Communicating Process Architectures 2007
Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch
IOS Press, 2007
© 2007 The authors and IOS Press. All rights reserved.

A Process Oriented Approach to USB Driver Development

Carl G. RITSON and Frederick R.M. BARNES

Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NF, England.
{cgr,frmb}@kent.ac.uk

Abstract. Operating-systems are the core software component of many modern computer systems, ranging from small specialised embedded systems through to large distributed operating-systems. The demands placed upon these systems are increasingly complex, in particular the need to handle concurrency: to exploit increasingly parallel (multi-core) hardware; support increasing numbers of user and system processes; and to take advantage of increasingly distributed and decentralised systems. The languages and designs that existing operating-systems employ provide little support for concurrency, leading to unmanageable programming complexities and ultimately errors in the resulting systems; hard to detect, hard to remove, and almost impossible to prove correct.

Implemented in occam-π, a CSP derived language that provides guarantees of freedom from race-hazards and aliasing error, the RMoX operating-system represents a novel approach to operating-systems, utilising concurrency at all levels to simplify design and implementation. This paper presents the USB (universal serial bus) device-driver infrastructure used in the RMoX system, demonstrating that a highly concurrent process-orientated approach to device-driver design and implementation is feasible, efficient and results in systems that are reliable, secure and scalable.

Keywords. occam-pi, operating-systems, RMoX, concurrency, CSP, USB, embedded-systems, PC104

Introduction

The RMoX operating-system, previously presented at this conference [1], represents an interesting and well-founded approach to operating-systems development. Concurrency is utilised at the lowest level, with the operating-system as a whole comprised of many interacting parallel processes. Compared with existing systems, that are typically sequential, RMoX offers an opportunity to easily take advantage of the increasingly multi-core hardware available — it is scalable. Development in occam-π [2,3], based on CSP [4] and incorporating ideas of mobility from the π-calculus [5], gives guarantees about freedom from race-hazard and aliasing error — problems that quickly become unmanageable in existing systems programmed using sequential languages (which have little or no regard for concurrency), and especially when concurrency is added as an afterthought.

Section 1 provides an overview of the RMoX system, its motivation, structure and operation. Section 2 provides a brief overview of the USB hardware standard, followed by details of our driver implementation in section 3. An example showing the usage of the USB driver is given in section 4, followed by initial conclusions and consideration for future and related work in section 5.

1. The RMoX Operating System

The RMoX operating-system is a highly concurrent and dynamic software system that provides an operating-system functionality. Its primary goals are:

- *reliability*: that we should have some guarantees about the operation of the system components, possibly involving formal methods.
- *scalability*: that the system as a whole should scale to meet the availability of hardware and demands of users; from embedded devices, through workstations and servers, to massively parallel supercomputers.
- *efficiency*: that the system operates using a minimum of resources.

The majority of existing operating-systems fail to meet these goals, due largely to the nature of the programming languages used to build them — typically C. Reliability within a system utilising concurrency requires that we have a solid understanding of that concurrency, including techniques for formal reasoning. This is simply not the case for systems built with a *threads-and-locks* approach to concurrency, as most operating-systems currently use. The problem is exasperated by the use of 3rd-party code, such as device-drivers provided by specific hardware vendors — the OS cannot guarantee that code being "plugged in" interacts in a way that the OS expects. Getting this right is up to the hardware vendor's device-driver authors, who are unlikely to have access to every possible configuration of hardware and other device-drivers which the OS uses, in order to test their own drivers.

Scalability is a highly desirable characteristic for an OS. Most existing operating-systems are designed with specific hardware in mind, and as such, there is a wealth of OSs for a range of hardware. From operating-systems specific to embedded devices, through general-purpose operating-systems found on workstations and servers, up to highly concurrent and job-based systems for massively-parallel supercomputers. Unfortunately, most operating-systems fail to scale beyond or below the hardware for which they were originally intended. Part of the scalability problems can be attributed to concurrency — the mechanisms that existing systems use to manage concurrency are themselves inherently unscalable.

A further issue which RMoX addresses is one of efficiency, or as seen by the user, performance. *Context-switching* in the majority of operating-systems is a notoriously heavyweight process, measured in thousands of machine cycles. Rapid context-switching is typically required to give 'smooth' system performance, but at some point, the overheads associated with it become performance damaging. As such, existing systems go to great lengths to optimise these code paths through the OS kernel, avoiding the overheads of concurrency (specifically context-switches) wherever possible. The resulting code may be efficient, but it is hard to get right, and almost impossible to prove correct given the nature of the languages and concurrency mechanisms used. Furthermore, the OS cannot generally guarantee that loaded code is well behaved — either user processes or 3rd-party drivers. This results in a need for complex hardware-assisted memory protection techniques.

In contrast, the RMoX OS *can* make guarantees about the behaviour of foreign code — we insist that such code conforms. Fortunately, the occam-π compiler does this for us — it is one-time effort for the compiler writer. Clearly there are issues relating to *trust*, but those are orthogonal to the issues here, and are well addressed in other literature (related to security and cryptography). Having assurances that code running within the OS is well-behaved allows us to do away with many overheads. Most notably, the context-switch (including communication) can be measured in tens of machine cycles, orders of magnitude smaller than what currently exists. With such small overheads, we can use concurrency as a powerful tool to simplify system design. Furthermore, the resulting systems are scalable — we can run with as few or as many processes are required.

1.1. Structure

The structure of the RMoX operating-system is shown in figure 1, with detail for the "driver.core" shown. The network is essentially a client-server style architecture, giving a guarantee of deadlock freedom [6].

