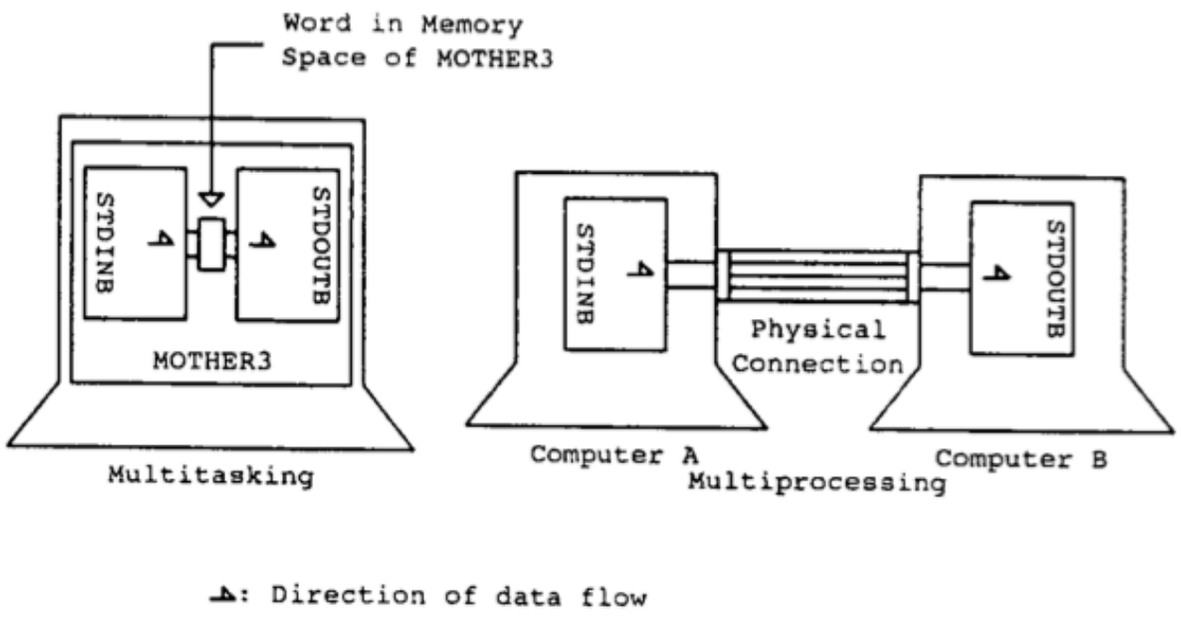
occam Transpiler Workshop

Presentation: Saving the Hard Stuff

Lawrence J. Dickson, PhD
Space Sciences Corporation
Lemitar, NM, USA
larry@spacesciencescorp.com

- **Structure and value of the occam language**
- **Key components of occam programming**
- **Transforming Go constructs into equivalents of these components**
- **The complete picture via lex/yacc grammar and descent ordering**
- **Making Go do occam things: demonstrations**
- **Extension to modern hardware communication (block and network)**

Value of static occam2 language

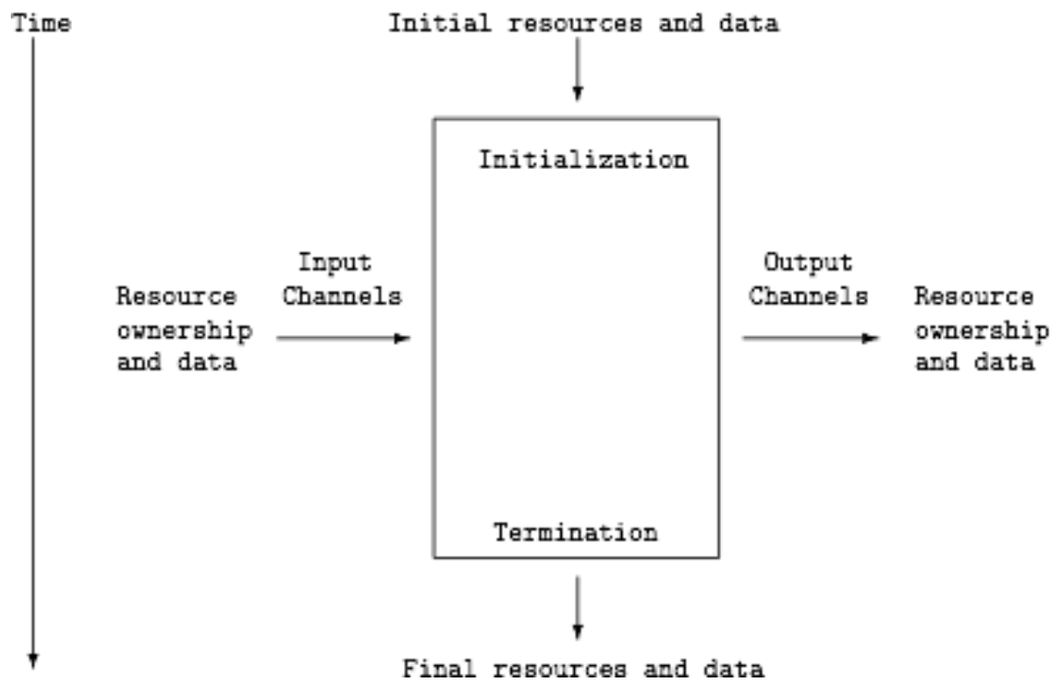


- Small and simple, yet capable of doing all computing tasks
- Hardware/software equivalence (HSE), perfect for IoT
- See drawing above (DOS, 1996) for simple HSE illustration
- Clear, obvious parallel coding using formally verifiable CSP
- Massive legacy of the Transputer, currently being reborn
- Very responsive, with no need for an operating system
- Capable of extension while still remaining static (HSE, CSP)
- Adapted to rigorously correct modeling of complex systems

