Kent Academic Repository

Cupitt, John R. G. (1989) *The design and implementation of an operating system in a functional language.* Doctor of Philosophy (PhD) thesis, University of Kent.

Downloaded from

https://kar.kent.ac.uk/94289/ The University of Kent's Academic Repository KAR

The version of record is available from

https://doi.org/10.22024/UniKent/01.02.94289

This document version

UNSPECIFIED

DOI for this version

Licence for this version

CC BY-NC-ND (Attribution-NonCommercial-NoDerivatives)

Additional information

This thesis has been digitised by EThOS, the British Library digitisation service, for purposes of preservation and dissemination. It was uploaded to KAR on 25 April 2022 in order to hold its content and record within University of Kent systems. It is available Open Access using a Creative Commons Attribution, Non-commercial, No Derivatives (https://creativecommons.org/licenses/by-nc-nd/4.0/) licence so that the thesis and its author, can benefit from opportunities for increased readership and citation. This was done in line with University of Kent policies (https://www.kent.ac.uk/is/strategy/docs/Kent%20Open%20Access%20policy.pdf). If you ...

Versions of research works

Versions of Record

If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

Author Accepted Manuscripts

If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in ***Title of Journal*** , Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

Enquiries

If you have questions about this document contact ResearchSupport@kent.ac.uk. Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our Take Down policy (available from https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies).

THE DESIGN AND IMPLEMENTATION OF AN OPERATING SYSTEM IN A FUNCTIONAL LANGUAGE

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
John R. G. Cupitt
October 1989

Abstract

Operating systems are notoriously difficult programs to write. This thesis deals with the design and implementation of a small operating system in the purely functional language Miranda[1][26]. Functional languages promise extreme brevity of expression, combined with the possibility of formal verification. This thesis aims to show that an operating system in a functional language can be both easy to write and easy to reason about.

[1]Miranda is a trademark of Research Software Ltd.

Acknowledgements

I should like to gratefully acknowledge the help and encouragement of my supervisor David Turner during the course of this research. I should also like to thank Simon Thompson for several lengthy discussions, and for help when I was stuck while reading up on the more theoretical aspects of the subject. Many thanks go to Mark Longley, Gareth Howells and Rafael Lins for their ideas about functional programming, and to my family for their love.

The quotes in the chapter headings are taken from UBIK by Philip K. Dick and should not be read.

Contents

Abstract ii

Acknowledgements iii

1 Introduction 1

1.1 Basic Operating System Structure 2

1.2 Comparison with Other Work 5

2 Non-determinism and Functional Programming 6

2.1 Introducing Non-determinism 7

2.2 Stoye's Sorting Office 11

2.3 Cleaning up the Types 12

2.4 Synchronised Streams 13

2.5 Removing Processes 17

2.6 Controlling Devices 18

2.7 Switching to Continuations 18

2.8 A Small Kernel 22

2.9 A Worked Example 28

2.10 An Outline of a Semantics for the KAOS Kernel 30

2.11 A Sample Proof . . . 32

2.12 Discussion . . . 37

3 The Structure of KAOS **38**

3.1 The Real KAOS Kernel . . . 39

3.1.1 The KAOS Kernel Interface . . . 39

3.1.2 Extensions to Miranda . . . 42

3.2 KAOS Message Structure . . . 43

3.3 The Event Manager . . . 45

3.4 Device Drivers . . . 47

3.5 The File System . . . 51

3.6 The Resource Tracker . . . 52

3.6.1 Tracking Process Creation . . . 54

3.7 Current Difficulties . . . 55

3.8 Implementation . . . 57

4 A Better Kernel for KAOS **58**

4.1 What is Missing? . . . 58

4.1.1 Non-determinism . . . 58

4.1.2 Memory Management . . . 60

4.1.3 Exception Handling . . . 61

4.2 New Features for KAOS . . . 61

4.3 Discussion . . . 64

5 Programming Operating System Processes **65**

5.1 Higher Order Functions for Sequencing . . . 66

5.2 The `stdio` Library . . . 69

5.3 Programming System Processes 71

6 KAOS Applications **80**

6.1 The KAOS shell 80

6.1.1 Shell Command Syntax 80

6.1.2 Programs Available 82

6.1.3 Shell Commands 84

6.1.4 Variable Substitution 85

6.1.5 Command Sequencing 86

6.1.6 Shell Syntax 87

6.1.7 A Sample Session 87

6.1.8 Inside the Shell 88

6.2 `medit` — A Screen Editor Under KAOS 88

6.2.1 Changing the Input Style 89

6.2.2 Isolating File Input and Output 90

6.2.3 Cosmetic Changes 90

7 Conclusions and Future Developments **92**

A KAOS Kernel Source **97**

A.1 Introduction 97

A.2 Type definitions 98

A.3 Useful functions 102

A.4 Kernel functions visible to the outside world 103

A.5 Kernel device control functions 104

A.6 The scheduler 106

A.7 Process removal 111

A.8 Message passing 113

A.9 Process creation 115

B KAOS Libraries **117**

B.1 Introduction 117

B.2 `misc.m` — Various useful functions 117

B.3 `char.m` — Useful character functions 120

B.4 `key.m` — Maintain keyed lists 122

B.5 `lift.m` — Add an error element to a type 127

B.6 `errs.m` — KAOS error codes 128

B.7 `collection.m` — Manage hetrogenous lists 130

C Interactions **133**

C.1 Introduction 133

C.2 `lowinter.m` — The core of `interact` 134

C.3 `inter.m` — Useful functions over `lowinter.m` 138

List of Figures

1 The Canonical Operating System 4
2 A Parallel Solution to the Hamming Numbers Problem 8
3 A Network Requiring a Non-deterministic Solution 9
4 The Sorting Office 12
5 A Process 20
6 Two Processes 21
7 Sketch of the Major KAOS System Processes 45

"Friends, this is clean up time, and we're discounting all our silent, electric Ubiks by this much money. Yes, we're throwing away the bluebook. And remember: every Ubik on our lot has been used only as directed."

Chapter 1

Introduction

It has been claimed[2] that operating systems are among the most complicated things ever constructed by man. Because of this complexity, as any computer user can tell you, they are invariably riddled with bugs. Proponents of functional languages have claimed for years that the functional paradigm makes for far faster and less troublesome development of programs, and even opens up the possibility of the application of formal methods to large systems. This thesis describes the design and implementation of KAOS (for Kent Applicative Operating System), a small multi-user operating system developed entirely in the purely functional language Miranda.

There is another reason why an operating system written in a functional language might be an interesting exercise. Functional languages are usually used in an environment in which a user enters an expression and waits for the value to be printed. It is not clear how one might express an interactive process as a function — this is still a research topic, see for example [25]. Operating systems perhaps represent the archetype of this sort of programming problem: an operating system must control the execution of other programs, interface to physical devices, provide interconnections

between programs and users and so on. If we can successfully tackle these problems in a purely functional style, then it will be quite a step forward for functional programming.

The remainder of this first chapter tries to explain what an operating system should do and argues for a structure along the lines of UNIX[1]. Chapter two discusses the features that need to be added to a functional language to make it possible to write an operating system, and presents a tiny operating system kernel based on these ideas. It ends with a sketch of a semantics for an even smaller operating system. Chapter three presents the complete KAOS kernel and describes in detail the additions to Miranda that were necessary in order to implement it. It goes on to cover the higher levels of KAOS, explaining how KAOS handles interrupts, the file system and resource allocation. Chapter four describes the features present in operating systems like UNIX but still missing from KAOS and then shows how these capabilities might be added. Chapter five discusses the problems involved in systems programming in functional languages and presents a systematic scheme for handling side effects and interrupts. Chapter six discusses the two large applications that run under KAOS: the KAOS shell and a port of the Miranda screen editor, `medit`. Finally, chapter seven lists the areas in which KAOS is still deficient and suggests some avenues for future research.

1.1 Basic Operating System Structure

An operating system provides a cushion between the user and the machine. We can list four basic characteristics which an operating system should possess

- The operating system should allow several people to share the machine.

[1] UNIX is a trademark of AT&T and Bell Laboratories in the USA and other countries

- It should manage a persistent set of objects for each user.
- It should provide a convenient environment for each user to operate in. It should make it easy to manipulate objects in the persistent store, start up programs and so on.
- It needs to provide an environment for programs. It should be easy to write programs such as editors and compilers, and they should have controlled, high-level access to the machine's facilities.

These demands have a number of simple consequences for the design of an operating system.

A multi-user operating system will need to support a set of independent processes and these processes should have shared access to the resources of the machine, such as printers, discs, terminals and so on. How are these processes to exist comfortably together? The operating system must clearly provide a *virtual machine* for each process: each virtual machine will be a cleaned up version of the real underlying machine. The function of the operating system is then to provide a 'nice' mapping of a set of virtual machines onto the single physical machine. The environment for programs demanded above will then be a set of libraries providing a pleasant interface to a virtual machine, and the environment for the user will be a program which allows this mapping to be controlled.

It is perhaps harder to see that the requirements above also imply *non-determinism*. As new computations are started by each user, the operating system will need to perform them not in some predefined order, but rather overlapped in some way. It clearly is no good at all if one user wanting to know the seven millionth digit of π prevents anyone else from using the machine for ten minutes! Moreover, we cannot

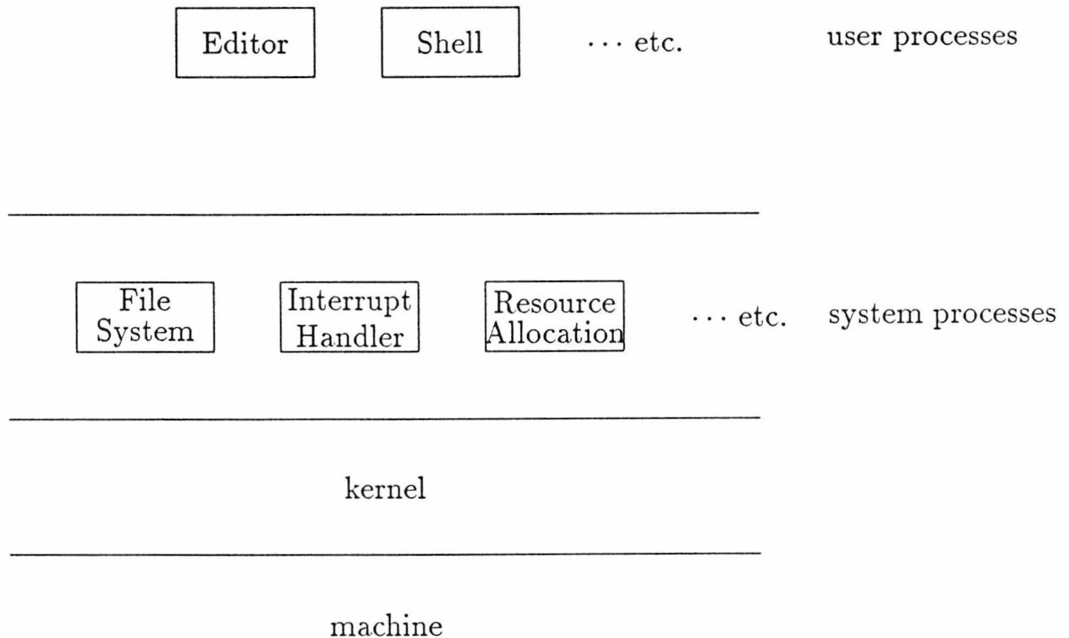


Figure 1: The Canonical Operating System

decide in advance how processes are to be overlapped — it will depend of their relative execution speeds. Any system whose result depends on the exact number of CPU cycles a computation will take must be non-deterministic. This problem is discussed at greater length in chapter two.

While the lowest levels of the operating system will clearly have to deal with non-determinism, this is not a complication we want any higher up than strictly necessary — it can only make reasoning harder. We therefore split the operating system into two main levels: we have a *kernel* which supports non-deterministic parallel execution of a set of processes, and over this a *system layer* in which a number of privileged address the problems of resource allocation and sharing. Finally, user processes sit over the system layer. Figure 1 gives an overview of this canonical operating system.

1.2 Comparison with Other Work

There have been a number of operating systems written in functional languages. Perhaps the one closest to KAOS is the operating system produced by Stoye for his SKIM II machine[21]; the scheme KAOS uses for introducing non-determinism is based on Stoye's. This was not intended to be a full operating system, it was just a collection of utilities produced during the development of the machine.

Jones has produced a number of small operating systems written in functional languages[15, 14]. These are based on the `merge` operator described in section 2.1. These systems were not produced as practical operating systems, but rather as programming exercises.

More complete systems have been produced or proposed, for example Nebula[8, 19], but these tend to be based on the somewhat *ad-hoc* use of operators like `merge`, and as a result seem almost impossible to reason about.

In summary, KAOS is a relatively complete operating system in a functional language with the beginnings of a formal semantics. We believe that it would make a sound basis upon which to construct a fully practical multi-user operating system in a functional language.

"Instant Ubik has all the flavour of just-brewed drip coffee. Your husband will say, Christ, Sally, I used to think your coffee was only so-so. But now, wow! Safe when taken as directed."

Chapter 2

Non-determinism and Functional Programming

There are several problems to be overcome before a functional language can be used to write an operating system. The most obvious is one of efficiency — in the past, functional languages have been hundreds of times slower than imperative languages. This does now seem to be changing with the development of new implementation techniques such as TIM[5] and the C.M.-C.M.[11]. At a slightly more subtle level, it's quite difficult to find an elegant way of representing many operating system concepts in functional languages. In particular, parallelism in the form exhibited by operating systems like UNIX require one, as we noted in the introduction, to introduce non-determinism.

This chapter first outlines a number of the schemes that have been proposed for adding non-determinism to functional languages. We cover in detail the scheme we shall use and present a listing of a fairly complete kernel written in this style. The chapter ends with a sketch of a semantics for our kernel.

2.1 Introducing Non-determinism

We first present another perspective on the reasons for the introduction of non-determinism. We shall consider the sorts of process networks that can be represented in functional languages, and aim to show that these networks are inadequate as the basis for an operating system.

A deterministic process can be conveniently represented as a lazy function from a list of the messages sent to it to a list of the messages it sends to other processes. The process has internal state, as each output item can be a function of the entire input history up to the moment when the output item is sent. In Miranda such a process has type

```
process * ** == [*] -> [**]
```

We can form networks of these processes by simply using the output from one process as the input to another. A good example of this style of programming is the well known communicating-process solution to the Hamming numbers problem. Here one has to print in ascending order all numbers of the form $2^a3^b5^c$ where a, b, c can be any natural numbers. The network shown in figure 2 is described by the Miranda function output below

```
output ::  [num]
output
      = 1:(mult 2 output $join mult 3 output) $join mult 5 output
        where
        mult n = map (*n)
        join (a:x) (b:y)
              = a:join x (b:y), a < b
              = b:join (a:x) y, a > b
              = a:join x y, a = b
```

See [17, 28] for more on this style of programming.

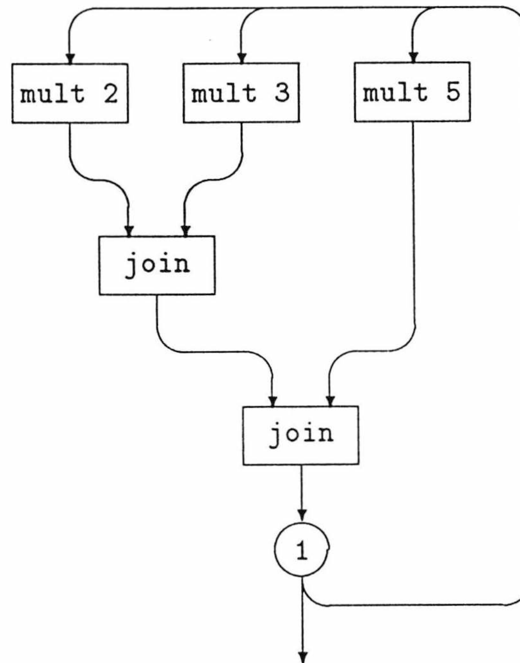


Figure 2: A Parallel Solution to the Hamming Numbers Problem

Now consider the network in figure 3. Here we have two processes both communicating with a single filing system process. How are we to represent this network in a functional language? The problem is that the filing system needs to know in what order the processes will send requests; it cannot simply take a request first from one process and then from the other, as this would require the processes to emit requests alternately!

One of the simplest solutions is to require processes to emit special 'null' messages at regular intervals to mark the passage of time. These null messages were dubbed *Hiatons* by Wadge, as they mark a 'hiatus' of activity on a channel. So for example, in figure 3 the processes would be required to emit a hiaton perhaps every second or so. The filing system can then simply alternate between the processes; if one process is not using the filing system for a while, then the filing system will just see a hiaton rather than having to wait until the process does want to use the filing system again.

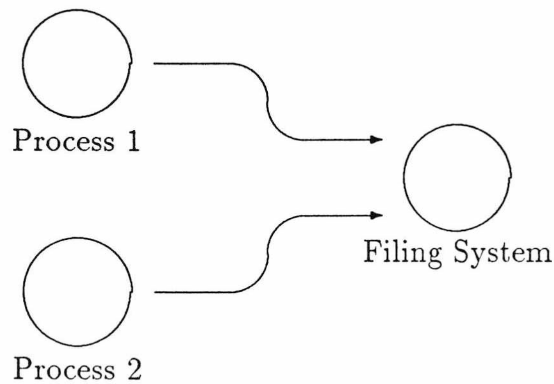


Figure 3: A Network Requiring a Non-deterministic Solution

This discipline has to stretch throughout the whole operating system. Every process is under obligation to emit regular hiatons if everything is to keep running. From a practical point of view, this is very inefficient. We have what is known in operating system terminology as *busy waiting*: the entire system is in constant motion as hundreds of hiatons move between idle processes. More seriously, networks constructed with hiatons cannot be made *bottom avoiding*. Suppose an undebugged user process reduces to $\perp$ and stops sending hiatons? There is a good chance that the whole system will stop running!

It should be clear that no scheme in a pure functional language can describe a bottom avoiding network, since bottom avoidance is essentially non-deterministic. We are forced to introduce some new primitive.

One of the most common non-deterministic primitives is the `merge` operator, as used by Henderson[6]. This is a function which evaluates two lists in parallel, producing a single list of elements in order of computation time. `merge` might be written in Miranda as

```
merge ::  [*] -> [*] -> [*]
```

```
merge x y
      = test x, ready x
      = merge y x, otherwise
        where
        test [] = y
        test (a:b) = a:  merge y b
```

(Where `ready` is a hypothetical (and very unsound!) 'function' that side-effects its argument, performing some fixed number of reductions and returning `True` if it manages to reduce it to a base type, a constructor function or a lambda within that time, and `False` otherwise. Note that this is a very disciplined form of `merge`: it is both fair and bottom avoiding. The issues connected with this are discussed at greater length in section 2.12)

We now have another problem. By introducing this operator we have lost the ability to reason equationally about our programs. For example

```
merge [2] [4]
```

might reduce to

```
[2,4]
```

We would then expect

```
merge [1+1] [4]
```

to have the same value, but of course it won't! This might well reduce to

```
[4,2]
```

Other types of `merge` can be even stranger. If the two streams being merged come from different processors, perhaps over a network, then `merge` applied to the *same* arguments might yield different values on different occasions. All these things

combine to make reasoning about our programs far more complicated. A number of models for functional programs incorporating `merge` have been put forward[18, 1], but all are rather unwieldy. Clearly, `merge` is a bad thing and ought to be used as little as possible. In [22], Stoye suggested a variation on `merge` which confines its use to a single instance outside the language. We shall now consider this scheme in detail.

2.2 Stoye's Sorting Office

Stoye represents each process running in the machine as a lazy deterministic function from the stream of messages sent to the process to a stream of messages the process wishes to send to other processes. Each message sent needs to have the address of the destination process attached to it. Using Miranda we can write this as

```
process == [object] -> [(address, object)]
```

Connecting together all the processes, Stoye has a *sorting office*. This acts rather like a postman, taking pairs of addresses and objects from the output of each process and feeding the objects into the inputs of the destination processes; see figure 4. The key point here is that the sorting office acts *non-deterministically*. It reduces all the processes concurrently, routing messages not in some predefined order, but rather in the order in which the processes produce them.