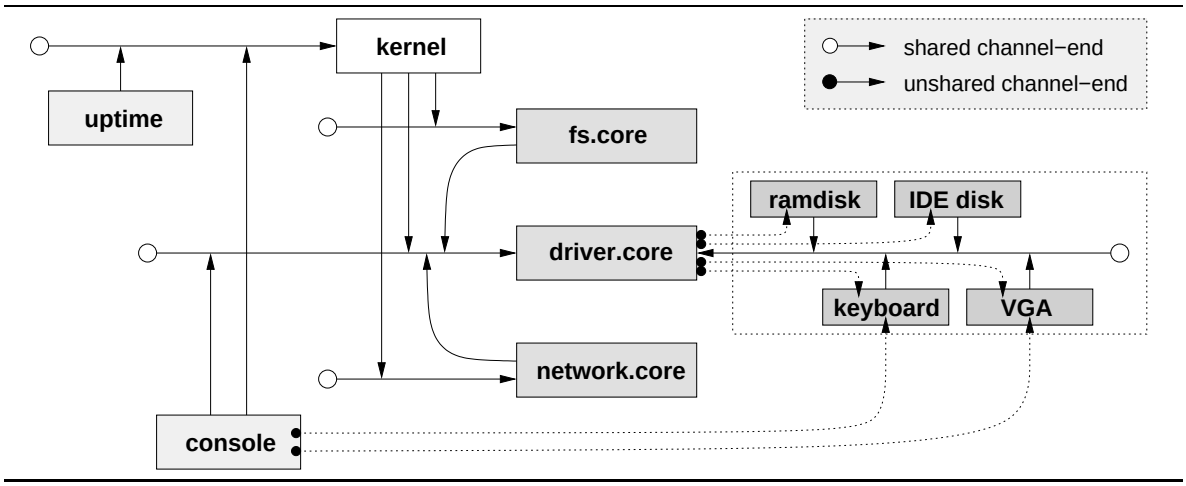

Figure 1. RMoX operating-system process network.

There are three core services provided by the RMoX system: device-drivers, file-systems and networking. These simply provide management for the sub-processes (or sub-process networks) that they are responsible for. When a request for a resource is made, typically via the 'kernel' process, the relevant 'core' process routes that request to the correct underlying device. Using mobile channels, this allows *direct* links to be established between low-level components providing a particular functionality, with high-level components using them. Protocols for the various types of resource (e.g. file, network socket, block device-driver) are largely standardised — e.g. a file-system driver (inside "fs.core") can interact with any device driver that provides a block-device interface. Since such protocols are well defined, in terms of interactions between processes, building pipelines of processes which layer functionality is no issue. Some consideration must be given to shutting these down correctly (i.e. without inducing deadlock); fortunately that process is well understood [7].

As the system evolves, links established between different parts of the system can result in a fairly complex process network. However, if we can guarantee that individual components interact with their environments in a 'safe' way (with a per-process analysis performed automatically by the compiler), then we can guarantee the overall 'safe' behaviour of the system — a feature of the compositional semantics of CSP as engineered into the occam-π language. This type of formalism is already exploited in the overall system design — specifically that a client-server network is deadlock free; all we have to do is ensure that *individual* processes conform to this.

The remainder of this paper focuses on the USB device-driver architecture in RMoX. Supporting this hardware presents some significant design challenges in existing operating-systems, as it requires a dynamic approach that layers easily — USB devices may be plugged-in and unplugged arbitrarily, and this should not break system operation. The lack of support for concurrency in existing systems can make USB development hard, particularly when it comes to guaranteeing that different 3rd-party drivers interact correctly (almost impossible in existing systems). RMoX's USB architecture shows how concurrency can be used to our benefit: breaking down the software architecture into simple, understandable, concurrent components; producing a design that is scalable, and an implementation that is reliable and efficient.

2. The Universal Serial Bus

The *Universal Serial Bus* (USB) [8,9] first appeared in 1996 and has undergone many revisions since. In recent years it has become the interface of choice for low, medium and high speed peripherals, replacing many legacy interfaces, e.g. RS232, PS/2 and IEEE1284. The range of USB devices available is vast, from keyboards and mice, through flash and other storage devices, to sound cards and video capture systems. Many classes of device are standardised in documents associated with the USB, these include human-interface devices, mass-storage devices, audio input/output devices, and printers. For these reasons adding USB support to the RMoX operating system increases its potential for device support significantly. It also provides an opportunity to explore modelling of dynamic hardware configurations within RMoX.

2.1. USB Hardware

The USB itself is a 4-wire (2 signal, 2 power) half-duplex interface, supporting devices at three speeds: 1.5 Mbps (low), 12 Mbps (full) and 480 Mbps (high). There is a single bus master, the host controller (HC), which controls all bus communication. Communication is strictly controlled — a device cannot initiate a data transfer until it has been offered the appropriate bandwidth by the HC. The topology of a USB bus is a tree, with the HC at the root. The HC provides a root hub with one or more ports to which devices can be connected. Additional ports can be added to the bus by connecting a hub device to one of the existing bus ports. Connected hubs are managed by the USB driver infrastructure, which maintains a consistent view of the topology at all times. Figure 2 shows a typical arrangement of USB hardware.

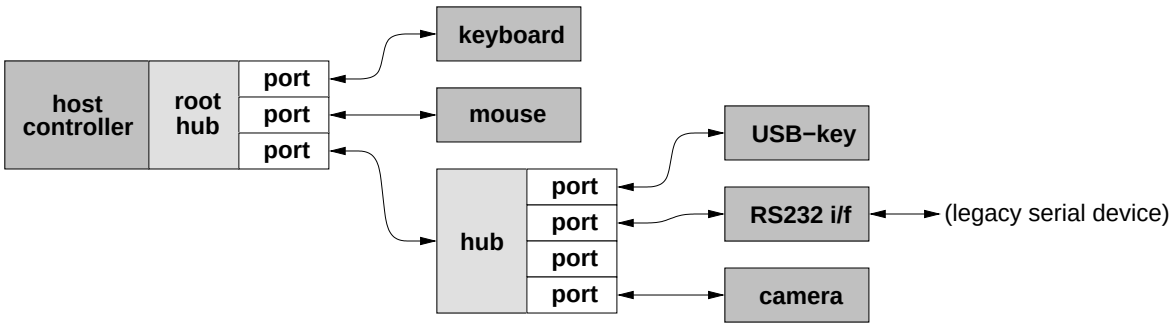

Figure 2. Example USB hardware tree.

Unlike more traditional system busses, such as PCI [10], the topology the USB is expected to change at run-time. For this and the reasons above, access to bus devices is via communication primitives provided by the USB driver infrastructure, rather than CPU I/O commands or registers mapped into system memory. Although it should be noted that this difference does not preclude the use of DMA (direct memory access) data transfers to and from bus devices.