Causal CSP and Ground Truth

- "Causal" essentially means "use input ALTs"
  The restriction means someone has to commit absolutely, to avoid the situation in the cartoon. With DTA-ACK communications this means input ALTs; REQ-DTA would mean output ALTs.
- "Ground truth" uses real, runnable Transputer code to prove equivalence
  The critical thing is that timing is handled, by simply counting Transputer cycles. This proves that equivalence implied by CSP is achievable by responsive hardware. It could use other hardware, of course, but static Transputer occam is by far the easiest and most robust.
- Example: shelf/store FIFO (causal, rotating index) = standard buffer
  This was detailed in my Fringes at CPA2017. It has wide implication for real hardware, e.g. connections in TCP/IP.
- Both causal CSP and ground truth exhibit hardware/software equivalence
  The causality sequences are very closely examined. See my Fringes at CPA2017. The fact that occam is a static language is critical here.
- All this is offered for critique by world CSP experts at this CPA conference

Structure of occam



- Processes can be either hardware or software
- Processes use fixed, known resources
- Processes, once underway, share only by channel communication
- These channels can be either hardware or software
- Processes can subdivide either in time (SEQ) or space (PAR)
- Complex shared communications work by ALT (=select)
- All the above are language primitives, not system calls
- There is no hidden system or driver code; all code works the above way
- The effect of a process on its surroundings can be fully described
- These descriptions can be nested for robust, predictable programs

Key components of occam

- Standard sequential “Von Neumann” imperative structures
  These include IF, WHILE, sequence, call, etc; all translatable with ease into most languages.
- PAR, which subdivides resources to parallel sub-processes
  These are not “spawned”, but terminate together, giving their resources back to the parent. Processes must be parallel to communicate via channels.
- Channels, which can communicate certain data and timing
  When both sides are ready, the communication happens promptly, and then both processes may continue. The communication is immediate (unbuffered) and handshaken.
- The channels are ALT-enabled, with recipient deciding which will flow
  This splits a communication in half, with a select happening before the handshake or ACK. It, not threading, is needed for many-to-one communication designs to work.
- Resource arrays, including memory, are unfragmented and top-down
  This means static, and diverges from the post-1995 direction of almost all computing, even occam-Pi! A tight grip on resources, which always have a hardware equivalent, makes formal verification and HSE easy.
- Complex tasks therefore require transmuting resources in place
  Knowledge of type sizing, abbreviation, and in-place retyping (RETYPES) is therefore a necessity, used frequently in concert with general-purpose (counted array) communication.

Requirements on Go

```
Decode
                      gob     28.17 MB/s       (39.8 MB in 1.41s)
                     json     30.17 MB/s       (113.8 MB in 3.77s)
                  memdump   1031.54 MB/s       (113.2 MB in 0.11s)
```

Reference: https://github.com/alexflint/go-memdump

- Garbage collection must be disabled for the life of occam programming
  The static occam language allocates all resources from the top down, and all changes must be controlled by the program. References say this is done by “GOGC=off” or “GOGC=<negative percent>”
- The `unsafe` package must be safe
  This means the memory mapping must be clear and unfragmented. See documentation of `unsafe`.
- Type bitmaps must be explicitly known (e.g. little-endian)
  The occam language achieves clarity by streaming real data, not abstractions.
- Object-oriented constructs like “interfaces” must be avoided
  The pure bitlevel data knowledge and determinism of occam takes care of the problems which “interfaces” and the like are built to combat (usually very inefficiently - see table above, where `memdump` uses `unsafe` and avoids interfaces).
- These requirements, though frightening to “language lawyers”, seem easily satisfied in real Go installations.
  For example, the Go playground reveals itself to be little-endian with obvious mapping between int64 and [2]int32.

PAR implementation

Use goroutines with atomic countdown

- Global to the PAR: int32 kount and chan bool alldone
- Preset kount to member count minus 1
- At end of each member, decrement kount by 1
- If member finds kount negative, output to alldone
- In parent, await input from alldone

func that is invoked must be locally declared

- The decrement and test must use global for this PAR
- Therefore a general or library func cannot be called
- However, it may be invoked within local func
- This is like time.Sleep within whistle
- The fmt.Println in whistle is for debugging only
- The use of local variable kountdown in test is **required**!
- Otherwise, more than one PAR member may output on alldone

```
package main

import "fmt"
import "time"
import "sync/atomic"

var kount int32
var alldone chan bool

func whistle(ind int) {
  time.Sleep(time.Duration(ind) * time.Second)
  kountdown := atomic.AddInt32(&kount, -1)
  fmt.Println("whistle", ind, kountdown)
  if kountdown < 0 {
    alldone <- true
  }
}

func main() {
  alldone = make(chan bool)
  kount = 2
  for i:=0; i<3; i++ {
    go whistle(i)
  }
  <- alldone
  fmt.Printf("Type of alldone is %T\n", alldone)
  time.Sleep(3 * time.Second)
  fmt.Println("Done after wait")
}
```

`ALT` with Boolean guards

Use the `when` construct if guarded

- https://groups.google.com/forum/#!topic/golang-nuts/ChPxr_h8kUM
- Ross Cox's `when` construct works for any `chan`
- The func type is same as the `chan` type
- The `nil` is a do-nothing channel that never wins
- The result is same as occam

```
notinput[i] & masin[i] ? thewait
```

A select on a chan works if not guarded

- The Go would be like

```
case thewait = <- masin[i] :
```

- The corresponding occam would be like

```
masin[i] ? thewait
```

- Notice that a replicated ALT must be expanded in Go

Replicated select left out of Go

- https://groups.google.com/forum/#!msg/golang-nuts/ZXDfIkVk54g/BcrR50rVrdsJ

On Monday, February 18, 2013 at 6:01:26 PM UTC-8, Russ Cox wrote:

. . .