We can use `merge` to write a sorting office in Miranda

```
sorting ::  [(address, object)]
sorting
      = mergelist [
                process1 (input_for address_1),
                process2 (input_for address_2),
                ...  etc.
        ]
```

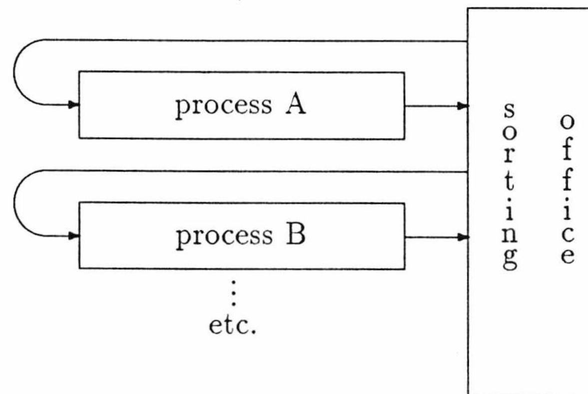


Figure 4: The Sorting Office

```
where
mergelist = foldr merge []
input_for adr = [ x | (d, x) <- sorting; d = adr ]
```

We now only have a single instance of merge in the entire system. We have split the world into two levels: ordinary functional programs inside each process to which we can apply all of the usual proof techniques, and a simple non-deterministic agent linking them all together.

2.3 Cleaning up the Types

In [27], Turner suggested a method for giving individual types to processes. Each process consumes a list of objects of some specific type and produces a list of `message`.

```
process * == [*] -> [message]
```

Here, `message` is an opaque type. The only way that processes can make `message`s is by using special wrapper functions. These take an object of the input type of some

process and wrap it up (together with the address of that process) inside a `message`.

```
wrapper * == * -> message
```

So a process can only send a `message` to another process if it possesses a wrapper function for that process.

Wrapper functions are made by the sorting office when a new process is created. Processes are created by using a special wrapper called `spawn` to wrap up the function that is to be the new process, and then sending the resulting `message` to the sorting office. The sorting office sets up the new process, hands it its wrapper and starts it going. Obviously, owning your own message wrapper is not very useful. The new process will have to send this new wrapper off to some other processes using wrappers it inherited from its parent.

```
spawn :: (wrapper * -> process *) -> message
```

(Turner handled process creation in a slightly different manner, but I believe the spirit is the same.)

By requiring processes to use wrappers to send messages we have allowed each process to have a different input type, but kept static typechecking. This is an important addition to the flexibility of the system.

2.4 Synchronised Streams

Our next step is to require that processes emit the special message `WAIT` immediately prior to reading their input. This has the effect of making public the process's dependencies between input and output. Stoye has this idea in his thesis[21] and uses it to get correct synchronisation for interactive programs, but he never applies it to the sorting office.

If we do this, processes then make enough information available about how they are going to behave to allow us to write a version of the sorting office that doesn't use `merge`. The advance is that now the sorting office can differentiate between processes that it can safely demand more output from and those that are blocked on input by applying these rules whenever a process emits a message

- If the process emits the special `WAIT` message and the total number of `WAIT`s the process has emitted so far is greater than the total number of messages sent to the process, then the process has become blocked on input.
- If the process emits an object to be sent to some destination process d and the total number of messages sent to d is equal to the total number of `WAIT`s emitted by d, then d was previously blocked but is now runnable again.

(These rules express *asynchronous* message passing, where processes only ever stop if they have no input available. It is straightforward (if a little fiddly) to extend these rules to express *synchronous* message passing, where senders block until their message has been read. Synchronous message passing is generally thought to be more fundamental, and in fact is the style we will use later.)

These rules are easily expressed in Miranda. A message is now either a `WAIT` or a `SEND`. For each process running, we will keep track of the output list for that process and the number of messages that that process has yet to consume

```
process_state == ([message], num)
message ::= WAIT | SEND process_address object
process_address == num
```

The state of the sorting office as a whole comes in two parts. First, we have a list of the `process_state`s for all the processes currently in the machine. Process

addresses will be indexes into this list. The second part is a run queue — a list of the addresses of all the processes which are currently runnable. The process whose address is at the head of the run queue is the next to be run

```
machine_state == ([process_state], [process_address])
```

The sorting office is a function taking a machine_state to a [message]. It finds a possible interleaving of the process' output lists. We can also see the sorting office as controlling the execution of processes. Each time it demands the next output item from a process, it is forcing the evaluation of a small part of that process.

There are three main cases — the first is that the run queue is empty, that is, that there are no runnable processes in the operating system. We have either deadlocked or the operating system has terminated. If the run queue is not empty, then we demand the next output item from the process at the head of the run queue, and choose one of the other two cases depending on whether or not it is the special message WAIT.

```
sorting ::  machine_state -> [message]
sorting (plist, run)
      = [], run = []
      = handle_wait, x = WAIT
      = handle_send, otherwise
        where
        (p:tailrun) = run
        (x:tailproc, waiting) = plist!p
        run' = postfix p tailrun
```

Second case — the process has emitted the WAIT message. If this process has no messages waiting for it, it becomes blocked on input. If there are messages waiting, we can just put its address back on the end of the run queue.

```
handle_wait
     = x:sorting (plist', tailrun), waiting = 0
     = x:sorting (plist', run'), otherwise
```

```
        where
        plist' = update p (tailproc, waiting - 1) plist
```

(`update` is a function to replace an element of a list. `update n x l` is a new version of `l` in which element `n` has been replaced by `x`.)

Final case — the process is sending a message to another process. If the waiting count on the destination process was zero, then it was previously blocked on input and should now go back on the run queue. Otherwise, we can just put the sender's address back on the end of the run queue.

```
handle_send
      = x:sorting (plist'', run''), dwaiting = -1
      = x:sorting (plist'', run'), otherwise
        where
        plist' = update p (tailproc, waiting) plist
        plist'' = update d (dproc, dwaiting + 1) plist'
        run'' = postfix d run'
        (SEND d obj) = x
        (dproc, dwaiting) = plist!d
```

We make the input stream for each process by applying some suitable filter function to the output of `sorting`. For example, we might run a process `wombat` at address zero by

```
runwombat ::  [message]
runwombat
      = sorting ([(wombat wombat_input, 0)], [0])
        where
        wombat_input = [ x | SEND d x <- runwombat; d = 0 ]
```

(Note that for the sake of simplicity we are not using Turner's type scheme here. We are also assuming that processes produce an infinite list of messages once they are started.)

Our new sorting office is not quite the same as the original, however. Execution switches between processes only when they send or receive messages, never while a process is actually computing: it is no longer bottom avoiding. Things are not hopeless however. The merge-free sorting office does make a useful prototype and in some applications this kind of security is not necessary. Chapter 4 discusses these issues at greater length.

It is interesting to note that this form of parallelism occurs in conventional operating systems as well. Systems such as NeWS[24] and MultiFinder for the Macintosh series of computers are based on *non-preemptive schedulers* similar to the one presented above.

2.5 Removing Processes

We need some way of removing processes from the system once they have done their job. The obvious way of doing this is to allow processes to reduce to `[]` — the sorting office can spot this and then take the process out of the process list.

Adding process removal to the sorting office introduces a nasty problem: the wrapper functions described in section 2.3 can persist after the process to which they connect has gone. We need to add the stipulation that after using a send wrapper, the next item on the process's input is a special value indicating whether or not the message was sent successfully.

We also need some way of forcibly removing processes from the system. A simple way of doing this is to make the sorting office hand a new process a kill wrapper along with its send wrapper. Using this kill wrapper will cause the sorting office to remove this process.

```
killwrapper :: message
```

```
spawn :: (wrapper * -> killwrapper -> process *) -> message
```

As with send wrappers, kill wrappers can become out of date. We need to say that after using a kill, the next item on the process's input is a success or failure value.

2.6 Controlling Devices

We need to add facilities for controlling devices such as discs and terminals. We do this by adding more special wrapper functions and more guarantees about what will appear next in the process's input stream. For example, a process might talk to a disc with wrappers like

```
write_disc :: blockNumber -> block -> message
```

where the next item on the process's input is a success or failure value, and

```
read_disc :: blockNumber -> message
```

where the next item on the process's input is either failure, or success coupled with the `block` that was read.

It seems sensible to change our terminology slightly — while 'sorting office' is quite acceptable for something that simply routes messages, we have now added enough extras to allow us to call what was the sorting office the *kernel* of an operating system.

2.7 Switching to Continuations

If one tries to implement a kernel based on the above sketch, things rapidly become very awkward indeed.

- The curious guarantees about the next item on the input of a process after a send, kill or device message has been sent require that we delay adding anything to the output of `sorting` unless we know that it will be used straight away. The behaviour of the sorting office is far simpler if we use continuations, where input and output are tied together.

- Continuations will work as well with a strict language. Stream based systems depend very heavily on the precise semantics of the lazy evaluation of lists — not always as simple a thing as one might think, as [25] shows.

- As we will see, the strange type `message` that was present in the stream based model disappears when we switch to continuations.

Continuations are used in Denotational Semantics[23] to model constructions such as `goto` statements. The idea is that instead of having functions always returning their results to their caller, functions are instead passed a number of other functions, one of which they can choose to pass a value on to. In general, a function using continuations might look something like

$$f : \underbrace{\alpha \rightarrow \cdots \rightarrow \beta}_{a} \rightarrow \underbrace{(\gamma \rightarrow \cdots \rightarrow \delta)}_{b} \rightarrow \cdots \rightarrow \underbrace{(\epsilon \rightarrow \cdots \rightarrow \delta)}_{c} \rightarrow \underbrace{\delta}_{d}$$

Where a represents the 'ordinary' arguments to f, b represents the first continuation available to f and c represents the final continuation available. d is the result of the entire computation.

Instead of being lazy functions over lists, processes are now simply objects of type `process` (corresponding to δ in the above sketch). This is an opaque type whose internal structure does not yet concern us. New processes are now handed three functions as they start up: a *send* function which can be used to send messages to

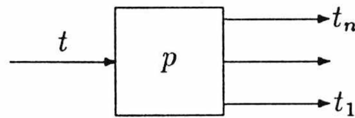


Figure 5: A Process

this process, a *receive* function which this process can use to get its next item of input and finally a *kill* function which can be used to remove this process.

```
send * == * -> process -> process
receive * == (* -> process) -> process
killfun == process -> process
newProcess * == send * -> receive * -> killfun -> process
```

A send function takes as arguments an item of the input type of the process to which the function sends and a continuation. When the message has been accepted, control passes to the continuation. A receive function takes a continuation expecting a message of the input type. This continuation is handed the next message sent to that process. A kill function takes a single continuation. Control passes to this continuation when the process has been removed. Clearly, if a process executes its own kill wrapper, the continuation will never be used. As was hinted at above, communication is intended to be *synchronous*: senders block until the message has been read by the receiver, receivers block until a message arrives.

So for example, consider the process p shown in figure 5. This takes the type t as input, and can send messages of type t_1 to t_n. Its type would be

$$p :: \texttt{send } t_1 \texttt{ -> } \cdots \texttt{ -> send } t_n \texttt{ -> newProcess } t$$

New processes are created with `spawn`, a function exported by the kernel. This takes two arguments: the function which is to be the new process and a continuation

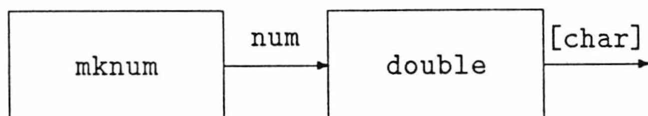


Figure 6: Two Processes

to which control passes when the process has been created.

```
spawn ::  newProcess * -> process -> process
```

Devices can be controlled with more functions exported by the kernel

```
readDisc ::  blockNumber -> (block -> process) -> process
writeDisc ::  blockNumber -> block -> process -> process
```

As an example, here are two processes exchanging messages (See figure 6). mknum makes a series of numbers and pipes them to double. double doubles each one, converts it to a [char] and sends it off to some other process.

mknum posts its numbers off using a send function it was handed as it started up. As mknum never reads any input, we leave its input type as the empty tuple.

```
mknum ::  send num -> newProcess ()
mknum outfun mknumsend mknumreceive mknumkill
      = loop 42
        where
        loop n = outfun n (loop (n + 1))
```

double is itself handed a send function as it starts up. It first spawns mknum, handing it a send function

```
double ::  send [char] -> newProcess num
```

```
double sendfun doublesend doublereceive doublekill
      = spawn (mknum doublesend) (double' sendfun doublereceive)
```

And then loops, reading numbers sent by mknum, doubling them, converting them to [char] and sending them off

```
double' ::  send [char] -> receive num -> process
double' sendfun doublereceive
      = loop
        where
        loop = doublereceive loop'
        loop' n = sendfun (show (n * 2) ++ ", ") loop
```

2.8 A Small Kernel

This section presents a listing of a kernel based on continuations. Appendix A contains the source to the complete kernel used in the current implementation of KAOS. It is a greatly expanded version of the kernel below. For simplicity we will omit process removal, but we will include a version of the type discipline described in section 2.3. As there is no process removal, for the definitions below read

```
newProcess * == send * -> receive * -> process
```

We hold the current state of a process as an algebraic type. The * represents the input type of the process.

```
process_state *
    ::= RUN process [(*, process_id)] |
        IN (* -> process) |
        OUT process [(*, process_id)]
```

RUN The process is runnable. We need to keep the current continuation and a queue of processes waiting to send to this process. For each waiting process, we hold

the message the process is trying to send and the `process_id` of the sender. `process_ids` are small integers used to identify processes.

`IN` The process is blocked on input. We need only keep track of the continuation.

`OUT` The process is blocked on output. We need to hold the continuation and a list of the processes waiting to send to this process.

We will need to store a list of these things inside the kernel. Unfortunately, the * will take a different binding for each process and Miranda requires that every element of a list has the same type. We get around this problem with the special function `changetype`, whose purpose is to smuggle things past the typechecker

```
changetype ::  * -> **
changetype = id
```

(This is only intended to give the reader an idea of what `changetype` is like. This is most definitely not a legal definition in Miranda.)

It is interesting to note that it is possible to hide the use of `changetype` inside a much smaller abstype, and then to write the kernel in terms of this. Section B.7 shows how this might be done. A kernel written in this way is however far more cumbersome.

We will hold the process states as `[process_state item]`, where `item` is supposed to represent some untyped expression. We turn `*`s into `item`s and back using `changetype`. Our use of message passing functions ensures that playing with types in this way is safe — these functions guarantee that `*`s will always be `changetype`d back into their original form.

```
item == num || Could be anything
```

The state of the kernel is again a pair consisting of a list of all the processes and

a run queue.

```
machine_state == ([process_state item], [process_id])
process_id == num
```

A `process_id` is again an index into the `[process_state item]`. We adopt the convention that the current process is the one whose `process_id` is at the head of the run queue.

We are now able to reveal exactly what a `process` is — it's a function from a `machine_state` to a list of `sys_message`, the Miranda console message type.

```
process == machine_state -> [sys_message]
abstype process
with    writeTty ::  [char] -> process -> process
        spawn ::  newProcess * -> process -> process
        start ::  newProcess * -> [sys_message]
        schedule ::  process
        pre_send ::  process_id -> send *
        pre_receive ::  receive *
```

`writeTty` sends a list of characters to a terminal. There should clearly be more of these device functions, but `writeTty` serves to give the flavour.

```
writeTty ::  send [char]
writeTty str cont mstate = Stdout str:cont mstate
```

(Note that we could make `writeTty` call the scheduler, rather than always returning to its caller.)

`spawn` creates a new process and adds it to the end of the run queue. Note how we create the message passing functions for the new process by partially parameterising `pre_send`, instantiating the type variable with the input type of the new process.

```
spawn ::  newProcess * -> process -> process
spawn newprocess cont (process_list, run)
```

```
        = cont (process_list', run')
          where
          newpid = #process_list
          run' = postfix newpid run
          process_list' = postfix newp process_list
          newp = RUN (newprocess ptx prx) []
          ptx = pre_send newpid
          prx = pre_receive
```

schedule puts the old current process at the end of the run queue and executes the next process. If the run queue is empty, the kernel can just exit.

```
schedule ::  process
schedule (process_list, (a:rest))
      = cont (process_list, run')
        where
        run' = postfix a rest
        (RUN cont pend) = process_list!(hd run')
schedule (process_list, [])
      = [Stdout "run queue empty"]
```

start sets up the first process and calls the scheduler. We type this at the Miranda prompt to get off the ground.

```
start ::  newProcess * -> [sys_message]
start root
      = schedule ([rootp], [0])
        where
        rootp = RUN (root rtx rrx) []
        rtx = pre_send 0
        rrx = pre_receive
```

pre_receive first checks the queue of pending messages attached to the receiver. If there are no other processes waiting, the receiver becomes blocked on input. Otherwise, we copy a message across and restart the sender. Section 2.9 contains a worked example — it may be helpful to read that in conjunction with these definitions.

```
pre_receive ::  receive *
```

```
pre_receive rcont (process_list, run)
      = switch (process_list!rpid)
        where
        (rpid:rest) = run
```

There are no processes waiting to send to this process. We block this process on input, remove it from the run queue and exit through the scheduler.

```
switch (RUN oldcont [])
      = schedule (process_list', rest)
        where
        rnew = IN (changetype rcont)
        process_list' = update rpid rnew process_list
```

There are processes waiting. We take the sender at the head of the queue, copy the message across, change the sender from OUT to RUN, add the now runnable sender to the run queue and exit through the scheduler.

```
switch (RUN oldcont ((x, spid):other))
      = schedule (process_list'', run')
        where
        run' = postfix spid run
        (OUT scont spend) = process_list!spid
        snew = RUN scont spend
        process_list' = update spid snew process_list
        rnew = RUN (rcont (changetype x)) other
        process_list'' = update rpid rnew process_list'
```

pre_send switches on the state of the receiver. If the receiver is blocked on input, we can just copy the message across. Otherwise, we have to block the sender on output and add it to the end of the receiver's pending list.

```
pre_send :: process_id -> send *
pre_send rpid x scont (process_list, run)
      = switch (process_list!rpid)
        where
```

```
(spid:rest) = run
```

The receiver is blocked on input. We copy the message across, change the receiver back into RUN, add it to the run queue and exit through the scheduler.

```
switch (IN rcont)
      = schedule (process_list'', run')
        where
        run' = postfix rpid run
        (RUN oldcont spend) = process_list!spid
        snew = RUN scont spend
        process_list' = update spid snew process_list
        rnew = RUN (rcont (changetype x)) []
        process_list'' = update rpid rnew process_list'
```

The receiver is running and thus not interested in receiving. We change the sender into OUT, add it to the end of the receiver's pending list and exit through the scheduler.

```
switch (RUN rcont rpend)
      = schedule (process_list'', rest)
        where
        (RUN oldcont spend) = process_list!spid
        snew = OUT scont spend
        process_list' = update spid snew process_list
        rnew = RUN rcont (postfix (changetype x, spid) rpend)
        process_list'' = update rpid rnew process_list'
```

And finally, if the receiver is itself blocked on output, the sender has to queue up on that.

```
switch (OUT rcont rpend)
      = schedule (process_list'', rest)
        where
        (RUN oldcont spend) = process_list!spid
        snew = OUT scont spend
        process_list' = update spid snew process_list
        rnew = OUT rcont (postfix (changetype x, spid) rpend)
        process_list'' = update rpid rnew process_list'
```

2.9 A Worked Example

This section runs through the evaluation of `start (double writeTty)`, with the intention of animating the definitions above.

