A Process Oriented Approach to USB Driver Development

Fred Barnes and Carl Ritson, Systems Research Group
Computing Laboratory, University of Kent, UK
{ F.R.M.Barnes , C.G.Ritson }@kent.ac.uk

University of Kent

Contents

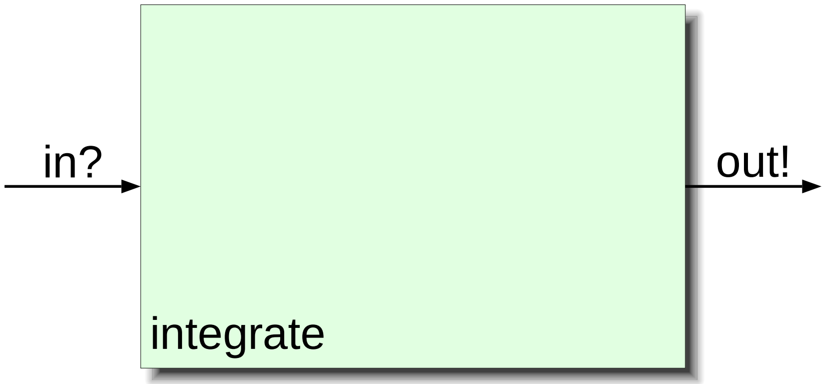
- A brief introduction to occam-π
- The RMoX operating system
- USB hardware
- Process-oriented USB (in layers)
- Conclusions and further work

A Brief Introduction to occam-pi

- **Process oriented** language: systems built from layered networks of communicating processes
  - semantics primarily from Hoare's **CSP** (communicating sequential processes)
  - incorporates ideas of **mobility** from Milner's **π-calculus**
- Language elements include:
  - **channels**: one-to-one, one-to-any, any-to-one, any-to-any synchronous unbuffered communication
  - **barriers**: synchronisation between multiple processes (CSP event)
  - **mobiles**: movement semantics for **data**, **channel-ends**, **processes**
  - **dynamic process creation**: for building dynamically evolving systems
- Strong **formal** concurrency mechanisms make occam-π suitable for building many types of system, both simple and complex

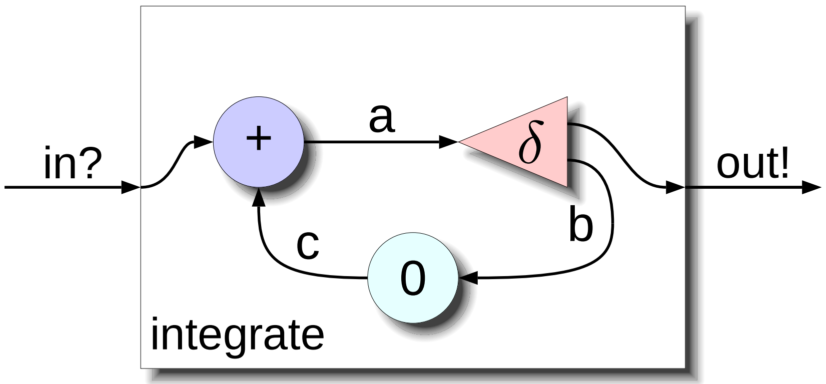
Simple Processes

- Serial integrator:




```
PROC integrate (CHAN INT in?, out!)
  INITIAL INT total IS 0:
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      total := total + x
      out ! total
:
```

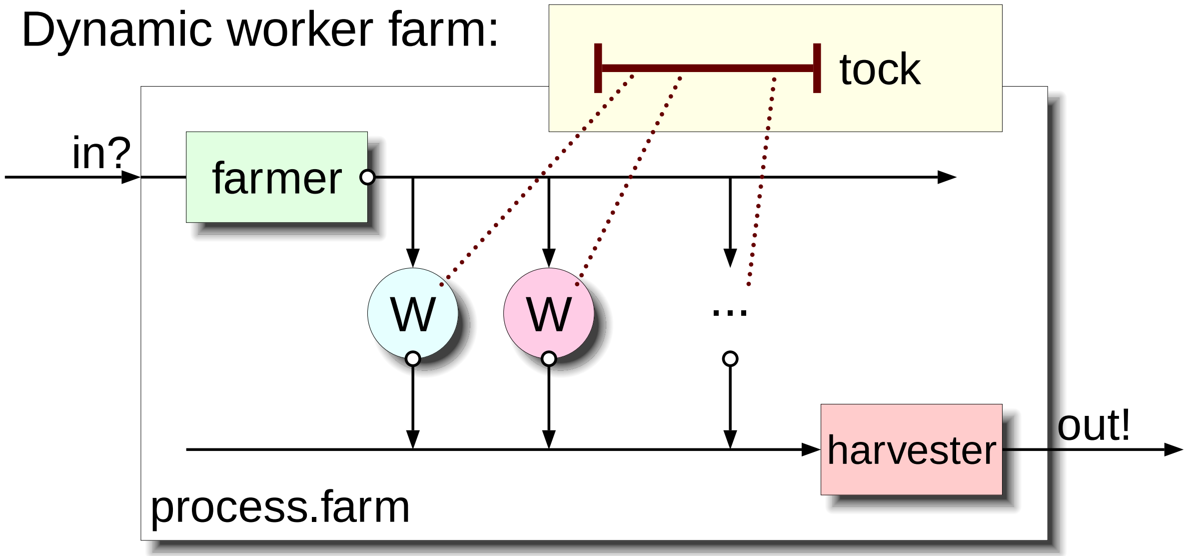
- Parallel integrator:

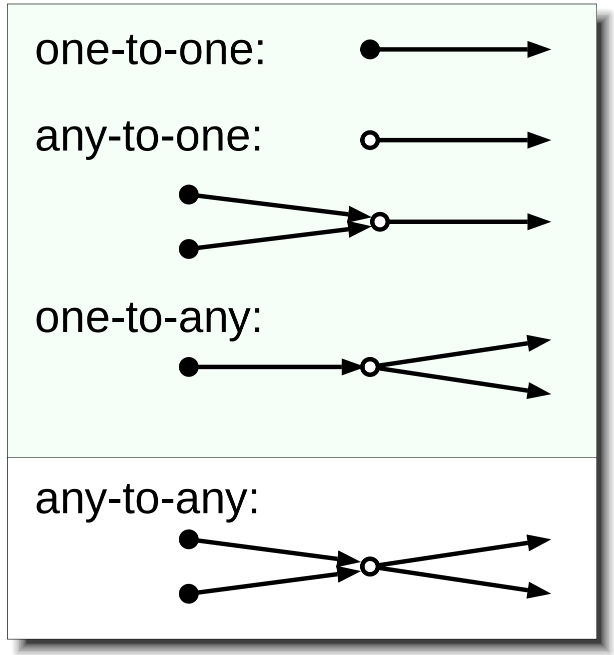



```
PROC integrate (CHAN INT in?, out!)
  CHAN INT a, b, c:
  PAR
    plus (in?, c?, a!)
    delta (a?, out!, b!)
    prefix (0, b?, c!)
:
```

Less Simple Processes

- Dynamic worker farm:





- Channel-type primer:
  - bundles of channels declared as a **record-type**
  - **mobile** ends (moved around for network reconfiguration)
  - both **shared** and **unshared** ends (supporting one-to-one, ..., any-to-any)
- Generally refer to the two ends as **client** and **server**, denoted '!' and '?' respectively — because that's the most common usage pattern, but not enforced