Regarding array select, I will repeat what Nigel found me saying earlier: "we intentionally left array select out. It's an expensive operation and can often be avoided by arranging for the various senders to send on a shared channel instead."

```
func when(b bool, c chan int) chan int {
  if b {
    return c
  }
  return nil
}

func parent(masin, masout [3]chan int) {
  var notinput [3]bool
  var thewait, ind, m int
  // more code . . .
    for k:=0; k<3; k++ {
      select {
        case thewait = <- when(notinput[0], masin[0]) :
          ind = 0
        case thewait = <- when(notinput[1], masin[1]) :
          ind = 1
        case thewait = <- when(notinput[2], masin[2]) :
          ind = 2
      }
      // more code . . .
}
```

```
// The logic behind this decision to reject the
// replicated select in Go is unfortunate.
// It is not a question of efficiency, but a question
// of modeling. Also, they seem to do complicated
// things with semaphores rather than the simple
// implementation of occam ALT.
```

Counted array protocol

Two Go channels are used

- Slices are used to hold data, not arrays
- The `chan` in the forward direction is a slice chan
- The `chan` in the back direction is a slice len count chan
- The slice chan does not send the data
- The slice chan sends pointer, length, and capacity
- The receiver pulls the data with a copy (memcpy)
- Count is min of slice size and buffer size
- The receiver returns count of data actually pulled

This detects success/failure information

- If receiver can find room for all data then success
- Otherwise, receiver detects failure
- If sender receives count equal to its slice len, success
- Otherwise, sender detects failure

Unclear whether zero-length is allowed

- The occam standard is unclear on this point
- Either possibility can be handled via branches

occam `chan of SP` is supported

- This occam protocol uses 16-bit count
- It is vital for general link communication
- It is vital for communication with an alien host like a PC
- It permits completely general freeform communication

```
func snd(c chan []int, cb chan int) {
  var sl int
  src := [8]int{8, 7, 6, 5, 4, 3, 2, 1}
  s4 := src[0:4]
  c <- s4
  sl = <- cb
  // more code
}

func rcv(c chan []int, cb chan int) {
  var tl int
  var tgt [6]int
  t4 := <- c
  tl = len(t4)
  if tl > 6 {
    tl = 6
  }
  copy(tgt[0:tl], t4)
  fmt.Println("rcv tgt[0:tl]", tgt[0:tl])
  cb <- tl
  // more code
}
```

Multidimensional arrays and slices

```
package main

import "fmt"

func main() {
  var a [5]int
  var b [4][3][2]int
  fmt.Printf("Type of a: %T, type of b: %T\n", a, b)
  fmt.Printf("Type of b[1]: %T, type of b[3][2]: %T\n", b[1], b[3][2])
  fmt.Printf("Type of b[1:3]: %T, type of b[3][0:1]: %T\n", b[1:3], b[3][0:1])
  fmt.Printf("Type of b[2][1][0]: %T, type of b[2][1][0:1]: %T\n", b[2][1][0], b[2][1][0:1])
}
```

Go arrays are organized exactly the same as occam arrays

- Rightmost index is fastest, both in type definition and in expression or variable
- An n+1-dimensional array is an array of n-dimensional arrays
- The members of an array are packed sequentially with no fragmentation
- The operation of the `unsafe` package implies this non-fragmentation
- The above program generates output that implies this organization

occam abbreviations are a subset of Go slices

- An occam abbreviation equals a Go slice with a single slice dimension
- The slice dimension is to the left of a type, or to the right of an expression or variable
- This corresponds to a contiguous sub-block of an array's memory
- The equivalent of occam `IS` or `RETYPES` always results in a slice
- Go also permits multiple slice dimensions, which do not correspond to an occam entity
- For example, `[][]byte` would correspond to multiple strings of variable length
- This could be implemented in C as an array of pointers which point to real byte arrays
- But it only works in occam if a constant inner dimension is present

Go equivalents of IS and RETYPES

IS is mostly equivalent to slices

- See previous slide for slices being made from arrays
- Slices can be made from slices too
- This takes care of occam array abbreviations
- Restrictions on abbreviation are imposed by occam only
- The unclear case is variable (non-VAL) scalar, not array
- Care must be taken that Go does not lose the variable
- If necessary, use the RETYPES techniques below

RETYPES requires unsafe and reflect

- GARBAGE COLLECTION MUST BE DISABLED!
- Unfragmented memory map found by unsafe package
- Calculations involving type size must be carried out
- The technique byte-overlays new slice on old array/slice
- Similar approach can be used for scalar RETYPES
- https://stackoverflow.com/questions/51187973/how-to-create-an-array-or-a-slice-from-an-array-unsafe-pointer-in-golang/51188315

