RMoX: A Raw Metal occam Experiment

Fred Barnes[†] (frmb2@ukc.ac.uk)
Christian Jacobsen[†] (clj3@ukc.ac.uk)
Brian Vinter[‡] (vinter@imada.sdu.dk)

[†] Computing Laboratory, University of Kent

[‡] Department of Maths and Computer Science, University of Southern Denmark

Contents

- Introduction
- Dynamic occam
- Design:
  - the ‘kernel’
  - device drivers
  - filesystems
  - networking
  - top-level and console
- Implementation:
  - accessing hardware
  - real and user-mode RMoX
- Conclusions

Motivation

- Do we really need another OS ?
- Most existing operating systems suffer:
  - from software error
  - from high overheads
  - a lack of scalability
- We want an operating system that:
  - has a rigorous design
  - is free from software error
  - is fast!

Introduction

- Raw Metal occam:
  - runs on the "raw" hardware
  - with some help from the UTAH Flux OSKit
  - and a modified KRoC run-time system
- CSP design
- Dynamic occam implementation

Dynamic occam

RMoX extensively uses two of the new dynamic features of KRoC/occam:

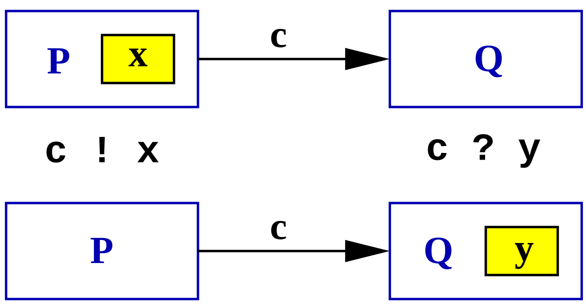
- mobile channel-bundle ends
- dynamic process creation (fork)

These support:

- dynamic network expansion and re-configuration
- scalable server farms

Mobile Data

- Provides a movement semantics for assignment and communication
- Implementation supports both *static* and *dynamic* mobiles
- Unit-time assignment and communication
- Unit-time ‘garbage-collection’ (strict aliasing)



- ‘P’ can no longer access ‘`x`’ — compiler enforced

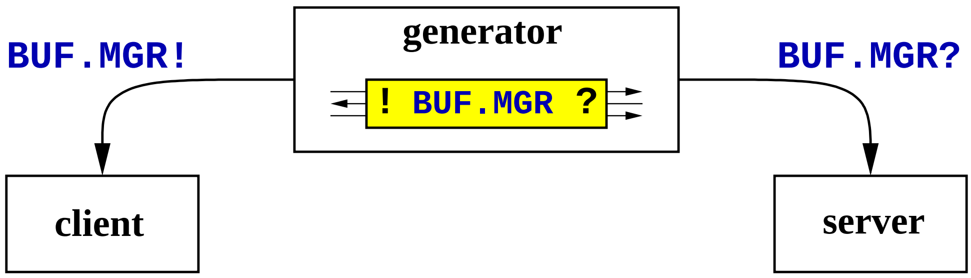
Mobile Channel-Ends

Ends of channel-types — structured *bundles* of channels:


```
CHAN TYPE BUF.MGR
  MOBILE RECORD
    CHAN INT req?:
    CHAN MOBILE []BYTE buf!:
    CHAN MOBILE []BYTE ret?:
:
```

- Channel bundles have two ends: "?" (or *server*) and "!" (or *client*)
- Direction of communication is specified (from the '?' view)

Mobile Channel-Ends



Code to setup the network is trivial:

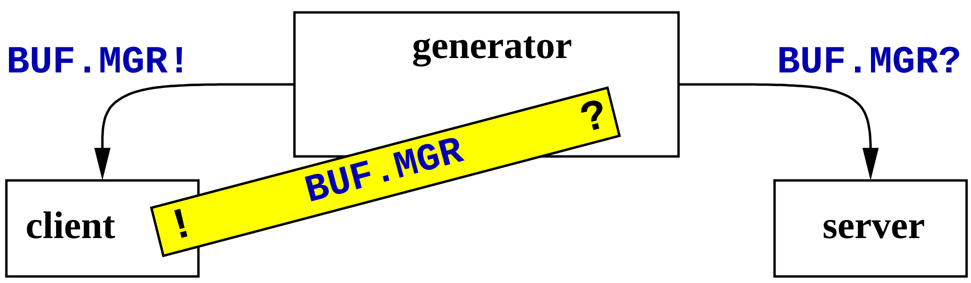
```
CHAN BUF.MGR! cli.chan:
CHAN BUF.MGR? svr.chan:
PAR
  generator (cli.chan!, svr.chan!)
  client (cli.chan?)
  server (svr.chan?)
```

Mobile Channel-Ends

The ‘generator’ process creates the channel bundle then communicates the *ends*:

```
PROC generator (CHAN BUF.MGR! cli.chan!,
                CHAN BUF.MGR? svr.chan!)
  SEQ
    ...  create channel bundle
    cli.chan !  buf.cli
    ...
:
```

