Communicating Process Architectures – 2003
Jan F. Broenink (Eds.)
IOS Press, 2003

RMoX: A Raw-Metal occam Experiment

Fred BARNES†, Christian JACOBSEN† and Brian VINTER‡

† *Computing Laboratory, University of Kent,*
Canterbury, Kent, CT2 7NF, England.
{frmb2,clj3}@kent.ac.uk

‡ *Department of Mathematics and Computer Science,*
University of Southern Denmark, Odense, Denmark.
vinter@imada.sdu.dk

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. This paper presents RMoX: a highly concurrent CSP-based operating-system written in occam. The motivation for this stems from the overwhelming need for reliable, secure and scalable operating-systems. The majority of operating-systems are written in C, a language that easily offers the level of flexibility required (for example, interfacing with assembly routines). C compilers, however, provide little or no mechanism to guard against race-hazard and aliasing errors, that can lead to catastrophic run-time failure (as well as to more subtle errors, such as security loop-holes). The RMoX operating-system presents a novel approach to operating-system design (although this is not the first CSP-based operating-system). Concurrency is utilised at all levels, resulting in a system design that is well defined, easily understood and scalable. The implementation, using the KRoC extended occam, provides guarantees of freedom from race-hazard and aliasing errors, and makes extensive use of the recently added support for dynamic process creation and channel mobility. Whilst targeted at mainstream computing, the ideas and methods presented are equally applicable for small-scale embedded systems — where advantage can be made of the lightweight nature of RMoX (providing fast interrupt responses, for example).

1 Introduction and Motivation

This paper presents RMoX, a highly concurrent CSP [1, 2, 3] based operating-system written in an extended variant of the occam [4] multiprocessing language.

On commodity hardware, operating-systems are the software component that control access to the physical hardware, manage applications, and provide services to applications (such as networking and structured file storage — where these are required). Clearly, it is essential that operating-systems function correctly (as their specification intended), including graceful error handling. Unfortunately, many operating-systems do not behave this way.

In our view, mainstream commodity operating-systems (such as Windows[tm] and Linux[tm]) suffer from three major problems. Firstly there is the issue of implementation errors (bugs) — often resulting in undesirable (and sometimes dramatic) system failure. The second is the problem of scalability: most operating-systems do not scale well over large numbers of processors (in either SMP or distributed environments). Finally is the issue of performance. Many of the mechanisms which provide security (at the hardware level), do so at great cost (such as memory-management and operating-system call gates). Much of this is in response to the first problem (of software error), requiring hardware support to ensure that software components do not interfere with each other destructively.

Bugs within the operating-system itself present a significant problem: at best they are an inconvenience; at worst they cost: in lost work and time (whilst the system is fixed).

The majority of software errors within an operating-system are due either to programmer error or incorrect specification. As the size and complexity of the system increases, the number of errors introduced also increases. The counter-acting force is time — the longer a system remains in operation, the more bugs that will be discovered and 'fixed'. Of course, without some level of formal verification (either manual or automated), there can be *no guarantee* of software correctness (particularly when the specification is deficient).

The C programming language [5], as used to implement the majority of operating-systems — and in some ways was designed for that purpose — provides no mechanisms for guarding against data aliasing and race-hazard errors. Within an operating-system such errors can be catastrophic, are easily introduced, and can be extremely hard to locate.

Operating systems written in C are not at a complete loss however; language research elsewhere has produced 'safer' dialects of C, such as Cyclone [6], that provide greater protection against aliasing (and some race-hazard) errors.

1.1 Concurrency in Operating Systems

The C language provides no natural support for concurrency, which presents a significant challenge to the operating-system implementor. Concurrency is required for some reasons and desirable for others.

One of the many tasks performed by an operating-system is the scheduling of application processes. Although processes might not interact between themselves, they will all interact with the operating-system. Many interactions between applications and the operating-system are *blocking* (socket reads, for example). Clearly, having one process blocked in the kernel should not prevent other processes from being scheduled. Furthermore, a process that is busy performing a computational task should not be allowed to 'hog' the processor.

Linux, for example, handles most concurrent interaction using 'wait-queues', structures where processes may wait inside the kernel, to be later rescheduled by another process (for instance, on completion of an I/O event, or a message from another process). At this level (and considering only single-processor systems), race-hazard errors are not too common and could only be caused through mis-handling of asynchronous events (typically hardware interrupts). A good description of the Linux kernel (and the various wait-queues, including the run-queue) can be found in [7].

The step in complexity from a uni-processor to a multi-processor kernel (for Linux) is significant, particularly for shared-memory multi-processing. Protection from race-hazards is largely handled using spin-locks to protect critical regions of the kernel. As Linux has evolved, this locking has become increasingly more fine-grained — now allowing multiple processes to be executing concurrently inside the kernel (across multiple processors). This is complex, and frequently a source of error (particularly in device-drivers). The slightest mis-placement of a 'lock' or 'unlock' call can have grave consequences. Errors of this type can remain undetected for a long time, and might never be discovered at all.

Unfortunately, programmers who work on the Linux kernel are often left with the common view that "concurrent programming is hard". This very generalised statement is wrong — CSP and its implementations have proved that already. The problem here is that concurrent programming in C is hard — and there is no easy or clean solution to that, particularly where operating-systems are concerned.

Concurrency (with CSP semantics) *is* desirable however, particularly for operating systems. Good concurrent design can produce systems that are easily scalable — this is simply not the case for many non-concurrent languages, C in particular.

1.2 The RMoX Operating System

The problems of existing operating-systems, particularly their lack of concurrency, has led us to RMoX: an experimental, highly concurrent, CSP-based dynamic operating system written in occam. Traditional occam is not particularly suitable for modern-day operating-system implementation. Its lack of *any* dynamic capability presents a significant problem, in addition to a lack of multi-level priority support.

Fortunately, previous and on-going work in KRoC/Linux [8] has added the dynamic capabilities required. Mobile data-types [9] are used to provide unit-time data *movement* operations (such as returning data blocks from a block device driver to a file-system driver). Much of the network connectivity (within RMoX itself) is handled using *shared* channel-types. Two significantly useful features are provided by these: the ability to share the resources of a server with multiple clients (in such a way that security and fairness are enforced by the compiler); and the ability to *move* channel-ends around a process network. These, combined with the `FORK` dynamic process creation mechanism, provide an ideal way to build dynamically scalable process networks. One type of network built in this way — the process farm — is discussed in detail in [10, 11] and [12]. RMoX, however, is a somewhat more general-purpose process network.

The underlying concept of RMoX is simple: a scalable concurrent system that provides 'operating system' functionality, including device-drivers, file-systems and networking. As a general guide, we aim towards providing a POSIX [13] compatible interface. POSIX clearly defines, for example, what operations must be supported on files, or what operations must be supported on sockets. It does not define how such functionality should be implemented, however.

One particularly significant gain from using a concurrent language is the removal of explicit scheduling control from operating system components. In Linux, for example, when a process wishes to block inside a device-driver, it must be placed on a wait-queue before calling the main '`schedule()`' function to schedule another process. This is a scheduling issue, and should not be the (direct) concern of the device driver. In fact, much of the scheduler interaction within the Linux kernel is done to provide simple process synchronisation (most obviously, communicating through a pipe). Calls to device drivers may need to synchronise with hardware events, which again, is usually done using wait-queues (and possibly some unpleasant interrupt handling). Code of this nature simply does not exist in RMoX— at least as far as the operating-system components are concerned. Device drivers, for example, are active processes — if a device driver wishes to perform a delay, or interact with another component (to which it is connected via a channel), it simply does so.