occam-pi and Related Tools

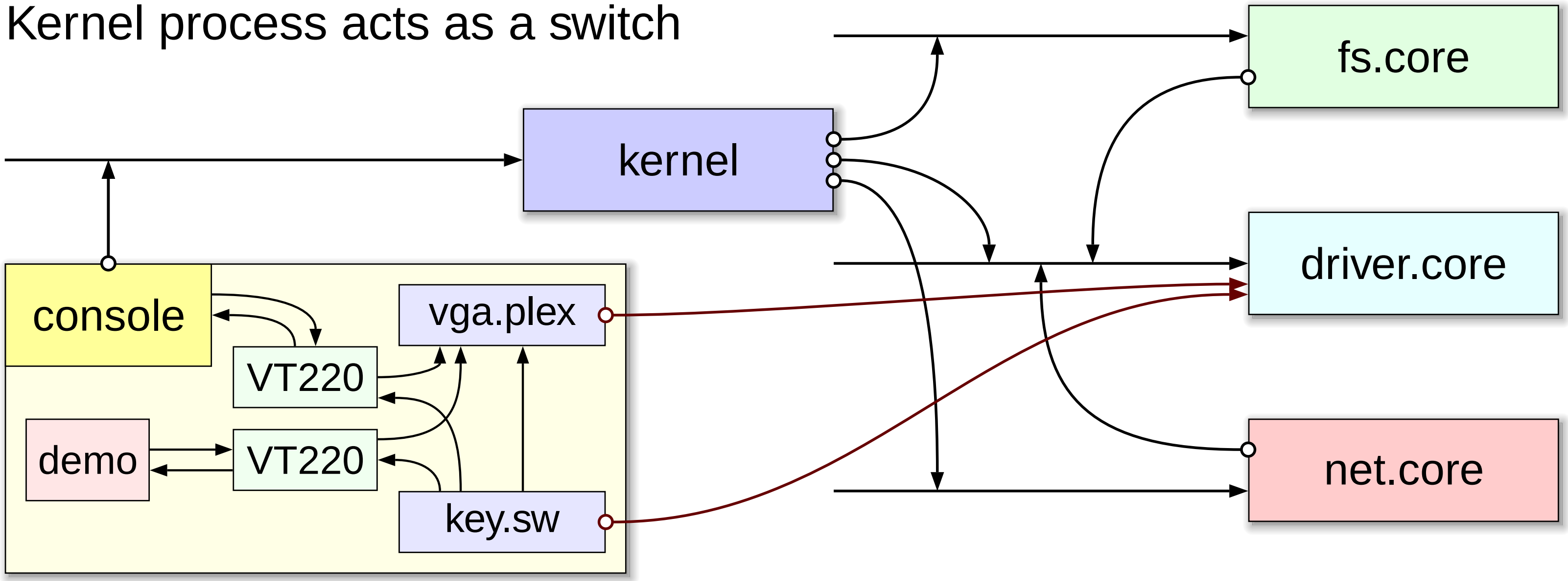
- For occam-π, have:
  - **KRoC**: compiles down to i386 native code, ˜50ns context-switch
  - **Transterpreter**: portable interpreter, ˜1us context-switch
- For expressing CSP ideas in other languages:
  - **JCSP** and **CTJ** for Java
  - **C++CSP** for C++
  - similar things cropping up here and there (concurrent programming is of increasing interest, again)
- The memory footprint for occam-π parallel processes is comparatively small:
  - can handle **millions** of simple processes on a modern desktop PC

The RMoX Operating System

- RMoX is an operating-system built using occam-π, utilising concurrency at its lowest level — currently for Intel Pentium based hardware
- EPSRC funded project (EP/D061822/1) to develop RMoX for PC/104 systems
- Theory is: build an OS out of layered networks of communicating concurrent processes, and it will be:
  - **scalable**: from small embedded systems, through general-purpose (desktop) computers, up to massively parallel supercomputers
  - **reliable**: freedom from race-hazard and aliasing errors
  - **efficient**: low overheads (sub 100ns context-switch), no need for heavyweight memory-management (maybe)

The RMoX Operating System

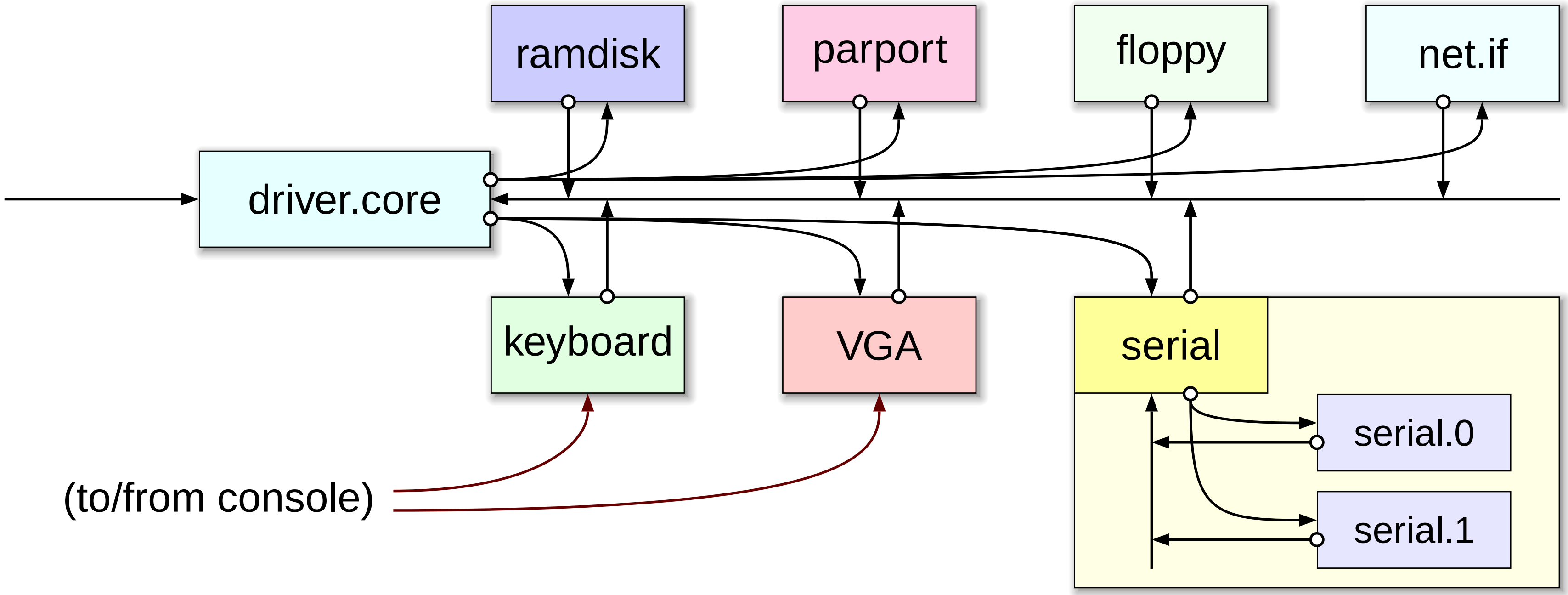
- System design is fairly straightforward — operating-systems provide services
- Kernel process acts as a switch