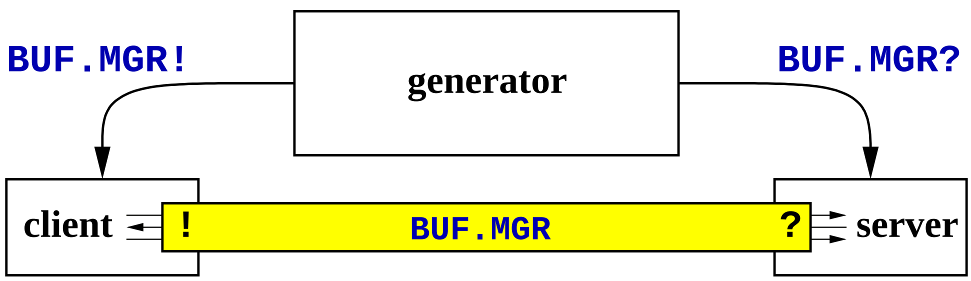
After the client end is communicated:



Mobile Channel-Ends

```
PROC generator (CHAN BUF.MGR! cli.chan!,
                CHAN BUF.MGR? svr.chan!)
  SEQ
    ...  create channel bundle
    cli.chan ! buf.cli
    svr.chan ! buf.svr
:
```

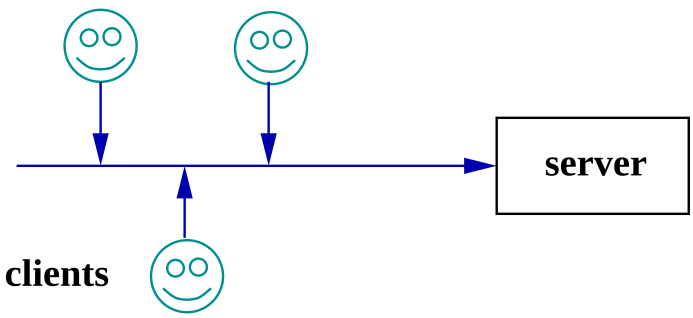
After the server end is communicated:



The 'client' and 'server' processes now communicate directly using the channel bundle

Securely Shared Channel-Ends

Channel-ends may also be declared *shared*, enabling the creation of multi-client multi-server process networks




```
SHARED BUF.MGR! buf.cli:
BUF.MGR? buf.svr:
SEQ
  buf.cli, buf.svr := MOBILE BUF.MGR
  PAR
    server (buf.svr)
    ..  client processes using buf.cli
```

Securely Shared Channel-Ends

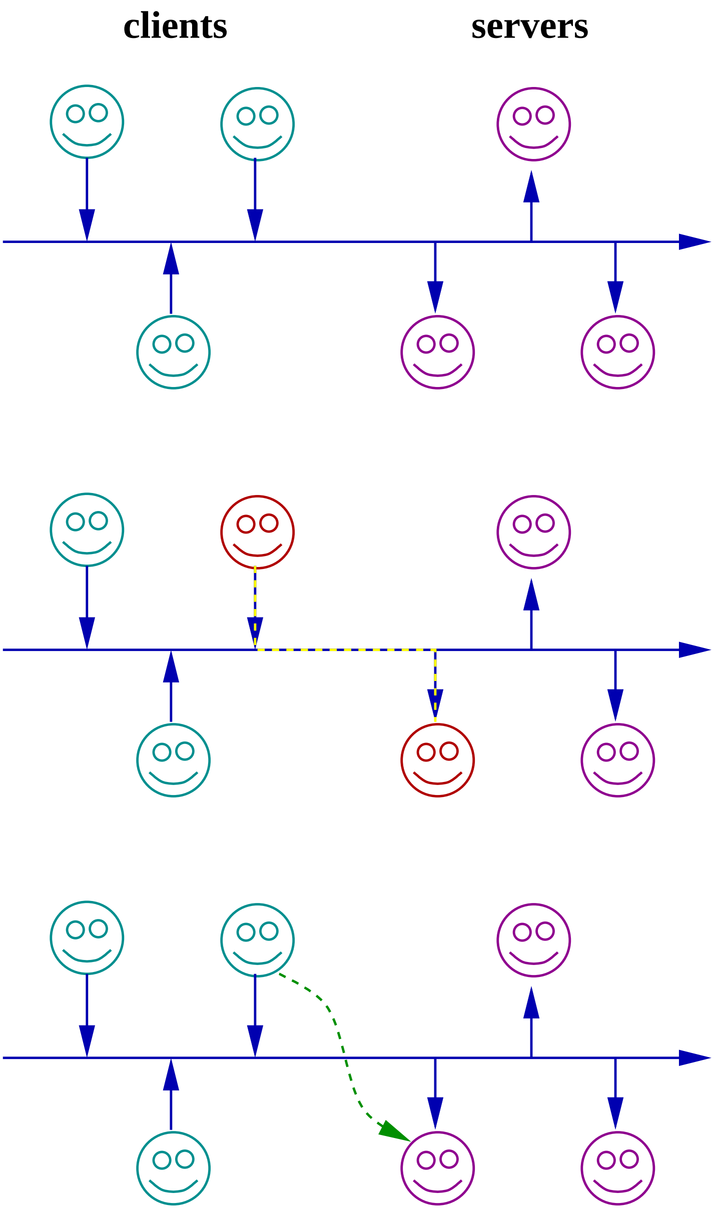
- Processes using *shared* ends must *claim* that end before using the channels within
- A channel-end may not be moved whilst claimed

The process body of a simple client could be:

```
MOBILE []BYTE buffer:
CLAIM to.svr
  SEQ
    to.svr[req] !  ...
    to.svr[buf] ?  buffer
    ...  use ‘buffer’
    to.svr[ret] !  buffer
```

But this prevents other clients/servers from interacting...

