing performed by h/w Interrupt **S**ervice **R**outines (**ISRs**)
- h/w generates interrupt (signal) when event occurs
 - e.g. h/w timer tick
- signal causes processor to execute ISR
 - no s/w involved in invocation of ISR!

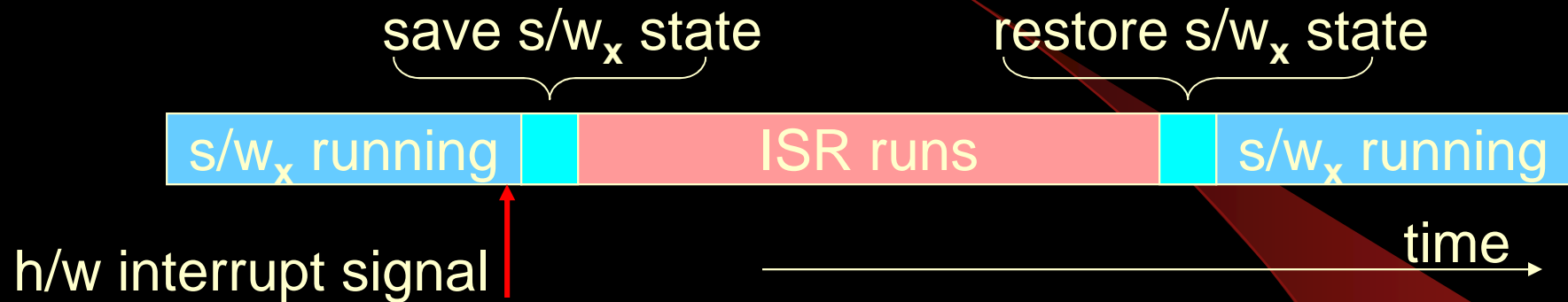
If you don't recall about **interrupts** – be sure to read about them in any microprocessor system text! See doc link on webpage.

ISR Related Control Flow

1. current s/w **state** is saved on stack
(registers: including status (e.g. flags) and program counter)
→ the current software is suspended!
(**interrupted!** pre-empted!)
2. **ISR runs**
3. prior **state** (1) is restored and s/w continues

If you don't recall about **interrupts** – be sure to read about them in any microprocessor system text! See doc link on webpage.

Interrupt & ISR



- Similar to a h/w invoked function call
- **NO** s/w involved in invocation!!
- interrupted s/w (s/w_x) does not “know” it was momentarily suspended or that the ISR executed! (i.e. that s/w_x was pre-empted)

Event-Driven Mindset: Interrupts & Concurrency

- processor is shared between the **threads of control** associated with **ISRs** and the sequential thread of the **main** program
 - shared processor = virtual concurrency
- h/w interrupts are **asynchronous**
 - the result of the actions of active hardware devices
- ISRs run due to h/w event handling, not due to sequential s/w sensing of events!

To use Interrupt-Driven Approach:

- place **high-priority** processing in **ISRs**
- place **low-priority** processing in main (sequential) program
- ISRs and main must **communicate**
- main requests that high-priority work to be performed by ISRs
- ISRs inform main of completed work
- communicate using **shared variables**

Recall Previous Timer Example

poll h/w timer

```
if ( tick )  
{ count++;
```

high priority work

← put in **ISR**
(no poll!)

```
if ( count == 1000 )  
{ count = 0; }
```

← where to put ?
share count ?

```
update display
```

low priority work

← put in **main**

```
}
```

Timer Example Revised

Suppose ISR and main **share**: boolean **SECOND**

- in ISR: `count ++;`

count is not shared

```
if ( count == 1000 )
{
    count = 0;
    SECOND = TRUE; }

```

- in main:

```
poll:
if ( SECOND )
{
    SECOND = FALSE;
    update display }

```

shared variable
initial value =
FALSE

Recall (Half of) Stream-2-Pipe Example:




- suppose streams and pipe are services by h/w ISRs:
 - **ISRA** – receives a **Data packet** of Stream A data
 - **ISRB** – receives a **Data packet** of Stream B data
 - **ISRP** – transmits **Pipe packets**