2.2. USB Interfaces

Each device attached to the bus is divided into interfaces, which have zero or more endpoints, used to transfer data to and from the device. Interfaces model device functions, for example a keyboard with built-in track-pad would typically have one interface for the keyboard, and one for the track-pad. Interfaces are grouped into configurations, of which only one may be active at a time. Configurations exist to allow the fundamental functionality of the device to change. For example, an ISDN adapter with two channels may provide two configurations: one con-

figuration with two interfaces, allowing the ISDN channels to be used independently; and another with a single interface controlling both channels bound together (*channel bonding*).

Individual *interfaces* may also be independently configured with different functionality by use of an "alternate" setting. This is typically used to change the transfer characteristics of the interface's endpoints. For example, a packet-based device interface, such as a USB audio device, may have alternate settings with different packet sizes. Depending on the bus load or other conditions, the driver can select the most appropriate packet size using an "alternate" setting.

Figure 3 illustrates the hierarchy of configurations, interfaces and endpoints, with an active configuration, interface and endpoint, shown down the left-hand side of the diagram.

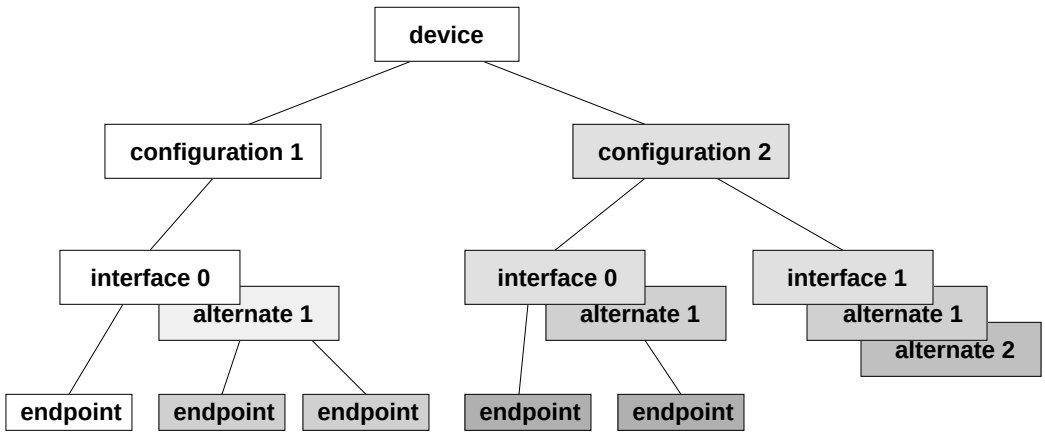

Figure 3. USB configuration, interface and endpoint hierarchy.

2.3. USB Interface Endpoints

Endpoints are the sinks and sources for communications on the bus. Bus transactions are addressed first to the device, then to an endpoint within it. A software structure known as a *pipe* is used to model the connection between the host and an endpoint, maintaining the state information (not entirely dissimilar to the structure and state maintained by a TCP connection). With a few exceptions (detailed later), communication on these pipes is logically the same as that on occam channels: unidirectional, synchronous and point-to-point. At the lower bus protocol level, acknowledgements, sequence numbers and CRC checks exist which reinforce these characteristics.

There are four different types of endpoint defined by the USB standards, each of which specifies how the communication 'pipe' should be used:

- *Control*, uses a structured message protocol and can exchange data in either direction. A setup packet containing the request is transferred from the host to the device, followed by zero or more bytes of data in a direction defined by the request type. These are used to enumerate and configure devices, and are also used by many USB device classes to pass information, such as setting the state of keyboard LEDs.
- *Bulk*, exchanges data unidirectionally on demand, no structure is imposed on the data. These are the most similar to a traditional Unix 'pipe'. They are used by storage devices, printers and scanners.
- *Interrupt*, these act similarly to *bulk* except data is exchanged on a schedule. At a set interval, the host offers bus time to the device and if it has data to transfer, or is ready, then it accepts the bandwidth offered. Alternatively, the device delays using a negative acknowledgement, and the transfer is tried again at the next specified interval. This

process continues for as long as the host desires. For example, the typical keyboard is offered a transfer every 10ms, which it uses to notify key-state changes.

- *Isochronous*, like *interrupt* these also use a schedule. The difference is that isochronous transfers are not retried if the device is not ready or a bus error occurs. Since isochronous transfers are not retried, they are permitted to use larger packets than any of the other types. Isochronous transfer are used where data has a constant (or known maximum) rate and can tolerate temporary loss; audio and video are the typical uses.

2.4. Implementation Challenges

There are a variety of considerations when building a USB device-driver 'stack'. Firstly, the dynamic nature of the hardware topology must be reflected in software. Traditional operating systems use a series of linked data-structures to achieve this, with embedded or global locks to control concurrent access. The implementation must also be fault-tolerant to some degree — if a user unplugs a device when in use, the software using that device should fail gracefully, not deadlock or livelock.

As USB is being increasingly used to support legacy devices (e.g. PS/2 keyboard adaptors, serial and parallel-port adapters), the device-driver infrastructure needs to be able to present suitable interfaces for higher-level operating system components. These interfaces will typically lie *underneath* existing high-level device-drivers. For instance, the 'keyboard' driver (primarily responsible for mapping scan-codes into characters and control-codes, and maintaining the shift-state), will provide access to any keyboard device on the system, be it connected via the onboard PS/2 port or attached to a USB bus. Such low-level connectivity details are generally uninteresting to applications — which expect to get keystrokes from a 'keyboard' device, regardless of how it is connected (on-board, USB or on-screen virtual keyboards). Ultimately this results in a large quantity of internal connections within the RMoX "driver.core", requiring careful design to avoid deadlock.

In addition to handling devices and their connectivity, the USB driver is responsible for managing power on the bus. This essentially involves disallowing the configuration of devices which would cause too much current to be drawn from the bus. Devices are expected to draw up to 100 mA by default (in an unconfigured state), but not more than 500 mA may be drawn from any single port.

3. Software Architecture

All device-driver functionality in RMoX is accessed through the central "driver.core" process (figure 1), which directs incoming requests (internal and external) to the appropriate driver within. To support the dynamic arrival and removal of devices, a new "dnotify" device-driver has been added. This is essentially separate from the USB infrastructure, and is responsible for notifying registered listeners when new devices become available or old ones are removed.