The SliceHeader output is unfragmented

- The technique constructs a slice from raw memory
- The experiment shows standard (little-endian) order

```
package main

import (
  "fmt"
  "unsafe"
  "reflect"
)

func main() {
  var try64 [2]int64
  var data []int32
  try64[0] = -4
  try64[1] = 3
  p := uintptr(unsafe.Pointer(&try64))
  sh := (*reflect.SliceHeader)(unsafe.Pointer(&data))
  sh.Data = p
  sh.Len = 4
  sh.Cap = 4
  fmt.Println(try64)
  fmt.Println(data)
}
```

occam Grammar Descent Levels

Syntactic objects have descent levels

- These are based on the BNF or grammar
- An object of higher descent level can include a lower level object, but not vice versa
- Example: an **assignment** can include an **expression**, but not vice versa (unlike C)
- If each can include the other, they are on the same descent level
- Example: an **operand** can include an **expression**, which in turn can include a more deeply nested **operand**
- All PROCESS LEVEL objects are on the same descent level
- All EXPRESSION LEVEL objects are on the same descent level
- Other groupings have several levels each, but higher are above lower
- LINE LEVEL and above have `NEWLINE`s, those below never do

The catch

- To reach this obvious descent, one obscure change had to be made in the BNF
- Inline value processes (which insert **process** inside **operand** or **expression.list**) are replaced by `ANY FUNCTION`s

```
#INCLUDE "hostio.inc" --  -- contains SP protocol
--{{{  PROC waitandtrigger(CHAN OF SP fmas, tmas)
PROC waitandtrigger(CHAN OF SP fmas,
  tmas)
  INT thewait,
    thetime:
  [2]INT thewaits :
  VAL INT thesize IS SIZE "Beware the jabberwock my son,* -- Start
                          * the jaws that bite,* -- Example of middle
                          * the claws that catch." : -- Example of end
  TIMER clock:
  INT16 len:
  SEQ
    SEQ j = 0 FOR 2
      SEQ
        []BYTE thewaitbytes RETYPES thewait:
        fmas ? len::thewaitbytes
        thewaits[j] :=
          thewait
    SEQ j = 0 FOR 2
      SEQ
        thewait := thewaits[j]
        clock ? thetime
        clock ? AFTER (thetime PLUS thewait)
        []BYTE thewaitbytes RETYPES thewait:
        tmas ! len::thewaitbytes
:
--}}}

LEGEND:

PROCESS LEVEL

LINE LEVELS

MIDDLE LEVELS

EXPRESSION LEVEL

SIMPLE LEVELS
```

Replace inline value processes

Old form below

```
total := subtotal + (INT sum:
                     VALOF
                       SEQ
                         sum := 0
                         SEQ i = 0 FOR SIZE v
                           sum := sum + v[i]
                     RESULT sum
                    )
```

```
ANY FUNCTION uniqname()
  INT sum:
  VALOF
    SEQ
      sum := 0
      SEQ i = 0 FOR SIZE v
        sum := sum + v[i]
    RESULT sum
:
total := subtotal + uniqname()
```

New form above

The equivalent `ANY FUNCTION` construction

- The example is taken from the occam 2 Reference Manual, page 65, Chapter 11, Functions
- The **value.process** (highlighted part) is shifted verbatim as shown
- `ANY` on the `ANY FUNCTION` means type must be calculated from `RESULT` type(s)
- The `ANY FUNCTION` is given a unique name by a preprocessor
- The `ANY FUNCTION` is defined below all definitions of globals used by the statement that needs the value process (this is always possible)
- This revision causes descent levels to clarify as in previous slide
- ***The author would like to know if anyone actually ever used the inline value process***

OCCAMLIB Grammar

The grammar is equivalent to the occam 2 syntax

- It uses the standard open-source flex (lex) lexer and bison (yacc) parser
- The biggest change is the `ANY FUNCTION` replacing inline value process (previous slide)
- A BNF error is corrected by adding **andor.expression** and **andor.operator** to distinguish the Boolean operators that do not need parentheses
- New syntactic objects **comms.type**, **counting.type**, and **data.type** are added to subdivide **primitive.type** and more accurately reflect the syntax
- The program is assumed to be a number of `PROC`s and typed `FUNCTION`s, which corresponds to the usage of the occam toolset used with the occonf configurer

The lexer detects `NEWLINE` and indentation

- Continuation lines, full-line comments, and preprocessor directives do not make `NEWLINE`s
- When combined with the elimination of inline value processes, this makes detection of indentation straightforward in the lexer

The parser uses `%glr-parser` and `%dprec` to converge

- Several changes, most notably combining **channel**, **port**, and **timer** into one, are made to permit parsing to converge
- Type information will allow them to be distinguished later
- In three places, `%dprec` is used to guide the multiple paths found by the parser
- Other ambiguities are solved by the quick death of the false path
- Details are in the Details-occam-parse.pdf document
- Complete history, lexer and parser source, and test files are in Details-occam-parse.tgz

Usefulness of the Grammar

The completeness of the grammar guides Transpiler creation

- The `y.output` file completely describes the parsing
- The grammar description has 262 lines
- The parser finds 574 states
- It is easy to do exhaustive searches to determine important properties of the language