`start` will build an initial `machine_state` and then call the scheduler.

```
startp = double writeTty (pre_send 0) pre_receive
start_state = ([RUN startp []], [0])
schedule start_state
```

`schedule` rotates the run queue and chooses the next process to be executed. Obviously, this will be the one we have just created.

```
double writeTty (pre_send 0) pre_receive start_state
```

`double` spawns `mknum`

```
spawn (mknum (pre_send 0)) (double' writeTty pre_receive) start_state
```

And `spawn` adds `mknum` as a new process, before returning to `double'`.

```
newp = mknum (pre_send 0) (pre_send 1) pre_receive
new_state = ([RUN startp [], RUN newp []], [0,1])
double' writeTty pre_receive new_state
```

`double'` now reads its input

```
pre_receive loop' new_state
```

And will clearly have to become an `IN` until `mknum` is run. `pre_receive` puts `loop'` back in the state and calls the scheduler again.

```
newer_state = ([IN (changetype loop'), RUN newp []], [1])
```

```
schedule newer_state
```

And `schedule` obviously has no choice but to run `mknum`.

```
mknum (pre_send 0) (pre_send 1) pre_receive newer_state
```

`mknum` generates the first number, and sends it off.

```
pre_send 0 42 (loop 43) newer_state
```

`pre_send` looks at the destination process, sees that it is `IN` at the moment, passes the message across, changes the `IN` into a `RUN` and calls the scheduler.

```
(process_list, run) = newer_state
(IN (changetype loop')) = process_list!0
(RUN newp []) = process_list!1
newest_state = ([RUN (loop' 42) [], RUN (loop 43) []], [1,0])
schedule newest_state
```

`schedule` rotates the run queue, choosing process `0` as the next to be run.

```
final_state = ([RUN (loop' 42) [], RUN (loop 43) []], [0,1])
loop' 42 final_state
```

`double'` has now received its first number. It doubles it, turns it into a `[char]` and calls `writeTty`.

```
writeTty "42, " loop final_state
```

And `writeTty` prints the characters to the screen.

```
Stdout "42, ":loop final_state
```

2.10 An Outline of a Semantics for the KAOS Kernel

This section sketches a very simple operational semantics for a cut down version of the kernel presented in this chapter. This is only intended to hint at how one might go about proving properties of collections of KAOS processes — a great deal of extra machinery would be required to turn this into a usable theory.

Consider the Miranda type `process`

```
abstype process
with    spawn :: newProcess * -> process -> process
        start ::  newProcess * -> [sys_message]
        writeTty ::  [char] -> process -> process
        pre_send :: process_id -> send *
        pre_receive :: receive *

newProcess * == send * -> receive * -> process
send * == * -> process -> process
receive * == (* -> process) -> process
```

We shall represent states of the operating system by finite lists of expressions of type `process`. We shall ignore the possibility that an expression might evaluate to $\perp$.

When reduced, expressions will reach one of the forms

$$\texttt{spawn}\ (\lambda x, y.a)\ b \tag{1}$$

$$\texttt{pre_send}\ i\ z\ a \tag{2}$$

$$\texttt{pre_receive}\ (\lambda x.a) \tag{3}$$

$$\texttt{writeTty}\ s\ a \tag{4}$$

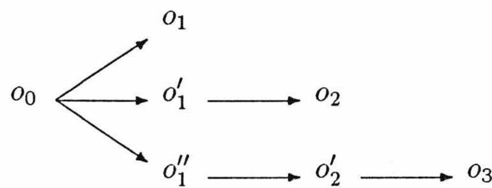
where $a, b \in$ `process`, $i \in$ `num`, $s \in$ `[char]` and z is any Miranda value.

We give a semantics to our operating system by defining a number of *rewriting rules* which show how an operating system state can be transformed once all of its elements are in one of the forms above. Note that there will usually be more than one possible rewrite. The choice as to which rewrite is used represents the non-determinism in our system.

$$\frac{\begin{array}{c}[p_0, \ldots, p_k, \ldots, p_n] \\ p_k = \texttt{spawn}\ a\ b\end{array}}{[p_0, \ldots, b, \ldots, p_n, a\ (\texttt{pre_send}\ (n+1))\ \texttt{pre_receive}]} \tag{5}$$

$$\frac{\begin{array}{c}[p_0, \ldots, p_s, \ldots, p_r, \ldots, p_n] \\ p_s = \texttt{pre_send}\ r\ x\ a \\ p_r = \texttt{pre_receive}\ b\end{array}}{[p_0, \ldots, a, \ldots, b\ x, \ldots, p_n]} \tag{6}$$

From a starting state o we can construct a *derivation tree* of states reachable from that state. For example



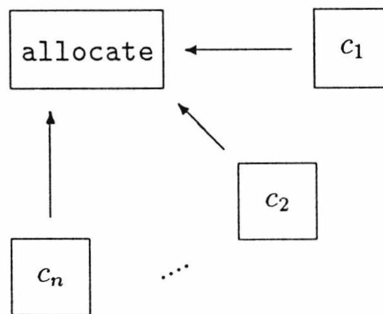
Derivation trees will usually be infinite.

We prove properties of an operating system by making a hypothesis about the structure of the derivation tree of some operating system state. We prove these hypotheses by induction over the depth of the derivation tree of that state.

2.11 A Sample Proof

This section sketches a proof of a small collection of processes using the semantics outlined in the previous section.

The example we shall use is resource allocation. The process `allocate` will organise sharing of a resource among a number of client processes represented by `client` below. Clients wishing to use the resource send `allocate` a REQuest message. When the resource becomes free, `allocate` sends back an acknowledgement message. When the client has finished, it sends a RELease message back to `allocate`, which is then free to grant the resource to some other client.



`allocReq` is the input type for our allocator.

```
allocReq
    ::= REQ (send ()) |     || Request with ack attached
        REL
```

`allocate` can be in two states: either the resource is free, in which case the allocator is simply waiting for a REQ message to arrive, or it is allocated to some process, in which case the allocator is waiting for a REL message to arrive.

There is a slight complication: a REQ message might arrive while the allocator is waiting for a REL message. If this happens, then the allocator has to put the request on to the end of a queue of pending requests. When the REL message does arrive and

the allocator goes back to wait for another REQ message, we have to make sure it first checks the queue to see if any requests are pending. The function representing the allocator state in which the resource is free therefore has to take a queue of pending requests as an argument.

For simplicity we are ignoring the problem of distributing the allocator's send function to the various clients.

```
allocate :: newProcess allocReq
allocate tx rx = free rx []
```

The resource is free. If there are any pending requests for the resource, then we send a grant to the process at the head of the queue. If there are no pending requests for the resource, then we have to wait for a REQ to arrive.

```
free :: receive allocReq -> [send ()] -> process
free rx (ack:rest)
      = ack () (alloced rx rest)
free rx []
      = rx unpack
        where
        unpack (REQ ack)
              = ack () (alloced rx [])
```

The resource is granted to some process. We wait for the next message. If it is another REQ, then we put that request on the end of the queue of pending messages. If the next message is a REL, then we can return to the state in which the resource is free.

```
alloced :: receive allocReq -> [send ()] -> process
alloced rx q
      = rx unpack
        where
        unpack REL = free rx q
```

```
unpack (REQ ack) = alloced rx (postfix ack q)
```

The process we shall use for the clients. This simply sits in a tiny loop, requesting the resource, waiting for it to be granted and then freeing it again.

```
client :: send allocReq -> newProcess ()
client alloc tx rx
      = req
        where
        req = alloc (REQ tx) wait
        wait = rx free
        free () = alloc REL req
```

There are two properties of the allocator that we might want to prove: safety and liveness. Safety here means that the allocator is well behaved, that is, that it never issues spurious grants. Liveness here would be that we never ignore a client forever, that is, that all requests are eventually followed by a grant. Unfortunately, our semantics is not strong enough for us to be able to prove liveness properties. We would need some fairness conditions attached to the rewriting rules. This is discussed further in section 2.12.

Instead we shall prove that our allocator is safe. We express this property as a predicate on operating system states. We shall take as our starting point the state

$$o_0 = [\texttt{free}\ r\ \texttt{[]}, \texttt{req}_1, \ldots, \texttt{req}_n]$$

Now consider a state o_k some way into the running of the program. It could have the form

$$o_k = [\texttt{free}\ r\ q, c_1, \ldots, c_n] \qquad (7)$$

$$\texttt{[pre_send}\ d_0, \ldots, \texttt{pre_send}\ d_k\texttt{]} = q \qquad (8)$$

where

$$\forall_{0 \leq i \leq k} \quad . \quad c_{d_i} = \texttt{wait} \tag{9}$$

$$\forall_{1 \leq i \leq n, \texttt{pre_send}\ i \notin q} \quad . \quad c_i = \texttt{req} \tag{10}$$

Equation 9 says that any process on the queue of pending requests will be `waiting` (see the definition of `client`) and equation 10 says that all other processes are waiting to make a request.

o_k could also have the form

$$o_k = [\texttt{alloced}\ \texttt{rx}\ q, c_1, \ldots, c_n] \tag{11}$$

$$[\texttt{pre_send}\ d_0, \ldots, \texttt{pre_send}\ d_k] = q \tag{12}$$

where

$$\forall_{0 \leq i \leq k} \quad . \quad o_k!d_i = \texttt{wait} \tag{13}$$

$$\exists_{1 \leq r \leq n} . c_r = \texttt{free ()} \quad \wedge \quad \forall_{1 \leq i \leq n, \texttt{pre_send}\ i \notin q, i \neq r} . c_i = \texttt{req} \tag{14}$$

Equation 13 again means that all processes on the queue of pending requests are equal to `wait` and equation 14 means that there is at least one process about to release the resource, and all other processes are waiting to make a request.

We shall say that an operating system state conforming to one of the sets of conditions above is *stable*.

Now we have only to show that from any stable starting state, we must always reach another stable state after a finite number of rewrites. The start state o_0 is clearly stable, as it matches equation 7 and satisfies the conditions associated with it.

Now consider an arbitrary stable state o_k. We must show that o_k will inevitably rewrite to another stable state within a finite number of transitions.

There are two cases: our stable state o_k could match either equation 7 or equation 11. For reasons of space we shall only explore the first case. We can immediately produce two more subcases

Case 1 $q =$ `[]`. In this case, `free` will wait for the next message to arrive. This in turn generates n subcases, one for each of the clients which could get to send the next request message. If client c_l gets to send to the allocator, then by application of rule 6, o_{k+1} will have the form

$$[\texttt{unpack (REQ (pre_send } l\texttt{))}, c_1, \ldots, c_{l-1}, \texttt{wait}, \ldots, c_n]$$

which is equal to

$$[\texttt{pre_send } l \texttt{ () (alloced rx [])}, c_1, \ldots, c_{l-1}, \texttt{wait}, \ldots, c_n]$$

Now there is only a single o_{k+2}, which by rule 6 will be

$$[\texttt{alloced rx []}, c_1, \ldots, c_{l-1}, \texttt{free ()}, \ldots, c_n]$$

This matches equation 11 and satisfies the associated conditions and so is therefore stable.

Case 2 $q =$ `(pre_send` d_0`:rest)`. In this case, o_k is equal to

$$[\texttt{pre_send } d_0 \texttt{ () (alloced rx rest)}, c_1, \ldots, c_{d_0-1}, \texttt{wait}, \ldots, c_n]$$

which by application of rule 6 is uniquely related to

$$[\texttt{alloced rx rest}, c_1, \ldots, c_{d_0-1}, \texttt{free ()}, \ldots, c_n]$$

which matches equation 11, satisfying the associated conditions and is therefore also stable.

2.12 Discussion

There are a number of points which should be made about the semantics presented above.

The system suggested allows one at most to prove *partial correctness* of an operating system, that is, that we cannot reason about operating systems in which some processes may evaluate to $\perp$. This is necessary if the same semantics is to be used for both the Miranda kernel in appendix A and for the putative bottom avoiding kernel discussed in chapter 4. We should perhaps not dodge this issue.

We also make no mention of *fairness*. As we saw in the example, we are unable to demonstrate many properties which it would be nice to be able to prove. We would have to add extra conditions to the definition of the rewriting rules to disallow derivations which involved indefinite delaying of a process which was ready to perform some action. Fairness is notoriously difficult to formalise, see for example [20].

"Can't make the frug contest Helen; stomach's upset. I'll fix you Ubik! Ubik drops you back in the thick of things fast. Taken as directed, Ubik speeds relief to head and stomach. Remember: Ubik is only seconds away. Avoid prolonged use."

Chapter 3

The Structure of KAOS

There has not been much written about the higher levels of functional operating systems. Most papers, such as [16, 21, 19], describe in detail how they tackle the problems of representing non-determinism and controlling devices, but say little about how they build an operating system on top of these facilities. As we have seen, considerations concerning the representation of non-determinism have already given us a gross structure for our operating system: a collection of processes exchanging messages. We can now borrow many structuring ideas from more conventional message passing operating systems.

This chapter first describes the differences between the actual KAOS kernel and the skeleton kernel outlined in the previous chapter, then describes the general form of messages exchanged by KAOS system processes and finally covers in detail the sorts of messages to which each type of system process responds.

3.1 The Real KAOS Kernel

The real KAOS kernel is very similar in structure to the tiny kernel outlined in the previous chapter, but with additions for process removal and device control. Appendix A contains a complete listing of the kernel source — a total of around 400 lines of Miranda.

3.1.1 The KAOS Kernel Interface

As the KAOS kernel allows processes to be removed, send functions can now become out of date: we need to allow for the possibility that a process might use a send function after the process to which that function connects has been removed. The continuations supplied to KAOS send functions therefore expect a success or failure message back from the kernel.

```
send * == * -> (lift () -> process) -> process
```

where `lift` adds an error element to a type

```
lift *
    ::= P * | I sysErr            || Proper and Improper
sysErr
    ::= ERR_EOF [char] |
        ERR_INAPPROPRIATE [char] |
        ERR_NOPROCESS [char] |
        ...  etc.
```

(The `[char]` attached to each constructor in `sysErr` is intended for helpful diagnostics, rather than for system use.)

Receive functions are augmented in the same way. A receive function can fail, for example, if it is used by some process other than the one for which it was made.

```
receive * == (lift * -> process) -> process
```

Kill functions can also fail if the process to which they refer is removed.

```
killfun == (lift () -> process) -> process
```

And `spawn` can also fail in KAOS.

```
newProcess * == send * -> receive * -> killfun -> process
spawn ::  [char] -> newProcess * -> (lift () -> process) -> process
```

(The `[char]` is the name the kernel will use for the process; it is used only for debugging. When the kernel detects deadlock, it tries to show the state of all the processes in the system. The `[char]` is used to label each process in this list.)

There is a function for producing Miranda's `sys_messages`. This is used by parts of the operating system to help produce debugging output, but is not available to user processes.

```
sys ::  sys_message -> process -> process
```

All devices in KAOS are identified by unique `deviceIds`. For example, the current version of KAOS usually has a disc as device `0` and terminals as devices `1` and `2`. Kernel functions are provided to read and write single disc sectors

```
discBlock == [char] || Always the same length
blockNum == num
readDisc
     :: deviceId -> blockNum -> (lift discBlock -> process) ->
        process
writeDisc
     :: deviceId -> blockNum -> discBlock ->
```

```
        (lift () -> process) -> process
```

Functions are provided to read from and write to terminals.

```
readTty
     :: deviceId -> (lift [char] -> process) -> process
writeTty
     :: deviceId -> [char] -> (lift () -> process) ->
        process
```

Many processes can read from or write to each terminal. In particular, a terminal can have several pending reads. Terminals can be in either `LINE` or `CBREAK` mode. In `LINE` mode, `readTty` requests return an entire line of input; in `CBREAK` mode they return every time the user hits a key. `CBREAK` mode is used only by the KAOS screen editor. A process is allowed to `setTtyState` while other processes are reading.

```
lineState ::= LINE | CBREAK
setTtyState ::  deviceId -> lineState -> process -> process
```

The `start` function from the previous chapter is now called `kaos` and takes two extra parameters: a list of the UNIX path names it is to read and write the discs from, and a list of the UNIX path names it is to read and write the terminals from.

```
kaos ::  [[char]] -> [[char]] -> newProcess * -> [sys_message]
```

So for example, KAOS is usually entered with

```
kaos ["os/sys/ker/disc.image"] ["/dev/tty", "/dev/tty01"]
       (root ":disc/sys/startup")
```

which sets up one disc and two terminal devices, then runs `root`, the KAOS boot process, passing in the name of the boot file on the KAOS file system.

3.1.2 Extensions to Miranda

Various features had to be added to Miranda to make the KAOS kernel described above possible. These are

`changetype` This is used in a few places in the kernel. It is not available to ordinary Miranda programs! The previous chapter describes in detail where it is used and why.

`nb_read` This is a version of the ordinary Miranda `read` function that does not block when input is unavailable, instead returning special *hiaton* characters. It is used inside the kernel to implement `readTty`.

An addition to Miranda would also be necessary to implement versions of `readDisc` and `writeDisc` which actually side-effected a disc (or a large UNIX file). Instead, the current version of the KAOS kernel represents the disc as a large list which is read from a UNIX file at start up and written back again when the kernel exits.

`load` This would be used to add new code to the operating system at run time. The idea is that you pass `load` a binary file you have just read from a disc, and the name of the object you wish to extract from the binary. `load` scans the binary, extracts the named object and returns a graph for it.

```
load ::  [char] -> [char] -> lift *
```

As with the Miranda `show` system, `load` is not a polymorphic function, it is a reserved identifier standing for a family of monomorphic functions. One of these functions is chosen at compile time, according to the type context that `load` appears in.

load will fail if it cannot find the name, or if the object it finds is not an instance of the type of load in this context. So for example, you might write a function to load a device driver as

```
loadDriver ::  [char] -> [char] -> lift (deviceDriver *)
loadDriver = load
```

(Note that, unlike a conventional operating system, we need no inverse of load for removing processes — the garbage collector will do this for us when the process finishes.)

load has not actually been added to the language yet, so the current version of KAOS is simply one huge function.

3.2 KAOS Message Structure

All KAOS system processes have a large algebraic type as their input, with one constructor for each of the sorts of requests you can make. All the messages you send to system processes have to have a reply function attached which the system process can use to tell you whether your request was successful or not. In general then, the input type of a system process might look something like

```
marsupReq
    ::= MARSUP_POSSUM possumType (send (lift ())) |
        MARSUP_WOMBAT wombatType (send (lift wombatId)) |
        ...  etc.
```

(Here a `MARSUP_POSSUM` expects a `possumType` as an argument, and sends back `lift ()` and `MARSUP_WOMBAT` expects a `wombatType` as an argument and returns `lift wombatId`.)

If you are a process wishing to talk to marsup, you should have a constructor of your input type dedicated to receiving replies from it.

```
myInput
    ::= FROM_MARSUP marsupReply |
        ...  etc.
```

where

```
marsupReply
    ::= MARSUP_CONFIRM (lift ()) |
        MARSUP_WOMBATID (lift wombatId) |
        ...  etc.
```

(Most KAOS system processes will have an equivalent to marsupReply defined for you.)

So for example, here's a process which registers a new wombat and then exits if it was successful. It expects to be passed a send function for marsup.

```
askForWombat ::  send marsupReq -> newProcess myInput
askForWombat marstx mytx myrx mykill
      = marstx (MARSUP_WOMBAT anotherWombat
                (mytx.FROM_MARSUP.MARSUP_WOMBATID)) test
        where
        test (P ()) = ask' myrx mykill

ask' ::  receive myInput -> killfun -> process
ask' myrx mykill
      = myrx unpack
        where
        unpack (P (FROM_MARSUP (MARSUP_WOMBATID x)))
              = mykill undef, isproper x
              = error "bad wombat!", otherwise
```

(No one would actually write this. There are a number of libraries available which hide all the details of sending and receiving messages.)