- Console process provides a basic user-interface
- Connections within dynamically formed and dismantled as required

The RMoX Operating System

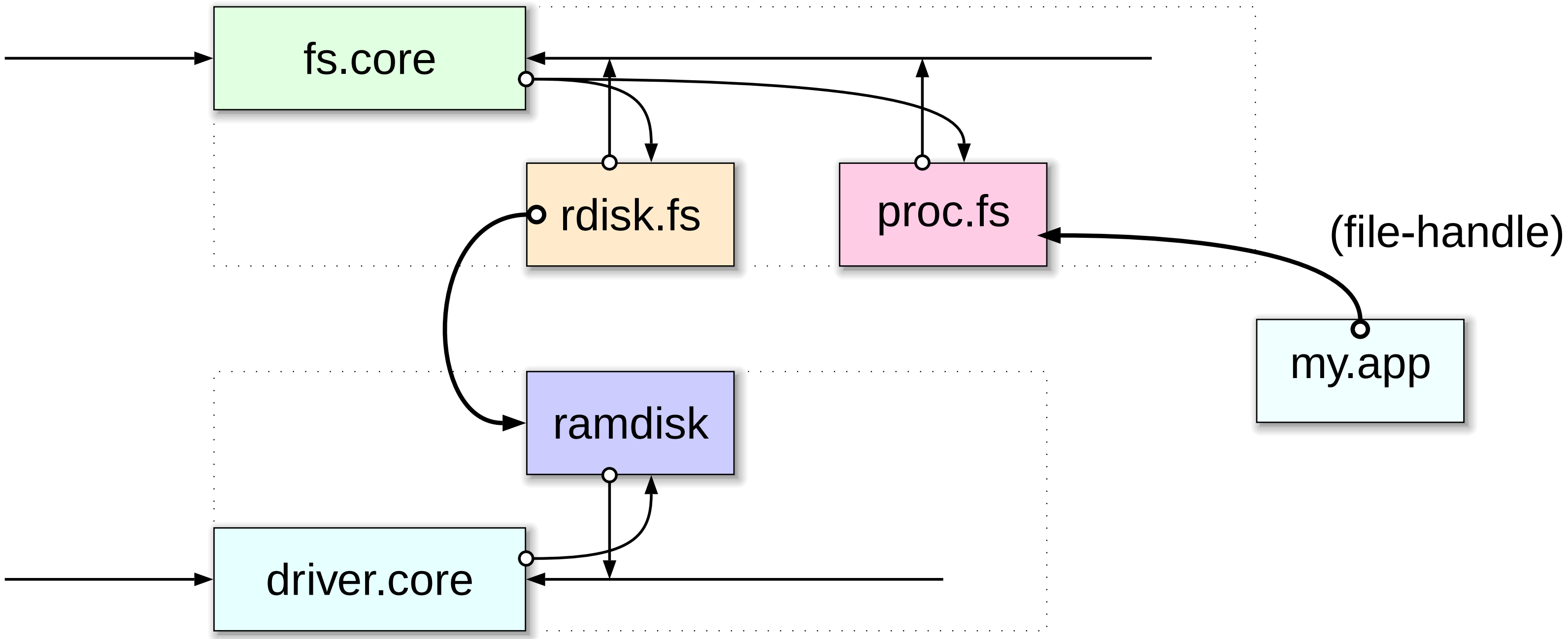
- The various “core” components utilise internal concurrency