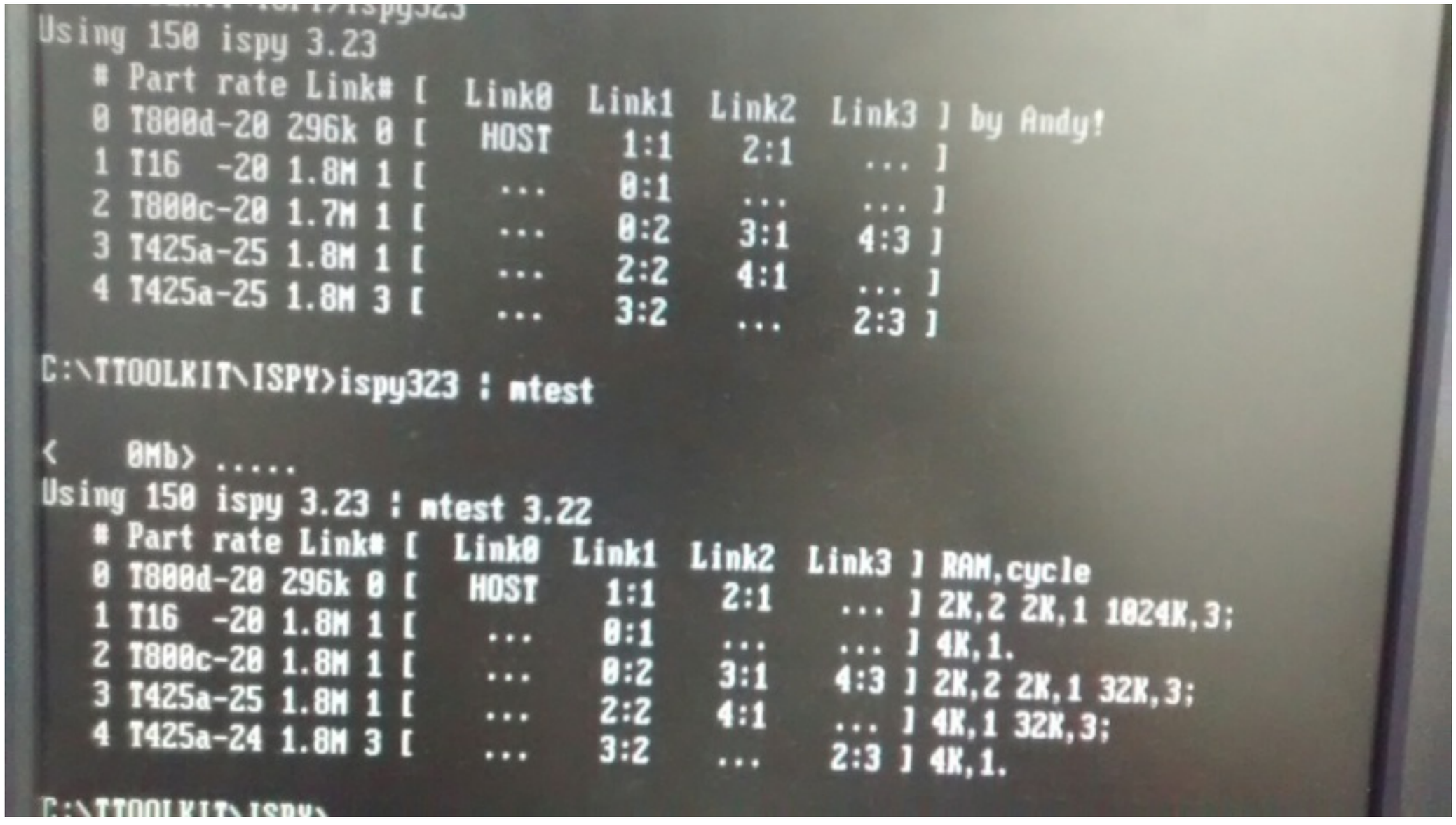
Example: a C-like syntax is possible

- Replacing every semicolon with a comma yields the same states
- Every `RIND` (single relative indent) is necessarily preceded by a `NEWLINE`
- This and a few other details show that a C-like-appearing syntax will trivially work

The entities that must be mapped into Go are few in number

- All statements (end in `NEWLINE`s) are either resources (colon-terminated) or live (not colon-terminated)
- 7 normal live leaves, 12 live leaves are empty structures (trivial)
- 7 live structures (basic)
- 5 live structures (dependent general guards)
- 1 quasi-live structure (`VALOF`) found in `FUNCTION`
- Sub-statement breakdown leads to expressions and elements (expressions and names)
- The most complex feature is the `PROTOCOL` used for a channel
- Given counted array protocol (above), it is soluble using a set of different Go channels to represent one occam channel

Legacy system, B008 with 4 TRAMs



- Transputer 0 is the root, connected to the DOS PC host.
- Transputer 2 is the parent, connected (`hardaltt` case) to 0, 3, 4.
- Transputer 1 is part of the B008, not used.

How test ALTbtes2 works

A parent communicates with three children, twice each

- Each child gets both its wait times at the beginning
- It waits the first time, communicates, waits the second time, communicates, quits
- The parent (master) cycles twice
- Each cycle, it listens to children in order (ALT), then clears a Boolean so it will not listen again

Child 0 gets two short times

- The first time it wins the race, but then is forced by the booleans to wait till others come in
- It actually tries to send second time long before first parent cycle is over
- As soon as parent turns on second cycle and sets its boolean, it communicates quickly