This clear split, between concurrent processes and the mechanisms scheduling them, simplifies operating-system implementation greatly. For scheduling occam processes within RMoX, a modified version of the CCSP [14] run-time kernel is used. The modifications are primarily the disabling of operating-system calls within CCSP itself (such as the mechanisms used to implement blocking system calls and terminal I/O). Within RMoX, any equivalent functionality is provided by active processes, with appropriate low-level support for handling interrupts.

1.3 Paper Overview

This paper is divided into 3 main sections. The first, section 2, presents the overall design of RMoX and its various sub-systems (as networks of occam processes).

Section 3 examines the low-level infrastructure that exists 'underneath' the occam kernel. Much of the work here is related to bootstrapping RMoX, and providing a C (or assembler) interface for accessing the low-level hardware — memory management and interrupts for example.

Given that RMoX is essentially an occam *program* in its own right, running it within the KRoC/Linux environment (as an ordinary Linux process) is possible — with appropriate replacements for the low-level handling and some device-drivers. Section 4 describes UM-RMoX (User-Mode RMoX), that provides this functionality.

RMoX is still in the early stages of development, and cannot nearly be considered complete. Currently, RMoX is sufficiently capable to implement small-scale embedded-systems, although the work to date has been done on Intel Pentium based machines. Much of the framework for future development is in place however. Section 5 discusses the work already done and presents suggestions for the future development of RMoX.

2 Design

The core of RMoX is a client-server network, connected mostly by shared channel-bundles. The 'static' component of the network is comprised of three main server-farms, plus a 'console' process and some auxiliary processes, as shown in figure 1.

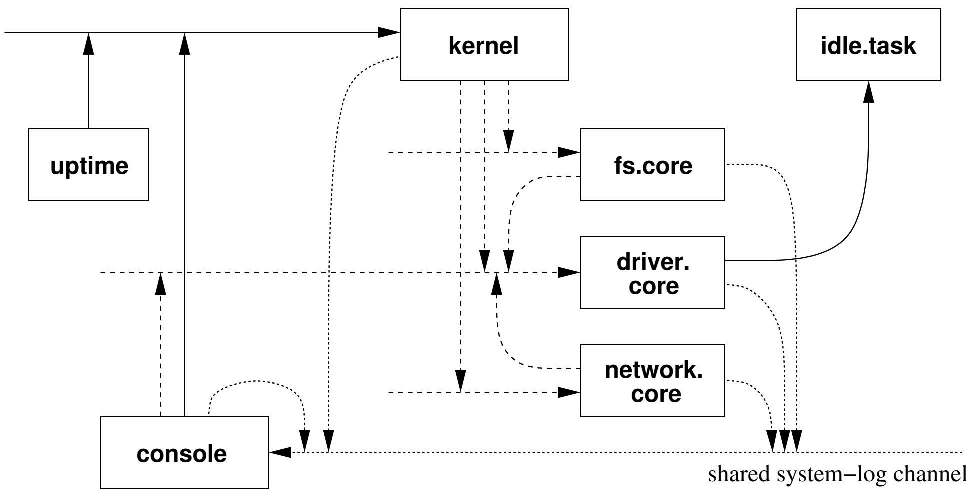

Figure 1: Initial RMoX top-level process network

The '`kernel`' process provides what could be considered the main system interface (system-call interface on Linux). The (client shared) channel-bundle on the left-hand side, initially connected to the '`uptime`' and '`console`' processes, is the main point for system interaction. The channel-type used is defined as:

```
CHAN TYPE CT.KERNEL
  MOBILE RECORD
    CHAN P.KERNEL.IN in?:       -- requests
    CHAN P.KERNEL.OUT out!:     -- responses
:
```

The two variant protocols '`P.KERNEL.IN`' and '`P.KERNEL.OUT`' cover the available range of operating-system 'calls' and responses. In addition to maintaining some local state (for example, the host-name and uptime), the function of the '`kernel`' process is to re-direct requests to the relevant sub-system (the file-system core for example). The kernel process *will* also be responsible for ensuring some security — e.g. that a process attempting to change the host-name is indeed allowed to.

2.1 Device Drivers

At the lowest level of the system are device drivers. The '`driver.core`' process itself is a pure server, with the responsibility of maintaining a *farm* of device-driver processes. Figure 2 shows an example of this for an active system. Initially, there are no device driver processes. These are `FORK`ed by the '`driver.core`' in response to an appropriate request.

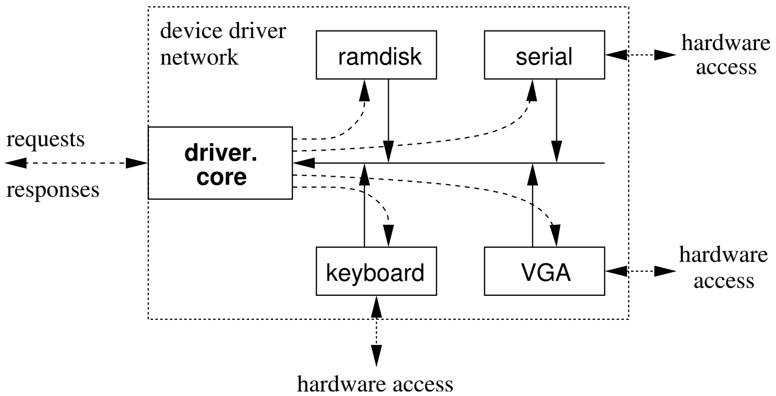

Figure 2: RMoX device driver process network

Requests are made to the '`driver.core`' process using the connected client-shared channel-bundle. This is defined as the type '`CT.DRIVER`', and in a similar way to the '`CT.KERNEL`' channel-type, has two channel fields: one for requests and one for responses:

```
CHAN TYPE CT.DRIVER
  MOBILE RECORD
    CHAN P.DRIVER.IN in?:
    CHAN P.DRIVER.OUT out!:
:
```

Currently, only a small number of operations are supported by '`driver.core`', defined in the '`P.DRIVER.IN`' protocol:

```
PROTOCOL P.DRIVER.IN
  CASE
    connect.to.driver; MOBILE []BYTE
    start.driver; MOBILE []BYTE
:
```

The range of possible responses is significantly broader however, and reveals much about the way we intend RMoX to work:

```
PROTOCOL P.DRIVER.OUT
  CASE
    driver.started
    no.such.device
    device.busy
    input.device; CT.INPUT!
    shared.input.device; SHARED CT.INPUT!
    output.device; CT.OUTPUT!
    shared.output.device; SHARED CT.OUTPUT!
    block.device; CT.BLOCK!
    shared.block.device; SHARED CT.BLOCK!
:
```

When a 'connect.to.driver' message is sent, and a device driver with the specified name exists, that request is simply forwarded. If the device can be opened, the device driver responds with the client-end of a channel-bundle, of the type 'CT.INPUT', 'CT.OUTPUT' or 'CT.BLOCK', depending on the type of device, that may or may not be shared.

Devices are referred to by name, using a simple hierarchical naming scheme. Many device drivers will be trivial — responsible for only a single piece of hardware. A driver such as 'serial', however, may be responsible for multiple pieces of hardware (each serial port in this case). In this particular case, the 'serial' driver manages a farm of serial-handling processes, as shown in figure 3.