- Drivers themselves may be concurrent internally
- Some drivers mostly complete, others under construction

The RMoX Operating System

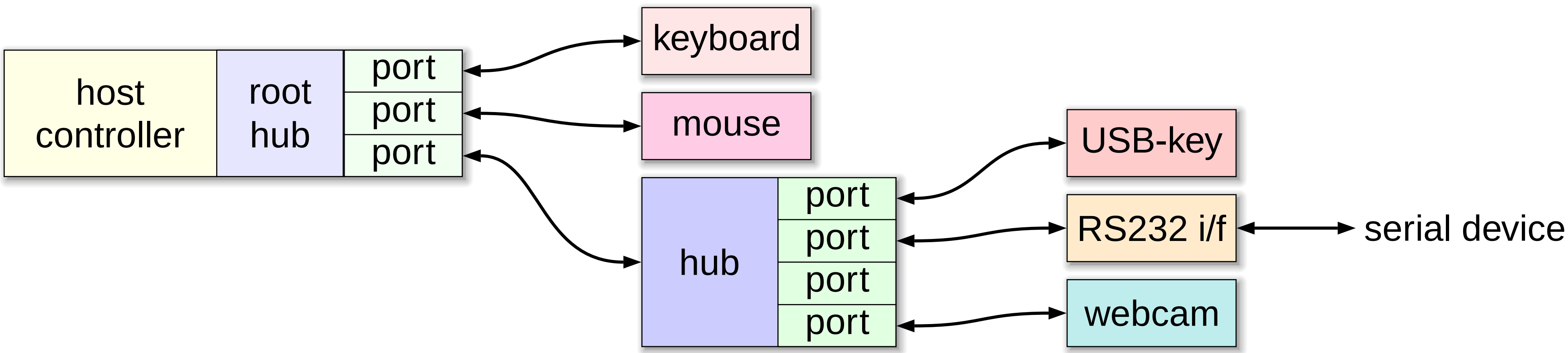
- Largely a **client-server** architecture internally — deadlock free



- Device drivers are the bottom-level 'server' components
- Process networks for things such as PCI and USB drivers reflect hardware organisation — not a complex mass of sequential code!

USB Hardware

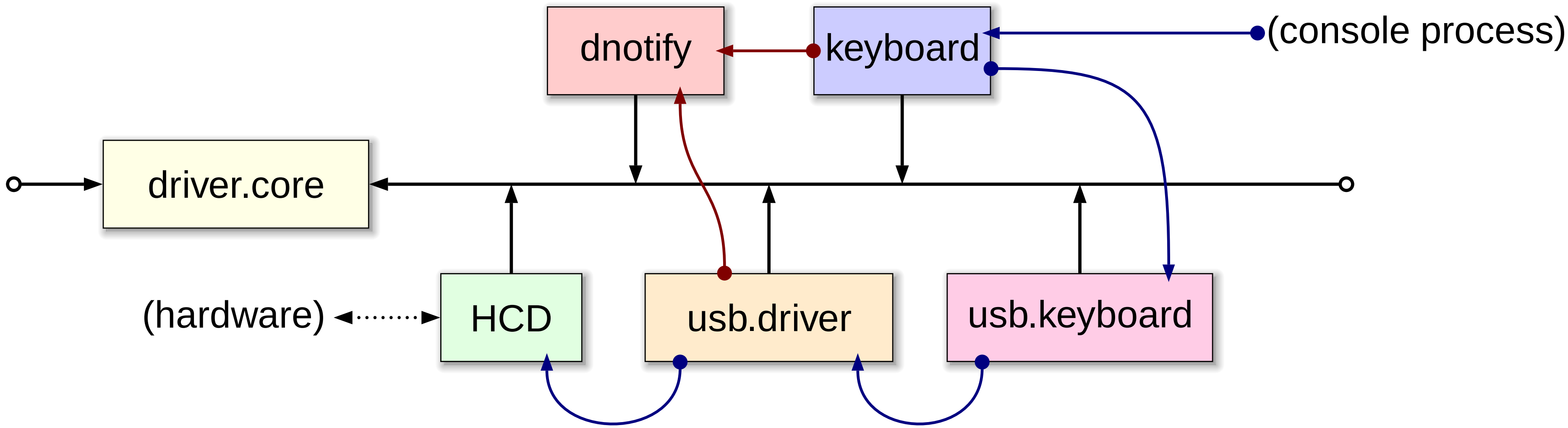
- USB, the **Universal Serial Bus**, is a 4-wire half-duplex peripheral interconnect
  - supports devices at three speeds: 1.5, 12 and 480 MBps
  - bus is strictly controlled: single **host controller** polls devices and offers bandwidth for transfers — UHCI, EHCI and OHCI standards say how
- Hardware structure is a tree:



- Devices can be added and removed randomly — software must cope!
- Range of devices is complex: simple peripherals, legacy interfaces, networking, ...

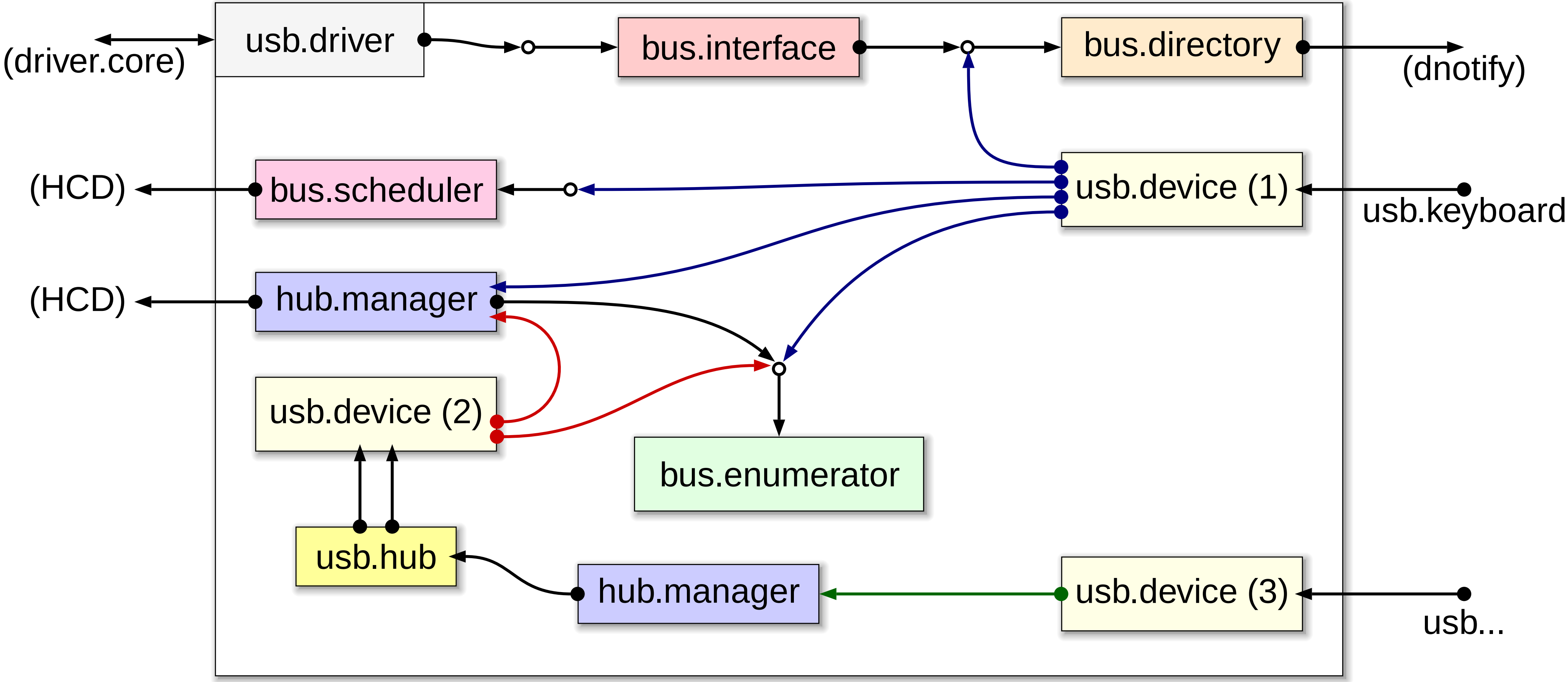
Process Oriented USB

- USB infrastructure lies mostly in the “driver.core”:



- **Device-notify** driver acts as a registration point, indicating when new USB devices are connected — routes connections between relevant device
- The **usb.driver** acts as a coordinator, with processes representing the physical structure — created and destroyed dynamically

