

## Computing Surface Network (CSN)

- produced by Meiko scientific in 1988
  - message passing interprocess and interprocessor communication mechanism
- originally ran on T400 (transputer) based supercomputers
- later ported to
  - T800 (transputer)
  - i860 based supercomputers
  - sparc based supercomputers
  - fugitsu 200 Mflops/sec
- heterogeneous processor based supercomputer

## Modula-2 implementation csn.def csn.mod

- almost the same interface as the Meiko CSN
- essentially provides a transport layer service between processes
- communicating processes need to
  - open a transport: `csn.Open (tpt)`
  - register a name for a transport:  
`csn.RegisterName (tpt, 'sink')`
  - look up the other processes transport:  
`csn.LookupName (sinkId, 'sink')`
- the lookup function yields the other processes `netid`
- `netid` value is used as a destination in transmitting data

## ex1.mod

- sink receives a single string from source

```

PROCEDURE Sink ;
VAR
  tpt   : Transport ;
  actual: CARDINAL ;
  netid : NetId ;
  status: CsnStatus ;
  buffer: ARRAY [0..80] OF CHAR ;
BEGIN
  IF csn.Open(tpt) # CsnOk
  THEN
    Halt('failed to open transport',
         __LINE__, __FILE__)
  END ;

  (* we register our transport *)
  IF csn.RegisterName(tpt, 'sink') # CsnOk
  THEN
    Halt('failed to register name',
         __LINE__, __FILE__)
  END ;

```

## Sink (continued)

```

netid := NullNetId ;
status := Rx(tpt, netid, ADR(buffer),
            HIGH(buffer), actual) ;
IF status=CsnOk
THEN
  WriteString('sink received string: ') ;
  WriteString(buffer) ; WriteLn
ELSE
  Halt('Rx failed', __LINE__, __FILE__)
END
END Sink ;

```

## Source

```

PROCEDURE Source ;
VAR
  tpt   : Transport ;
  sinkId: NetId ;
  status: CsnStatus ;
  buffer: ARRAY [0..80] OF CHAR ;
BEGIN
  IF csn.Open(tpt) # CsnOk
  THEN
    Halt('failed to open transport',
         __LINE__, __FILE__)
  END ;

  IF csn.LookupName (sinkId, 'sink') # CsnOk
  THEN
    Halt('failed to lookup filter program',
         __LINE__, __FILE__)
  END ;

```

## Source (continued)

```

  StrCopy('Hello world', buffer) ;

  status := csn.Tx(tpt, sinkId,
                  ADR(buffer), StrLen(buffer)) ;
  IF status#CsnOk
  THEN
    Halt('failed to send string to sink process',
         __LINE__, __FILE__)
  END
END Source ;

```

## Transmit/Receive

- transmit function: `csn.Tx(tpt, sinkId, address, bytes)`
  - `tpt` is the transport which has been opened
  - `sinkId` is the netid of the destination transport
    - which will have been found using `csn.Lookup`
  - `address` address of variable, array etc.
  - `bytes` is size of variable
- address and size can be found out by:
  - `ADR(variable)`
  - `SIZE(variable)`

## csn.Rx

- receive function:
  - `csn.Rx(tpt, sourceId, address, bytes, actual)`
- parameters
  - `sourceId` is the senders netid
  - `address` is where the incoming data will be placed
  - `bytes` the maximum length of incoming message
  - `actual` is the length of data received
- note we can receive `<=bytes` of data
  - why is this useful?

## Non blocking communication

- the previous `csn.Tx` and `csn.Rx` functions cause processes to block whilst data is being sent and received
- sometimes may wish allow processing and communication to occur in parallel
  - why?
- non blocking primitives allow this
  - `csn.TxNb`
  - `csn.RxNb`

## Non blocking communication

- however if we allow caller of `csn.Tx` to return before data is sent
- then the process may need some method of finding out when the data has been received
  - `csn.Test`
- in fact every non blocking operation (`csn.Tx` and `csn.Rx`) *must* be followed *some time* by a call to `csn.Test`

## ex2.mod

- `ex2.mod` is much the same as `ex1.mod` except:
  - it continually transmits and receives data "hello world"
  - it receives data using `csn.RxNb`

## ex2.mod: csn.RxNb

- ```

IF csn.RxNb(tpt, ADR(buffer),
            HIGH(buffer), actual) # CsnOk
THEN
  Halt('failed to queue buffer',
       __LINE__, __FILE__)
END ;

netid := NullNetId ;
bufptr := NIL ;
CASE csn.Test(tpt, {CsnRxReady}, MAX(CARDINAL),
             netid, bufptr, status)
OF
  CsnRxReady: WriteString(buffer)
ELSE
  Halt('unexpected return value from csn.Test',
       __LINE__, __FILE__)
END
      
```

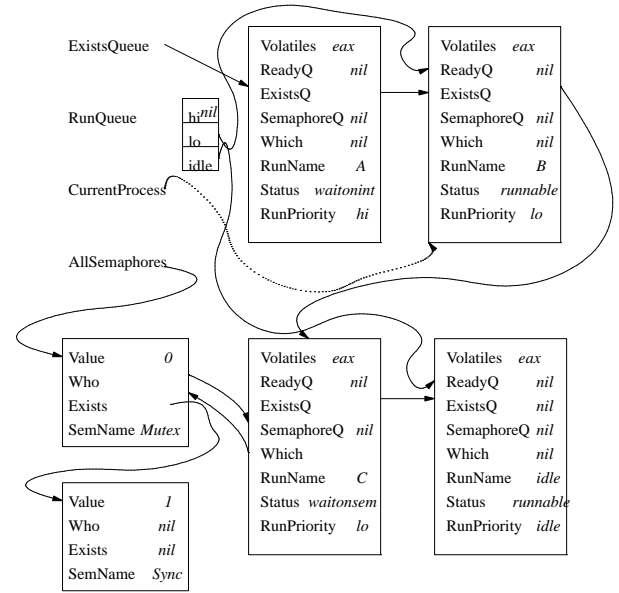
## csn.TxB parameters

- `csn.Test (tpt, {CsnRxReady}, MAX(CARDINAL), netid, bufptr, status)`
- parameters: three refer to the actual `csn.Test`
  - `tpt` transport created by `csn.Open`
  - flags must we wait until `RxReady` or `TxReady` or both
  - timeout value (`MAX(CARDINAL)`) max number of millisecs to wait
- remaining three parameters are filled in by the test but refer to the completed operation
  - `netid` is the netid of the transport communicating with `t`
  - `bufptr` is the start of the block of memory which has completed
  - `status` is the status of the completed operation

- `RunQueue`, `CurrentProcess`, `ExistsQueue` and `AllSemaphores`

## Tutorial: Executive

- draw the executive data structures which coordinate process activity



- draw 4 processes, 2 semaphores
  - you may simplify the list structure to the bare minimum
  - include appropriate global variables:

## Tutorial: Processes

- draw the runqueue, semaphore queue for the two processes:

```

PROCEDURE Process1 ;           PROCEDURE Process2
BEGIN                           BEGIN
    LOOP                         LOOP
        Wait (Sem) ;             Wait (Sem) ;
        (* critical region *)    (* critical r
        Signal (Sem)             Signal (Sem)
    END                           END
END                               END
    
```

- and redraw the queues showing each change as the processes execute.