The USB driver infrastructure is built from several parts. At the lowest level is a *host controller driver* (HCD), that provides access to the USB controller hardware (via I/O ports and/or memory-mapping). The implementation of one particular HCD is covered in section 3.3. At the next level is the "usb.driver" (USBD) itself. This process maintains a view of the hardware topology using networks of sub-processes representing the different USB busses, acting as a client to HCD drivers and as a server to higher-level drivers. Figure 4 shows a typical example, using USB to provide the 'console' with access to the keyboard.

The "usb.keyboard" process uses the USBD to access the particular keyboard device, and provides an interface for upstream "keyboard" processes. Such a "keyboard" process might actively listen for newly arriving keyboard devices from "dnotify", managing them all together — as many existing systems do (e.g. pressing 'num-lock' on *one* of the keyboards causes *all* num-lock LEDs to toggle).

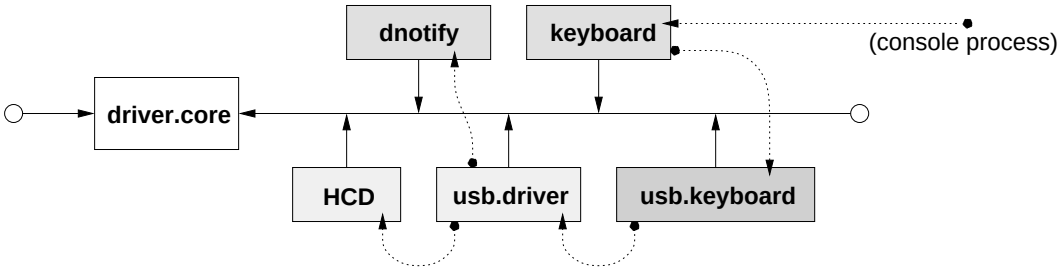

Figure 4. USB device-driver top-level components.

3.1. USB Driver Structure

Processes outside the USB driver can gain access to the USB at three levels: bus-level, device-level and interface-level. The "usb.driver" contains within it separate process networks for each individual bus — typically identified by a single host controller (HC). These process networks are highly dynamic, reflecting the current hardware topology. When a host controller driver instance starts, it connects to the USB driver and requests that a new bus be created. Mobile channel bundles are returned from this request, on which the host controller implements the low-level bus access protocol and the root hub. Through this mechanism the bus access hardware is abstracted. Figure 5 shows the process network for a newly created bus, with three connected USB devices, one of which is a hub. For clarity, some of the internal connections have been omitted.

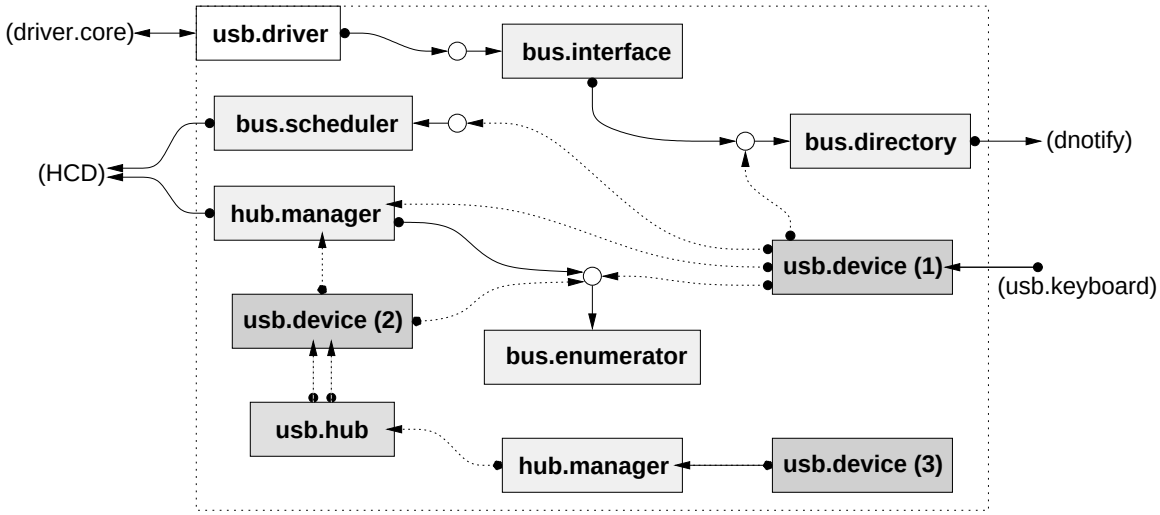

Figure 5. USB device-driver bus-level components.

Within each bus sub-network are the following processes:

- "bus.interface" provides mediated access to the bus, specifically the bus directory. It services a channel bundle shared at the client-end, which is held by the USB driver and other processes which request bus-level access.
- "bus.directory" maintains a list of all devices active on the bus and client channel-ends to them. Attempts to open devices and interfaces pass through the directory which resolves them to channel requests on specific devices. When devices and interfaces are added or removed from the directory, their information is propagated to the 'dnotify' driver which acts as a system wide directory of *all* devices (not just USB).
- "bus.enumerator" is responsible for assigning device addresses (1-127), and acts as a mutex lock for bus enumeration. The lock functionality is necessary as only one device maybe enumerated on the bus at any given time. When a device is first connected

it does not listen to the bus. After its port is reset it begins listening to the bus and responding to queries on the default address (0). The USB driver then sends a "set address" request to the default address.

- "bus.scheduler" is responsible for managing bus bandwidth and checking the basic validity of bus transactions. The USB standard dictates that certain types of traffic may only occupy a limited percentage of the bus time (specific values depend on the bus revision). If there is sufficient bandwidth and the request is deemed valid then it is passed to the HCD for execution.
- "hub.manager", of which there may be many instances, one for each hub and one for the root hub, are responsible for detecting device connection, disconnection, and initiating associated actions such as enumeration or device shutdown.

From Figure 5, it is possible to see that a hierarchy exists between the "hub.manager", "usb.hub" and "usb.device" processes. The "usb.hub" process converts the abstract hub protocol used by the "hub.manager" process into accesses to the hub's device endpoints. The root hub, not being an actual USB device, is implemented directly by the HCD in the abstract protocol of the "hub.manager" and hence no "usb.hub" process is necessary.

During the enumeration of a port, the "hub.manager" process *forks* a "usb.device" process, passing it the client-end of a channel bundle. The channel bundle used is client/server plus notify, and contains three channels: one from client to server, and two from server to client. The client is either listening on the 'notify' channel or making a request using the client/server channels. The server process normally requests on the client/server channel pair; if it wishes to 'notify' the client then it must do so in parallel, in order to maintain deadlock freedom.