Process Oriented USB

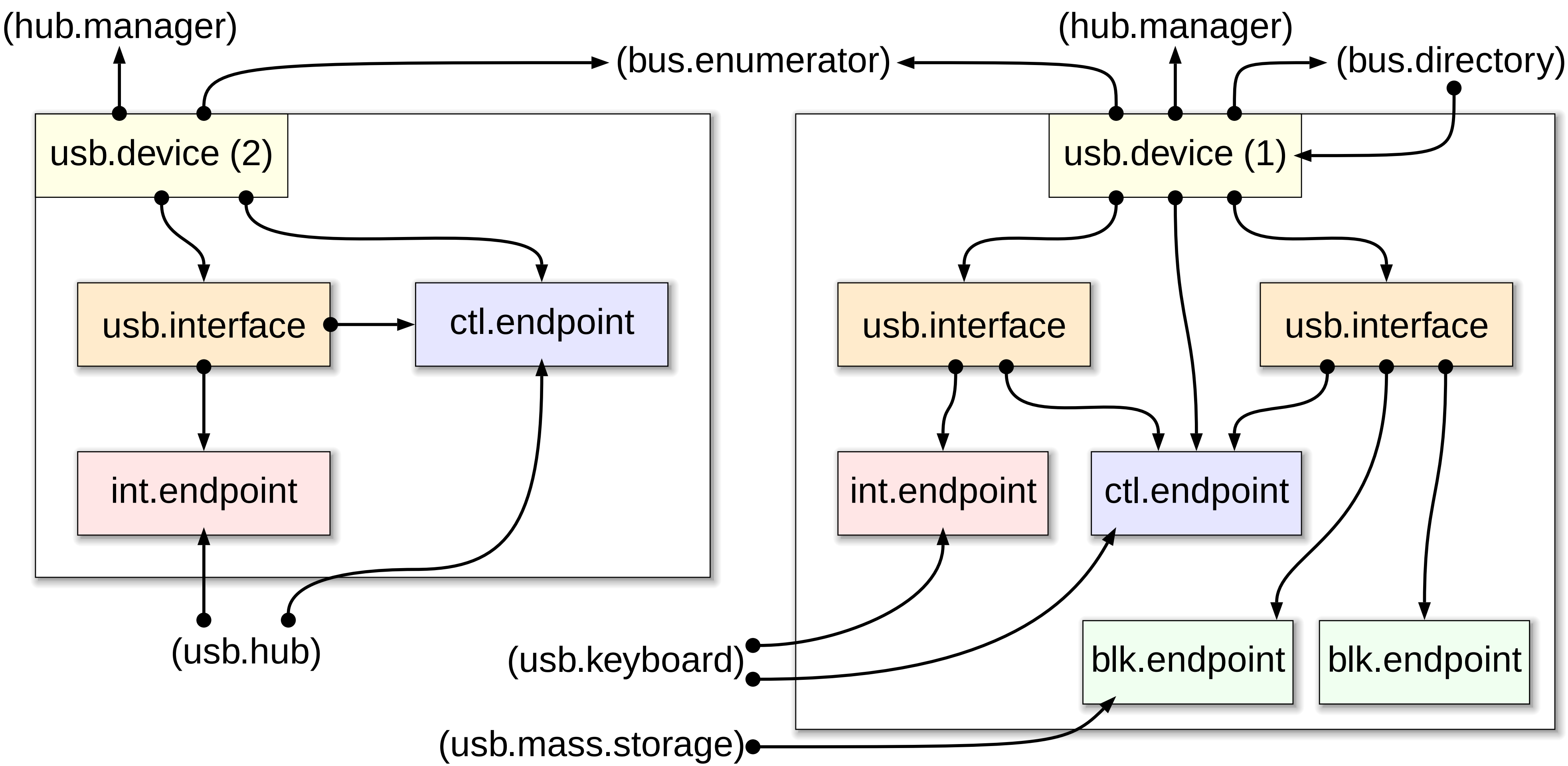


(some internal connections omitted)