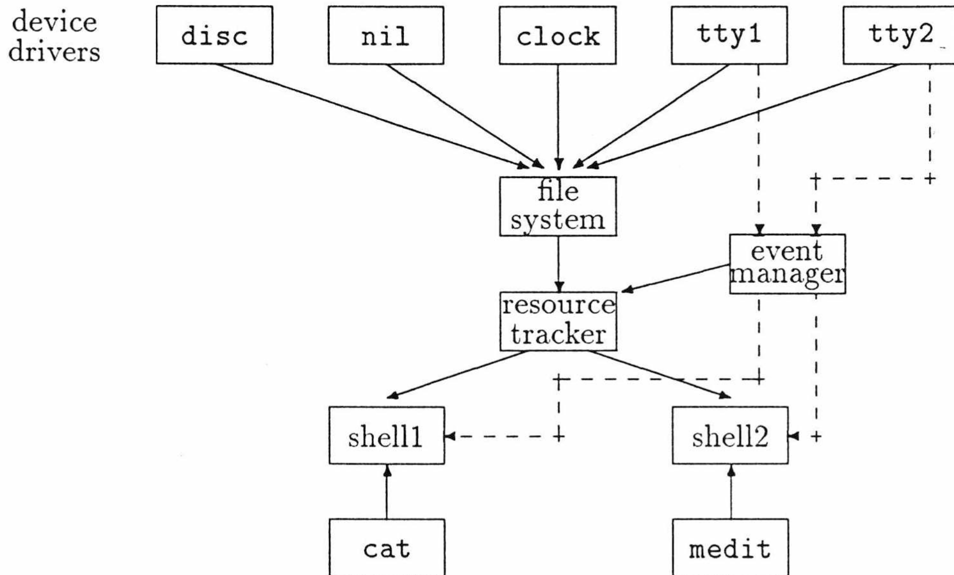


Figure 7: Sketch of the Major KAOS System Processes

The next few sections describe the input types of the major forms of system processes. Each section has roughly the same structure: a few paragraphs explaining why the process is necessary, the type itself, a brief explanation of each request and finally another few paragraphs of discussion. Figure 7 roughly shows the relationships between the processes discussed below for a two user version of KAOS.

3.3 The Event Manager

In a message passing operating system, interrupts are conveniently represented by small messages which arrive on a process's input when the interrupt occurs. For example, when the process controlling access to a terminal sees an arriving `^C` character, it should send a series of messages off to any processes which are under the control

of that terminal and which are interested in interrupts. To help route and coordinate interrupt messages, KAOS provides a system process called the *event manager.*

An *event* is some kind of external event, such as the user hitting `^C`. Processes which want to hear about the event declare an *interest* in it. When a process decides that the event should occur (perhaps the terminal driver has just seen the `^C` character arrive) it *signals* it and the event manager sends all interested processes an *interrupt* message.

```
eventReq
    ::= EVENT_NEW [char] (send (lift ())) |
        EVENT_INTEREST [char] (send ()) (send (lift eventId)) |
        EVENT_UNINTEREST eventId (send (lift ())) |
        EVENT_SIGNAL [char] (send (lift ())) |
        EVENT_REMOVE [char] (send (lift ()))

eventReply
    ::= EVENT_EVENTID (lift eventId) |
        EVENT_CONF (lift ())

eventId == num
```

`EVENT_NEW name reply` Create a new event with the given name. Events are identified by `[char]`. For example, the `^C` event associated with `tty1` is called `"CTRLC-TTY1"`. The event manager replies with a success or failure message. Creating an event can fail if the name supplied is already in use by an existing event.

`EVENT_INTEREST name signal_fn reply_fn` Declare an interest in an existing event. When the event is signalled, the event manager will use the `signal_fn` to send you `()`. The event manager replies with an `eventId` which you can use at some later point as an argument to `EVENT_UNINTEREST`.

`EVENT_UNINTEREST eventId reply` Renounce interest in an event.

`EVENT_SIGNAL name reply` Send () to all the processes which have expressed an interest in the named event. The event manager replies when all the interested processes have received their interrupt messages.

`EVENT_REMOVE name reply` Remove the named event. This will fail if any processes are still interested in the event.

Interrupt messages are the only ones in KAOS which have no reply field. This was rather a late decision, made to help prevent deadlocks which were occurring frequently when one typed ahead and then hit `^C`. See section 3.7 for more comments on this issue.

3.4 Device Drivers

The other main class of activity in KAOS is reading and writing `[char]` to and from terminals and discs. A device driver sits on top of the kernel device functions like `readDisc` and `writeDisc`, trying to provide a clean and consistent interface to the rest of the world.

At the most basic level, a driver accepts `driverReq` messages

```
driverReq
    ::= DRIVER_FILEREQ fileReq |
        DRIVER_STREAMREQ streamReq |
        DRIVER_SYSDRIVERREQ sysDriverReq
```

`sysDriverReq` is used to control the behaviour of the driver. It looks like

```
sysDriverReq
    ::= DRIVER_FORMAT pathName (send (lift ())) |
        DRIVER_MOUNT pathName (send (lift ())) |
        DRIVER_UNMOUNT (send (lift ()))
```

```
pathName == [fileName]
fileName == [char]
fileNameLength = 15
```

A `fileName` is a list of characters padded on the right with spaces to at least `fileNameLength`. Having a limit on the maximum length of file names can be rather annoying, but it does greatly simplify the device drivers, and as KAOS has no users apart from the author, the complaints are easy to ignore. `pathName`s are intended to behave rather like UNIX path names; KAOS allows you to open files called `"mydir/fred"`, for example.

`DRIVER_FORMAT path reply` The driver should in some way reset the physical device it controls. A disc driver would reformat the disc, a terminal driver might perhaps set the baud rate. The `pathName` can be interpreted in any way the driver sees fit. For example, the current KAOS disc driver expects a path like `"kaos root/20"`, where the first component is the title you wish to give the disc volume and the second is the number of sectors the driver should try to format for.

`DRIVER_MOUNT path reply` All drivers are sent this message when they are linked to the file system. The path here contains start up information of a more specific kind. For example, the current KAOS terminal driver expects a path like `"CTRLC-TTY1/1"`, where the first component is the name of the event the driver should signal when it sees a `^C` character and the second is the `deviceId` the driver should use for its `readTty` and `writeTty` calls.

`DRIVER_UNMOUNT reply` Drivers are sent this message when they are about to be unmounted. They should tidy up, and if all is ok, they should send `P ()` back

before killing themselves. If the driver has problems tidying up, perhaps because there are still some open files or because there is a hard device error when it tries to flush its caches, then it should send an error back and *not* kill itself.

All drivers are expected to support a subset of `streamReq`. These messages have to do with opening, reading, writing and closing files.

```
streamReq
    ::= STREAM_OPEN pathName (send (lift fileId)) |
        STREAM_CLOSE fileId (send (lift ())) |
        STREAM_READ fileId (send (lift [char])) |
        STREAM_WRITE fileId [char] (send (lift ())) |
        STREAM_SEEK fileId filePointer (send (lift ())) |
        STREAM_TRUNCATE fileId (send (lift ())) |
        STREAM_IOCTL fileId [char] (send (lift ()))

streamReply
    ::= STREAM_FILEID (lift fileId) |
        STREAM_CONF (lift ()) |
        STREAM_CHARS (lift [char])

fileId == num
filePointer == num
```

`STREAM_OPEN path reply` The file referenced by the path is opened, and a `fileId` that can be used in subsequent requests is returned. If the file does not exist, you get a `ERR_NOTFOUND` error back. You can have more than one open on a file; multiple readers and writers have the 'expected' behaviour. You can open directories as files and read from them (but not write to them). Files are opened with the file pointer set at zero.

There are no file modes — facilities such as open for append and open for read only are supposed to be provided by a higher level in the operating system.

`STREAM_CLOSE fileId reply` Closes the specified file.

`STREAM_READ fileId reply` Reads the next 'lump' from the file. Quite how much is read is left to the driver to decide. For example, the terminal driver will return single characters when in `CBREAK` mode and whole lines when in `LINE` mode. If you try to read past the end of the file you get an `ERR_EOF` back.

`STREAM_WRITE fileId chars reply` Writes the characters to the file. The file will magically 'grow' if you try to write past the end of it.

`STREAM_SEEK fileId filePointer reply` Moves to the given position. If you try to move the pointer past the end of the file you get an error and the pointer is not moved. This request is only supported by 'directory' style devices — it clearly makes no sense on a terminal.

`STREAM_TRUNCATE fileId reply` The length of the file is reduced, so that it is equal to the current value of the file pointer. This can cause confusion if other processes have opened the file, so it is only allowed if this `fileId` is the only one open on this file. Again, this makes no sense on a terminal.

`STREAM_IOCTL fileId command reply` Performs some kind of device specific control function. The only `IOCTL` supported at the moment is to the terminal driver. `"LINE"` puts the terminal into `LINE` mode, `"CBREAK"` puts it into `CBREAK` mode.

Only directory style devices are expected to support `fileReq`. These are the requests dealing with general file management.

```
fileReq
    ::= FILE_DELETE pathName (send (lift ())) |
        FILE_MKDIR pathName (send (lift ())) |
        FILE_CREATE pathName (send (lift ())) |
        FILE_RENAME pathName pathName (send (lift ()))
```

```
fileReply
    ::= FILE_CONF (lift ())
```

`FILE_DELETE path reply` Deletes the named file. You are not allowed to delete non-empty directories or files which are open.

`FILE_MKDIR path reply` Creates a new directory. If there is already a file with that name, it is deleted subject to the conditions above.

`FILE_CREATE path reply` Creates a new file. Again, any existing file with this name is deleted.

`FILE_RENAME frompath topath reply` The file at `frompath` is moved to `topath`. If `topath` already exists, it is deleted subject to the usual conditions. You can move open files and full directories.

3.5 The File System

The file system gathers together all the device drivers under one roof. It interprets the first component of a `pathName` as the name of the device driver to which the rest of the request should be sent. Library functions are available to turn strings like `":disc/myfile/fred"` into the appropriate `pathName`.

```
fsReq
    ::= FS_FILEREQ fileReq |
        FS_STREAMREQ streamReq |
        FS_SYSFSREQ sysFsReq

sysFsReq
    ::= FS_MOUNT (send driverReq) pathName (send (lift ())) |
        FS_UNMOUNT fileName (send (lift ())) |
        FS_FORMAT pathName (send (lift ()))
```

`FS_MOUNT driver_send path reply` Adds a new device driver to the file system. The first component of the path is stripped off, and the rest passed down to the driver in a `DRIVER_MOUNT` message. If the driver reports a successful mount, the file system adds the driver to its list of mounted devices and returns a success message. From then on, any `fileReq` or `streamReq` to a path whose first component matches the first component of the mount path will be redirected to the new driver.

`FS_UNMOUNT name reply` Sends a `DRIVER_UNMOUNT` message to the driver. If this returns successfully, the driver is removed from the file system's list of devices and a success message sent back to the unmounter.

`FS_FORMAT path reply` Sends a `DRIVER_FORMAT` message down to the device. This really has nothing to do with the file system process, but is grouped in the same type for security. These are all messages which ordinary processes should be unable to send.

Another function performed by the file system is to ensure that the `fileIds` used by `streamReq` are unique across all the mounted device drivers.

3.6 The Resource Tracker

When a process is killed on an operating system like UNIX, all the resources allocated to it are automatically released. To help provide similar facilities, KAOS has a process called the *resource tracker* which coordinates all resource allocation. To kill a process safely, you send a message to the resource tracker; this calls the kernel `kill` function for the process and then releases all the resources the process had taken.

KAOS also tracks process creation, so that when you kill a process you also automatically kill all of its children. This requires quite a complex protocol and is described in its own subsection below.

```
trackMgrReq
    ::= TRACK_TRACKREQ processId trackReq
```

When `track` sees an incoming request, it needs to know which process sent it. A `send trackReq` function is in fact a `send trackMgrReq` that has had a `processId` buried inside it. Every process running under the supervision of the resource tracker thus has its own personal `send trackReq`.

```
trackReq
    ::= TRACK_OPEN pathName (send (lift fileId)) |
        TRACK_CLOSE fileId (send (lift ())) |
        TRACK_NEWEVENT [char] (send (lift ())) |
        TRACK_REMOVEEVENT [char] (send (lift ())) |
        TRACK_INTEREST [char] (send ()) (send (lift eventId)) |
        TRACK_UNINTEREST eventId (send (lift ())) |
        TRACK_FORK (send (send trackReq)) |
        TRACK_FORKROOT (send (send trackReq)) |
        TRACK_ENDFORK killfun (send (lift ())) |
        TRACK_KILL (send (lift ())

trackReply
    ::= TRACK_CONF (lift ()) |
        TRACK_TRACKWRAP (send trackReq) |
        TRACK_FILEID (lift fileId) |
        TRACK_EVENTID (lift eventId)

processId == num
```

Note that not all system requests go through `track`; only the ones which cause some kind of allocation to take place.

`TRACK_OPEN path reply` Passes a `STREAM_OPEN` message on to the file system, makes a note of the `fileId` that was allocated and returns it to the caller. All the

rest of these messages do much the same kind of thing, except for the messages concerned with process creation (see below) and

`TRACK_KILL reply` Kills off the process and all its children, then releases any resources they have allocated.

3.6.1 Tracking Process Creation

Creating a new process which the resource tracker can follow is rather an intricate operation.

There are in fact two different kinds of process creation. `TRACK_FORK` is used to tell the resource tracker that you wish to fork a child process, `TRACK_FORKROOT` is used to indicate that the process you wish to fork should be at the root of a new process tree. This second kind of forking is only supposed to be used by system processes. The description below only talks about `TRACK_FORK`, but it should be clear how `TRACK_FORKROOT` would work.

- The parent-to-be posts a `TRACK_FORK` message to the resource tracker using its personal track function. It includes a send function that the tracker then uses to send back the child-to-be's personal track function.

- The parent waits for the child's track function to arrive, then calls the kernel `spawn` function, passing the child's personalised track function. The parent then waits for a confirmation message to arrive from the child.

- The first thing the child does is to send a `TRACK_ENDFORK` message to the tracker, including in the message its kill function (which the tracker will later use to remove it.)

- The tracker makes a note of the child's kill function, then posts a confirmation message back to the child.
- The child sees the confirmation arriving from the tracker, then posts a confirmation to the parent and starts executing.
- The parent sees the child's confirmation message and resumes execution. To kill the child, it sends a `TRACK_KILL` message using the child's personal track function.

(User processes see none of this detail — they simply call a library function which spawns the child and returns a tracked kill function.)

3.7 Current Difficulties

In our view, the major difficulty with the current design has to do with an aspect of the system processes which this chapter has been very careful to avoid up to now.

As an example, here is a deadlock problem that appeared while developing KAOS. At this early stage, interrupt messages still had reply fields and the event manager would only reply to an `EVENT_SIGNAL` message when all of the interested processes had replied to their interrupt messages.

- The user types ahead a short way, and then hits `^C`.
- The terminal driver spots the `^C` character, and sends a signal to the event manager. It then waits for the event manager to reply, indicating that all the interested processes had seen the signal.
- The event manager sends an interrupt message to the shell, and waits for the shell to reply.

- The shell meanwhile has finished processing its input, and tries to print a new prompt (before looking at its input again). It sends "`kaos> `" off to the terminal driver and waits for a reply.
- ...but naturally, the terminal driver never sees the message — it's still waiting for a reply from the event manager. We have deadlocked!

There are various ways around this problem. The simplest (and the solution the current version of KAOS uses) is to not have reply fields on interrupt messages. The event manager will then be able to signal all of the interested processes, even if they are not listening for interrupt messages at the time. This seems rather unsatisfactory however, as the root of the problem seems to lie in the way processes behave, and in particular in the way that the current method for handling multiple input streams works. The shell should have been able to respond to an interrupt message even while it was busy talking to the device driver, the device driver should have been able to print some output even while waiting for the event manager to reply. In operating system terminology, KAOS processes are *single-threaded* rather than being *multi-threaded.*

Another problem has to do with the level at which the filing system works. It would be nice if it were possible to write a device driver that mimicked a UNIX pipe, allowing something like

```
kaos> cat :disc/fred >:pipe/newpipe &
kaos> cat <:pipe/newpipe >:tty1
```

to copy "`:disc/fred`" through "`:pipe/newpipe`" and onto the screen. Unfortunately by the time the `STREAM_OPEN` message reaches the driver, the driver can have no way of knowing whether it should be an open for reading or an open for writing,

that is, which end of the pipe this STREAM_OPEN should be attached to. This is simple to fix: STREAM_OPEN should take a set of flags as an argument.

Large message passing operating systems often have amazingly complicated message passing semantics, allowing processes to attach priorities to messages, to time stamp them, and so on. Whether the KAOS message model is suitable for larger systems remains to be seen.

There must also be questions about the adequacy of the semantics presented for the event manager. While the existing scheme seems to work reasonably well for ^C events, it's easy to imagine situations in a more grown up operating system where it would be woefully inadequate.

3.8 Implementation

KAOS is quite a large program by Miranda standards: almost 14,000 lines of source. This divides up as (roughly) 700 lines for the kernel, 4,000 lines for the disc device driver, 1,000 lines for the terminal device driver, about 1,500 lines for the three system processes, 1,500 lines for the shell and its commands, 4,000 lines for the screen editor and the rest in various libraries. Execution speed is more or less what one might expect. On an Orion-1/05 (a 4 VAX MIPS machine) with 700,000 cells of heap space it takes about a second for the kaos> prompt to come back after you hit return.

"Perk up pouting household surfaces with new miracle Ubik, the easy-to-apply, extra-shiny, nonstick plastic coating. Saves endless scrubbing, glides you right out of the kitchen! Entirely harmless if used as directed."

Chapter 4

A Better Kernel for KAOS

As we noted in section 2.4, the Miranda kernel is really useful only as a prototype. If we want an operating system written in a functional language to be used in the same situations as conventional systems such as UNIX, then there are many more problems we have to address. This chapter tries to list the areas in which the Miranda kernel is deficient and then briefly describes a kernel that could address these problems.

4.1 What is Missing?

The Miranda kernel is missing several important features of conventional operating system kernels. These are discussed individually in the sections below.

4.1.1 Non-determinism

The Miranda kernel is missing the important property of *bottom avoidance*: that is, if one process running contains a bug and goes to $\perp$, it is very important that the whole machine does not stop; we have to reintroduce the non-determinism we were

so careful to take out in section 2.4.

It is interesting to note that a primitive like `ready` (see section 2.1) makes it very easy to add bottom avoidance to the Miranda kernel; we only need to rewrite the function `schedule` (see section 2.8). `ready` applied to an object of type `process` will evaluate to `True` when the process tries to examine its missing argument, the `machine_state`. As `process` is an `abstype`, this can only be done by one of the functions in the `abstype`. Hence `ready` will return `True` when the process tries to call a kernel function.

So the modification we have to make is that before committing itself to a particular process as the next to perform a kernel call, the scheduler should check that the process is ready. If the current process is not ready, then the scheduler should try the next on the run queue. This requires the addition of a single line

```
schedule ::  process
schedule (process_list, (a:rest))
      = cont (process_list, run')
                , ready cont
      = schedule (process_list, run')
                , otherwise
        where
        run' = postfix a rest
        (RUN cont pend) = process_list!(hd run')
schedule ([], process_list)
      = [Stdout "run queue empty"]
```

A kernel to be used as the basis for a real operating system would also need some system for giving different processes different amounts of processor time based on their relative importance.

4.1.2 Memory Management

The Miranda kernel has no control over how much memory each processes uses. A process containing a bug might use up the heap for the whole machine, stopping every process. In a paper describing an earlier version of KAOS[27], Turner suggested a scheme which put each process in a separate address space. This nicely solves the memory problems, but forces us to think very carefully about message passing. Turner required that message passing be hyperstrict, that is, objects to be sent in messages should be completely reduced; message passing then simply consists of copying fully evaluated acyclic subgraphs between processes. While this successfully prevents processes sending strange objects such as infinite lists in messages, it also prevents the passing of functions, and it seems absolutely necessary to be able to send send functions (message wrappers in the terminology used in Turner's paper) in order to be able to set up networks with the correct interconnections.

For example, suppose we have some persistent process implementing, say, a file system in our operating system. When a new process is created, we can easily arrange for it to be passed the send function for this file system process, enabling the new process to send messages to it. What we cannot do, unless we are allowed to pass send functions in messages, is arrange for the file system process to be given the send function for our new process. Our new process will be able to write files, but not able to read them!

The solution seems to be to require message passing to be hyperstrict, but to simply say that we allow functions through. This is exactly what the standard Miranda function `force` does. This will work well, provided that the only functions we try to send are send functions and compositions of send functions with constructors. If a programmer tries to send, say, a function which implements a screen editor, then it

will work but the operating system will run very slowly while all three megabytes of graph are copied and reconstructed.