Client/server plus notify channel bundles, already mentioned, are used between hubs and devices. When the "hub.manager" detects that a port has been disconnected, it notifies the devices attached to it. This is done by passing the server-end of the channel bundle to a newly forked process, in order to prevent the hub blocking whilst it waits for the device to accept the disconnect notification. The forked process performs the aforementioned parallel service of client/server and notify channels. A similar pattern is also used between the underlying hub driver ("usb.hub" or "HCD") and the "hub.manager" to notify of changes in the hub state (port change or hub disconnect).

3.2. USB Device Structure

Figure 6 shows the internal structure of the "usb.device" processes, and within these 'interface' and 'endpoint' processes. With the exception of the default control endpoint, these form the structure described in 2.2 (figure 3), and model the hierarchy defined in the USB specification directly as processes. When a device is configured (non-zero configuration selected), it forks off interface processes to match those defined in the configuration (read from the device). The interfaces in turn fork endpoints to match their current alternate setting. Changing an interface's alternate setting causes the endpoints to be torn down, and changing the configuration of the device tears down all interfaces and endpoints.

Devices, interfaces and endpoints maintain a channel bundle, the client-end of which is given out when they are "opened". This channel-end is *not* shared, so that the process can track the active client. If the device is disconnected, or the interface or endpoint is torn down, then it continues to respond to requests (with errors) until the client-end of this "public" channel bundle is returned, after which it may shutdown (and release its resources). As the USB topology is expected to change during normal system operation (adding and removing devices), so the process network must not only grow, but safely shrink. Maintaining these public channel-ends as exclusive (unshared) allows us to guarantee this safety.

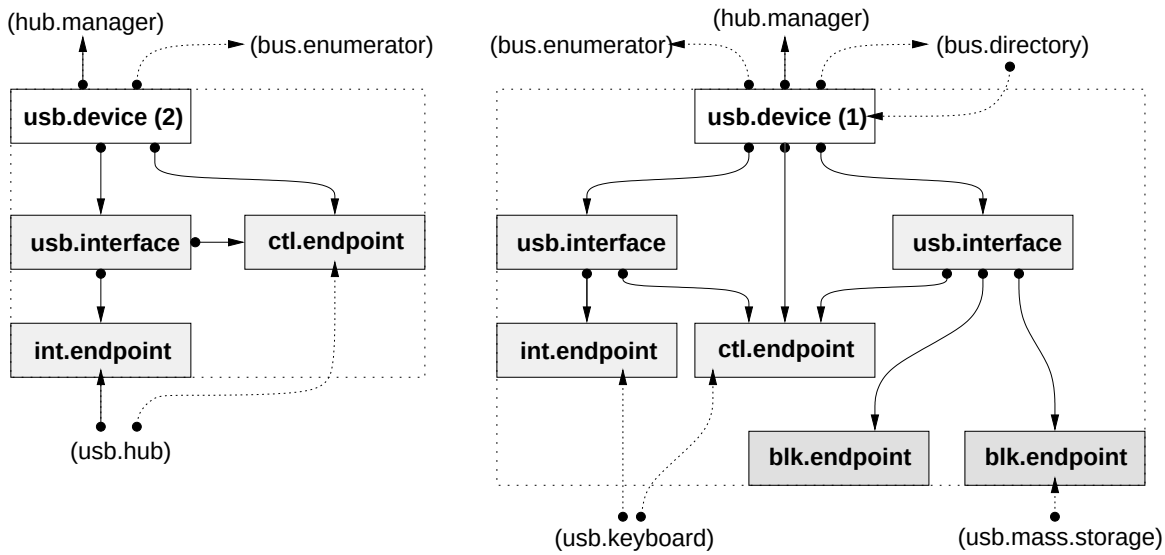

Figure 6. USB device-driver device-level components.

It is however, still possible to safely share resources if the need arises by issuing a separate channel bundle to each client that opens it. When all channel-ends have been returned, the resource may safely terminate. This pattern is used for control endpoints, which due to their structured data transfers can be safely used by many clients at once. Additionally, the default control endpoint must be accessible to all interfaces and their clients. Shared access to devices, interfaces and other endpoints does not typically make sense (given the nature of devices), and hence is not implemented. If we do later decide to introduce sharing, it can be added at a 'higher-level' within the process network.

Requests to open a device come in over the device-directory interface channel. If the device is not already open then it returns that client channel-end via the directory. Requests to open interfaces are passed first to the associated device, which in turn queries the interface over its internal channel. Interfaces may also be opened through the device's channel-end. Using the first approach it is possible open an interface without first opening its associated device (which may already be open). This allows interfaces to function independently and separates functions from devices — i.e. the keyboard driver only uses the keyboard interface, without communicating with the associated device. Endpoints are only accessible through their associated interface — this makes sense as a driver for a function will typically be the only process using the interface and its endpoints.

Care must be taken when implementing the main-loop of the endpoint processes, such that the channel from the interface is serviced at a reasonable interval. This is mainly a concern for interrupt endpoints, where requests to the bus could wait for a very long period of time before completing. For all other endpoint types, bus transactions are guaranteed to finish within a short period of time, hence synchronous requests are guaranteed to complete promptly. The consequence of ignoring this detail would be that the system could appear to livelock until some external event (e.g. key press, or device removal) occurs, causing a pending interrupt request to complete.

3.3. USB UHCI

A number of host controller standards exist, of which UHCI (*Universal Host Controller Interface*) is one. These allow a single USB host controller driver to be written such that it supports a range of host controller hardware. RMoX has drivers for the UHCI, OHCI and EHCI standards. The UHCI [11] standard, released by Intel in 1996, is the simplest and shall

be used as an example to explore how data is transferred efficiently from endpoints to the bus. Figure 7, expands the HCD part of Figure 4, as implemented by the UHCI driver.