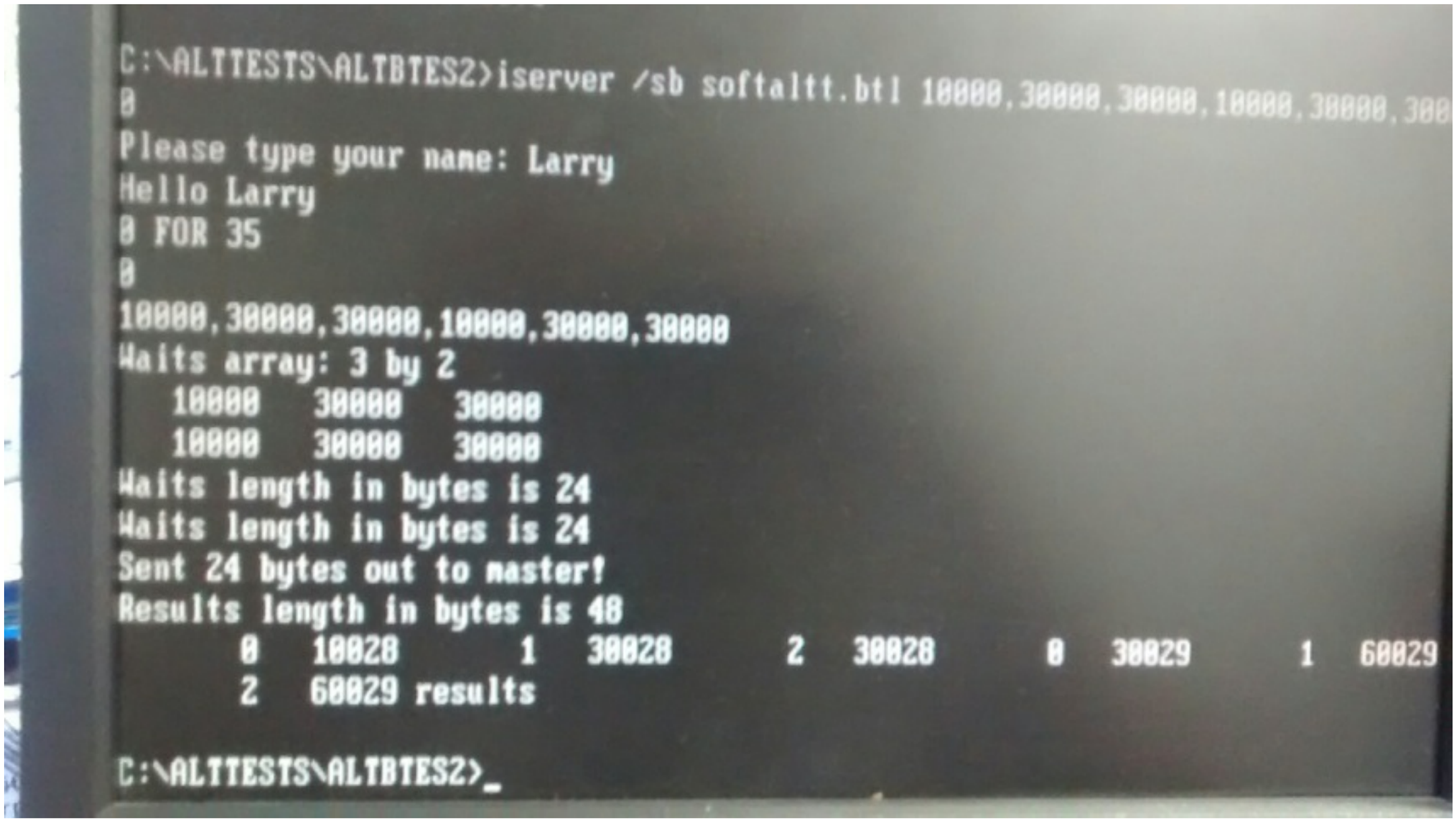
This tests both the ALT and the boolean

- The ALT listens to whatever child comes ready first
- The boolean prevents any child from communicating twice in a cycle
- The timings show that the parent and the children are operating independently
- The orderly shutdown shows that the PAR implementation is working
- Other tests with different times demonstrate all these results

ALTbtes2/master.occ

```
#INCLUDE "hostio.inc" --  -- contains SP protocol
--{{{  PROC master([3]CHAN OF SP masin, masout)
PROC master([3]CHAN OF SP masin, masout)
  [3]BOOL notinput :
  INT starttime, thetime, thewait, thewaitinput, m:
  VAL INT dummy IS 0:
  TIMER clock:
  INT16 len:
  [2][3]INT waits:
  [6][2]INT results :
  VAL []BYTE dummybytes RETYPES dummy:
  VAL INT16 lenone IS INT16 (SIZE dummybytes) : -- 4
  SEQ
    []BYTE waitsbytes RETYPES waits:
    masin[0] ? len::waitsbytes -- len better be 24
    clock ? starttime
    m := 0
    SEQ j = 0 FOR 2
      SEQ
        -- give each child its wait length
        SEQ i = 0 FOR 3
          VAL []BYTE onewaitbytes RETYPES waits[j][i] :
          masout[i] ! lenone::onewaitbytes
    SEQ j = 0 FOR 2
      SEQ
        notinput := [TRUE, TRUE, TRUE]
        -- Listen three times
        -- Each time, remove the winner from the next ALT by a boolean
        []BYTE thewaitinputbytes RETYPES thewaitinput :
        SEQ k = 0 FOR 3
          ALT i = 0 FOR 3
            notinput[i] & masin[i] ? len::thewaitinputbytes
              SEQ
                clock ? thetime
                results[m][0] := i
                results[m][1] := thetime MINUS starttime
                m := m + 1
                notinput[i] := FALSE
    VAL []BYTE resultsbytes RETYPES results :
    SEQ
      len := INT16 (SIZE resultsbytes) -- 48
      masout[0] ! len::resultsbytes
:
--}}}
```