The kernel now has to be able to copy cyclic graphs between address spaces. There are a number of ways of doing this, all of which have some overhead compared with the copying of acyclic graphs[4]. On the plus side, these schemes all preserve sharing, which straightforward copying of acyclic graphs would not. The degree to which this would slow down message passing is an open question.

While the previous missing feature could be easily added to a Miranda kernel given a new primitive, it is very hard to conceive of a primitive that might allow the addition of a separate address space system to KAOS.

4.1.3 Exception Handling

The final serious problem with the Miranda kernel lies with the difficulties raised by exception handling. Currently, if a process contains a bug and attempts to divide by zero, the whole operating system will crash. Again, the damage should ideally be limited to the abrupt removal of a single process from the system.

4.2 New Features for KAOS

A replacement kernel could be written in a conventional language such as C which would address most of the problems outlined above. This section briefly describes a design for such a kernel.

The C kernel should be implemented as a modification to the abstract machine supporting Miranda. It should timeslice reduction between the processes running, maintain separate address spaces for each process, copy cyclic graphs between the

processes, and so on. There are a number of additions that would need to be made to the kernel interface to control these new features. These are

- We would need some mechanism to allow priorities to be attached to processes. This could be done most easily by adding a priority function to `newProcess`

  ```
  priority
        == priorityLevel -> (lift () -> process) -> process
  ```

  Rather as with `killfun`, the new process would be free to pass this to other process which it felt ought to be able to set its priority. Again, as with `killfun`, we would have to build some mechanism on to process start up to pass priority functions for processes to the resource tracker.

- We need some way of allowing processes to interrogate the kernel, to find out how many processes are running, how much of the CPU each is using, and so on. Again, this could be done by the addition of another function to `newProcess`

  ```
  stats
        == (lift processStats -> process) -> process
  ```

  Which would return some data structure containing information about this process. As with `priority`, we would want some mechanism for passing stats functions back to the resource tracker.

- We need some mechanisms for controlling the separate address spaces. We need some way of finding out when a process runs out of heap, and we need some way of expanding or shrinking the amount of heap a process is allocated.

Another parameter to `spawn` could specify the amount of memory a process was given to start with, and another function in `newProcess` could change heap allocations

```
setsize
     == bytes -> (lift () -> process) -> process
```

Its harder to decide how we could most easily detect heap exhaustion. Perhaps the simplest way would be to add another parameter to `spawn`, giving a send function to be used by the kernel when the new process ran out of memory. See the final version of `spawn` below for a possible type.

- When a process is removed by the kernel following an exception such as divide by zero, we need some way for the kernel to inform other processes that this has happened, so that resources claimed by the dead process can be released by the operating system. As with the memory exhaustion case above, the simplest way is to pass a send function for the kernel to use to `spawn`.

In summary, the new kernel will need some changes to the type of `spawn`. The new `spawn` will have type

```
spawn
     :: [char] -> newProcess * -> bytes -> send exception ->
        (lift () -> process) -> process
newProcess *
     == send * -> receive * -> killfun -> priority -> stats ->
        setsize -> process
exception
    ::= EX_HEAPFAULT |
        EX_DIVIDEBYZERO |
        ...
```

4.3 Discussion

It is interesting to note how little the above sketch depends on the details of the machine that KAOS is running on. It could map as easily onto a network of machines, such as Transputers, as onto a shared memory multiprocessor machine, such as Grip, as onto a conventional machine like the Orion 1/05 that KAOS runs on at the moment.

Many of the problems with the current kernel would not occur in a functional programming language with a Martin-Löf style type system[9]. Programs written under this type system are guaranteed to not reduce to $\perp$, reducing the problems associated with non-preemptive scheduling and are also guaranteed not to attempt things like division by zero, reducing the problems caused by the lack of exception handling noted above. Type systems of this sort can also describe the type of `load` (see section 3.1.2) and `[process_list item]` (see section 2.8), removing the need for the games we have been playing with the type system. Such type systems do not make predictions about the space behaviour of programs, however; some form of run time memory management would be needed even for an operating system running under a Martin-Löf type system.

It is also worth noting that the additions described above are either cosmetic, such as `stats`, or only affect the operation of the machine in exceptional circumstances, such as when a process goes wild and consumes all of its heap or perhaps hits $\perp$. Provided that you are happy with partial correctness proofs, the semantics presented in section 2.10 will still hold for the new kernel. This issue is discussed at greater length in section 2.10.

"Has perspiration odour taken you out of the swim? Ten-day Ubik deodorant spray or Ubik roll-on ends worry of offending, brings you back to where the happening is. Safe when used as directed in a conscientious program of body hygiene."

Chapter 5

Programming Operating System Processes

Chapter 3 discussed the functions that the system processes have to perform, without saying quite how one is to write such a process in a functional language. This chapter describes the techniques used in KAOS and presents fragments from the inside of the disc device driver.

Writing operating system code presents special difficulties for the programmer. As an example, consider a device driver for a disc. It has to turn messages like 'write the string `"cat"` into the file referenced by handle `131072` at the current file pointer position' into a series of `readDisc` and `writeDisc` kernel calls. This is quite difficult, for a number of reasons.

The driver needs to *side effect* the disc drive. As the driver processes a request, it makes a series of changes to the disc; keeping track of the state of the disc during the request and being able to take the appropriate recovery action if disaster should

strike is tricky. For example, what should happen if one of the sectors fails during a write operation? That sector should clearly be taken off the disc's free sector map and another allocated instead. What happens if a sector fails and there is no replacement available because the disc is full? The half completed write operation will have to be undone. Anticipating all the possible combinations of errors and programming in appropriate recovery strategies is very tough.

The disc file system is one of the most crucial parts of the operating system from the point of view of overall operating system speed. There are a great many optimisations possible to minimise the number of `readDisc` and `writeDisc` calls needed to service a request. These range from simple block caches, to sequencing disc operations in order to minimise head movement. Programming in these optimisations requires a good deal of thought.

To help tackle these problems, KAOS uses a library of functions based on the type `interact`. `interacts` come with structuring primitives tailored to the problems of exceptions, side effects and parallelism.

5.1 Higher Order Functions for Sequencing

In [25], Thompson introduced a scheme for handling interactions with the user in a functional language which hid the details of stream input and output. In the same spirit, KAOS uses a type `interact` to hide the details of messages and continuations. Appendix C contains the complete source to the `interact` library. It may be helpful to read this source in conjunction with the explanation below.

An `interact` is a 'function' that can side effect input and output. It is a function in the sense that it has a value it returns: `interacts` may be composed, mapped along lists, used to fold up lists, etc.

`interact`s are made with `return`

```
return ::  * -> interact *
```

So for example, an `interact` which took two numbers and returned their product might be written

```
silly :: num -> num -> interact num
silly a b = return (a*b)
```

`interact`s can be composed with `comp`

```
comp ::  (* -> interact **) -> interact * -> interact **
```

So, for example, you might write

```
silly 4 $comp silly 2 3
```

which would (eventually) return 24. More usefully, you can define equivalents of the standard `fold` and `map` functions. For example, one library provides `ifoldr`

```
ifoldr ::  (* -> ** -> interact **) -> ** -> [*] -> interact **
ifoldr int start = foldr (comp .  int) (return start)
```

`interact` provides a simple kind of multiple input port. `get` gets the next message of the right 'flavour'

```
get ::  (inputType -> bool) -> interact inputType
```

where `inputType` is the input type for this process and occurs <u>`%free`</u> in the script defining `interact`. The first argument to `get` is a predicate that will be applied to incoming messages. The first to satisfy the predicate will be the message that `get` returns. Messages which fail the predicate are saved in a buffer that will be searched the next time `get` is called.

You often simply want the next message to arrive.

```
getNext :: interact inputType
getNext = get (const True)
```

`interact` provides a simple exception handling system. `catch` runs an `interact`, trapping to an error handler if anything goes wrong.

```
catch :: (sysErr -> interact *) -> interact * -> interact *
```

and `raise` signals an error, returning control to the error handler associated with the most recently encountered `catch`

```
raise :: sysErr -> interact *
```

So for example, many of the kernel functions return `lifted` types; it's very handy to have something to turn these explicit error values into `raise`s.

```
hideLift :: interact (lift *) -> interact *
hideLift int
      = test $comp int
        where
        test (I err) = raise err
        test (P x) = return x
```

And vice versa

```
revealLift :: interact * -> interact (lift *)
revealLift int
      = catch (return . I) ((return . P) $comp int)
```

Things become more difficult if the `interact`s inside a `catch` send or receive any messages. Execution can pass to the error handling `interact` without the error handler having any idea exactly how far through a sequence of messages the faulty

`interact` got. To let `raise` pass information back, `interact` carries about a `%free` object of type `userState`. Three functions are available to manipulate the state

```
getState :: interact userState
```

returns the current state.

```
putState :: userState -> interact ()
```

changes the current state, and

```
applyState :: (userState -> userState) -> interact ()
```

applies a function to the current state.

5.2 The stdio Library

KAOS provides a library of `interacts` called `stdio` to hide the details of communication with the KAOS system processes. The complete library contains around thirty functions; a few are described here to help give a flavour of the sorts of abstractions that are possible.

`stdio` maintains some state using the `getState`, `putState` and `applyState` functions described in the previous section. The state includes the various send functions `stdio` needs to communicate with the system processes, and also the buffers needed by the `readLine interact` described below.

```
openFile :: pathName -> interact fileId
```

Sends a `TRACK_OPEN` message to the resource tracker (see chapter 3) and returns the

reply. An error is `raised` if there are any problems.

```
readChars :: fileId -> interact [char]
```

Reads the next block from the file, `raiseing` `ERR_EOF` if there is no more input available.

```
readLine :: fileId -> interact [char]
```

Reads the next `'\n'` terminated line of text from the file, again `raiseing` `ERR_EOF` if there is no more text available. This function is useful, as `readChars` returns an amount of text determined by the driver for the particular device you are reading from. This is only a line of text in the case of the tty driver, the disc driver returns text in lumps approximately equal to the size of the sectors of the disc it is controlling. Text returned from the driver but not needed by this call to `readLine` is saved in a buffer ready for the next call.

```
forkProcess
      :: [char] -> application * -> send (lift ()) ->
         interact trackedKill
```

Starts up a new process. `application *` is `stdio`s equivalent of `newProcess *`. It includes all of the extra parameters that processes that are to run under `stdio` need. The `send (lift ())` is the send function that the instance of `stdio` attached to the child will use to post back the child's exit status. If the child exited because of a `raise`, then you will see an error value arriving. Normal termination returns `P ()`. The `trackedKill` is a send function that the parent can use to signal to the resource tracker that it wants the child to be killed.

```
mountDriver :: deviceDriver * -> pathName -> interact ()
```

Forks and mounts a new device driver.

```
interestEvent :: [char] -> send () -> interact eventId
```

Express interest in an event. When the event occurs, the `send ()` passed will be used

to notify you.

```
stdioInterest ::  [char] -> interact eventId
```

Make stdio express interest in an event. When stdio sees the message indicating that the event has occurred it executes a special *signal* interaction for you. You can set this signal with

```
setSignal ::  interact * -> interact ()
```

This signal interact could set a flag in the state to indicate to the rest of your program that the signal had occurred, or could perhaps raise an exception.

5.3 Programming System Processes

This section presents some code fragments from the inside of the disc device driver, illustrating how the driver addresses the problems outlined at the start of this chapter.

As these pieces of code come from a complex program, it is not really possible within a reasonable amount of space to give enough background information for a complete understanding of all the ins and outs. The presentation below tries to give enough information to make the general outline of the code comprehensible without overwhelming the reader with detail.

The first example comes from the script which contains the functions used to maintain directories. KAOS directories are ordinary files containing a list of objects of type dirEntry (encoded as a [char], of course). examineNext is a function which when given a fileId for an open directory either returns the next dirEntry, or if it reaches the end of the file raises an ERR_NOTFOUND, since the directory must have been completely searched without a match being found.

```
examineNext :: fileId -> interact dirEntry
```

```
examineNext fid
      = handle ERR_EOF handler (
                (return . hd . ctod) $comp
                readStream dirEntryLength fid
        )
        where
        handler err = raise (ERR_NOTFOUND "not in directory")
```

where handle is a library function built on catch — it lets you name a particular error you wish to trap. ctod turns a [char] into a [dirEntry].

Now it's easy to search a whole file for a particular dirEntry. There is a slight complication: when a file is deleted, all the driver does is set the fileType field in its dirEntry to UNUSED. searchStream has to be careful to ignore any dirEntrys that are marked as UNUSED.

```
searchStream :: fileName -> fileId -> interact dirEntry
searchStream name fid
      = test $comp examineNext fid
        where
        test entry
              = return entry
                        , fname = name & ftype ~= UNUSED
              = searchStream name fid
                        , otherwise
                where
                (ftype, fname, ibn) = entry
```

State changing operations can often be split into three stages: an *open* phase when some kind of resource is claimed, a *processing* stage during which the resource is manipulated and finally a *close* stage which releases the resource. If the processing stage can fail in some way with an exception, it is often necessary to insert a catch to ensure that the close operation is always performed, otherwise there is perhaps some danger of leaving the resource in an inconsistent state.

careful performs a processing interact, being careful to tidy up even if the process fails.

```
careful
      :: (* -> interact ()) -> (* -> interact **) -> * ->
         interact **
careful tidy process x
       = dotidy $comp catch (((.).comp) raise dotidy) (process x)
         where
         dotidy y = tidy x $then return y
```

Note that the error handler here itself contains a raise. then performs left-to-right sequencing of interacts which return no result. It is defined as

```
then :: interact () -> interact * -> interact *
then int1 int2 = (const int2) $comp int1
```

This next interaction makes any stream reading operation 'safe'. It opens a stream, performs some arbitrary interaction on it and then closes the stream. It remembers to perform the close even if the interaction it called raised an error.

```
handoverFileId
      :: fileType -> blockNum -> (fileId -> interact *) ->
         interact *
handoverFileId ftype ibn inter
       = careful closeStream inter $comp openStream ftype ibn
```

And now we can write a safe directory searching function, which will work whatever happens.

```
searchDir :: fileName -> blockNum -> interact dirEntry
searchDir name ibn
       = handoverFileId DIRECTORY ibn (searchStream name)
```

As a more complex example, here is the code used to extend a file. In the extend operation, more blocks have to be allocated from the disc's free sector map and linked

on to the existing blocks of the file. This example illustrates all the problems discussed at the beginning of this chapter: the extend function has to be able to cope gracefully with sectors failing at arbitrary points during the extend operation.

Before we can examine the details of the `extendStream` function, we first have to say something about the representation of files on the KAOS disc and the data structures the device driver has to maintain in order to be able to keep track of them.

Disc sectors come in three parts

```
driverBlock
     == (blockNum,                        || Next block
         blockNum,                        || First block
         [char])                          || Data
```

The first field is the `blockNum` of the next block in the file, the second field is the `blockNum` of the first block in the file, and the final field is the data itself. The `[char]` is always `blockLength` characters long.

Files come in two levels. Simple files are blocks chained together by their next block fields. Files at this level have to be a multiple of the block size in length. The last block in a file has its next block field set to zero.

Inodes are simple files holding information about another simple file. Each stores

- the length of the file in characters,
- the size of the map,
- a map of the file. This is a list of the `blockNums` of the blocks making up the body of the other file.

Since we only know the length of the inode to the nearest `blockLength`, we have to hold the length of the map explicitly.

When a file is opened, the whole of the inode is read into the inode structure maintained by the driver. Changes to the inode are made in memory. When the file is closed again, if the inode has changed, it is rewritten on the disc. This is far simpler than having to change the inode as well as the main body of the file every time the length of the file changes. It has the disadvantage that until the file is closed, the disc is in an inconsistent state. The inconsistency is not serious, however. If disaster were to strike before the file was closed, then the only problem would be that files whose inodes were not updated will be truncated somewhat. A disc recovery program could easily find these truncated files and rebuild the inodes correctly. Having said that, this is a deficiency in this version of the driver that should perhaps be fixed in the next version.

The types used below are

```
openFiles
     == ([streamBlock],                  || One per fileId
         [inodeInfo],                    || One per file
         [(blockNum, inodeId)])          || Map iids to inodeIds
```

The first component of `openFiles` has a `streamBlock` for every stream currently open (the `fileId` passed to `extendStream` was an index into this list); the second has an `inodeInfo` for every open file. Note that open files and open streams are not the same thing! We could for example have the same file opened twice, with both `streamBlocks` sharing a single `inodeInfo`.

The real device driver uses `keyedLists` (see appendix B) rather than ordinary lists for these structures; ordinary lists are used here to help simplify presentation.

`streamBlocks` and `inodeInfos` have the following structure

```
streamBlock
     == (blockId,                        || Current in cache
         blockNum,                       || Block's address on disc
```

```
            filePointer,                || Current file pointer
            num)                        || Pointer into inode info

inodeInfo
      == (blockNum,                     || blockNum of inode on disc
            fileType,                   || Need to track this
            blockMap,                   || Map for file
            num,                        || Current file length
            bool,                       || Dirty bit
            num,                        || Number of opens
            blockMap)                   || Map for inode on disc
```

It's not necessary to explain the use of all these fields, except to note that the current length of the file is held inside the `inodeInfo` for that file.

There are two cases

- See if we can satisfy the request just by changing the length field in the inode, that is, without allocating a new block. If so, it's very easy: we just change the length field in the copy of the inode we are holding in memory.
- If not, then we have to allocate a new block, link it on (by rewriting the current last block), and recurse. We have to be very careful about error handling. More comments below on this.

All `extendStream` does is to extract the stream data structures from the state, checking that the stream we are interested in does indeed exist. The stream data structures are then passed on to `extend'`.

```
extendStream :: num -> fileId -> interact ()
extendStream amount fid
      = extend' amount fid $comp extractStream fid
```

And here are the two cases we mentioned above. If the amount of new space needed is less than the space that is unused within the last sector of the file, then all

we need do is change `len`. Otherwise, we read the final block of the file into memory, preparatory to allocating a new block and linking it on.

```
extend' :: num -> fileId -> openFiles -> interact ()
extend' amount fid ofiles
      = replaceStream ofiles'
                , amount < available
      = tweak $comp getBlock ftype lastbn
                , otherwise
        where
```

Take the state apart.

```
        (strbs, inodes, index) = ofiles
        (sb, bn, ptr, iid) = strbs!fid
        (ibn, ftype, map, len, idirty, nopens, imap) = inodes!iid
```

Useful things we calculate. `lastbn` is the `blockNum` of the current last-block-in-file, `offset` is the position of `len` within the last block, and `available` is the amount we can increase `len` by without allocating more space.

```
        lastbn = last map
        offset = len mod blockLength
        available = blockLength - offset
```

Rebuilding the state for the easy case. Here we can just bump up `len`.

```
        inodei' = (ibn, ftype, map, len + amount, True, nopens, imap)
        inodes' = update iid inodei' inodes
        ofiles' = (strbs, inodes', index)
```

And the hard case. We have just grabbed the current last block of the file. We allocate a new block, rewrite the old last block (to link it on to the new block), write out the new last block, change `openFiles` to reflect the new length and recurse.

We consider error recovery separately. The general philosophy behind error recovery in the driver is to either complete successfully, or to fail and leave things as

you found them. In the recursive case, we only make two changes: we allocate the new sector and rewrite the last block of the file. Hence in the case of an error after the allocation we have to remember to to deallocate the sector again, and in the case of an error after we have rewritten the last block we need to write back its original contents.

This is not quite the situation we had in the previous example, where we had to remember to close the stream again, even if the stream operation had trouble. Here we only want to deallocate only if the write has trouble. The error function we need is called `errundo` (perhaps meaning 'undo in case of error') and is defined as

```
errundo
     :: (* -> interact ()) -> (* -> interact **) -> * ->
        interact **
errundo undo int x
       = catch hand (int x)
        where
        hand err = undo x $then raise err
```

And finally, here is the recursive case.

```
tweak (0, first, data)
      = errundo deallocate tweak' $comp allocate
        where
        tweak' newbn
              = errundo (const restore) (const link) $comp
                putBlock ftype lastbn (newbn, first, data)
                where
```

The two operations be can perform: `link`ing on the new block and recursing, and undoing the link should we fail further down the line.

```
restore
      = putBlock ftype lastbn (0, first, data)
link
      = putBlock ftype newbn
```

```
                    (0, first, testBody) $then
            extend' (amount - available)
                    fid ofiles''
```

Rebuilding `openFiles` for the recursive case. `len` should point into the first character of the new block.

```
inodei'' = (ibn, ftype, postfix newbn map,
           len + available, True, nopens, imap)
inodes'' = update iid inodei'' inodes
ofiles'' = (strbs, inodes'', index)
```

As these examples have hopefully shown, systems programming is much more difficult than you might expect. Practical experience has shown that abstractions in the style of `interact` help make these problems manageable.