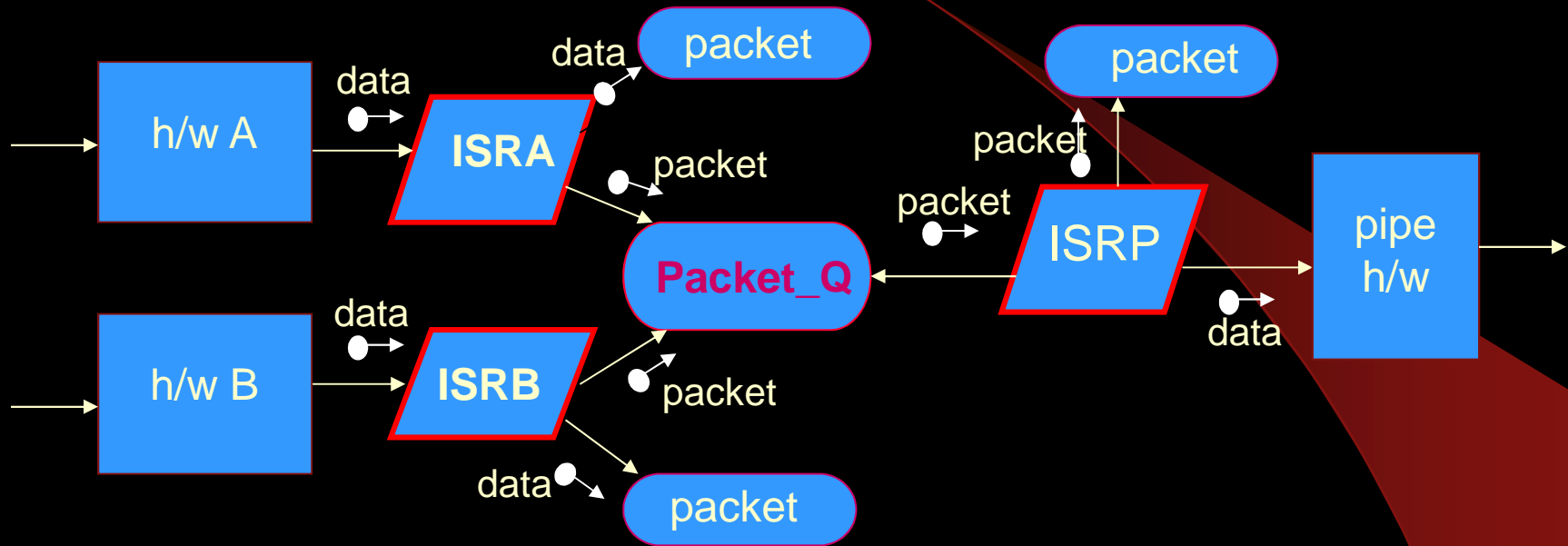
Stream-2-Pipe Communication

- ISRs **share** a queue (**Packet_Q**) to exchange packets
- **ISRA** and **ISRB** produce packets as they are received
- when packet of data is received it is put in **Packet_Q**
- **ISRP** consumes packets by transmitting them
- when ISRP is idle, it gets a packet from **Packet_Q**
- instance of classical producer/consumer problem

used widely to illustrate
operating system issues



Stream-2-Pipe: Pictorial Representation



Delays?

Issues to Expose: **SYNCHRONIZATION**

- among concurrent activities
- e.g. transmit on pipe cannot proceed without data from streams
 - pipe transmission must **wait** for work
- frequent requirement in concurrency!

Issues to Expose: Buffer Management

- how do ISRA and ISRB obtain empty packet buffers for receiving packets?
- what does ISRP do with an empty packet buffer after transmitting a packet?
- static vs. dynamic schemes?
- what happens if no buffers/memory available?

Issues to Expose: **INTERFERENCE**

Potential for INTERFERENCE:

- concurrent activities share **Packet_Q**

INTERFERENCE occurs when
simultaneous concurrent activities
corrupts a shared resource

- modification is concurrent with “other”
access

Critical Sections

- a region of code that has the potential to cause interference is called a **critical section**
- the **existence** of a critical section does not guarantee interference – often depends on specific access sequences and timing
- interference may not show up in testing !
– hard to debug!