Soft altbtes2 run

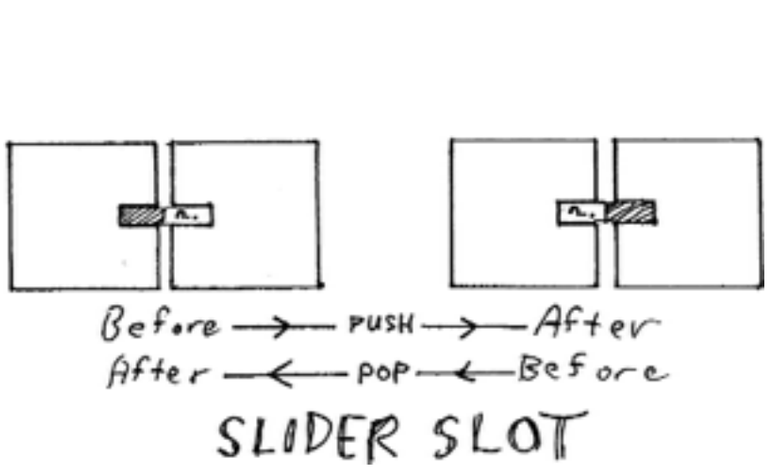


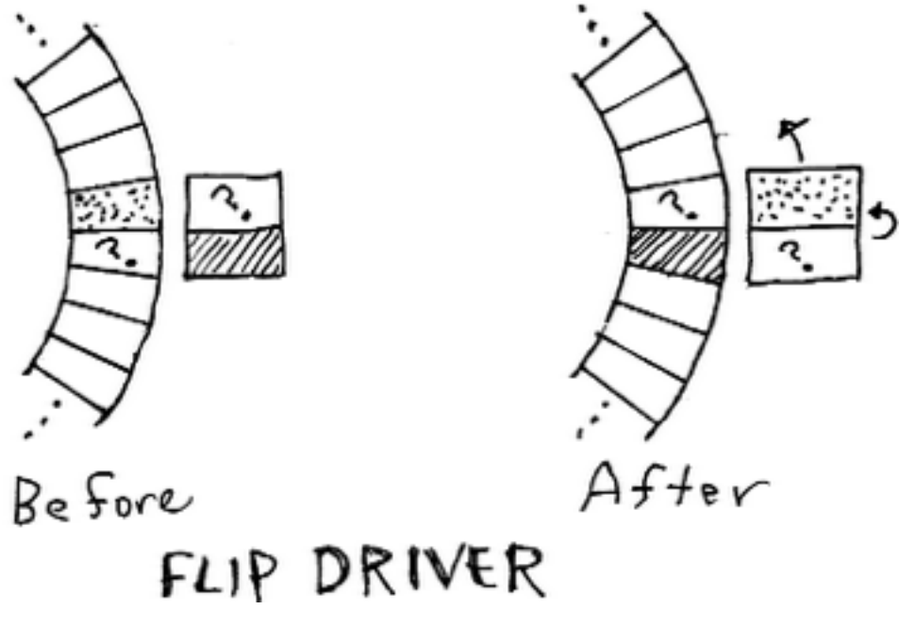
- Parent and 3 children are all soft processes on the root.
- Boolean prevents child 0 from communicating second time till all three children have communicated once.

Hard altbtes2 run

- Child 0 is on root, parent on Transputer 2, other children on 1 and 3
- Boolean prevents child 0 from communicating second time till all three children have communicated once.

The Detachable (block device)





Hardware memory stick is analogue of static mobile block

- This concept is explored in my *Crawl-Space Computing* (Amazon, 2014)
- The static mobile is from Fred Barnes' thesis, *Dynamics and Pragmatics for High Performance Concurrency*, University of Kent, 2003, section 4.2.1 - 4.2.3; implemented in occam-Pi

HSE means that an empty slot must be supported

- This is not the same as "undefined" as treated in Barnes, section 4.6
- The hardware analogy is extremely useful in defining this correctly

A Detachable may be like a link: external to the hardware process

- The definition of the Detachable must enclose all devices that use it
- Detachables would be supported in the configuration (.PGM) file, like links
- A Detachable cannot be created or destroyed, only moved