It might be objected that `interact` is not really in the spirit of functional programming; that it just implements a tiny procedural language inside Miranda. We would say that all large functional programs define their own set of higher order functions for structuring, tailored to their own problems. It is not surprising that the primitives chosen for an operating system are concerned with the problems of exceptions, side effects and parallelism.

"Taken as directed, Ubik provides uninterrupted sleep without morning-after grogginess. You awaken fresh, ready to tackle all those little annoying problems facing you. Do not exceed recommended dosage."

Chapter 6

KAOS Applications

This chapter briefly describes the facilities provided by the two main KAOS application programs, `sh` the KAOS shell and `medit8`, a full screen editor that runs under KAOS. It ends with a discussion of the special difficulties that arise in writing programs that are to run under systems like KAOS.

6.1 The KAOS shell

KAOS has a simple shell which provides similar facilities to the standard UNIX shells. It is a user level process just like any other, relying heavily on the `stdio` library described in the previous chapter.

6.1.1 Shell Command Syntax

A shell command consists of a number of blank separated words, the first of which is the name of the program to be run and the rest are either arguments which are to be

passed to the program, or modifiers which tell the shell about the environment the program is to be run in.

There are four modifiers available: output, input and error redirection and backgrounding. If a word starts with a `'>'` character, the rest of the word is interpreted by the shell as a path name to which the program's output should be redirected. If a word starts with `'<'`, the rest of the word is interpreted as a path name for input redirection and if a word starts with `'#'`, as error redirection. If the last word in a shell command is a `"!"`, then the program is run in the background. The shell will print a message on the terminal when a backgrounded program terminates.

At the moment there is no notion of a current directory; path names have to all be given in full. The syntax of a path name in KAOS is

pathname : ':' < devicename >< tailpath >

tailpath : ϵ |
'/' < filename >< tailpath >

(Where file names and device names are both strings of up to `fileNameLength` characters. See section 3.4 for another view of this.)

So for example:

```
ls :disc/sys
```

lists the `sys` directory on the KAOS disc, and

```
cat :disc/sys/newshell >:tty2 !
```

copies the file `newshell` off the KAOS disc onto device `tty2` in the background.

There are a variety of devices available: `:disc` is the main filesystem, `:nil` is a 'black hole' which simply swallows anything sent to it, `:tty1` is the terminal and `:clock` is a time of day clock. There may be other `tty` devices, depending upon the configuration of the system.

All commands return success or failure codes. If a command finishes successfully, the shell simply prompts for more input. If a command fails, then the shell will attempt to give some diagnostics. Background commands always produce a message when they finish.

6.1.2 Programs Available

There are a number of programs that can be run from the shell. These vary from tiny utility programs like `cat` and `echo`, to large applications like the screen editor `ed`. All can have their input, output and error streams redirected, all can be backgrounded and all can be interrupted if they are run in the foreground.

`cat` Interprets its arguments as a list of filenames which are each read in turn and copied to `cat`'s standard output. If there are no arguments, `cat` just copies standard input to standard output.

`rm` Removes the list of filenames given as arguments.

`ls` If the named files are directories, `ls` writes a pretty directory listing to its standard output. If the files are not directories, then KAOS crashes.

`mv` Renames a file on a device. This cannot be used across devices, for obvious reasons. For example,

```
mv :disc/sys/newshell :disc/test
```

moves `newshell` into the root directory, renaming it as `test`. Only two arguments are allowed, both of which have to be complete path names.

`echo` Writes its arguments to `stdout`. For example,

```
echo hello sailor!  >:tty2
```

writes `hello sailor!` on `:tty2`.

`if` Tests its arguments for textual equality, returning an error if they are not all the same. This can be used in conjunction with the sequencing operators described below.

`fmt` Reformats the named devices. This is rather a dangerous command and should be used with care!

`mkdir` Makes a series of new directories.

`sh` Runs a sub shell. If there are no arguments, `sh` reads commands from its standard input. Otherwise it runs itself on the script named in the first argument, passing in the remaining arguments to the script.

`ed` Runs a version of `medit`, the Miranda screen editor, on the first file named. If there are no arguments, `ed` starts up on an empty file.

`mount` Mounts a device. The first argument is the name of the driver to be mounted (currently only `tty`, `clock`, `disc` and `nil` are allowed), the second argument is a path to mount at. For example,

```
mount tty :tty2/CTRLC-TTY2/2
```

mounts another tty driver, which will signal interrupts on `CTRLC-TTY2` and which will access kernel device number 2.

`unmount` Unmounts a series of devices. For example,

```
unmount :tty1 :tty2
```

`help` Prints a manual page for `sh`. The help system is rather rudimentary.

6.1.3 Shell Commands

There are a number of commands which are built directly into the shell. These behave rather differently to the programs described above: although they can have their input, output and error streams redirected, they cannot be run in the background, and they cannot be interrupted, with the exception of `wait`.

`jobs` Lists all the background jobs currently executing.

`wait` Wait until all backgrounded jobs finish.

`create` Creates a new event owned by the shell. Only one argument is allowed.

`setenv` Sets an environment variable. For example,

```
setenv ENV_SIGNAME CTRLC-TTY2
```

sets variable `ENV_SIGNAME` to `CTRLC-TTY2`. The environment is a list of mappings from `[char]` to `[char]`, passed down from parent to child processes. It includes things like `ENV_STDOUT`, etc.

`showenv` Display the current environment.

`unsetenv` Unset a variable.

`exit` Causes the shell to exit prematurely. If there are any arguments, then the shell exits with a `USERERROR` with the arguments attached to it. If there are no arguments, then the shell exits normally. For example,

```
exit wrong number of args!
```

`^D` (CONTROL-D) Generates an end-of-file from the keyboard. The sample session below shows how it can be used.

`^K` (CONTROL-K) The shell will attempt to kill the foreground job. The shell does not let you kill background jobs.

6.1.4 Variable Substitution

Before command lines are parsed, KAOS performs variable expansion. Occurences of `$(fred)` are replaced by the contents of the environment variable `fred`. Expansions can be nested: `$(fred$(bill))` expands `bill` first, then looks for a variable whose name starts with `fred`, followed by the contents of `bill`. As an abbreviation, the brackets may be omitted if the name of the variable is a single character.

A number of environment variables are defined automatically. These are:

`ENV_STDIN`, `ENV_STDOUT`, `ENV_STDERR` These hold the file handles for the streams the shell is currently working on.

`ENV_SIGNAME` This holds the name of the event the shell is listening for interrupt messages on. If it is undefined when the shell starts up, then the shell will not look out for interrupts.

`0` This is the name of the script the shell is currently processing. It is set to `stdin` if the shell is just reading from its standard input.

1, 2, ... These are the arguments the shell was passed. They may or may not be all defined!

n This is the number of arguments the shell was passed, not including argument 0.

* Is arguments 1 to n joined together with spaces.

% Is arguments 2 to n joined together with spaces. This is useful for recursing down argument lists, in the absence of proper string operators.

6.1.5 Command Sequencing

Several commands may be put on a line, joined together with "&", "|" or ";" and grouped with "(" and ")". These operators let you do simple command sequencing.

command1 & command2 Means do command2 unless command1 fails. This is supposed to look like conjunction: the whole expression only succeeds if both of its components succeed.

command1 | command2 Means do command2 unless command1 succeeds. This is like disjunction: the whole expression succeeds if one of its components succeeds.

command1 ; command2 Means do command2 whether command1 succeeds or not.

These operators are left associative and all have equal binding power. You can use "(" and ")" to force grouping when necessary.

So for example, here is a shell script which will act a little like the UNIX cp command: it will copy all its initial arguments to its final argument with cat. Note how we do tests and recursion.

```
( if $n 0 | if $n 1 ) | ( cat $1 >$($n) & sh $0 $% )
```

6.1.6 Shell Syntax

The syntax accepted by the shell can be summarised as

line	:	< element > | < element >< op >< line >
element	:	< command > | '(' < line >< ')' >
op	:	'&' | '|' | ';'
command	:	< name >< args >< bged >
args	:	ϵ | < arg >< args >
bged	:	ϵ | '!'

6.1.7 A Sample Session

Below is a sample KAOS session, intended to illustrate the use of the commands above. As a general rule of thumb, the system tends to work like UNIX: if you do what comes naturally, most things work.

```
kaos> mount disc :disc/3/0
kaos> ls :disc
:disc
kaos> cat >:disc/jane
Now is the winter of our discontent
Made glorious summer by this OS of Kent.
^D
kaos> ls :disc
:disc
1) jane :  FILE
kaos> cat :disc/jane
Now is the winter of our discontent
Made glorious summ^K
kaos> cat :disc/jane >:disc/fred &
kaos> jobs
1) cat :disc/jane >:disc/fred
kaos> rm :disc/jane
```

```
shell: 'rm :disc/jane' failed: file is open
kaos>
shell: 'cat :disc/jane >:disc/fred': terminated
kaos> cat >:disc/jane
cat :disc/jane
^D
kaos> sh :disc/jane >:disc/fred
kaos> cat :disc/fred
cat :disc/jane
kaos> unmount :disc
kaos> ^D
```

6.1.8 Inside the Shell

Internally, the shell is structured as an event loop. At any time, it can receive an exit message from a background process, a CONTROL-C interrupt message or a new line of input from the user. This presents structuring problems, as `interacts` such as `readLine` (see section 5.2) cannot be used asynchronously.

The solution adopted was to split the shell into two processes. A front end calls `readLine` repeatedly, posting lines of text to the other half of the shell as they arrive. This other half simply calls `getNext` repeatedly, switching to different functions depending on the message.

6.2 `medit` — A Screen Editor Under KAOS

`medit` was originally written by Robert Duncan[3] to run inside the normal Miranda environment. Substantial changes had to be made to make it work comfortably under KAOS. It is interesting to note that the changes described below took only a fews days hacking, despite the complexity of `medit` and the relative lack of program

documentation. This ease of maintenance seems to be due to a combination of a program which was written with updates to Miranda in mind; referential transparency, which makes it possible to understand small parts of large systems in isolation; and a powerful type system, which catches most errors at compile time.

6.2.1 Changing the Input Style

The original version of `medit` had two main functions, `line_mode` and `screen_mode`. `screen_mode` was for moving around the document: it allowed you to search for strings, move to specific lines, scroll forwards and backwards by a page and so on. If you tried an editor command that `screen_mode` did not understand, such as inserting a character, it would extract the current line of text from the document and pass it to `line_mode`. `line_mode` simply edited a single line of text from the document. It allowed you to cursor left and right along the line, insert and delete text from the current line, and so on. If you tried an editor command that `line_mode` did not understand, it would return the edited line together with the editor command that failed. `screen_mode` would put the new line back in the document and then try to interpret the command itself.

This was clearly an unsuitable structure for a KAOS application, as one of the things being passed between `line_mode` and `screen_mode` was the stream of commands being typed at the keyboard, and KAOS does not have this form of lazy input. What was needed was a single function taking an editor command and an editor state to a new state and some characters to be used to update the screen.

The editors state was expanded to hold a flag indicating whether the editor was in line mode, and if so to then also hold the appropriate extra pieces of state information. Every time the edit function is called, it looks at the editor command and decides

if it should be handled by `line_mode` or `screen_mode`. It then checks the state the previous editor command was processed in. If this command is to be processed in the same state, then it can simply extract the right part of the state and pass it to either `line_mode` or `screen_mode`. If not, then before it can call either function, it has to switch states by calling a function to extract the current line from the document or a function to replace the edited current line in the document.

6.2.2 Isolating File Input and Output

`medit` originally had its file input and output handling scattered throughout the code. Under KAOS, input and output has to be done with `interacts` — to prevent `interact` spreading throughout the program it was necessary to move all the file handling to the top level. Again, this involved quite substantial changes to the structure of the program.

6.2.3 Cosmetic Changes

A number of other changes were made to `medit`. The system for binding editor commands to key sequences was originally rather more complex and slow than was strictly necessary. It was replaced by a system which associated a single keystroke with a single command. `medit` originally checked every character it sent to the screen to see if it was printable ASCII, displaying it as a `'?'` if not. Removing this feature speeded up screen output dramatically. The terminal database used by `medit` was improved, and the functions for extracting the description of the current terminal are now called just once, during program start up. Previously, `medit` had performed this extraction operation every time it needed to output some screen control codes. Support was also added for the INSERTCHAR and DELETECHAR operations provided by VT100

terminals, speeding up character insertion.

The version of `medit` running at the moment under KAOS comes in two parts: a single 400 line file written using `interacts` which handles all the communication with the operating system, and another 3500 lines written in an ordinary functional style which defines a single function `edscreen`, of type

```
edscreen
     :: window -> edit_command -> screenstate ->
        (screenstate, [char])
```

which processes a keystroke, returning a new editor state and some characters to be sent to the screen.

"Pop tasty Ubik into your toaster, made from fresh fruit and healthful all-vegetable shortening. Ubik makes breakfast a feast, puts zing into your thing! Safe when handled as directed."

Chapter 7

Conclusions and Future Developments

Chapter two discussed the problems of introducing non-determinism to a functional language, developing a model of parallelism which was flexible enough for implementation in either a functional language or an imperative one. It then presented an operating system kernel written entirely in Miranda and went on to sketch roughly how one might go about proving properties of parallel functional programs. Chapter three described the insides of KAOS, an operating system based on the ideas developed in chapter two and written entirely in Miranda. Chapter four showed what features of conventional operating systems are not covered by the kernel from chapter three, and suggested how a kernel written in an imperative language could solve these problems while maintaining compatibility with the Miranda kernel. Chapter five explained a systematic scheme for writing operating system code in a functional language, giving examples of how one handled error recovery and side effects. Finally, chapter six described two of the larger applications that run under KAOS: the shell,

and the screen editor `medit`.

There are a number of problems with the current design of KAOS. As section 3.7 explained, there are questions about the adequacy of the design of several of the system processes, and about the very simple message passing semantics adopted by KAOS. The semantics presented in section 2.10 is also very incomplete: a great deal of work would be necessary if something along those lines were to become a reasonable system for proving properties of collections of processes.

If KAOS were to be taken any further, perhaps the next step should be a proper kernel, as described in chapter 4, and the implementation of a Miranda compiler in Miranda, so that KAOS could become a complete development environment. Only by using the system on a daily basis could one get a feel for its shortcomings and some ideas about the future development of operating systems in functional languages.

Bibliography

[1] M. Broy. *Nondeterministic Dataflow Programs: How to Avoid the Merge Anomaly*. Science of Computer Programming 10 (1985) 65–85.

[2] P. Denning, R. Brown. *Operating Systems*. Scientific American, September 1984.

[3] R. Duncan. *Using the Miranda Screen Editor MEDIT*. University of Kent Computing Laboratory, 1986.

[4] Fenichel, Yockelson. *Garbage Collection in a Virtual Storage Situation*. CACM, vol. 12, no. 11, p. 611, 1969.

[5] J. Fairbairn, S. Wray. *TIM — A simple, lazy abstract machine to execute supercombinators*. Third conference on Functional Programming Languages and Computer Architecture, Springer Lecture notes in Computer Science 274.

[6] P. Henderson. *Purely Functional Operating Systems*. Functional Programming and its Applications, eds Darlington, Henderson and Turner. Cambridge University Press, 1982.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[8] K. Karlsson. *Nebula — A Functional Operating System*. Laboratory for Programming Methodology, Department of Computer Science, Chalmers University, 1981.

[9] P. Martin-Löf. *An Intuitionist Theory of Types: Predictive Part*. Logic Colloquium 1973, eds. Rose and Shepherdson, North Holland 1975.

[10] G. Kahn, D. McQueen. *Coroutines and networks of parallel processes*. IFIP 77, North Holland.

[11] R. D. Lins. *C.M.-C.M. A Categorical Multi-Combinators Machine*. UKC Computing Laboratory Report 57, December 1988.

[12] M. Longley. *MOOSE*. UKC Computing Laboratory Report No 55.

[13] M. Longley. *Continuations → Continuations = Interactions*. UKC Computing Laboratory Report No 63.

[14] S. B. Jones. *A Range of Operating Systems Written in a Purely Functional Style*. University of Stirling Department of Computer Science, Technical report TR16.

[15] S. B. Jones. *Abstract Machine Support for Purely Functional Operating Systems*. Oxford University Programming Research Group Technical Monograph 34, August 1983.

[16] S. B. Jones, A. F. Sinclair. *Functional Programming and Operating Systems*. The Computer Journal, Vol. 32, No. 2, 1989.

[17] G. Kahn, D. McQueen. *Coroutines and networks of parallel processes*. IFIP 77, North Holland.

[18] J. Misra. *Equational Reasoning About Nondeterministic Processes.* Published where??

[19] N. Perry. *Short Note: Towards a Functional Operating System.* Imperial College report.

[20] A. Pnueli. *Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends.* LNCS 224, March 86.

[21] W. Stoye. *The Implementation of Functional Languages using Custom Hardware.* PhD Thesis, Cambridge University Computer Laboratory, December 1985.

[22] W. Stoye. *A New Scheme For Writing Functional Operating Systems.* Cambridge University Computer Laboratory Technical Report 56, 1984.

[23] C. Strachey, C. P. Wadsworth. *Continuations — a Mathematical Semantics for Handling Full Jumps.* PRG-11, Programming Research Group, University of Oxford, 1974.

[24] Sun Microsystems, *The NeWS Window System: Technical Overview.* October 1986.

[25] S. J. Thompson. *Interactive functional programs: a method and a formal semantics.* UKC Computing Laboratory Report No 48, November 1987.

[26] D. A. Turner. *An Overview of Miranda.* SIGPLAN Notices, December 1986.

[27] D. A. Turner. *Functional Programming and Communicating Processes.* Conference proceedings for PARLE, June 1987.

[28] D. A. Turner. *SASL Language Manual.* St. Andrews University Department of Computational Science, December 1976.

Appendix A

KAOS Kernel Source

```
>|| kernel.m - KAOS kernel, V.5.0
```

Export the kernel calls and some useful type synonyms.

```
>%export
>       receive killfun newProcess send
>       discBlock discBlockLength blockNum lineState deviceId
>       process kaos spawn sys
>       writeTty readTty setTtyState writeDisc readDisc
```

Include various libraries. Their source appears in appendix B.

```
>%include "os/lib/misc"
>%include "os/lib/char"
>%include "os/lib/key"
>%include "os/lib/lift"
>%include <local/disgusting>
```

A.1 Introduction

The kernel manages the set of processes currently active in the system, switching execution between them, passing messages etc. It provides functions to control devices too.

This is a literate script; all lines starting with a > character are lines of program text. This appendix was generated automatically from the KAOS source.

A.2 Type definitions

We have functions for sending and receiving. Processes have a private receive function for accepting messages addressed to themselves, and a public send function they can pass to other processes who might wish to send to them.

To send a message, we pass the thing to be sent and a continuation expecting the result of the send. Send fails if the function is out of date; ie. is a send function for a process that has been removed from the system.

```
>send * == * -> (lift () -> process) -> process
```

To receive a message, we just pass a continuation expecting something of the appropriate type. Receives can fail if used by some process other than the one they were made for.

```
>receive * == (lift * -> process) -> process
```

A kill function removes a particular process. Each process starts up with its own kill function and is free to pass it to any other processes which it feels ought to be able to kill it. Kills can also fail, eg. if they become out of date. A process kills itself when it wishes to finish. The argument can be left `undef` in this case.

```
>killfun == (lift () -> process) -> process
```

A `newProcess` the thing we pass to `spawn`; it is a process expecting all the things processes are handed when they start up.

```
>newProcess * == send * -> receive * -> killfun -> process
```

A process is a huge abstype containing the entire state of the operating system. There are only three kernel calls available to the outside world

`kaos` Starts up the operating system. This is what one types at the Miranda prompt to boot, passing the root process and some other stuff.