Example: consider a static array implementation of **Packet_Q**

circular Q: (data structure)

- **Head** and **Tail** pointers (indices)
- remove @ **Head**
- **Tail** points to next available array element
- when reach end of array, wrap to start:
$$\text{index} = (\text{index} + 1) \bmod \text{Q_size}$$

Data Declarations

Q_Size = ***** ; // some constant

Packet_Q :

array [**0** .. **Q_Size - 1**] of **packet_buffer** ;

Head : integer ; // index of packet to remove

Tail : integer ; // index of next free array element

Count : integer ; // # of packets in Packet_Q

SHARED data!

Initial Values & Empty() Method

Initially:

Head = 0;

Tail = 0;

Count = 0;

```
boolean Empty() { return ( Count == 0 ); }
```

Add Method

```
Add ( P : packet_buffer )  
{ if Count >= Q_Size  
  { /*exception! Q full! */ exit ; }  
  Packet_Q [ Tail ] = P ;  
  Tail = ( Tail + 1 ) mod Q_Size ;  
  Count = Count + 1 ;  
}
```

NOTE: puts **P** in **Q** before adjusting **Tail** or **Count** !

Remove Method

```
Remove ( var P : packet_buffer )  
{ // assume Count > 0  
  P = Packet_Q [ Head ];  
  Head = ( Head + 1 ) mod Q_Size;  
  Count = Count - 1;  
}
```

NOTE: removes P from Q before adjusting Head or Count

Scenario

in a uniprocessor implementation, suppose:

- **ISRA** and **ISRB** finish receiving packets at approx. the same time
- independent reception – no interference
- both may attempt to access **Packet_Q.Add** concurrently
- accessing shared resource!

Add Method Details

- suppose **ISRA** calls **Add** first and is executing:

Packet_Q [**Tail**] = **P_A** ;

Tail = (**Tail** + 1) mod **Q_Size** ;

- suppose the compiled implementation of the 2nd line is:

temp = **Tail**_{old} ; // temp might be a register

temp = temp + 1 ;

temp = temp mod **Q_Size** ;

Tail_{new} = temp ;

ISRB Interrupts ISRA!

- suppose **ISRA** has executed:

Packet_Q [**Tail_{old}**] = **P_A** ;

temp_A = **Tail_{old}** ;

and is about to execute:

temp_A = **temp_A** + 1 ;

when an **interrupt** occurs and **ISRB**
begins to run

Data Corruption!

- when **ISRB** runs, **ISRA** has placed a packet in **Packet_Q**, but has not yet modified **Tail** and **Count**
- **ISRB** will overwrite the packet just added by **ISRA**, then adjust **Tail**, and then increment **Count**
- when **ISRA** resumes it will finish adjusting **Tail_{old}**, and then increment **Count**

Interference!

net result: (after both ISRs complete)

- **lost packet** P_A originally added by **ISRA**
– overwritten by P_B added by **ISRB**
- **Tail** is still correct (for the packets in Q)
but **Count** is **corrupted** (too large by one)
- Are there other interference problems?

Other Potential Interference

- **Add / Remove** concurrently
 - potential interference with **Count**
- **concurrent Add** when only one space left in **Packet_Q**
 - both calls could pass the “full” test before incrementing **Count**
 - overwrite a valid packet & increment **Count** beyond **Q_Size**

Race vs. Interference

- **race**: two concurrent activities have begun the process of accessing a shared resource
- one activity will get there first!
- a race is due to sharing resources, but a race (by itself) does not corrupt the resource
- race conditions are a **common occurrence** in event-driven systems

Critical Section Protection

- ensure mutually exclusive access to relevant shared resource(s)

Uniprocessor Solution:

- **disable interrupts** while processing critical sections
- keep critical sections short!
- which interrupts should be disabled?
 - all?
 - only those with potential to interfere?