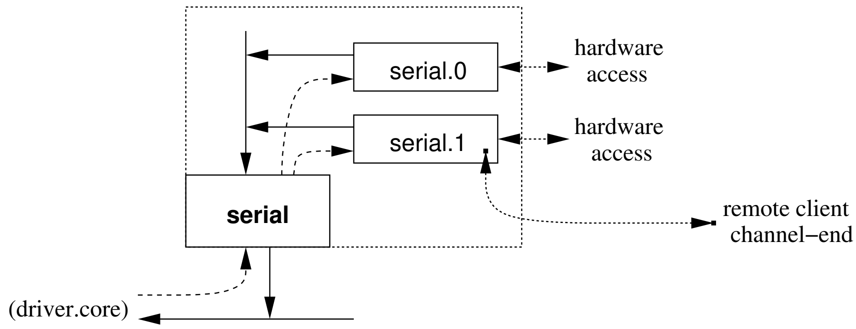

Figure 3: RMoX 'serial' device-driver process network

This example shows the serial network with two sub-processes, and an active connection to the 'serial.1' device. When this device is requested, the 'driver.core' process only examines the name up to the first period ('.') when deciding where to forward the request. The 'serial' device then examines the next component of the name and forwards it appropriately (or returns an error if the device does not exist).

The types 'CT.INPUT' and 'CT.OUTPUT' are used for character devices of the input and output variety respectively. The serial-port drivers within RMoX actually present two devices each — one for input and one for output. These are identified using the suffixes '.input' and '.output' respectively. The 'CT.INPUT' channel-type is defined as:

```
RECURSIVE CHAN TYPE CT.INPUT
  MOBILE RECORD
    CHAN P.INPUT.IN in?:
    CHAN P.INPUT.OUT out!:
    CHAN BYTE data.out!:
    CHAN CT.INPUT! return?:
:
```

The two channels 'in' and 'out' are used primarily for device-specific control, handled using the 'P.INPUT.IN' and 'P.INPUT.OUT' variant protocols. Data generated by a character device is simply sent byte-by-byte down 'data.out'. If a more substantial communication mechanism is required, the 'in' and 'out' channels can be used to provide it.

When a client process has finished using the services of a device driver, it returns its local end on the 'return' channel — using the recursive channel-type feature. This way, a device driver knows when a client process has finished using the device, and that action is synchronised. Currently, there is no return mechanism for devices that are *shared*, although this may be added later on (if the need for it arises).

The 'CT.OUTPUT' type is defined in a similar way, but whose BYTE channel is named 'data.in?' and where the device-specific protocols 'P.OUTPUT.IN' and 'P.OUTPUT.OUT' are used. The 'vga' driver in RMoX is a character output device, initially operating in a very 'dumb' mode. When more of the system starts up (section 2.4), this driver is put into a more complex mode of operation and communication continues on the 'in' and 'out' channels.

Block device drivers use the 'CT.BLOCK' channel-type and are specifically geared towards block-addressable devices (such as disks — including the 'ramdisk'). This channel-type is defined as:

```
RECURSIVE CHAN TYPE CT.BLOCK
  MOBILE RECORD
    CHAN P.BLOCK.IN in?:
    CHAN P.BLOCK.OUT out!:
    CHAN CT.BLOCK! return?:
:
```

The variant protocols 'P.BLOCK.IN' and 'P.BLOCK.OUT' provide mechanisms for reading and writing *blocks* of data, as well as querying the device for its block-size and total number of blocks.

2.2 File Systems

The file-system sub-system is built in a similar way to the device-driver sub-system — i.e. a farm of sub-server processes along with a controlling process ('fs.core').

The interface to the file-system is presented in a UNIX-like manner, supporting the familiar 'mount' and 'umount' operations (which POSIX does not specifically define). When RMoX starts up, there are no mounted file-systems. The first file-system mounted is a ramdisk file-system named 'ramdisk.fs', that is mounted at the root ('/'). Before a file-system can be mounted, the underlying device (if required) must be accessed. The ramdisk file-system uses the 'ramdisk' device-driver, that simply provides read and write access to a number of fixed-size blocks.

In the state where no file-systems are mounted, the majority of requests will return errors (with the exception of mounting a file-system on '/'). The rest of RMoX will continue to operate normally however.

Figure 4 shows the file-system network with two mounted file-systems, 'ramdisk.fs' and 'dev.fs'. 'dev.fs' is a *virtual* file-system that provides access to devices through the file-system interface. The traditional UNIX mechanism for device access is through the use of special (character and block) files, usually found in the '/dev' directory. The use of virtual file-systems for '/dev' within UNIX systems is becoming increasingly popular, however, and it makes good sense to do this — restricting the contents of '/dev' to the devices that are *actually* available, instead of wasting hundreds of directory-entries on devices that are not available.

Internally, the 'ramdisk.fs' process maintains a *block-bitmap* of allocated 'ramdisk' blocks, alongside an array of *inodes*. This is a very simplified version of many traditional UNIX file-systems, except that they retain the block-bitmap and inodes on the device itself (often with a layer of caching to improve performance).

File handling within the ramdisk is performed using FORKed 'file.handler' processes. When a file is opened, a channel-bundle of type 'CT.FILE' is created. The client-end is returned to the process that opened the file and the server-end is given to the newly created 'file.handler' process. In addition to servicing "open file" and similar requests, 'ramdisk.fs' acts as a server for the potentially many 'file.handler' processes. In particular this includes allocating blocks when writing to files, and manipulating inodes.

Figure 5 shows an example process connected to a 'file.handler' within the ramdisk. When such a process has finished with a file, the recursive channel-type feature is used to

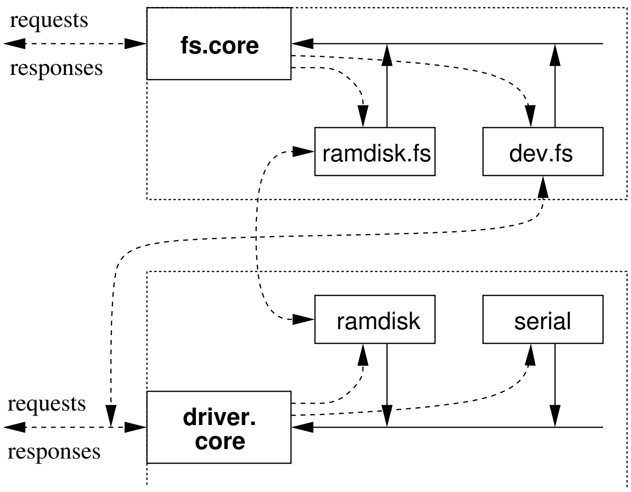

Figure 4: RMoX file-system driver network

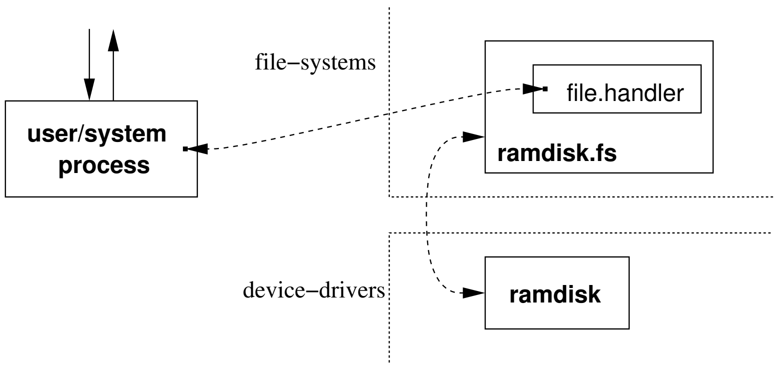

Figure 5: RMoX file handling

return the whole channel to the '`file.handler`' process, that then updates any inode states and shuts-down gracefully.