Detachables relate to Barnes mobilespace

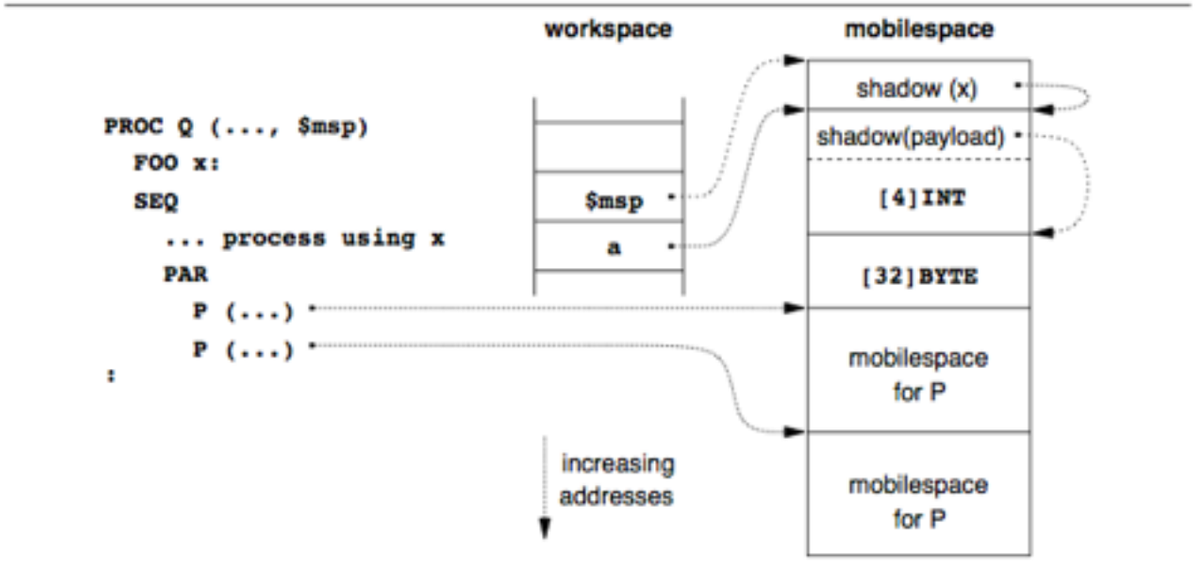


Figure 4.3: Allocation of process instances in mobilespace

Barnes, p 72: data structure in mobilespace

Modify Barnes shadow to accommodate both empty and undefined

- MINT can never be a pointer, so it can denote empty
- All pointers (and MINT) are even, so the LSB can denote whether defined or undefined

Proposed: if any part ever defined, whole is defined

- Barnes, section 4.6 (which is extremely complex) is then not necessary
- Definedness is passed up into including structures like static chain
- Once defined, need check only lowest structure to see defined marking is unneeded

New `unlatch` instruction “formats” (makes undefined)

- Undefinedness is spread to included structures using top pointers
- Even physically removed Detachables retain defined/undefined information
- A Detachable cannot be created or destroyed, only moved

The Attachable (network device)

Hardware: operator connects cable to server

- This implements socket TCP/IP and similar communication (network)
- Incoming call (`connect()`) is a cable proposing a connection
- Server may `accept()` or connection may fail (cable withdrawn): 3-way handshake
- Server has finite slot count (WACOMSAS or httpd): static mobile channels, standard many2one

TCP connection is a buffered 2-way link

- Due to the sliding window and the send buffer limit, data flow is a subset of buffered CSP link
- See https://devcentral.f5.com/articles/the-send-buffer-in-depth-21845

The connection has one out-of-band variant

- The two two-way FIN-ACKs are standard communications of the out-of-band FIN
- They can be sent independently by either side, leading to withdrawal of attachable cable
- The cable (whole IP world) is a permanent feature of Client

3-way handshake occam pseudocode

Server

```
PORT OF CABLE a :
PORT OF EVENT aack :
CHAN OF CONNECTION b :
CABLE cable :
CONNECTION connection :
BOOL success :
SEQ
  a ? cable
  -- process
  IF
    success
      SEQ
        b ! connection
        aack ! ANY
    TRUE
      aack ! ANY
```

Client

```
PORT OF CABLE a :
CHAN OF EVENT aack :
CHAN OF CONNECTION b :
CABLE cable :
CONNECTION connection :
BOOL success :
SEQ
  a ! cable
  ALT
    b ? connection
      SEQ
        success := TRUE
        aack ? ANY
    aack ? ANY
      success := FALSE
```

`aack` is explicit acknowledge of `a`

- The `PORT` does not acknowledge on input or await acknowledge on output
- The `CHAN OF EVENT` input does hardware handshake only (see Events in *The Transputer Databook*, INMOS, 1992). It is an `AFTER` timing test only.
- In the standard case (TCP/IP hardware) `b` and `aack` are the same hardware connection and direction. The ordering of the half-messages in Server success is always possible, due to the possibility of different processes controlling different channel directions on a single link.
- The third leg of the triple handshake is the acknowledge of `b`. In the failure case, the race-winning `aack` signal is treated as a NAK.
- The cable/connection combination is called a **ATTACHABLE** because Client lends it to Server.

Extended rendezvous equivalent

Server

```
CHAN OF EXTENDED RENDEZVOUS CABLE a :
CHAN OF CONNECTION b :
CABLE cable :
CONNECTION connection :
BOOL success :
a ?? cable
  SEQ
    -- process
    IF
      success
        b ! connection
      TRUE
        SKIP
-- aack follows here
```

Client

```
CHAN OF EXTENDED RENDEZVOUS CABLE a :
CHAN OF CONNECTION b :
CABLE cable :
CONNECTION connection :
BOOL success :
a !! cable & ALT
  b ? connection
    success := TRUE
  FAILURE -- aack wins
    success := FALSE
-- aack follows here if needed
```

`aack` is now included in `a`

- The extended rendezvous is a feature of occam-Pi
- See https://stackoverflow.com/questions/2862675/occam-pi-extended-rendezvous
- The usage here may not be the same as in occam-Pi
- Particularly, the construct with the ALT is invented by me
- It is as shown on the previous page, a race between the delayed ACK and the connection
- `FAILURE` means the ACK won; in case of success, the ACK is implicit at the closure

Conclusions and future directions

Classic, static occam can do everything in computing

- Missing ingredients (block, network) can be done statically
- Hardware/software equivalence can be explicitly enforced in every case (hardware analogue)
- All levels of computing (from system/driver to UI apps) fit the classic occam model
- This model expands in power without expanding in complexity
- Modern resource wealth means dynamic structures are unneeded in outer code
- They can remain in inner code (apps) inside an occam harness and still satisfy the model

Transpiling classic occam (to Go) is a tractable problem

- All the hard parts of occam requirements in Go are designed
- The size of the task is established by the full grammar for occam
- The same approach will be usable for other CSP-related languages (e.g. Rust)

The value is huge, because it gives formally verifiable IoT

- Causal CSP is a completely capable subset of CSP
- Causal CSP timing is established by Ground Truth, formally verifiable in occam
- HSE means component processes, once formally verified, build in nested fashion
- The occam harness allows predefined app code to be made part of this without much effort
- Maintenance becomes easy, because the external behavior of a process is totally defined