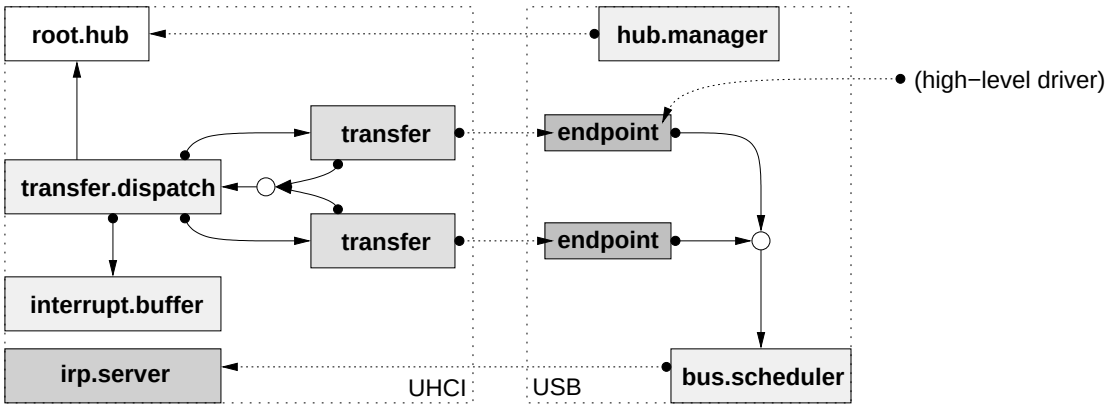

Figure 7. Overview of the 'uhci.driver' host controller driver.

The "uhci.driver" is broken down into four main processes (ignoring transfers, which are forked in response to demand as explained below):

- "root.hub" provides access to the hardware registers which implement the ports of the root hub, and receives relevant interrupt information from "transfer.despatch".
- "interrupt.buffer" receives interrupts from the underlying interrupt routing subsystem (part of the PCI driver). When an interrupt is received, the hardware status register is read, then cleared before the interrupt subsystem is told it can unblock the interrupt line the UHCI hardware is using. Status register flags are buffered and passed to the "transfer.despatch" process on request. The "interrupt.buffer" is similar in function to an interrupt handler subroutine in a traditional OS kernel, such as Linux.
- "transfer.despatch" manages all other registers of the UHCI hardware not handled by other processes. It also manages a set of linked data structures in system memory which are accessed by the UHCI hardware and is used to initiate and control bus transfers.
- "irp.server" (I/O request packet server) implements the HC protocols which the "bus.scheduler" process uses to schedule traffic. On receiving a transfer request from the "bus.scheduler" it forks off a transfer to handle that request.

From the descriptions above it is clear that the UHCI hardware registers are partitioned between the cooperating processes. This ensures that there are no shared resource race-hazards between processes within the driver. To further reinforce this, there are no shared memory buffers; all memory used is *mobile* and is *moved* between processes as appropriate.

As previously mentioned, the "irp.server" forks off a transfer process to handle each bus transfer request. As part of each request received from the "bus.scheduler" is a client channel-end. This is also passed to the transfer process during the fork. The endpoint that initiated the transfer holds the server-end of the channel bundle, and so provides a direct path between the endpoint and the driver.

The transfer process builds a set of linked data structures to describe the packets which will be exchanged on the bus. These data structures are then registered with the despatch process which links them into the hardware accessible data structures it maintains. In the same request, the transfer process also passes a client channel-end on which the despatch process can communicate with it. When the despatch process detects a hardware condition, and associated data structure changes that suggest the state of a transfer has changed, then it contacts the associated transfer process passing back any associated memory buffers. The transfer process then examines the data structures. Not all of the data structures which must

be examined are accessible to the despatch process, hence the transfer process implements this check.

Based on the state of the transfer data structures, the transfer process, when queried, tells the despatch process to continue, suspend or remove its transfer. If the transfer is complete or has failed then the transfer process notifies the endpoint, which in turn can decide to issue a new transfer or terminate the transfer process. This allows the network between the endpoint and despatch process, and any allocated data structures, to persist across multiple transfers, reducing communication and memory management overheads. This is legal in bandwidth scheduling terms as only interrupt and isochronous transfers are allocated bus bandwidth, based on their schedule, which cannot be changed once a request has begun. When the transfer is finally terminated the endpoint will notify the "bus.scheduler" that the bandwidth is once again free. However, it should be noted that for hardware reasons, control and bulk transfers do not use this *persistence* feature with the UHCI driver.

Memory buffers from the client are passed directly from endpoint to transfer process, and are used for DMA with the underlying hardware. This creates an efficient *zero-copy* architecture, and has driven investigation into extending the occam-pi runtime allocator to be aware of memory alignment in DMA memory positioning requirements.

4. Using the USB Driver

As an example of using the USB driver, we consider a version of the "usb.keyboard" process. Instead of connecting directly to "usb.driver", the USB keyboard driver registers the client-end of a "`CT.DNOTIFY.CALLBACK`" channel-bundle with the "dnotify" driver, requesting that it be notified about USB keyboard connections. This involves setting up a data-structure with details of the request and passing it along with the notification channel-end to the "dnotify" driver, using the following code:

```
-- USB device classes (HID or boot-interface) and protocol (keyboard)
VAL INT INTERFACE.CLASS.CODE IS ((INT USB.CLASS.CODE.HID) << 8) \/ #01:
VAL INT INTERFACE.PROTOCOL IS 1:

CT.DNOTIFY.CALLBACK? cb.svr:
SHARED CT.DNOTIFY.CALLBACK! cb.cli:
MOBILE []DEVICE.DESC intf.desc:
INT notification.id:
SEQ
  cb.cli, cb.svr := MOBILE CT.DNOTIFY.CALLBACK   -- allocate callback bundle
  intf.desc := MOBILE [1]DEVICE.DESC             -- allocate descriptor array

  intf.desc[0][flags]    := DEVICE.MATCH.TYPE \/
                            (DEVICE.MATCH.CLASS \/ DEVICE.MATCH.PROTOCOL)
  intf.desc[0][type]     := DEVICE.TYPE.USB.INTERFACE
  intf.desc[0][class]    := INTERFACE.CLASS.CODE
  intf.desc[0][protocol] := INTERFACE.PROTOCOL

  CLAIM dnotify!
    SEQ
      dnotify[in] ! add.notification; DNOTIFY.INSERTION; cb.cli; intf.desc
      dnotify[out] ? CASE result; notification.id
```

The resulting network setup is shown in the left-hand side of figure 8. The "usb.keyboard" driver then enters its main-loop, waiting for requests from either the driver-core, or "dnotify". When a USB keyboard is subsequently connected (or if one was already present), the notification is sent and "usb.keyboard" responds by forking off a driver process ("keyboard.drv").