Directory operations are performed in a similar manner, but use the services of a '`dir.-handler`' process — connected to the client process using a '`CT.DIR`' channel-bundle.

2.3 Networking

The networking infrastructure, specifically the provision of an IP [15] stack, has been largely designed and implemented by Adam Sampson, at student at UKC, as part of occamnet [16].

occam and CSP provide a very natural way of building network infrastructures. It makes good sense to build networking components (such as routers, multiplexors and filters) as communicating processes, that simply communicate 'packets' (plus any additional control information required). The use of `MOBILE` types ensures a minimum copying of data — the communication of a 'packet' will always be a unit-time (pointer-swapping) operation within occam.

Previous work in [17] discusses the building of ATM over DS-Link routing networks using occam and Transputer-based hardware [18, 19]. This is closely related to the work in occamnet, which instead routes IP packets over occam channels. One paper of interest is [20], that describes a method for transforming high-level protocol specifications into workable occam code, ultimately resulting in two communicating state-machines, one for sending and one for receiving. This technique has been used to implement a SLIP [21] protocol driver, that provides support for IP over serial lines (the Ethernet equivalent is ARP [22]).

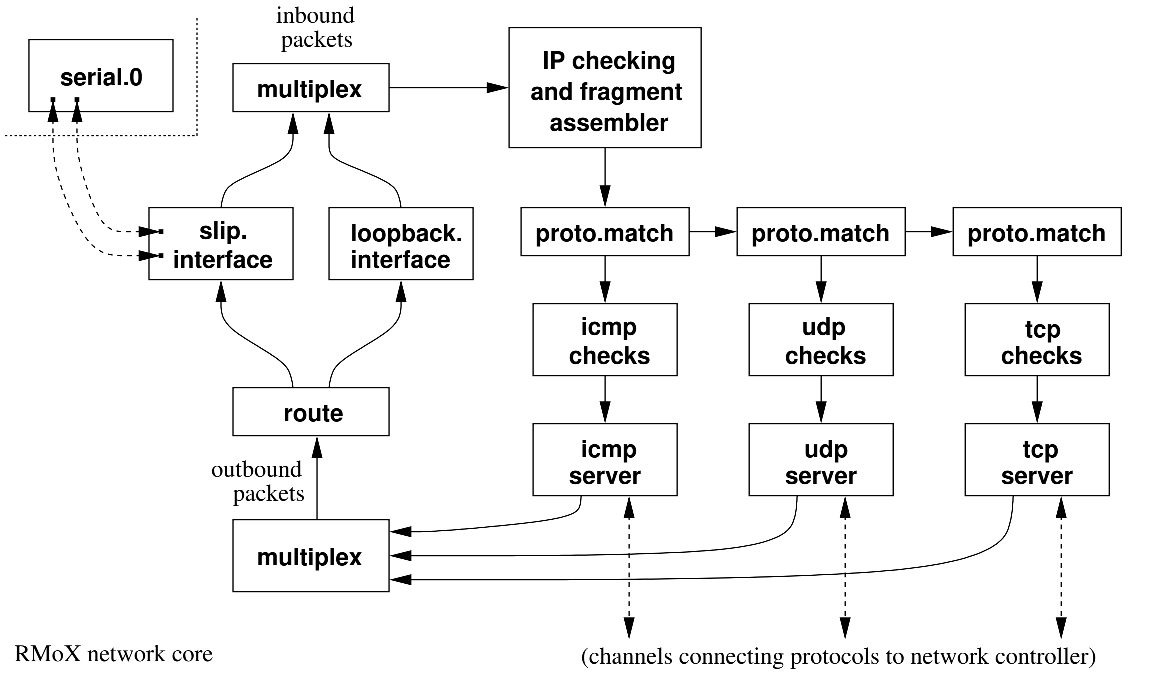

Figure 6: RMoX network core and protocol handling

Figure 6 shows the overall layout of the RMoX/occamnet networking infrastructure, with the SLIP interface at the top-left of the figure. Although it is not done this way currently, the interfaces and protocol stacks will ultimately become dynamic components, giving significantly greater control. At the time of writing, lack of support for some particular types of nested `MOBILE` structures has prevented this dynamic implementation, discussed further in section 5.

Internally, the '`icmp.server`', '`udp.server`' and '`tcp.server`' processes are implemented as process farms, in a style similar to that of RMoX's file-handling (as shown in figure 5).

2.4 The System Console

The '`console`' process in figure 1 provides the traditional "system-console" service. Additionally, it performs the initial system startup. Currently, this is the loading of the '`vga`', '`keyboard`' and '`ramdisk`' device drivers, followed by the mounting of '`ramdisk.fs`' at the file-system root.

Internally, the '`console`' is a network of sub-processes, using the '`vga`' and '`keyboard`' drivers to provide 6 *virtual-terminals*, that each support a basic VT220 functionality. Figure 7 shows this process network.

The first virtual console is used solely for the system log, and is where system messages are displayed (via the shared log channel). When the system first starts, there is only the '`console`' process, with no '`vt.handler`' processes. Until the '`vga`' driver is loaded (and subsequently used for log output), the system log is either simply discarded or written out using a lower-level function (such as any provided by the boot agent, described in section 3).

The '`vga`' driver provides support for six buffers, one of which may be the 'active' (visible) buffer. The switching of buffers is performed by the '`vtkeysw`' and '`vgaplex`' processes, triggered using the keys Alt-F1 through Alt-F6 (in a similar style to Linux).

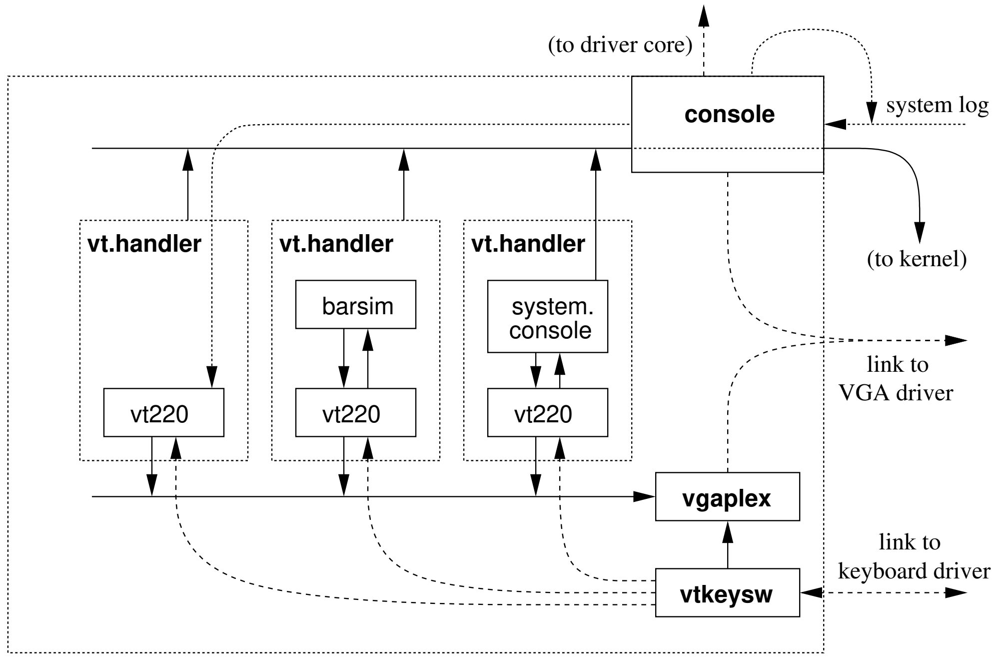

Figure 7: RMoX console process network

The last virtual console is used for the '`system.console`' process. This provides a very basic shell-like functionality, in particular, a set of built-in commands that communicate with the main '`kernel`' process to provide some basic system control (for example, '`mount`' and '`umount`'). Also provided are some more complex built-in commands, such as '`ls`', '`cat`' and '`wfln`' (write text to file). This 'shell' is particularly primitive and does not provide support for features such as pipes and input/output re-direction.