`spawn` Makes a new process out of the `newProcess` passed, returns an error value to the spawner.

`sys` Sends a `sys_message` to the outside world. Used only for producing debugging messages on the console.

And five more functions for accessing devices. All get passed a `deviceId`, which should be for a device of the right type.

`readDisc/writeDisc` Read and write a sector on the disc. Some quick defs. for disc blocks

```
>discBlock == [char]
>discBlockLength = 60     || Very small for the moment
>blockNum == num
```

`readTty/writeTty/setTtyState` Read and write characters to the terminal. Select either raw or cooked input with `setTtyState`.

```
>lineState ::= LINE | CBREAK
```

See the implementation equations for more detailed comments on these functions.

```
>abstype process
>with   kaos :: [[char]] -> [[char]] -> newProcess * ->
>               [sys_message]
>       spawn
>            :: [char] -> newProcess * -> (lift () -> process) ->
>               process
>       sys  :: sys_message -> process -> process
```

Device functions.

```
>       readDisc
>            :: deviceId -> blockNum -> (lift discBlock -> process) ->
>               process
>       writeDisc
>            :: deviceId -> blockNum -> discBlock ->
>               (lift () -> process) -> process
>       readTty
>            :: deviceId -> (lift [char] -> process) -> process
>       writeTty
>            :: deviceId -> [char] -> (lift () -> process) ->
>               process
>       setTtyState
>            :: deviceId -> lineState -> process -> process
```

And some internal functions that also need to be in the abstype.

```
>        schedule :: process
>        doDeviceNow :: deviceId -> process
>        doDeviceLater :: process
>        doRunnable :: process
>        exitKaos :: process
>        killWrap :: processId -> killfun
>        receiveWrap :: receive *
>        sendWrap :: processId -> send *
```

A process is a function taking the current `machineState` to a list of `sys_message` suitable for piping to the Miranda environment.

```
>process == machineState -> [sys_message]
```

The machine state is a `keyedList` of all the `processStates` (`processIds` are keys into this list), a list of the `processIds` of all the currently runnable processes and a list of device states. `deviceIds` are indexes into this list.

```
>machineState
>    == (keyedList (processState item),
>         [processId],
>         [deviceState]
>       )
>processId == listKey
>deviceId == num
>item == num     || Could be anything
```

`item` is supposed to represent some untyped fragment of combinator graph. We use `changetype` to turn things into items and back again; see send/receive at the end of this file. The head of the runnable list is the pid of the currently active process.

A process can be in one of four states

`RUN` The process is runnable. We need to hold the continuation for that process and a list of the processes which are currently waiting to send to it, together with the thing they are trying to send.

`OUT` The process is blocked waiting for output; it tried to send to a process that was not waiting for input. We need to hold the continuation, the process we are trying to send to and the list of processes currently blocked on this process.

`IN` The process is blocked waiting for input; it tried to read when there were no messages waiting. We only need to hold the continuation.

`DEV` The process is blocked on a device. Something in `deviceState` keeps track of the process' `processId` and will restart it when the device is ready. Likewise, `DEV` has to track the `deviceId` the process is waiting on, so we can tidy up `deviceState` if the process is killed. Again, we need to keep track of the processes blocked on this process.

The $*$ for a `processState` is the input type for that process, the `[char]` at the front of each constructor is the name of the process (useful for debugging).

```
>processState *
>    ::= RUN [char] process [(processId, *)] |
>        OUT [char] (lift () -> process) processId [(processId, *)] |
>        IN [char] (lift * -> process) |
>        DEV [char] (devReply -> process) deviceId [(processId, *)]
```

And the device state. For each device attached, we note the queue of processes blocked on it and some more stuff particular to each device type.

```
>deviceState == ([processId], deviceParticulars)
```

For the disc we just hold the current list of blocks and the name of the file it came from. For each tty we need to hold

- The input stream from that tty; ie. the result of something like `nb_read "/dev/tty14"`,
- The current line state for the tty; either `LINE` or `CBREAK`,
- The device name for the tty. Again, something like `"/dev/tty14"`. This is the name we `Tofile` to to write to the terminal, and `stty` to to change the line mode.

```
>deviceParticulars
>    ::= PART_DISC [discBlock] [char] |
>        PART_TTY [char] lineState [char]
```

KENT LIBRARY UNIVERSITY

Continuations in DEV are waiting for a devReply. This has various shapes, depending on the device the process is blocked on. Currently, things only get blocked on tty reads.

```
>devReply
>    ::= TTY_REPLY (lift [char])
```

A.3 Useful functions

This function is used by schedule (see below) to dump the process list. Without this function, we just get <unprintable> errors as processState is not exported (and I don't think should be) from this file.

```
>showprocessState :: (* -> [char]) -> processState * -> [char]
>showprocessState showthing (RUN name cont pend)
>      = "RUN \"" ++ name ++ "\" <process> " ++ showpend showthing pend
>showprocessState showthing (OUT name cont key pend)
>      = "OUT \"" ++ name ++ "\" <lift () -> process> <item> " ++
>        show key ++ " " ++ showpend showthing pend
>showprocessState showthing (IN name cont)
>      = "IN \"" ++ name ++ "\" <lift item -> process>"
>showprocessState showthing (DEV name cont did pend)
>      = "DEV \"" ++ name ++ "\"  <devReply -> process> " ++ show did ++
>        " " ++ showpend showthing pend
```

Show a pending list.

```
>showpend :: (* -> [char]) -> [(processId, *)] -> [char]
>showpend showthing pend
>      = "[" ++ foldl showit "" pend ++ "]"
>        where
>        showit sofar (key, x)
>              = showed, sofar = ""
>              = ", " ++ showed, otherwise
>                where
>                showed = "(" ++ shownum key ++ "," ++ showthing x ++ ")"
```

Replace the current process' continuation. This is called by spawn, sys etc.

```
>replaceRunnable :: process -> machineState -> machineState
>replaceRunnable cont' (plist, run, devs)
>     = (plist', run, devs)
>       where
>       pkey = hd run
>       (RUN name cont pend) = indexKey pkey plist
>       pstate' = RUN name cont' pend
>       plist' = updateKey pkey pstate' plist
```

Find a process' name. Useful for debugging.

```
>findName :: processState item -> [char]
>findName (RUN name cont pend) = name
>findName (OUT name cont key pend) = name
>findName (IN name cont) = name
>findName (DEV name cont did pend) = name
```

A.4 Kernel functions visible to the outside world

kaos evaluates some root process to a [sys_message]. We have to set up the initial deviceState: this involves doing some nb_reads and grabbing the initial disc states from UNIX. We set the terminals to ignore ^D's. We take lists of tty names and file names (which hold the discs) as arguments.

```
>|| kaos :: [[char]] -> [[char]] -> newProcess * -> [sys_message]
>kaos discs ttys root
>     = schedule (addProcess "root" root (emptyKey, [], devs))
>       where
>       makedisc name = ([], PART_DISC
>                             (choplist (chop discBlockLength)
>                                       (read name)) name)
>       maketty name = ([], PART_TTY (ttyInput name) LINE name)
>       devs = map makedisc discs ++ map maketty ttys
```

Repeatedly read from a terminal. This lets us catch ^Ds. We turn ^Ds into CR, ^D, CR. We have to pass the name around to stop lazy evaluation doing funny things.

```
>ttyInput :: [char] -> [char]
>ttyInput name
```

```
>       = join name
>         where
>         join name
>                 = nb_read name ++ ['\n','\04','\n'] ++ join name
```

spawn turns a newProcess into new process, adding it to the state. Currently spawn always succeeds.

```
>|| spawn
>||   :: [char] -> newProcess * -> (lift () -> process) -> process
>spawn name newjob cont state
>       = schedule state''
>         where
>         state' = replaceRunnable (cont (P ())) state
>         state'' = addProcess name newjob state'
```

And finally, sys sends a sys_message to the Miranda environment and then runs some other process.

```
>|| sys :: sys_message -> process -> process
>sys mess cont state
>       = mess : schedule state'
>         where
>         state' = replaceRunnable cont state
```

A.5 Kernel device control functions

The disc functions are very simple. We do the read/write then exit through the scheduler.

```
>|| readDisc
>||   :: deviceId -> blockNum -> (lift discBlock -> process) ->
>||      process
>readDisc did n cont state
>       = schedule state'
>         where
>         (run, plist, devs) = state
>         ([], PART_DISC disc discname) = devs!did
>         state' = replaceRunnable (cont (P (disc!n))) state
```

```
>|| writeDisc
>||   :: deviceId -> blockNum -> discBlock -> (lift () -> process) ->
>||      process
>writeDisc did n blk cont state
>     = schedule state''
>       where
>       (run, plist, devs) = state
>       ([], PART_DISC disc discname) = devs!did
>       disc' = update n blk disc
>       devs' = update did ([], PART_DISC disc' discname) devs
>       state' = (run, plist, devs')
>       state'' = replaceRunnable (cont (P ())) state'
```

`writeTty` just does a `sys` for you. It has to look up the name of the file to write to in the device list.

```
>|| writeTty
>||   :: deviceId -> [char] -> (lift () -> process) -> process
>writeTty did str cont state
>     = sys (Tofile devname str) (cont (P ())) state
>       where
>       (run, plist, devs) = state
>       (q, PART_TTY inp lst devname) = devs!did
```

This process becomes blocked on tty input. We add it to the end of the queue inside `ttyState` and change it into a `DEV`. The scheduler will test the input stream and restart the process if there is any available.

```
>|| readTty :: deviceId -> (lift [char] -> process) -> process
>readTty did cont state
>     = schedule state'
>       where
```

Take the state apart.

```
>       (plist, (ckey:rest), devs) = state
>       (q, part) = devs!did
```

Make the new device state. We just append the `processId` to the queue on this device.

```
>        q' = postfix ckey q
>        devs' = update did (q', part) devs
```

Make the new process list. We have to change the RUN into a DEV.

```
>        (RUN name oldcont pend) = indexKey ckey plist
>        pstate' = DEV name cont' did pend
>        cont' (TTY_REPLY x) = cont x
>        plist' = updateKey ckey pstate' plist
>        state' = (plist', rest, devs')
```

Set the lineState for a particular terminal.

```
>|| setTtyState :: deviceId -> lineState -> process -> process
>setTtyState did lst cont state
>     = sys (System str) cont state'
>        where
>        (plist, rlist, devs) = state
>        (q, PART_TTY inp oldlst devname) = devs!did
>        devs' = update did (q, PART_TTY inp lst devname) devs
>        state' = (plist, rlist, devs')
>        toterm = " > " ++ devname
>        str    = enterCbreak ++ toterm, lst = CBREAK
>               = enterLine ++ toterm, otherwise
```

sys_messages for changing the terminal mode.

```
>enterCbreak = "stty raw -echo"
>enterLine = "stty -raw echo"
```

A.6 The scheduler

Chose and run the next process. This is very confusing! In summary, we

- Are there any processes blocked on devices? If so, are there any devices ready? If we find a ready device and a waiting process, we change the process from DEV to RUN and schedule it next (by calling doDeviceNow). Otherwise, fall through to the next test.

- Are there any processes in the run queue? If there are, we choose one of these and run it (by calling `doRunnable`). Otherwise, fall through to the next test.
- Test again to see if there are any processes blocked on devices. If there are, then wait for the first device to become ready and then schedule its associated process (by calling `doDeviceLater`). Otherwise fall through to the next test.
- There are no runnable processes to be found. KAOS should exit, dumping the system state (by calling `exitKaos`).

Note: we have to be very careful to continually throw away hiatons. Every time we find a hiaton, unless we junk it we will not force another poll next time around. So, after scanning the tty list to find any ready terminals, we have to scan it again throwing away hiatons from the fronts of all the lists.

```
>|| schedule :: process
>schedule state
>       = doDeviceNow (hd ready) state', ready ~= []
>       = doRunnable state', run ~= []
>       = doDeviceLater state', waiting
>       = exitKaos state', otherwise
>         where
```

Take the state apart.

```
>         (plist, run, devs) = state
```

Look through all the devices for devices with non-empty queues.

```
>         waiting = or [ q ~= [] | (q, part) <- devs ]
```

Find all the ttys with non-empty queues and some stuff waiting to be read on their input. Build a list of the `processIds` of all these devices.

```
>         ready = [ id | (id, (q, PART_TTY inp lst devname)) <-
>                        zip2 [0..] devs;
>                        q ~= [] & hd inp ~= hiaton ]
```

`ready` will force evaluation of the heads of all the tty input streams with waiting processes. All the input streams with no ready input will now have hiatons on their fronts. We have to go through all the input streams throwing away one hiaton from the front of each.

```
>       devs' = [ (q, tidy part) | (q, part) <- devs ]
>       tidy (PART_TTY (a:x) lst devname)
>             = PART_TTY x lst devname, a = hiaton
>             = PART_TTY (a:x) lst devname, otherwise
>       tidy other = other
>       state' = (plist, run, devs')
```

Do a read from a tty. By the time we get called, we know that there is enough input available on this terminal. Roughly, we do

- Read the appropriate amount from the input stream,
- pull the blocked process out of the process list,
- hand the continuation the appropriate number of characters,
- put it back in the process list as a RUN and add its key to the run queue.
- run the process!

```
>|| doDeviceNow :: deviceId -> process
>doDeviceNow did state
>     = cont' state'
>       where
```

First we take the state apart.

```
>       (plist, run, devs) = state
>       (nkey: restq, PART_TTY inp lst devname) = devs!did
```

Read either the next line or the next character from inp. inp' is what's left after the read, dta is what we have read.

```
>       (txtline, restline) = chopuntil (='\n') inp
>       (chr: rest) = inp
>       inp'  = rest, lst = CBREAK
>             = restline, lst = LINE
>       dta   = [chr], lst = CBREAK
>             = txtline, lst = LINE
```

Take the blocked process out of the state and hand it the new stuff we have read.

```
>       (DEV name cont did' pend) = indexKey nkey plist
>       cont' = cont (TTY_REPLY (P dta))
>       pstate' = RUN name cont' pend
```

Rebuild the state, but with the ex-blocked process off the waiting list for the tty and back on the run queue. It's going to be the next thing we run, so its key goes at the head of the run queue.

```
>       run' = nkey:run
>       plist' = updateKey nkey pstate' plist
>       devs' = update did (restq, PART_TTY inp' lst devname) devs
>       state' = (plist', run', devs')
```

Chose and run a new process from the run queue. Put the old process key at the end of the run queue.

```
>|| doRunnable :: process
>doRunnable (plist, run, devs)
>     = cont (plist, run', devs)
>       where
>       (okey:rest) = run
>       run' = postfix okey rest
>       ckey = hd run'
>       (RUN name cont pend) = indexKey ckey plist
```

Wait for a device to become ready. We grab all the input streams from devs and look along them for the first non-hiaton. This is quite complicated! Because of the way `nb_read` works, we need to be very careful about our order of evaluation.

Once we have found a device one of whose processes could be run, we call `doDeviceNow` to schedule it.

```
>|| doDeviceLater :: process
>doDeviceLater state
>     = doDeviceNow did state'
>       where
```

First we take the state apart.

```
>       (plist, run, devs) = state
```

Get all the terminal input streams, paired with their deviceId. Filter out devices which have no waiting processes.

```
>       streams = [ (did, inp) | (did,
>                   (q, PART_TTY inp lst devname)) <-
>                           zip2 [0..] devs; q ~= [] ]
```

Useful functions: drop the head of every stream, find the streams that have output ready, test to see if any streams have output available.

```
>       step strs = [ (did, tl str) | (did, str) <- strs ]
>       ready strs = [ did | (did, str) <- strs; hd str ~= hiaton ]
>       avail strs = ready strs ~= []
```

Loop `step` along the streams, find the first step at which one stream has some input ready and find its `did`.

```
>       steps = iterate step streams
>       (found: junk) = dropwhile ((~) . avail) steps
>       (did: morejunk) = ready found
```

Replace all our shortened streams in the device list.

```
>       devs' = foldr plopback devs found
>       plopback (did, newstr) devs
>             = update did (q, PART_TTY newstr lst devname) devs
>               where
>               (q, PART_TTY inp lst devname) = devs!did
```

The new state.

```
>       state' = (plist, run, devs')
```

Shut down KAOS. We write the (presumably) modified discs back to UNIX with a series of `Tofiles`, then dump the system state.

```
>|| exitKaos :: process
>exitKaos (plist, run, devs)
>     = plopdiscs ++ [Stdout helpful]
>       where
```

```
>       plopdiscs = [ Tofile discname (concat disc) |
>                       (q, PART_DISC disc discname) <- devs ]
>       helpful
>             = "Run queue empty - dump of system state:\n" ++
>               showkeyedList
>                       (showprocessState (const "item"))
>                       plist
```

A.7 Process removal

Remove a process. There are several things to do to unlink the process

- Signal errors for any processes waiting to send to the process we are trying to remove.
- If the process we are removing is waiting on output, we have to remove its `processId` from the pending list of the process it is trying to send to.
- Remove the process from the RUN queue if necessary.
- Remove the process from the queues in the device list if necessary.

```
>removeProcess :: processId -> machineState -> machineState
>removeProcess pid (plist, run, devs)
>     = switch (indexKey pid plist)
>       where
>       switch (IN name cont)
>             = (plist', run, devs)
>               where
>               plist' = removeKey pid plist
>       switch (RUN name cont pend)
>             = foldr restartProcess st' pend
>               where
>               st' = (plist', run', devs)
>               plist' = removeKey pid plist
>               run' = filter (~=pid) run
>       switch (OUT name cont wpid pend)
>             = foldr restartProcess st' pend
>               where
```

```
>               st' = (plist'', run, devs)
>               plist' = removeKey pid plist
>               plist'' = removePending pid wpid plist'
>       switch (DEV name cont did pend)
>             = foldr restartProcess st' pend
>               where
>               (q, part) = devs!did
>               st' = (plist', run, devs')
>               devs' = update did (q', part) devs
>               q' = filter (~=pid) q
>               plist' = removeKey pid plist
```

Restart a process blocked on output.

```
>restartProcess :: (processId, *) -> machineState -> machineState
>restartProcess (pid, x) (plist, run, devs)
>     = switch (indexKey pid plist)
>       where
>       switch (OUT name cont wpid pend)
>             = (plist', pid:run, devs)
>               where
>               pstate' = RUN name (cont (I err)) pend
>               plist' = updateKey pid pstate' plist
>               err = ERR_NOPROCESS "receiver killed"
```

Remove a `processId` from a pending list.

```
>removePending
>    :: processId -> processId -> keyedList (processState item) ->
>       keyedList (processState item)
>removePending pid wpid plist
>     = switch (indexKey wpid plist)
>       where
>       fstneq a (b,x) = a ~= b
>       switch (RUN name cont pend)
>             = plist'
>               where
>               plist' = updateKey wpid
>                                   (RUN name cont pend') plist
>               pend' = filter (fstneq pid) pend
```

```
>       switch (OUT name cont wait pend)
>             = plist'
>               where
>               plist' = updateKey wpid
>                                (OUT name cont wait pend') plist
>               pend' = filter (fstneq pid) pend
>       switch (DEV name cont did pend)
>             = plist'
>               where
>               plist' = updateKey wpid
>                                (DEV name cont did pend') plist
>               pend' = filter (fstneq pid) pend
```

A.8 Message passing

First do a receive for the current process. We switch depending on the state of the pending list; if it's empty we block the process on input, otherwise we take the first `processId`, copy the item across and restart the sender.

```
>receiveProcess
>     :: (lift * -> process) -> machineState -> machineState
>receiveProcess rcont (plist, run, devs)
>      = switch (indexKey rpid plist)
>        where
>        rpid = hd run
>        switch (RUN name old [])
>              = (plist', run', devs)
>                where
>                plist' = updateKey rpid
>                                (IN name (changetype rcont)) plist
>                run' = tl run
>        switch (RUN rname old ((spid, x): rest))
>              = (plist'', run', devs)
>                where
>                (OUT sname scont swait spend) = indexKey spid plist
>                scont' = scont (P ())
>                rcont' = (changetype rcont) (P x)
>                plist' = updateKey spid
>                                (RUN sname scont' spend) plist
```

```
>               plist'' = updateKey rpid
>                                 (RUN rname rcont' rest) plist'
>               run' = postfix spid run
```