This initially connects to the USB *interface* specified in the notification (which will be for the connected keyboard), as shown in the right-hand side of figure 8. The code for this is as follows:

```
PROC keyboard.drv (VAL DEVICE.DESC device, SHARED CT.OUTPUT! keyboard,
                   SHARED CT.BLOCK! usb)
  CT.USB.INTERFACE! intf:
  INT result:
  SEQ
    -- connect to interface
    CLAIM usb!
      SEQ
        usb[in] ! ioctl; IOCTL.USB.OPEN.INTERFACE; device[address]
        usb[out] ? CASE result; result
        IF
          result = ERR.SUCCESS
            usb[device.io] ? CASE intf
          TRUE
            SKIP
    ...  get endpoints and start main loop
:
```

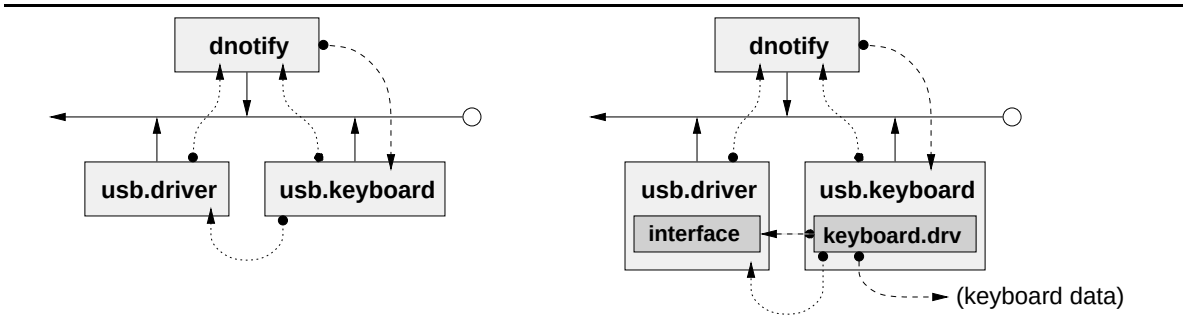

Figure 8. Setup of the 'usb.keyboard' device-driver

4.1. Using USB Interfaces

With a connection to the USB interface (in the variable '`intf`'), the keyboard driver requests connections to the *control* and *interrupt* endpoints of the USB interface. Discovering the identifier of the interrupt endpoint first involves querying the interface, simply:

```
MOBILE []BYTE endpoints:
SEQ
  intf[in] ! list.endpoints
  intf[out] ? CASE endpoints; endpoints
```

The returned mobile array is expected to be of length 1, containing the interrupt endpoint identifier. The control endpoint is identified separately, as there is at most one per interface. Connections to the endpoints are then established, resulting in connectivity similar to that shown in figure 6. The following code is used for this, omitting error-handling for brevity:

```
CT.USB.EP.CTL! ep0:
CT.USB.EP.INT! int.ep:
SEQ
  intf[in] ! open.endpoint; 0                    -- request control endpoint
  intf[out] ? CASE ctl.ep; ep0
  intf[in] ! open.endpoint; endpoints[0]         -- request interrupt endpoint
  intf[out] ? CASE int.ep; int.ep
```

In addition to listing and connecting to specific endpoints, the interface-level connection is used for listing and switching between alternative interfaces, retrieving information about the device, and other USB specific control.

4.2. Using Interrupt and Control Endpoints

From this point, the USB keyboard driver uses the two endpoint connections to receive keyboard data and control the keyboard. The receiver loop (using the interrupt endpoint) is structured in the following way:

```
packet := MOBILE [8]BYTE
INITIAL BOOL done IS FALSE:
WHILE NOT done
  SEQ
    int.ep[in] ! dev.to.host; packet             -- request 8 byte input
    int.ep[out] ? CASE complete; result; packet  -- response
    IF
      result > 0                                 -- received data
        process.packet (packet, keyboard!)       -- send keys to terminal
      result = 0                                 -- no data
        SKIP
      TRUE
        done := TRUE                             -- interrupt pipe error (exit)
```

The control endpoint is used to set the keyboard LEDs and keyboard rate, in addition to other USB control. The following code example is used to set the keyboard LEDs:

```
VAL BYTE type IS USB.REQ.TYPE.HOST.TO.DEV \/
                 (USB.REQ.TYPE.CLASS \/ USB.REQ.TYPE.INTERFACE):
MOBILE []BYTE data:
INT result:
SEQ
  data := MOBILE [1]BYTE
  data[0] := leds                                -- each bit represents an LED

  ep0[in] ! type; HID.REQ.SET.REPORT; (INT16 HID.REPORT.OUTPUT) << 8;
            INT16 (device[address] /\ #FF); data
  ep0[out] ? result; data                        -- get response

  IF
    result >= 0
      SKIP                                       -- success
    TRUE
      ...  report error
```

As can be seen, using control endpoints is moderately cumbersome, but this is to be expected given the vast range of USB devices available. However, general device I/O through the interrupt endpoint is largely straightforward.

Concurrency is a significant advantage in this environment, allowing a single device-driver to maintain communication with multiple endpoints simultaneously, without significant coding complexity. This particularly applies to situations where a single driver uses multiple USB devices, which may operate and fail independently. One example would be a software RAID (redundant storage) driver, operating over many USB mass storage devices, and presenting a single block-level interface in the RMoX device layer. Expressing such behaviours in non-concurrent languages in existing operating systems is complex and error-prone, primarily due to the lack of an explicit lightweight concurrency mechanism.

5. Conclusions and Future Work

In conclusion, we have designed and developed a robust and efficient process-orientated USB driver. Significantly, the process networks we have developed bare an almost *picture perfect* resemblance to the hierarchy presented in the USB standards and the network which exists between physical devices. Furthermore, as a feature of the development language and process-orientated approach, our driver components are scheduled independently. This allows us, as developers, freedom from almost all scheduling concerns. For example "hub.manager" processes can make synchronous device calls, without causing the entire system to cease functioning.