The other virtual terminals are currently either unused or used for demonstrators. Ultimately, these will provide system login points, once those components have been implemented. The processes within the virtual terminals are run at a lower priority than the rest of the operating-system. Although not currently the case (given the applications within RMoX at the time of writing), when the CPU load reaches 100%, we would wish the operating-system components to be scheduled in preference to user applications.

3 Implementation

So far, only the occam components of RMoX have been examined. Much more, however, is required in order to provide operating-system functionality.

The scheduling of occam processes is performed using a modified version of the standard CCSP run-time kernel, that uses lower-level components to provide access to memory regions, I/O regions and interrupts. The use of this extra layer provides a good hardware abstraction for CCSP, that will ultimately aid in the porting of RMoX to other architectures — RMoX is currently restricted to Intel Pentium [23] based architectures.

The current version of RMoX uses the Flux OSKit [24] to provide this functionality. The OSKit can provide much more, however, including schedulers, device drivers, file-systems and network protocol stacks. RMoX currently only uses some of the more basic features of the OSKit — those that provide access to memory, I/O regions and interrupts. The OSKit also provides the boot-loader for RMoX, a fairly complex procedure on Pentium based hardware.

Figure 8 shows the layering of components within RMoX. Hardware is either accessed directly (from RMoX device drivers), or indirectly from CCSP via the OSKit. Accessing hardware directly from occam is in fact relatively trivial. Much of this is related to occam's original application on the Transputer, where there was no need for an operating-system — the hardware performed scheduling and communication directly.

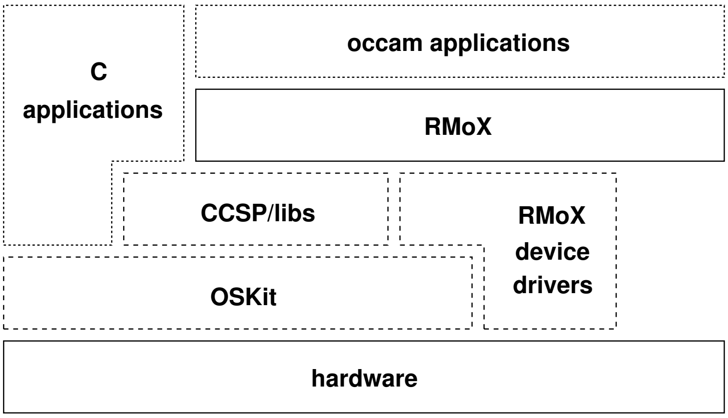

Figure 8: Components of RMoX

Section 3.1 discusses the accessing of (IBM-PC style) hardware from within occam. The provision for supporting concurrent C applications is discussed in section 3.2, although this has not been implemented yet.

3.1 Hardware Access from occam

At the lowest level, RMoX expects to have access to interrupts, memory-mapped I/O regions, and port-mapped I/O regions. The Transputer provided its interrupt mechanism using event-pins [25], mapped directly to communication in occam (using placed channels).

For RMoX, we currently use the OSKit's mechanism for accessing interrupts — which is to call a program-specified function asynchronously when the interrupt occurs. Internally, CCSP maintains an array of interrupt counters and an array of process 'hooks', one for each interrupt[1]. When an interrupt occurs, the appropriate counter value is incremented and a special 'interrupt' sync-flag [26] is set.

When CCSP 'sees' the interrupt flag, it re-schedules any corresponding process waiting for that interrupt, and clears the interrupt counter to zero. If there is no process waiting for an interrupt, nothing happens, and the counter remains unchanged. To synchronise on an interrupt, occam programs call a CCSP built-in that performs the wait (or returns immediately if interrupts have happened), declared with the (external) `PROC` signature:

```
PROC C.oos.wait.int (VAL INT int, RESULT INT count)
```

This waits for the interrupt specified by '`int`', returning the number of interrupts (since the last call) in '`count`'. Only one (occam) process may wait for each interrupt. If interrupt sharing is required, it must be implemented in occam (trivial).

The more natural representation of interrupts in occam is that of a channel communication. This can easily be engineered into a process, and adds very little overhead. However, there is no reason to use such an intermediary process, unless an RMoX component needs to `ALT` against an interrupt.

Accessing I/O memory and ports is somewhat more direct. To enable access to physical memory-ranges or I/O ranges, RMoX provides the following external C functions:

[1]The current implementation provides for 32 hardware interrupts, more than the 16 used by the majority of IBM-PC compatible (uni-processor) hardware.

```
PROC C.oos.mem.map.phys (VAL INT addr, size, VAL BOOL cache,
                         RESULT INT addr, res)
PROC C.oos.io.reserve (VAL INT start, size, RESULT INT res)
PROC C.oos.io.free (VAL INT start, size)
```

Once reserved, memory or I/O ports can be explicitly ‘PLACED’. Memory is mapped by the placement of ordinary occam data types — typically arrays or ‘PACKED’ RECORD types. I/O ports are accessed using ‘PLACED PORT’s in occam, and are accessed in the same way as channels (using input and output).

The ‘keyboard’ device driver, for example, uses both interrupts and I/O ports to communicate with the hardware. The main body of this process is simply:

```
INT kyb.ctrl:
INT res:
SEQ
  C.oos.io.reserve (KEYBOARD.IOADDR, KEYBOARD.IOADDR.LEN, res)
  ASSERT (res = 0)

  PLACED [KEYBOARD.IOADDR.LEN]PORT BYTE kyb.control AT KEYBOARD.IOADDR:
  WHILE TRUE
    BYTE kyb.status, kyb.val:
    SEQ
      -- wait for interrupt
      C.oos.wait.int (KEYBOARD.INTR)

      -- read data
      kyb.control[KEYBOARD.STATUS] ? kyb.status
      IF
        (kyb.status /\ KEYBOARD.STATUS.OBF) <> 0
          SEQ
            kyb.control[KEYBOARD.DATA} ? kyb.val
            out ! kyb.val
        TRUE
          SKIP
```

In the actual implementation, a loop is used to retry keyboard reads (up to $10,000$ times) if the ‘output buffer full’ (OBF) flag is not set.

3.2 Running C Applications in RMoX

RMoX, as viewed by the OSKit, is simply a large monolithic process — CCSP scheduling the occam processes. The OSKit, however, provides support for its own scheduling of applications, using a number of freely-available operating-system schedulers (such as those from Linux and OpenBSD).

Given suitable interfacing, these could be used to provide support for concurrent C applications, allowing a UNIX/POSIX environment to be run on-top of RMoX. Figure 9 shows the proposed method — using a special ‘posix.sysif’ component that is simultaneously an occam process (within RMoX) and a POSIX system-call interface (within the C process environment).

In this type of setup, the Flux OSKit would provide the main low-level scheduler, scheduling POSIX-interfaced processes alongside RMoX. Unlike C processes, we would not normally allow RMoX to be preemptively scheduled. This ensures that whenever RMoX (or any of its C/occam components) *are not* executing, then RMoX is in a known, safe state — the point at which it invoked the low-level scheduler.