Securely Shared Channel-Ends



Dynamic Process Creation

(using the FORK)

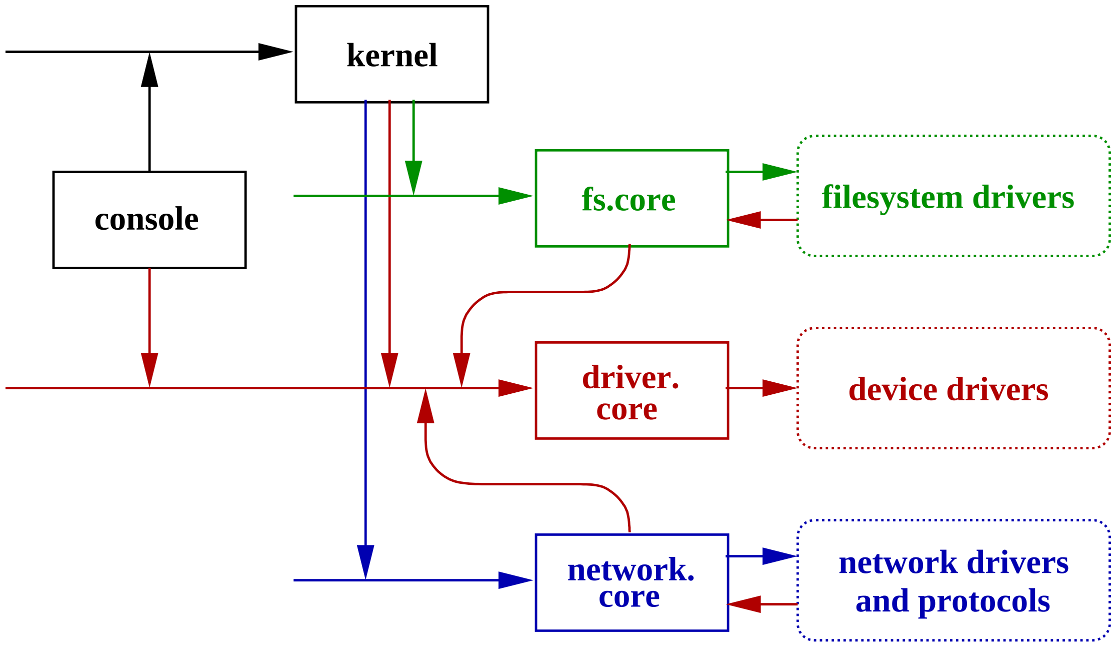
- Allows a process to dynamically create another process that runs concurrently
- Implementation supports the FORK of an arbitrary PROC, with parameter restrictions

```
WHILE test
  SEQ
    ...
    FORK P (n, in?, out!)
    ...
```

- Parameters to FORKed processes follow a *communication* semantics, instead of re-naming semantics
- An alternative syntax would be easy:

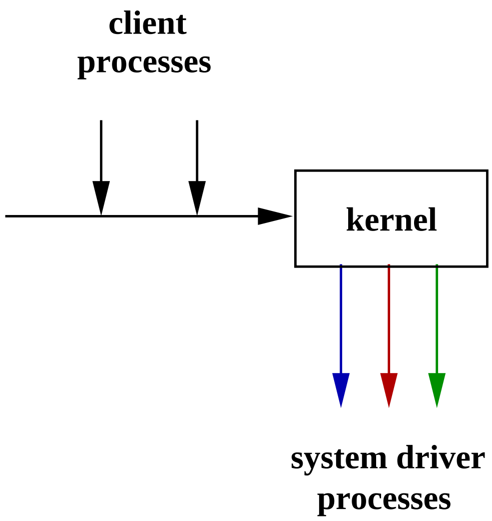
```
P ! (n, in?, out!)
```

Design of RMoX



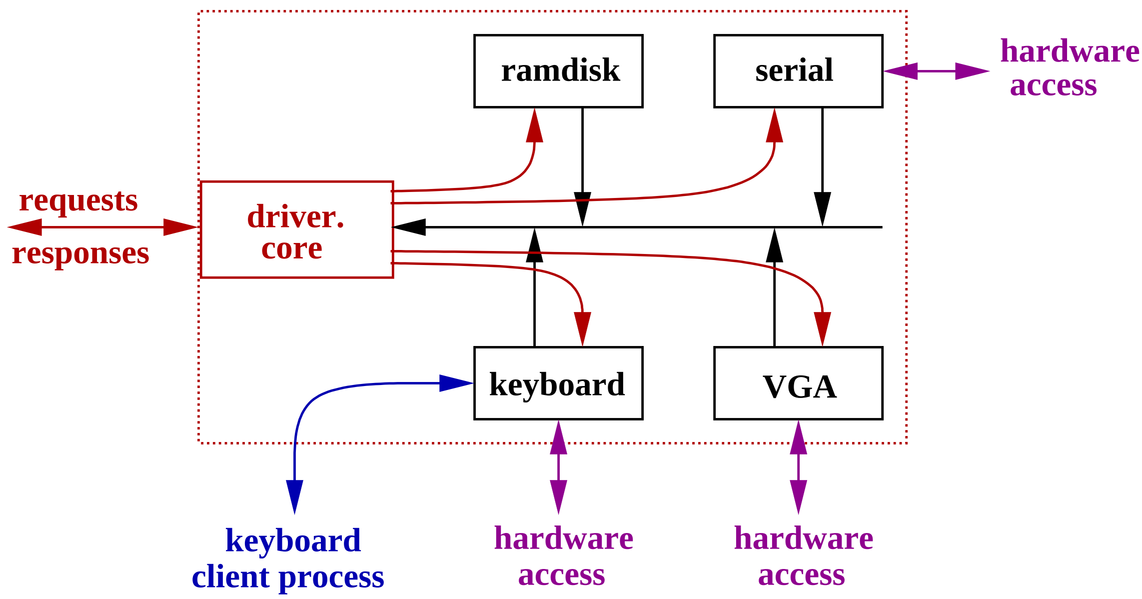
- Logical structure — client-server
- Built using channel-types and `FORK`
- Built with 'plug-in' components
- Scalable (as many components as needed)
- Dynamically extensible/degradable
- Fast ($<$ 100ns comms/cxt.sw.) [P3-800]