- Structure grows and shrinks dynamically as devices are added and removed

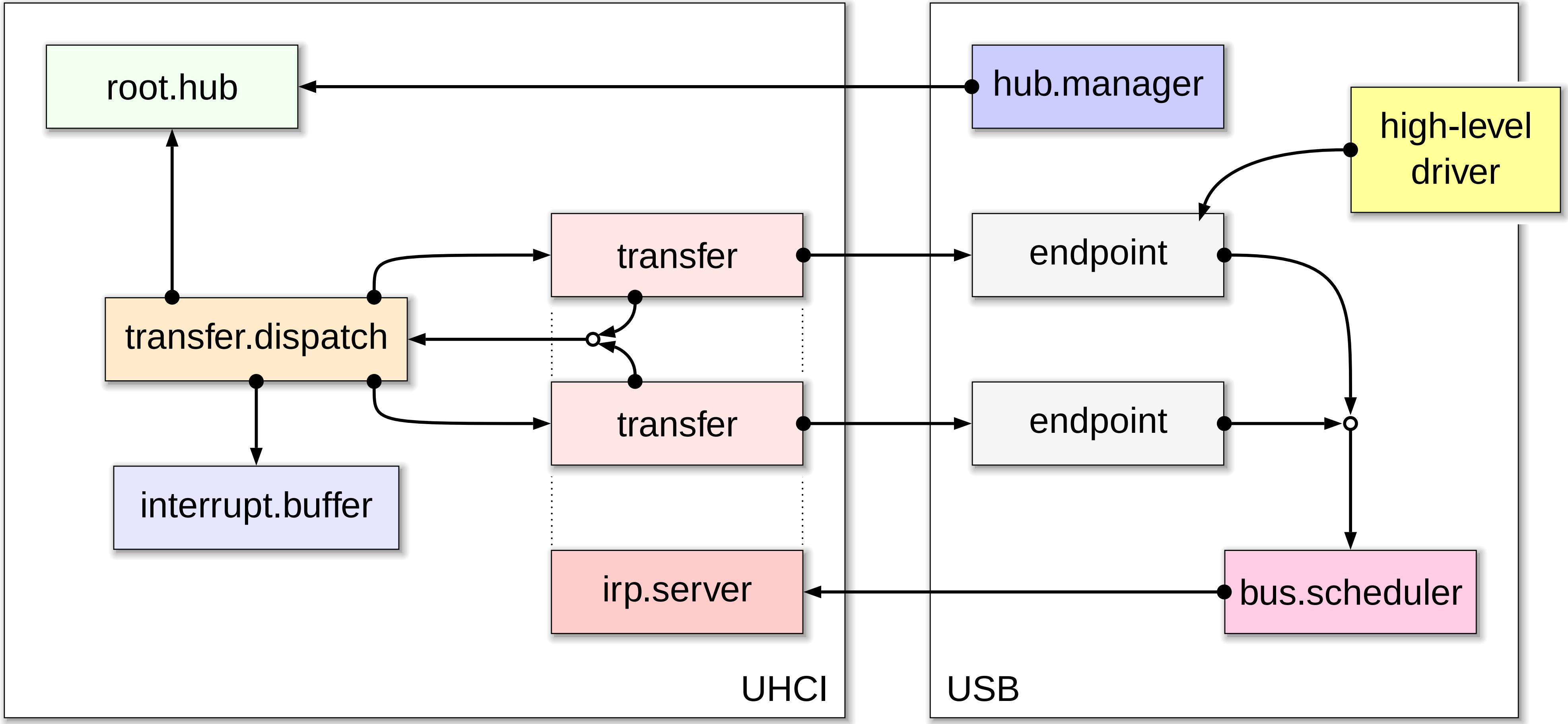
Process Oriented USB

- The **usb.device** processes have their own internal structure, reflecting the logical structure of USB devices (**interfaces** and **endpoints**):



Process Oriented USB

- The last puzzle piece is the linkage between the various **endpoint** processes and the underlying **host controller**:



Conclusions

- The resulting process network for any modest USB setup will be non-trivial (several hundred processes)
  - but built safely from **simple** self-contained components, assembled in a way which is **understandable**
- Software structure reflects hardware organisation
- Concurrency is a significant advantage for programming — a single device-driver (e.g. USB keyboard) can interact with multiple endpoints without complex coding
- Some care required in shut-down when a device is unplugged
  - dummy processes **forked** to service requests, preventing deadlock

Further Work

- USB stack itself works well (and, we expect, efficiently)
- Currently lacking higher-level components to use the underlying devices
  - have a working **USB keyboard** and can drive **USB audio** devices with wave data from **USB storage** devices (raw-blocks)
- Usage example is given in the paper, including the code that USB device-drivers use to connect and interface to the bus
- Ongoing development for **multi-processor** support, next steps will be proper **file-system** support
- Essentially open-source software, so any development effort very welcome!
- Questions?