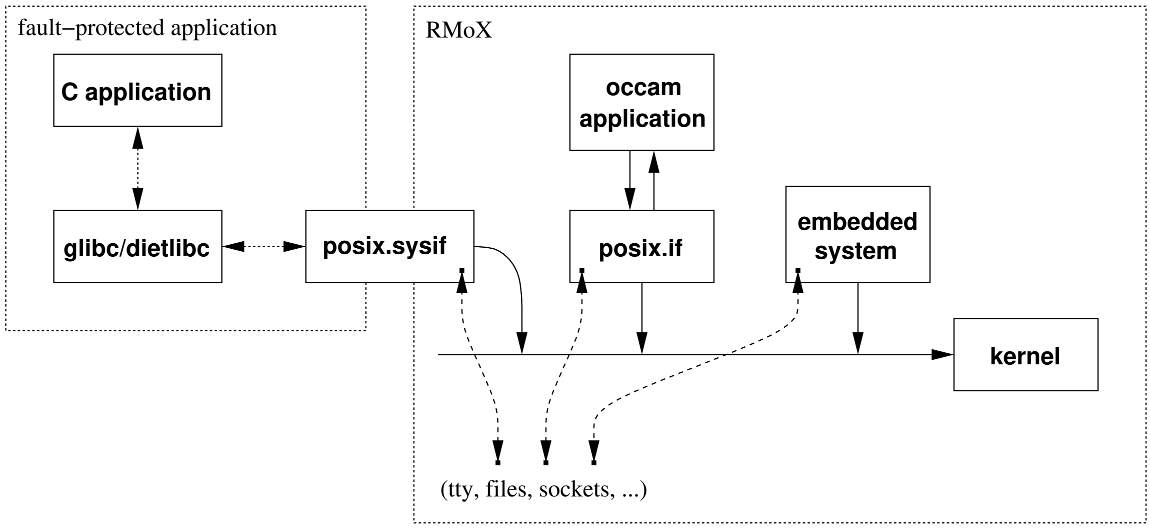

Figure 9: Running C applications alongside RMoX

This way, if a hardware interrupt occurs while a C process is executing, it can be pre-empted immediately, with RMoX resuming at a point where any process waiting for that interrupt will be scheduled first (assuming the absence of higher-priority processes). If a hardware interrupt occurs whilst RMoX is executing, the latency will depend on the frequency at which occam components interact with the scheduler (parallel granularity). Within RMoX, this is typically less than 1 micro-second (on modern hardware, with CPU speeds measured in giga-hertz).

At the time of writing, very little has been implemented with regards to supporting external C processes. Currently, RMoX has no underlying scheduler — it is the only 'process'[2] running. Instead of re-working parts of the Flux OSKit to meet RMoX's requirements (which is still a potential approach), a new low-level scheduler has been developed as part of another student project at UKC. This is discussed further in section 5.1.

4 User-Mode RMoX

User-Mode RMoX is a port of the RMoX kernel and certain drivers to the Linux operating system. In User-Mode RMoX, RMoX itself runs as a user-land process in the host operating system (Linux) and does therefore not have direct access to hardware. Instead, use is made of specific 'user-mode' device-drivers, that emulate any required devices (requiring blocking interaction with Linux in many cases). User-Mode RMoX is largely analogous to the User Mode Linux (UML) port [27].

User-Mode RMoX was initially developed to expedite the RMoX development cycle, which as in any operating system development, is rather cumbersome. Instead of booting from a floppy-disk or network (on raw hardware), which is relatively slow, RMoX is simply run as a Linux program — with almost instantaneous startup.

The reduction in RMoX development time is not the only motivation for User-Mode RMoX. Simply having a user-mode implementation provides easy access for development and general experimentation. Furthermore, User-Mode RMoX makes a good application for extensive testing of the new KRoC/occam features, dynamic channel and process creation in particular.

[2]In this context, a 'process' refers to all code and data contained within a particular *virtual machine*, whose protection is provided for by the hardware.

For RMoX itself, User-Mode RMoX can provide access to devices that are not currently supported within RMoX natively. For example, access to a complete file-system, networking and any (Linux supported) device in general.

This is done by providing file-system, device-driver and networking components that interface with the corresponding Linux/POSIX functionality — essentially glue-code. Interfacing in this way supports the development of components that range from providing the whole functionality, to those that provide only a basic functionality. For example, the RMoX socket interface can either be provided directly (by mapping one-on-one to UNIX sockets), or by using the proposed RMoX networking infrastructure (section 2.3), in combination with a virtualised low-level packet-interface (using the Linux 'tun-tap' driver, for example). In particular, this provides a good environment for developing higher-level RMoX components, that might require functionality not currently available when RMoX is run on raw-hardware.

4.1 User-Mode Alternatives

Other ways of achieving similar (user-mode) functionality do exist, however. For instance, being able to run an unmodified RMoX kernel in user-land is more preferable than running a version that has been modified. Attempts at this, however, have been largely unsatisfactory — for various reasons.

The most straightforward way to run the unmodified RMoX kernel in the host operating system is by the use of a PC *virtualiser*: VMWare, Plex86 and Bochs have all been used with varying degrees of success. In the process of virtualising, however, performance suffers — such systems must provide an entire virtual machine, on top of an existing operating-system. Additionally, there may be compatibility problems. In the case of VMWare, the virtual machine provided is incompatible with RMoX to the extent that attempts to boot RMoX result in VMWare crashing — sometimes taking the host operating-system (Windows or Linux) with it[3]

One advantage of PC virtualisers is that they provide "real" virtual hardware. This can be particularly useful when developing device-drivers whose hardware is particularly sensitive to mis-programming — for example, setting up a DMA (direct memory access) transfer incorrectly, often resulting in severe memory corruption. Device drivers, on the whole, are highly hardware dependant, and the virtual 'hardware' might not accurately model the real hardware.

The Flux OSKit [24], whose functionality RMoX uses, includes its own mechanisms for making a port of kernels built on top of it. This is provided by special 'user-mode' versions of OSKit libraries, plus some additional initialisation code. The modifications within the OSKit to support this are comparable with the modifications in RMoX that make it 'User-Mode RMoX'.

One potential benefit of porting RMoX to the user-mode OSKit is the availability of some OSKit drivers, that are also available in the 'real' OSKit implementation — and could be made available to RMoX. However, RMoX currently does not use any of the device-drivers available in the OSKit, largely as we intend to replace the OSKit with a more dedicated (and simpler) layer in the future (discussed in section 5.1).

4.2 Implementation

The porting of RMoX to a user-mode system is mostly trivial. This can largely be attributed to the fact that RMoX is essentially an occam process network, running on top of a modified CCSP (the run-time system that provides scheduling, communication and dynamic memory

[3]This may very well be due to VMWare's focus on virtualising Windows and Linux.

allocation). Modifying RMoX to run on the standard run-time system is simply a matter of removing or replacing the hardware-specific parts.

Drivers within RMoX access the hardware using memory-mapped I/O, port-based I/O and interrupts, as already discussed in section 3.1. In User-Mode RMoX, the I/O memory-mapper is replaced with a version that simply returns free memory blocks, which the driver can then probe (under the false impression that it is accessing hardware). In most cases, RMoX drivers will simply fail when initialised in the 'virtual' environment, correctly reporting that to the RMoX component(s) attempting to use them.

The only drivers which must be completely replaced for User-Mode RMoX are the 'keyboard' and 'vga' drivers, that provide the basic display and input functionality. In RMoX proper, these drivers use port-mapped I/O and interrupts (for the keyboard driver) to access the hardware directly. The interface to User-Mode RMoX is provided through 'xtextwindow' — a set of X11 routines that provide the display (in a standard X11 window), and the keyboard (through events received on that window).