The Kernel



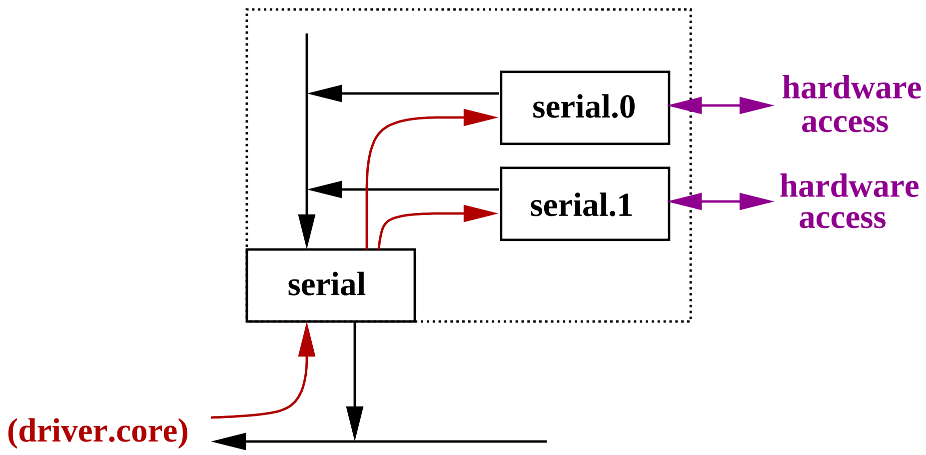
- Acts as an 'arbitrator' for system services
- POSIX based interface
  - but others are certainly possible
  - DOS, VMS, ...
- Has special privileges for 'process control'

Device Drivers



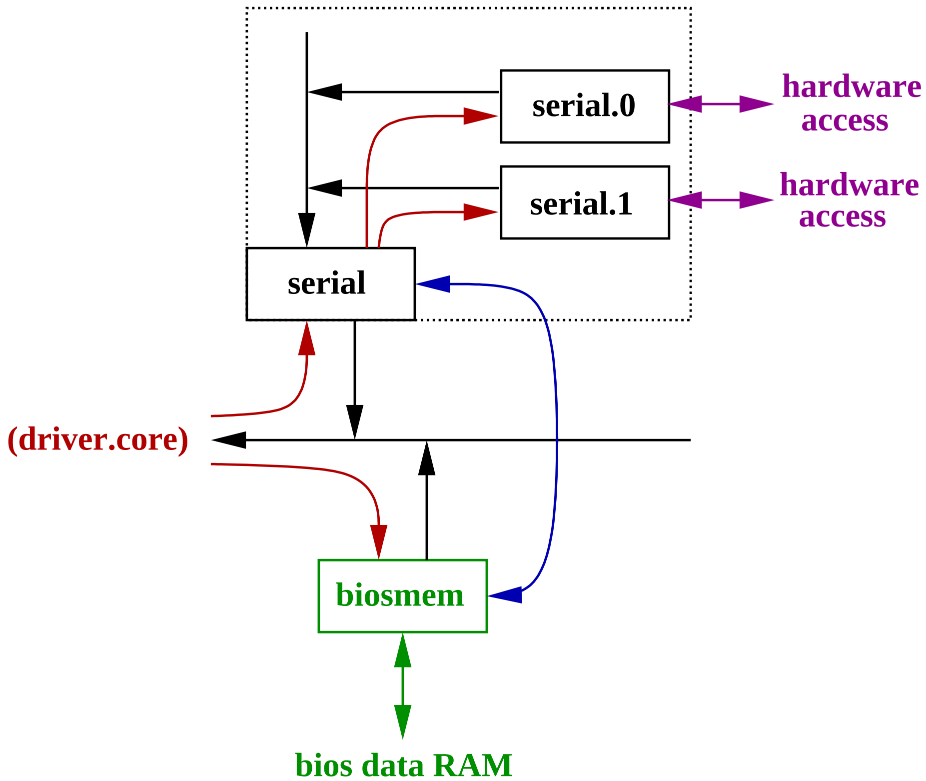
- ‘driver.core’ manages individual drivers – started by `FORK`ing them
- Requests for a specific device are passed to the right driver
- Driver returns a channel-type-end for interacting with the device
- Recursive channel-type is used to *snap back* the channel in on itself when done