Uniprocessor Solution

Common solution:

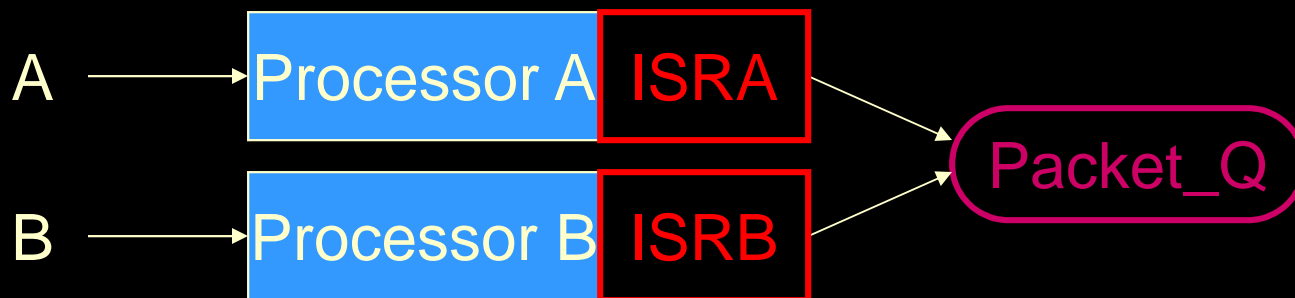
```
disable;  
    critical section    // protected!  
enable;
```

e.g.

```
disable;  
    Packet_Q.Add( myP )  
enable;
```

What about a Multiprocessor Solution ?

- recall stream-2-pipe example:
 - suppose the ISRs are implemented on independent processors & share memory
 - disabling ints on one processor won't stop interrupts on other processors!



Multiprocessor Solution

- use busy waiting and shared variables to ensure mutual exclusion
 - busy waiting ☹️
 - wastes CPU time!
- keep critical sections short 😊
 - minimize wasted time

Busy Waiting (Version 1)

share a boolean variable **Busy**

TRUE == resource is busy

FALSE == resource is available

Lock (var **Busy** : boolean)

```
{ while ( Busy ) { } // wait until available
  Busy = TRUE; // indicate resource busy
}
```


Busy Wait (version 1)

- PROBLEM! **non-atomic Lock!**
- more than one **processor** could pass busy wait loop before setting `Busy = TRUE`
- each would proceed assuming mutually exclusive access to resource

```
while ( Busy ) { }  
Busy = TRUE;
```

both processors
could reach here
before either sets
`Busy = TRUE`

Busy Wait (Version 2)

- use h/w enforced atomic operation to read and modify Busy
- **Test-And-Set TAS**
- functional syntax:
old_value **TAS** (variable, new_value)
- returns original value of variable (old_value), and sets variable to new_value
- typically locks system bus for duration of instruction

No Problem!

(as long as hardware supports TAS ☺)

```
myLock ( var Busy : boolean )  
{  
  while ( TAS ( Busy, TRUE) ) { }  
}  
      ↑  
  atomic operation
```

- Software-only solutions (no TAS) also exist for multiprocessor systems
e.g. **Lamport's bakery algorithm**

Summary of Motivation (1)

- **concurrency** has inherent difficulties:
 - potential for **interference**
 - need for **synchronization** of activities
 - need for **communication** among activities
 - **race conditions** (event-driven reality!)

Summary of Motivation (2)

- concurrent activities can arise in the **requirements** of an application
 - i.e. the system **must** support more than one input/output relationship concurrently
- concurrency in an **implementation** is the result of **design decisions**

Concurrency-Related Issues (1)

- **mindset:**

~~sequential (polling) vs.~~

event-driven (interrupts, multiprocessor)

- **priority:** some **activities** are high-priority, while others have lower-priority

- **h/w:** determines extent of concurrent capabilities of components

Concurrency-Related Issues (2)

- **culture:** "we do it this way here"
 - legacy
 - tools at hand
- **designer's artistic creation**
 - experience, problem solving
 - "on a previous project, a similar problem was solved by "

What SYSC 5701 Is

- concerned with using a process model to help reduce the development challenges for real-time applications
- primary concern: **designer's perspective!**
- simplifying the implementation of concurrency
- hide some machine details
- use “standard” process model
- simplifying the mapping of concurrency in requirements onto concurrency in implementation

Lampport on Concurrency (2009)

“Education is not the accumulation of facts. It matters little what a student **knows** after taking a course. What matters is what the student is able to **do** after taking the course. I've seldom met engineers who were hampered by not knowing facts about concurrency. I've met quite a few who lacked the basic skills they needed to think clearly about what they were doing.”

So ... Why are you Here?

- **IF**

Education is not the accumulation of facts. It matters little what a student knows after taking a course. What matters is what the student is able to do after taking the course.

- **THEN:**

What will you be able to do after completing a graduate degree?

What do you think a professor would answer?