The user-mode and 'real' RMoX systems are built from the same set of source files — with the exception of replacement device drivers and the user-mode interface. Use is made of the recently added pre-processor capability for this, that supports the conditional compilation of code based on the definition of a pre-processor constant (very much like the C pre-processor's '`#ifdef`' directive). The constant that selects a user-mode version of RMoX, '`USER.MODE.RMOX`', is set in the 'Makefile' (passed via the KRoC command-line to the occam compiler).

The changes required within RMoX (including its supporting code and build system), to support a user-mode implementation, are small — up to the point of providing specialised user-mode drivers. The provision of 'fake' I/O memory-mapping, and dummy handling for port-mapped I/O and interrupts, inside the run-time system (CCSP), also requires few changes.

4.3 Current and Future User-Mode Sub-Systems

The screen and keyboard drivers are perhaps the most important user-mode drivers, as without those it is not possible to interact with a running user-mode RMoX (until a method of accessing RMoX from the network becomes available, that is).

Two versions of these screen and keyboard drivers are currently supported — an X11 based interface ('xtextwindow'), and a text-console based interface. The X11 interface displays a window on the host display, providing keystrokes to the user-mode 'keyboard' driver and displaying data sent to the 'vga' driver. Unfortunately, concurrent interaction in Xlib (the X11 client library for C), is impractical — the mechanisms which Xlib uses internally to protect from race-hazards is non-compatible with the mechanism used to implement blocking system-calls [28] (that are required for the blocking interaction with Xlib).

This problem is solved by allowing device-drivers to request connections to each other, via the main '`driver.core`' process. The user-mode 'vga' driver is responsible for (serial) interaction with the underlying C code (and Xlib). The user-mode 'keyboard' driver requests a connection to the 'vga' driver, that returns a channel-end to which keyboard events are sent. As long as the structuring of inter-driver requests is strictly client-server, as is the case here, there is no possibility for deadlock. There are good reasons for including this functionality, however. For example, a software RAID [29] driver that connects to multiple physical disk drivers.

4.3.1 The Console Interface

The console interface for User-Mode RMoX is somewhat simpler than the X11 interface. RMoX, as a KRoC occam program, does not have the three standard top-level channels (keyboard, screen and error), and passing these channels down through the driver-core to the user-mode 'keyboard' and 'vga' processes would be messy.

Fortunately, KRoC itself implements the handling for the three top-level channels using three *real* occam processes (that access the UNIX keyboard, screen and error streams through standard interfaces — external C calling [30] and blocking system-calls). The user-mode 'keyboard' and 'vga' drivers simply replicate and extend this functionality — in order to support the RMoX interface. This also includes supporting VT220 escape sequences generated by RMoX during the initial startup (before its own VT220 handling processes, within the '`console`', are activated). This handling is also required in the X11 drivers, but given the relatively small number of different escape sequences generated, both implementations are reasonably simple.

The console interface is somewhat faster then the X11 interface, but requires a correctly sized terminal window to display output correctly — this particularly applies when running within an 'xterm' or similar virtual terminal.

4.3.2 Host File-System Access

User-Mode RMoX provides a complete file-system in the form of "host-fs" — the host file-system interface. This provides, for RMoX, all the POSIX defined file and directory handling operations (with the exception of memory-mapping), translating requests to Linux file-handling system calls. The 'host-fs' file-system driver is written entirely in occam, and uses the KRoC file-library for file and directory access [31] (that has undergone minor improvements and has been extended slightly, to support all the operations required by RMoX/POSIX).

Some complications arise from differences between RMoX and Linux directory handling. RMoX allows the 'seek' operation to be performed on directories — Linux does not. This is handled by having the 'host-fs' driver cache directory entries, such that 'seek' and 'read' operations can be handled in unit-time. No limits are imposed on the number of directories which can be cached in this way, although they could be, if required.

A virtual 'block' device driver is provided with User-Mode RMoX, that gives seekable block-structured access to a file on the host file-system. Such a file can easily be 'mounted' under Linux, using the 'loop-back' interface — provided, of course, that the file contains a file-system which Linux supports. More usefully, a file-system can be created on the file (from within Linux), and subsequently used for developing RMoX's own native file-system drivers (for the Linux second-extended file-system, for example). Since Linux supports seekable access to block-devices through the file-system, this mechanism can be used to provide User-Mode RMoX with access to physical disk partitions — assuming that the User-Mode RMoX process has sufficient privileges under Linux to open such devices.

Use could be made of this in a dual-boot RMoX and Linux/User-Mode RMoX system, with a partition used exclusively for RMoX. In this way, the partition can be accessed in both environments — either by using the user-mode 'file' block-device, or using a native RMoX block disk device-driver (that, currently, is only partially implemented).

The User-Mode Linux port provides some interesting variations from standard block device-drivers, such as sparse block devices and COW (copy-on-write) block devices [27]. Incorporating these ideas into User-Mode RMoX at some stage would be beneficial — allowing more conservative disk-usage for host 'file' file-systems (the size actually used by files and other information, not the size of the virtual device).

5 Conclusions and Future Work

At the time of writing, RMoX is very much incomplete, but much of the ground-work for future development is in place. Functional, but basic, instances of some components exist — such as the ramdisk file-system and serial device-driver (in addition to the keyboard and VGA drivers).

Looking towards the future, it is hoped that the development effort on RMoX can be stepped-up, now that a functional user-mode implementation exists, under which much development can be done.

At some point, we intend to replace the OSKit layer with a more specialised implementation — with some consideration to portability. The OSKit currently only supports Intel i386 based platforms. We aim to be broader than that, particularly considering the range and scale of hardware and applications for which RMoX, and CSP-based technologies in general, are suitable.

The long-term possibilities for RMoX are vast, particularly if our claims about security, scalability and reliability hold true in real-life situations. The first (significant) demonstrator produced will be RMoX combined with a modified version of the occam web-server [32], which we hope will perform relatively well in comparison with other web-servers, given the comparative immaturity of RMoX.

5.1 A Dedicated Low-Level Scheduler

One of the main motivations behind providing our own low-level scheduler, besides portability, is the added level of simplicity that it offers. The Flux OSKit is a very general-purpose tool-kit — as it was designed to be — and RMoX only uses a small proportion of its functionality.

The reason for wanting a low-level scheduling functionality (that involves hardware support for memory protection), is to support the concurrent execution of UNIX/POSIX processes — typically those written in C — as discussed in section 3.2. Such separate processes must still interact with RMoX however, since it provides the kernel (POSIX) functionality. In particular, we want RMoX to retain control at all times — i.e. it must never be preempted.

An experimental scheduler and memory-manager has been developed by Ramsay Taylor as a final-year project at UKC [33]. Currently, however, it is not sufficiently complete for use with RMoX, although a basic version is not far away.

5.2 Distribution and Replication

Figure 8 contains a gap. In the future, we intend to fill this space with functionality that provides distribution and replication for networked devices, as dynamically and transparently as possible. At one level, this could be applied to networks of workstations to provide the combined resources of all the nodes — a general-purpose distributed operating-system.

At another level, for example (and very hypothetically), such functionality could be used to inter-link the embedded control-systems of Earth orbiting satellites, to a variety of ends — from ensuring that they avoid each other (where possible/useful) to providing transparent replication of services in cases of failure.

Techniques for distributed operating-system control are already well-investigated, particularly in the networks and distributed systems field of research. It is highly likely that much of this previous work will be relevant when building a distributed RMoX.