Device Drivers II



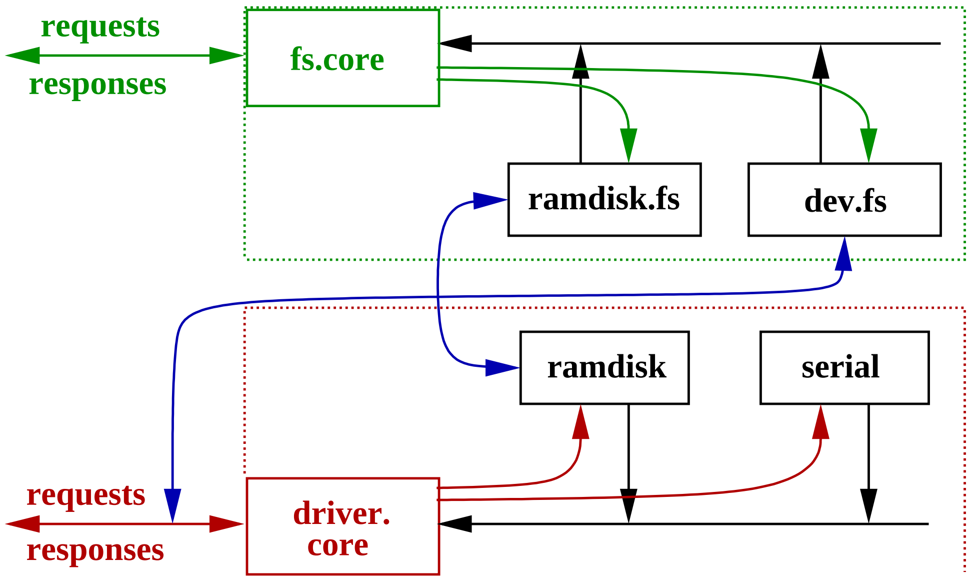
- A top-level driver may actually be a network of sub-processes
- Device naming scheme routes requests appropriately
- Useful for hierarchical device structures:
  - USB, IEEE-1284 (parallel), ...

Device Drivers III



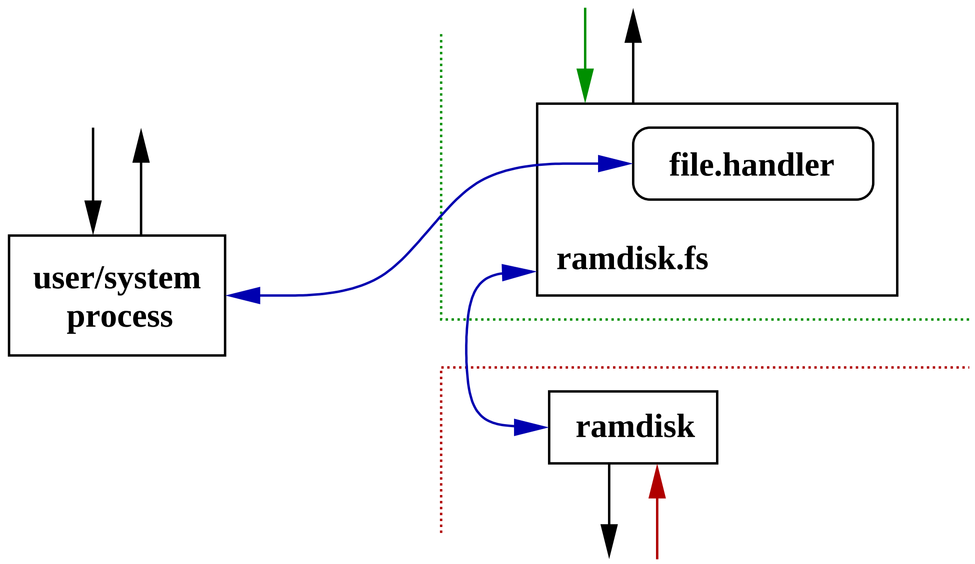
- Drivers may request connections to other device drivers
- Carefully controlled to avoid deadlock