RMoX itself still has far to go. The hardware platform for which we are developing is a PC104+ *embedded PC* — a standardised way of building embedded PC systems, with stackable PCI and ISA bus interconnects [12]. This makes a good initial target for several reasons. Firstly, the requirements placed on embedded systems are substantially less than what might be expected for a more general-purpose (desktop) operating-system — typically acting as hardware management platforms for a specific application (e.g. industrial control systems, ATM cash machines, information kiosk). There is, however, a strong requirement for reliability in such systems. Secondly, the nature of the PC104+ target makes the RMoX components developed immediately reusable when targeting desktop PCs in the future. Additionally, USB is being increasingly used for device connectivity within embedded PC104 systems, due to its versatility. Assuming a future RMoX driven ATM cash machine, adding a survellience camera would simply involve plugging in the USB camera, installing the appropriate video device-driver and setting up the application-level software (for real-time network transmission and/or storage on local devices) — this could be done without altering the existing system code at all, it simply runs in parallel with it. The builds are routinely tested on desktop PCs and in emulators as standard, exercising the *scalability* of RMoX. We also have a functional PCI network interface driver, and hope to experiment with distributed RMoX systems (across several nodes in a cluster) in the not too distant future.

In addition to the RMoX operating-system components is development work on the tool-chain and infrastructure. Developing RMoX has highlighted a need for some specific language and run-time features, such as the aforementioned allocation of aligned DMA-capable memory. A new occam-π compiler is currently being developed [13] which will allow the easy incorporation of such language features. There is also a need to stabilise existing occam-π language features, such as nested and recursive mobile data types, and port-level I/O.

5.1. Related Work

The most significant piece of related research is Microsoft Research's Singularity operating system [14], which takes a similarly concurrent approach to OS design. Their system is programmed in a variant of the *object-orientated* C# language, which has extensions for efficient communication between processes — very similar in principle and practice to occam-π's mobilespace [15]. The times reported for context-switching and communication in Singularity are some 20 times slower than what we have in RMoX, though their justification for it is incorrect in places (e.g. assuming occam processes can only wait on a single channel — not considering the 'ALT' construct). Some of the difference is correctly attributed to RMoX's current lack of support for multi-core/multi-processor machines. Fortunately, we know how to build these CSP-style schedulers for multi-processor machines, with comparatively low overheads, using techniques such as *batch-scheduling* [16], and are currently investigating this.

More generally, there is a wide range of related research on novel approaches to operating-system design. Most of these, even if indirectly, give some focus to the language

and programming paradigm used for implementation — something other than the *threads-and-locks* procedural approach of C. For example, the Haskell operating-system [17] uses a functional paradigm; and the Plan9 operating-system [18] uses a concurrent variant of C ("Alef"). However, we maintain the view that the *concurrent process-orientated* approach of occam-π is more suitable — as demonstrated by the general scalability and efficiency of RMoX, and the ease of conceptual understanding in the USB driver hierarchy — software organisation reflects hardware organisation.

A lot of ongoing research is aimed at making current languages and paradigms more efficient and concrete in their handling of concurrency. With RMoX, we are starting with something that is already highly concurrent with extremely low overheads for managing that concurrency — due in part to years of experience and maturity from CSP, occam and the Transputer [19].

Acknowledgements

We would like to thank the anonymous reviewers who provided valuable feedback and suggestions for improvement. This work was funded by EPSRC grant EP/D061822/1.

References

[1] F.R.M. Barnes, C.L. Jacobsen, and B. Vinter. RMoX: a Raw Metal occam Experiment. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, pages 269–288, Amsterdam, The Netherlands, September 2003. IOS Press. ISBN: 1-58603-381-6.

[2] Frederick R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent, June 2003.

[3] P.H. Welch and F.R.M. Barnes. Communicating mobile processes: introducing occam-pi. In A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[4] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-13-153271-5.

[5] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. ISBN-10: 0521658691, ISBN-13: 9780521658690.

[6] P.H. Welch, G.R.R. Justo, and C.J. Willcock. Higher-Level Paradigms for Deadlock-Free High-Performance Systems. In R. Grebe, J. Hektor, S.C. Hilton, M.R. Jane, and P.H. Welch, editors, *Transputer Applications and Systems '93, Proceedings of the 1993 World Transputer Congress*, volume 2, pages 981–1004, Aachen, Germany, September 1993. IOS Press, Netherlands. ISBN 90-5199-140-1. See also: `http://www.cs.kent.ac.uk/pubs/1993/279`.

[7] P.H. Welch. Graceful Termination – Graceful Resetting. In *Applying Transputer-Based Parallel Machines, Proceedings of OUG 10*, pages 310–317, Enschede, Netherlands, April 1989. Occam User Group, IOS Press, Netherlands. ISBN 90 5199 007 3.

[8] Compaq, Intel, Microsoft, and NEC. Universal Serial Bus Specification - Revision 1.1, September 1998.

[9] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. Universal Serial Bus Specification - Revision 2.0, April 2000. URL: `http://www.usb.org/developers/docs/usb_20_05122006.zip`.

[10] PCI Special Interests Group. PCI Local Bus Specification - Revision 2.2, December 1998.

[11] Intel. Universal Host Controller Interface (UHCI) Design Guide, March 1996. URL: `http://download.intel.com/technology/usb/UHCI11D.pdf`.

[12] PC/104 Embedded Consortium. PC/104-Plus Specification, 2001. URL: `http://pc104.org/`.

[13] F.R.M. Barnes. Compiling CSP. In P.H. Welch, J. Kerridge, and F.R.M. Barnes, editors, *Communicating Process Architectures 2006*, volume 64 of *Concurrent Systems Engineering Series*, pages 377–388, Amsterdam, The Netherlands, September 2006. IOS Press. ISBN: 1-58603-671-8.

[14] M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J.R. Larus, and S. Levi. Language support for Fast and Reliable Message-based Communication in Singularity OS. In *Proceedings of EuroSys 2006*, Leuven, Belgium, April 2006. URL: `http://www.cs.kuleuven.ac.be/conference/EuroSys2006/papers/p177-fahndrich.pdf`.

[15] F.R.M. Barnes and P.H. Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 243–264, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[16] K. Debattista, K. Vella, and J. Cordina. Cache-Affinity Scheduling for Fine Grain Multithreading. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 135–146, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[17] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in haskell. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 116–128, New York, NY, USA, September 2005. ACM Press.

[18] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs, 1995. Available from `http://www.cs.bell-labs.com/plan9dist/`.

[19] M.D. May, P.W. Thompson, and P.H. Welch. *Networks, Routers and Transputers*, volume 32 of *Transputer and occam Engineering Series*. IOS Press, 1993.