5.3 User-Mode RMoX

Work is currently underway to provide a network interface for User-Mode RMoX, at either the Ethernet or IP level, using the 'tun-tap' driver in Linux[4]. This provides what is essentially a virtual 'network-card', allowing development and testing of the RMoX networking infrastructure (protocol stacks in particular) within the user-mode environment.

Although RMoX currently has little use for pointer interfaces (typically mice and pen-based devices), the provision of a virtual 'mouse' driver from the X11 interface should be trivial, using a similar mechanism to the user-mode 'keyboard' driver, that collects X11 key events from the 'vga' driver. Providing a similar driver for the console user-mode interface is trickier, but can be done by collecting data from 'gpm', a program that provides 'cut and paste' functionality in the Linux virtual consoles. Collecting similar events from X11 virtual terminals (such as an 'xterm') is hard, and can only be done by connecting to the xterm's X11 window and collecting keyboard events manually (through Xlib). The use of a mouse in the RMoX console environment is fairly limited, however, especially when we intend to provide graphical interfaces in the future (both for the raw-metal and user-mode versions of RMoX).

Work is currently underway to provide a native (occam) port of the MINIX [34] file-system, that is small, portable and widely implemented. As there is currently no support in RMoX for a real physical block device (such as a hard-disk or floppy-disk), the MINIX file-system driver is being developed in User-Mode RMoX using a virtual 'file' block device to emulate the missing physical device support.

5.4 Future Uses for User-Mode RMoX

The current use of User-Mode RMoX is largely to improve the 'test-debug' cycle time of RMoX itself. However, as RMoX matures, it is conceivable that the role of User-Mode RMoX will shift away from a development-only tool to more general uses. One perceived use is that of student investigations into concurrent operating-system design and implementation, within a CSP/occam framework. Both the X11 and console interfaces can be run remotely, making such investigation easily accessible — the only requirements are a Linux system and remote terminal or X11 display clients.

One potential future use, in conjunction with the user-space KRoC.net infrastructure [35], is to provide an interim mechanism for future experiments with distribution and replication of RMoX— with one intention being the creation of a clustered operating-system environment, as described in section 5.2. On a single system, multiple instances of User-Mode RMoX could be run to provide a 'virtual cluster'. Such setups would also be useful for testing distributed applications, which we later intend to run on real hardware clusters. Also, for demonstrating and experimenting with cluster computing in CSP/occam, where a real cluster might not available.

Acknowledgements

We would like to thank Peter Welch, Dyke Stiles, Adam Sampson and Ramsay Taylor from UKC, for their various input, ideas and code. We also wish to thank the WoTUG/CPA community, who have provided much discussion and enthusiasm for this work. In particular, Alex Brodsky, Matt Pedersen and Peter Welch, who came up with the name "RMoX".

[4]Similar functionality also exists in other operating-systems, as it provides a convenient way of allowing applications to access raw IP/Ethernet interfaces — typically with a different IP than that of the host-system, which simply forwards packets to/from the 'tun-tap' interface(s).

References

[1] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

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

[3] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN: 0-13-674409-5.

[4] Inmos Limited. occam 2.1 Reference Manual. Technical report, Inmos Limited, May 1995. Available at: `http://www.wotug.org/occam/`.

[5] Brian W. Kernighan and Dennis M. Ritchie. *The C Progamming Language*. Prentice Hall, Englewood Cliffs, 1 edition, 1978. ISBN: 0-13-110163-3.

[6] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, California, USA, June 2002. Available at: `http://www.usenix.org/events/usenix02/jim.html`.

[7] Daniel P. Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly UK, 2nd edition, January 2003. ISBN: 0-596-00213-0.

[8] P.H. Welch, J. Moores, F.R.M. Barnes, and D.C. Wood. The KRoC Home Page, 2000. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/kroc/`.

[9] 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.

[10] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes: Part I. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 331–361, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[11] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes: Part II. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 363–380, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[12] F.R.M. Barnes and P.H. Welch. Prioritised dynamic communicating and mobile processes. *IEE Proceedings – Software*, 150(2), April 2003.

[13] International Standards Organization, IEEE. Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language], 1996. ISO/IEC 9945-1:1996 (E) IEEE Std. 1003.1-1996 (Incorporating ANSI/IEEE Stds. 1003.1-1990, 1003.1b-1993, 1003.1c-1995, and 1003.1i-1995).

[14] J. Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In B.M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 147–168, Amsterdam, the Netherlands, April 1999. WoTUG, IOS Press. ISBN: 90-5199-480-X.

[15] J. B. Postel. Internet protocol. RFC 791, Internet Engineering Task Force, September 1981.

[16] Adam T. Sampson. The design and development of occamnet, 2003. Project report, Computing Laboratory, University of Kent.

[17] 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.

[18] M. Homewood, D. May, D. Shepherd, and R. Shepherd. The IMS T800 Transputer. *IEEE Micro*, pages 10–26, October 1987.

[19] Inmos Limited. *The T9000 Transputer Instruction Set Manual*. SGS-Thompson Microelectronics, 1993. Document number: 72 TRN 240 01.

[20] Nathalie Supornpaibul. Protocol implementation in occam. *Transputer Applications*, 1989.

[21] J. L. Romkey. Nonstandard for transmission of IP datagrams over serial lines: SLIP. RFC 1055, Internet Engineering Task Force, June 1988.

[22] J. B. Postel and J. F. Reynolds. Standard for the transmission of IP datagrams over IEEE 802 networks. RFC 1042, Internet Engineering Task Force, February 1988.

[23] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1999. Available at: `http://developer.intel.com/design/PentiumIII/manuals/`.

[24] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux OSKit: A substrate for kernel and language research. In *Symposium on Operating Systems Principles*, pages 38–51, 1997. Software available from: `http://www.cs.utah.edu/flux/oskit/`.

[25] Inmos Limited. *The T9000 Transputer Hardware Reference Manual*. SGS-Thompson Microelectronics, 1993.

[26] P.H. Welch and D.C. Wood. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, volume 47 of *Concurrent Systems Engineering*, pages 143–166. World occam and Transputer User Group, IOS Press, Netherlands, March 1996. ISBN: 90-5199-261-0.

[27] Jeff Dike. User-mode Linux. In *Proceedings of the 5th Annual Linux Showcase & Conference*, Oakland, CA, USA., November 2001. Available from `http://www.linuxshowcase.org/dike.html`.

[28] F.R.M. Barnes. occwserv – an occam Web-Server, September 2001. CPA-2001 SIG presentation, University of Bristol, UK. Available at: `http://frmb.org/publications.html`.

[29] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[30] David C. Wood. KRoC – Calling C Functions from occam. Technical report, Computing Laboratory, University of Kent at Canterbury, August 1998.

[31] Fred Barnes. *Socket, File and Process Libraries for occam*. Computing Laboratory, University of Kent at Canterbury, June 2000. Available at: `http://www.cs.ukc.ac.uk/people/rpg/frmb2/documents/`.

[32] F.R.M. Barnes. occwserv: an occam web-server. In *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, IOS Press, Amsterdam, The Netherlands, September 2003.

[33] Ramsay G. Taylor. Implementing Memory Management for Highly Concurrent Systems, 2003. Project report, Computing Laboratory, University of Kent.

[34] Andrew S. Tanenbaum. MINIX: A UNIX clone with source code for the IBM PC. *login: the USENIX Association newsletter*, 12(2):3–9, March/April 1987. ISSN: 1044-6397.

[35] M. Schweigler, F.R.M. Barnes, and P.H. Welch. Flexible, Transparent and Dynamic occam Networking with KRoC.net. In *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, IOS Press, Amsterdam, The Netherlands, September 2003.