Filesystems



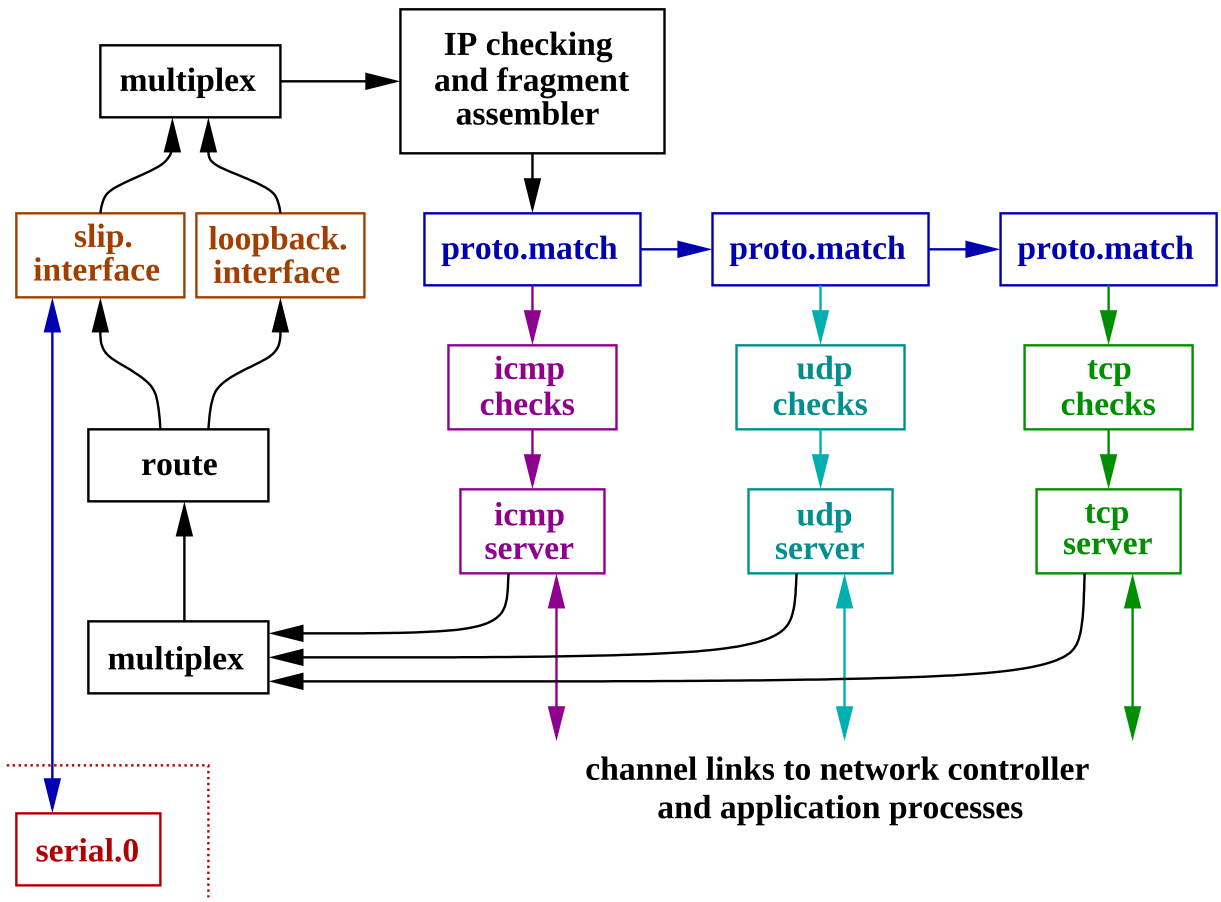
- Most filesystems will need a device to operate on
- Currently implementing a fairly traditional UNIX-style file-system
- The ramdisk is mounted on / when the system starts up

File handling



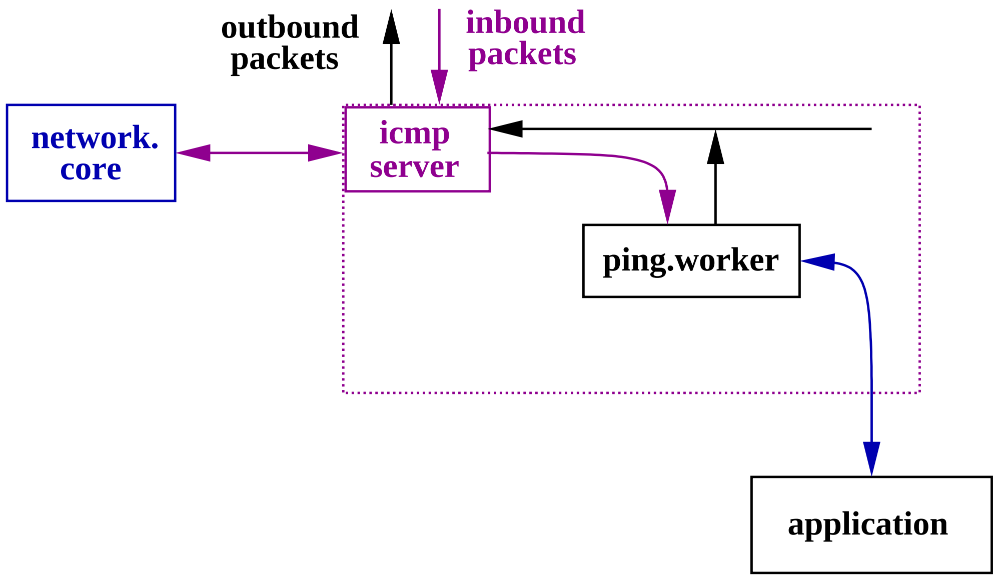
- When a file or directory is opened, 'ramdisk.fs' `FORK`s a handler process
- Handler processes (file or directory) return a *client channel-end*
- Handler services all requests on the file or directory by using that channel end.
- Channel-end *snapped-back* to finish

Networking



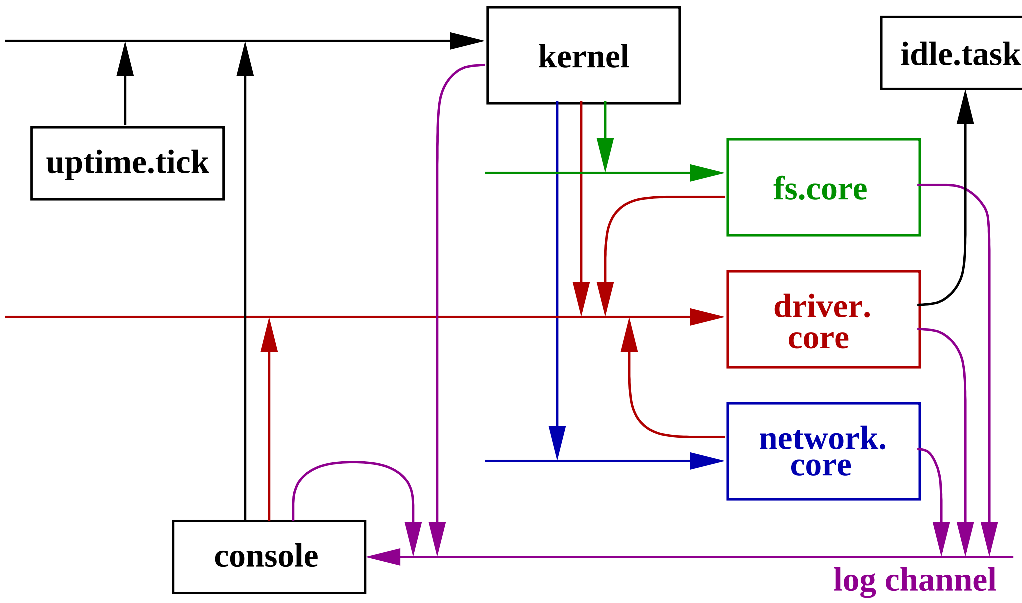
- Network infrastructure is from "occamnet" – a 3rd year project at UKC
- Clear and logical design

Networking II



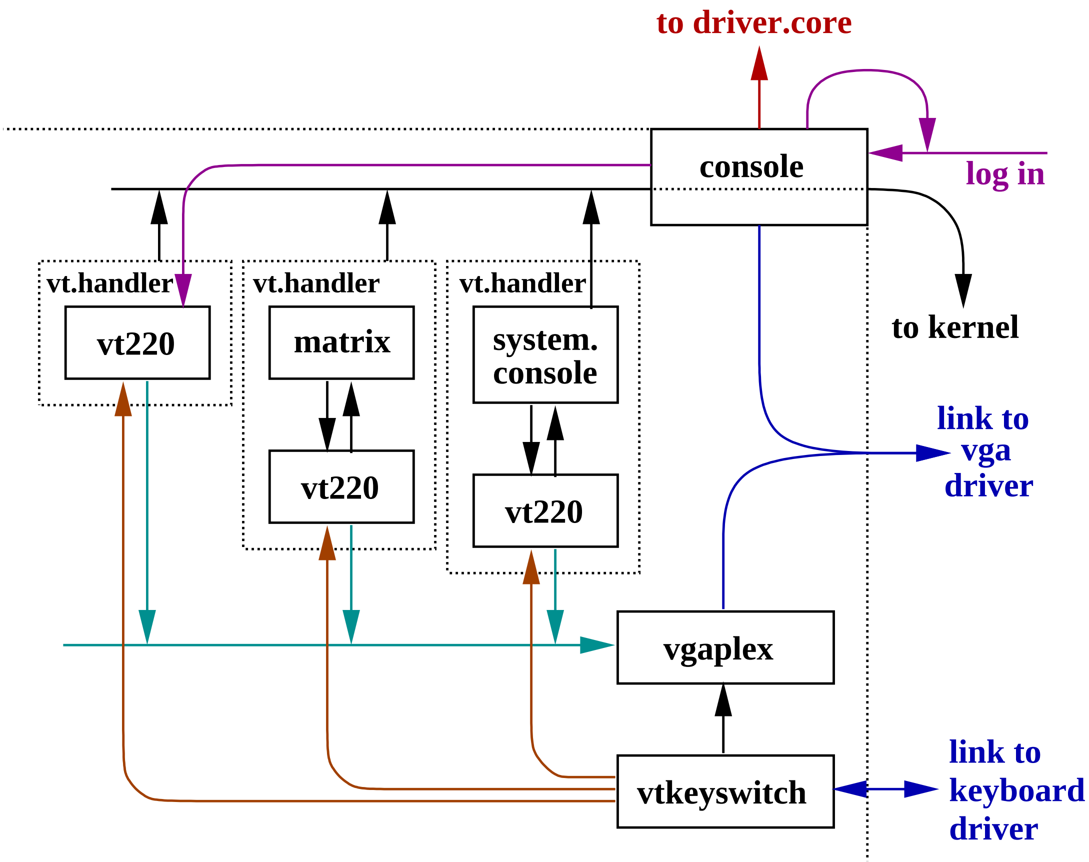
- Internal server-process structure similar to other RMoX server components

- Currently supported protocols are limited to ICMP and UDP
  - a simple DNS application exists
  - NFS is a larger, but feasible, project




```
PROC main ()
  CT.DRIVER? ct.driver.svr:
  SHARED CT.DRIVER! ct.driver.cli:
  ... other declarations
  SEQ
    ct.driver.cli, ct.driver.svr := MOBILE CT.DRIVER
    ... other initialisations
    SHARED CHAN BYTE debug.chan:
    PAR
      driver.core (ct.driver.svr, log.cli, debug.chan!)
      kernel (ct.kernel.svr, ct.driver.cli, ct.fs.cli,
              ct.network.cli, log.cli)
      ... other processes
      SEQ
        SETPRI (31)
        CLAIM debug.chan?
          idle.task (debug.chan?)
:
```

The console

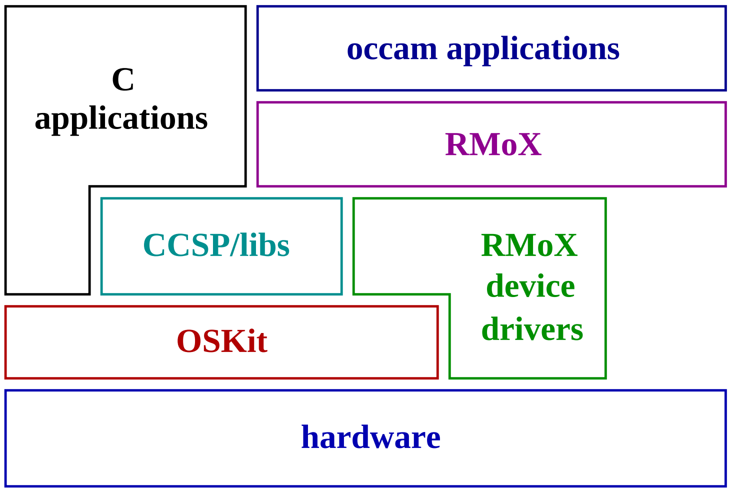


- Starts everything else off
- Handles the kernel log
- 'system.console' process provides prompt and basic commands

Implementation

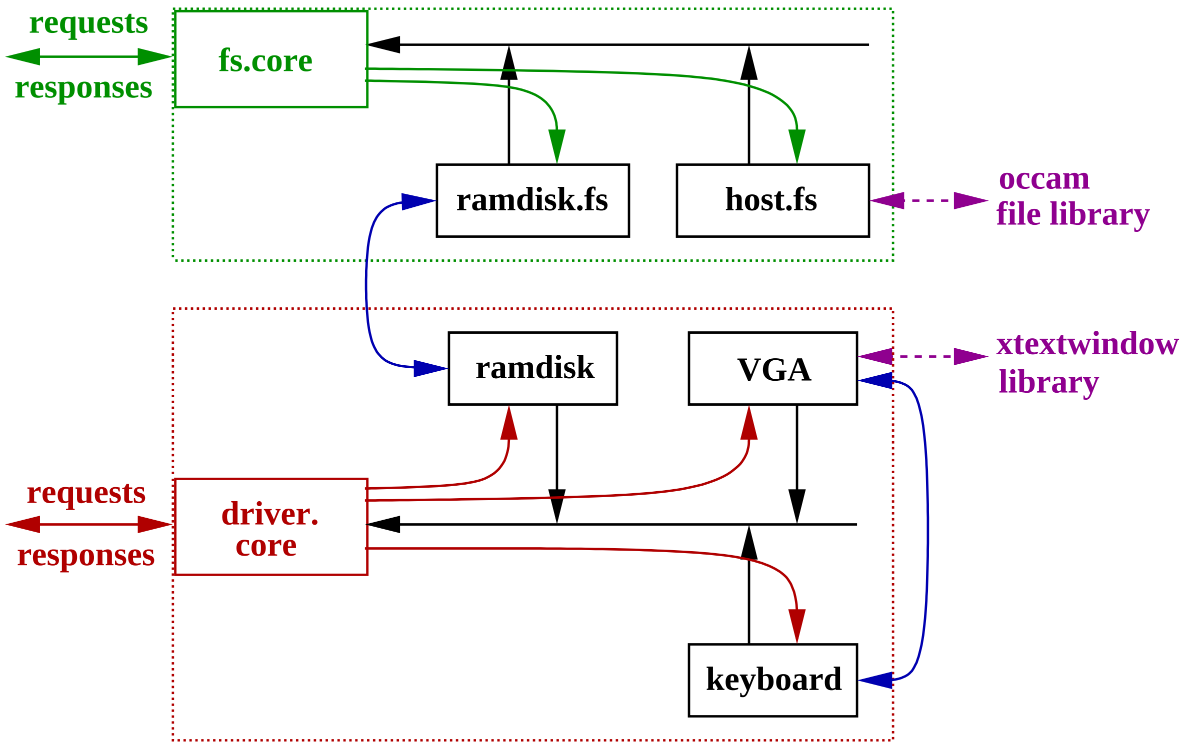
- Hardware is accessed in one of three ways:
  - IO port reads/writes
  - memory-mapped devices
  - interrupts
- Some devices may require polling:
  - not generally recommended
  - possible, with minor efficiency loss, through the use of timeouts
- RMoX requires a lower-level interface for managing these

Structure



- OSKit manages memory, IO ports and interrupts
- Once ‘allocated’, memory and IO ports can be used directly from occam
- CCSP schedules occam processes
  – and controls execution of C processes
- Interrupts are handled using a combination of the OSKit and CCSP

User-Mode RMoX



- RMoX is simply a dynamic set of communicating occam processes
  - ... and can be run as a normal KRoC/occam application
- User-mode RMoX (UM-RMoX) provides an abstraction for hardware, using the existing RMoX interfaces

User-Mode RMoX II

- Allows the core components of RMoX to be developed in a 'regular' environment:
  - good test for language extensions
  - significantly reduces test-debug time

- Cannot provide real hardware
  - emulation is only an approximation
  - IO port accesses are possible using `iopl()`

- Gives wider access for development and experimentation
  - student projects, ...

On-going Research

- Still under development, but progressing well (delayed by thesis)
- Reasonable number of basic devices now supported:
  - serial driver works well enough to launch a 'system.console' process on it
  - using FIFOs and interrupt-driven communication
- Basic file-system driver support:
  - 'ram.fs' and 'dev.fs' are implemented and work correctly
  - device filesystem mirrors the nested structure of devices
- Very fast! :-)