Send a message from the current process. We switch depending on the state of the process we are sending to. If it is waiting on input, we copy the item across and restart it. If it is OUT, RUN or DEV, we add the sender to the end of the pending list and block the sender.

```
>sendProcess
>    :: (lift () -> process) -> * -> processId -> machineState ->
>       machineState
>sendProcess scont x rpid (plist, run, devs)
>     = switch (indexKey rpid plist)
>       where
>       spid = hd run
>       (RUN sname old spend) = indexKey spid plist
>       switch (IN rname rcont)
>             = (plist'', run', devs)
>               where
>               spstate = RUN sname (scont (P ())) spend
>               rpstate = RUN rname ((changetype rcont) (P x)) []
>               plist' = updateKey spid spstate plist
>               plist'' = updateKey rpid rpstate plist'
>               run' = postfix rpid run
>       switch (OUT rname rcont rwait rpend)
>             = (plist'', run', devs)
>               where
>               spstate = OUT sname scont rpid spend
>               rpstate = OUT rname rcont rwait rpend'
>               rpend' = postfix (spid, changetype x) rpend
>               plist' = updateKey spid spstate plist
>               plist'' = updateKey rpid rpstate plist'
>               run' = tl run
>       switch (RUN rname rcont rpend)
>             = (plist'', run', devs)
>               where
>               spstate = OUT sname scont rpid spend
>               rpstate = RUN rname rcont rpend'
>               rpend' = postfix (spid, changetype x) rpend
```

```
>               plist' = updateKey spid spstate plist
>               plist'' = updateKey rpid rpstate plist'
>               run' = tl run
>       switch (DEV rname rcont did rpend)
>             = (plist'', run', devs)
>               where
>               spstate = OUT sname scont rpid spend
>               rpstate = DEV rname rcont did rpend'
>               rpend' = postfix (spid, changetype x) rpend
>               plist' = updateKey spid spstate plist
>               plist'' = updateKey rpid rpstate plist'
>               run' = tl run
```

A.9 Process creation

We make the various sends and receives for the new function and pop it into the state at the end of the run queue. Currently, `addProcess` always succeeds. Slight problem: the functions we pass to the new process need to know the key `addKey` will return. Solution: We `addKey` a dummy value, then `updateKey` it with the real thing when we know the key.

```
>addProcess
>    :: [char] -> newProcess * -> machineState -> machineState
>addProcess name newjob (plist, run, devs)
>     = (plist'', run', devs)
>       where
>       (newkey, plist') = addKey undef plist
>       plist'' = updateKey newkey newprocess plist'
>       newprocess = RUN name newjob' []
>       newjob' = newjob (sendWrap newkey)
>                        receiveWrap (killWrap newkey)
>       run' = postfix newkey run
```

The functions `addProcess` hands to a new process. First the kill function. We check to make sure the function is not out of date, then call `removeProcess`. We pass success/fail back to the killer and exit through the scheduler.

```
>|| killWrap :: processId -> killfun
>killWrap dpid kcont state
```

```
>       = schedule estate, ~checkKey dpid plist
>       = schedule state'', otherwise
>         where
>         (plist, run, devs) = state
>         estate = replaceRunnable (kcont (I err)) state
>         state'' = removeProcess dpid state'
>         state' = replaceRunnable (kcont (P ())) state
>         err = ERR_NOPROCESS "kill function out of date"
```

Receiving a message. Very easy! We don't need to do an isuptodate check.

```
>|| receiveWrap :: receive *
>receiveWrap rcont state
>       = schedule state'
>         where
>         state' = receiveProcess rcont state
```

Sending a message. We call sendProcess to do the dirty work, after checking that the function is not out of date. We exit through the scheduler.

```
>|| sendWrap :: processId -> send *
>sendWrap rpid x scont state
>       = schedule estate, ~checkKey rpid plist
>       = schedule state', otherwise
>         where
>         (plist, run, devs) = state
>         estate = replaceRunnable (scont (I err)) state
>         state' = sendProcess scont x rpid state
>         err = ERR_NOPROCESS "send function out of date"
```

Appendix B

KAOS Libraries

B.1 Introduction

This appendix contains the libraries used by the KAOS kernel. It is included to help understanding of appendix A rather than because it's of interest in its own right!

B.2 `misc.m` — Various useful functions

```
>|| misc.m - Other stuff


>%include "os/lib/char"
>%include "os/lib/lift"
```

A couple of useful little functions.

Given a score function on things and a list of things, chose the thing with the highest score. Return its index in the list and its score.

```
>chose_best :: (* -> num) -> [*] -> (num, num)
>chose_best score_fn things
>       = foldl f (0, sc a) (zip2 [1..] (map sc x))
>         where
>         f (p, s) (p', s')
>         (p', s±), s' > s
>                 = (p, s), otherwise
```

Update a list; replace the nth element by x.

```
>update :: num -> * -> [*] -> [*]
>update n x list
>      = error ("list too short for update; asked to update pos " ++
>                 show n ++ " in list of length " ++ show (#list))
>                        , ~(0 <= n < #list)
>      = take n list ++ [x] ++ drop (n+1) list
>                        , otherwise
```

Remove an element from a list. The nth is taken out and the rest shuffled up.

```
>remove :: num -> [*] -> [*]
>remove n list
>      = error ("list too short for remove; asked to remove element " ++
>                 show n ++ " in list of length " ++ show (#list))
>                        , ~(0 <= n < #list)
>      = take n list ++ drop (n+1) list
>                        , otherwise
```

Turn a num into 4 bytes and 4 bytes into a num. During debugging, these chaps sometimes have to be looked at. Store as ASCII until it seems to be working. This pair of functions will be

```
>|| ctoi :: [char] -> num
>|| ctoi [a,b,c,d]
>||   = (code a) * 16777216 +
>||     where
>||     (code b) * 65536 +
>||     (code c) * 256 +
>||     (code d)


>|| itoc :: num -> [char]
>|| itoc n
>||   = (decode a):(decode b):(decode c):(decode d):[]
>||     where
>||     a = n div 16777216
>||     b = (n div 65536) mod 256
>||     c = (n div 256) mod 256
>||     d = n mod 256
```

Have to be very careful about ctoi; numval is fussy about input formats!

```
>ctoi :: [char] -> num
>ctoi list
>       = numval list'
>         where
>         list' = "0", digs = []
>               = digs, otherwise
>         digs = takewhile isdigit list

>itoc :: num -> [char]
>itoc n = take 4 ((show n) ++ "    ")
```

Sort a list on a predicate. Predicate returns True if the two things passed are in the right order.

```
>psort :: (* -> * -> bool) -> [*] -> [*]
>psort pred []
>       = []
>psort pred (a:x)
>       = insert a (psort pred x)
>         where
>         insert a [] = [a]
>         insert a (b:x)
>               = a:b:x, pred a b
>               = b: insert a x, otherwise
```

Useful for input stream processing; chop a list in two. Kind of a combined take and drop. The two result lists ++ed together should be the same as the input list.

```
>chop :: num -> [*] -> ([*], [*])
>chop n list = (take n list, drop n list)
```

And a variant: chop until a predicate is true. We leave the element that caused us to stop scanning at the end of the first list (this is the behaviour one usually wants; see choplist below). This is supposed to be analogous to a repeat until loop — at least one element is always consumed.

```
>chopuntil :: (* -> bool) -> [*] -> ([*], [*])
>chopuntil pred list
>       = jiggle (chopwhile ((~) . pred) list)
>         where
>         jiggle (s, []) = (s, [])
>         jiggle (s, a:x) = (postfix a s, x)
```

Chop while a predicate is true. This is supposed to be analogous to a while loop — we can consume zero elements. DANGER: `choplist (chopwhile (~='\n'))` will not terminate! It'll loop on the first `\n` it hits.

```
>chopwhile :: (* -> bool) -> [*] -> ([*], [*])
>chopwhile pred list
>      = (takewhile pred list, dropwhile pred list)
```

Another variant: chop an entire list up. It should be possible to write this with a fold or something, but I can't quite see how. This is why we wanted `chopuntil` to behave as above:

```
choplist (chopuntil (='\n')) "my cat\nlikes" = ["my cat\n", "likes"]
```

```
>choplist :: ([*] -> ([*], [*])) -> [*] -> [[*]]
>choplist cf
>      = chp . cf
>        where
>        chp (blk, []) = [blk]
>        chp (blk, rest) = blk: chp (cf rest)
```

Find the index of something in a list. Return -1 for not found.

```
>findIndex :: (* -> bool) -> [*] -> lift num
>findIndex pred
>      = find 0
>        where
>        find n [] = I (ERR_NOTFOUND "findIndex fails")
>        find n (a:rest)
>              = P n, pred a
>              = find (n+1) rest, otherwise
```

B.3 char.m — Useful character functions

```
>|| char.m --- functions on ASCII characters
```

Lifted from Robert Duncan's libraries, 20th May 1987.

Some sensible names.

```
>escchar = '\27'
>delchar = '\127'
>belchar = '\7'
>newline = '\n'
>space = ' '
>tab = '\t'
>hiaton = '\191'
>killchar = '\11'
```

Convert c to its corresponding control character.

```
>ctrl c = decode (code c - code '@')
```

Character type classifications

```
>isupper c = 'A' <= c <= 'Z'
>islower c = 'a' <= c <= 'z'
>isalpha c = isupper c \/ islower c
>isdigit c = '0' <= c <= '9'
>isalphanum c = isalpha c \/ isdigit c
>isprint c = ' ' <= c < delchar
>ismark c = ' ' < c < delchar
>isspace = member " \t\n\r\f"
>isctrl c = c < ' ' \/ c = delchar
```

Change case of c.

```
>toupper c
>     = decode (code 'A' - code 'a' + code c), islower c
>     = c, otherwise
```

```
>tolower c
>     = decode (code 'a' - code 'A' + code c), isupper c
>     = c, otherwise
```

B.4 key.m — Maintain keyed lists

```
>|| key.m - keyed lists!
```

This script maintains a keyed list data structure. Keyed lists are used throughout KAOS for all kinds of things, from process tables in the kernel to file id tables in the disc device driver.

Export two versions of each function: the ones with an 'e' in front are lifted, the others bomb out with a conformality error if they have trouble.

```
>%export
>        keyedList emptyKey listKey
>        addKey updateKey removeKey indexKey checkKey
>        eaddKey eupdateKey eremoveKey eindexKey
>        showkeyedList unpackKey
```

```
>%include "os/lib/lift"
>%include "os/lib/misc"
```

For a keyed list of *, we keep a list of (key, *) pairs. Keys come in two parts

$$key = id * 65536 + < \text{index of item in } \texttt{keyedList} >$$

To look up an element in the list, we take the key apart, fetch that element from the list with '!' and check the ids. If they match, we have found it. Ids are supposed to be unique. They provide a measure of security as well as accelerating lookup. To help uniqueness of ids, we hold the next id we will allocate; to avoid constant #ing of lists before attempting subscripting, we hold the current length of the list.

Since the position of something in the list is bound up with its key, when removing things from the list we cannot simply take it out and shuffle everything else down. We replace the missing element with a ZOMBIE. When allocating new keys, we search the list for the first ZOMBIE and overwrite that.

And the abstype.

```
>abstype keyedList *
>with   emptyKey :: keyedList *
>       eaddKey :: * -> keyedList * -> lift (listKey, keyedList *)
>       eupdateKey :: listKey -> * -> keyedList * -> lift (keyedList *)
```

```
>       eremoveKey :: listKey -> keyedList * -> lift (keyedList *)
>       eindexKey :: listKey -> keyedList * -> lift *
>       addKey :: * -> keyedList * -> (listKey, keyedList *)
>       updateKey :: listKey -> * -> keyedList * -> keyedList *
>       removeKey :: listKey -> keyedList * -> keyedList *
>       indexKey :: listKey -> keyedList * -> *
>       showkeyedList :: (* -> [char]) -> keyedList * -> [char]
>       unpackKey :: keyedList * -> [(listKey, *)]
```

Ids, keys and indexes are just nums.

```
>listKey == num
>listId == num
>listIndex == num


>keyedList *
>    == ([keyListElement *],
>        listId,
>        num)


>keyListElement *
>    ::= ZOMBIE |                    || A place marker
>        ELEMENT listKey *
```

The implementation equations. These are straightforward.

Take `listKeys` apart and put them back together. Making a `listKey` is easy: we shift the id up 16 bits and add on the index.

```
>|| mkkey :: listId -> listIndex -> listKey
>mkkey id index = id * 2^16 + index
```

Take the id out of a key.

```
>|| ktoid :: listKey -> listId
>ktoid key = key div 2^16
```

Take the index out of a key.

```
>|| ktoin :: listKey -> listIndex
>ktoin key = key mod 2^16
```

Am empty keyedList!

```
>|| emptyKey :: keyedList *
>emptyKey = ([], 0, 0)
```

To add a new key, we search for a `ZOMBIE`; failing that we extend the list. The `lift` is in case we ever get around to adding error trapping. Currently `keyedLists` are limited to 65536 elements.

```
>|| eaddKey :: * -> keyedList * -> lift (listKey, keyedList *)
>eaddKey
>       = P . addKey


>|| addKey :: * -> keyedList * -> (listKey, keyedList *)
>addKey x klist
>       = switch (findIndex (=ZOMBIE) list)
>         where
>         (list, next, len) = klist
>         switch (I err)
>               = (key, (postfix ele list, next + 1, len + 1))
>                 where
>                 key = mkkey next len
>                 ele = ELEMENT key x
>         switch (P n)
>               = (key, (update n ele list, next + 1, len))
>                 where
>                 key = mkkey next n
>                 ele = ELEMENT key x
```

Update an existing element in a `keyedList`. We have to check that the index is sensible before we try to '`!`' the list with it! We then need to check that the position in question contains an `ELEMENT` rather than a `ZOMBIE`, and that the keys match. The non-error returning version dispenses with all this namby-pamby error checking.

```
>|| eupdateKey :: listKey -> * -> keyedList * -> lift (keyedList *)
>eupdateKey key x klist
```

```
>       = I errres, ~(0 <= index < len)
>       = switch (list!index), otherwise
>         where
>         errres = ERR_NOTFOUND "eupdateKey fails"
>         ele = ELEMENT key x
>         (list, next, len) = klist
>         index = ktoin key
>         switch ZOMBIE = I errres
>         switch (ELEMENT oldkey oldx)
>               = I errres, key ~= oldkey
>               = P (update index ele list, next, len), otherwise
```

```
>|| updateKey :: listKey -> * -> keyedList * -> keyedList *
>updateKey key x klist
>       = (update index ele list, next, len)
>         where
>         (list, next, len) = klist
>         index = ktoin key
>         ele = ELEMENT key x
```

Remove an element from a keyedList by key. In the ecase we check it's there first. We shorten the list if this is the last element and turn the element into a ZOMBIE otherwise

```
>|| eremoveKey :: listKey -> keyedList * -> lift (keyedList *)
>eremoveKey key klist
>       = I errres, ~(0 <= index < len)
>       = switch (list!index), otherwise
>         where
>         errres = ERR_NOTFOUND "eremoveKey fails"
>         (list, next, len) = klist
>         index = ktoin key
>         switch ZOMBIE = I errres
>         switch (ELEMENT oldkey x)
>               = I errres, oldkey ~= key
>               = P (take (len - 1) list, next, len - 1), index = (len - 1)
>               = P (update index ZOMBIE list, next, len), otherwise
```

```
>|| removeKey :: listKey -> keyedList * -> keyedList *
>removeKey key klist
>     = (take (len - 1) list, next, len - 1), index = (len - 1)
>     = (update index ZOMBIE list, next, len), otherwise
>       where
>       (list, next, len) = klist
>       index = ktoin key
```

Look up by key. As before we do several checks in the ecase.

```
>|| eindexKey :: listKey -> keyedList * -> lift *
>eindexKey key klist
>     = I err, ~(0 <= index < len)
>     = switch (list!index), otherwise
>       where
>       err = ERR_NOTFOUND "eindexKey fails"
>       (list, next, len) = klist
>       index = ktoin key
>       switch ZOMBIE = I err
>       switch (ELEMENT oldkey x)
>             = I err, oldkey ~= key
>             = P x, otherwise
```

```
>|| indexKey :: listKey -> keyedList * -> *
>indexKey key klist
>     = x
>       where
>       (list, next, len) = klist
>       index = ktoin key
>       (ELEMENT oldkey x) = list!index
```

Check that a key is valid.

```
>checkKey :: listKey -> keyedList * -> bool
>checkKey key klist
>     = isproper (eindexKey key klist)
```

Show a `keyedList`. Print as a list of pairs `(key, item)`. We export this, as it is sometimes necessary to call this function explicitly (see `kernel.m` for an example).

```
>|| showkeyedList :: (* -> [char]) -> keyedList * -> [char]
>showkeyedList showx keyed
>     = "keyedList [\n" ++
>       foldr comma [] (map showp (unpackKey keyed)) ++ "]"
>       where
>       showp (key, x) = shownum key ++ "," ++ showx x
>       comma new [] = new
>       comma new sofar = new ++ ";\n " ++ sofar
```

Return a list of the contents of a keyedList. Also exported for convenience, although it's rather unsafe.

```
>|| unpackKey :: keyedList * -> [(listKey, *)]
>unpackKey (list, next, len)
>     = [(key, x) | (ELEMENT key x) <- list]
```

B.5 lift.m — Add an error element to a type

```
>|| lift.m - things to add an error element to a type


>%export + "os/sys/hdr/errs"


>%include "os/sys/hdr/errs"        || KAOS errors
```

This is used throughout KAOS for functions that may fail. Also some functions for adding/stripping the improper element.

```
>lift * ::= P * | I sysErr


>addlift :: * -> (lift *)
>addlift st = P st


>striplift :: (lift *) -> *
>striplift (P x) = x
>striplift (I s) = error "striplift: can't strip improper element"
```

```
>isproper :: (lift *) -> bool
>isproper (P x) = True
>isproper (I s) = False


>isimproper :: (lift *) -> bool
>isimproper = (~) . isproper
```

B.6 errs.m — KAOS error codes

```
>|| errs.m - errors KAOS functions can return
```

The string attached to each error is intended to be used for diagnostics for the user rather than for machine consumption.

```
>sysErr
>    ::= ERR_INUSE [char] |
>        ERR_INAPPROPRIATE [char] |
>        ERR_NOPROCESS [char] |
>        ERR_BADNAME [char] |
>        ERR_NOTFOUND [char] |
>        ERR_ISFILE [char] |
>        ERR_NOSPACE [char] |
>        ERR_NOTIMPLEMENTED [char] |
>        ERR_OUTOFRANGE [char] |
>        ERR_OPENFILE [char] |
>        ERR_NOENCLOSING [char] |
>        ERR_NOTEMPTY [char] |
>        ERR_DEVICEERROR [char] |
>        ERR_EOF [char] |
>        ERR_USERERROR [char]
```

Check two sysErrs for equality .. we want to ignore the string.

```
>equalErr :: sysErr -> sysErr -> bool
>equalErr (ERR_INUSE x) (ERR_INUSE y) = True
>equalErr (ERR_INAPPROPRIATE x) (ERR_INAPPROPRIATE y) = True
>equalErr (ERR_NOPROCESS x) (ERR_NOPROCESS y) = True
>equalErr (ERR_BADNAME x) (ERR_BADNAME y) = True
```

```
>equalErr (ERR_NOTFOUND x) (ERR_NOTFOUND y) = True
>equalErr (ERR_ISFILE x) (ERR_ISFILE y) = True
>equalErr (ERR_NOSPACE x) (ERR_NOSPACE y) = True
>equalErr (ERR_NOTIMPLEMENTED x) (ERR_NOTIMPLEMENTED y) = True
>equalErr (ERR_OUTOFRANGE x) (ERR_OUTOFRANGE y) = True
>equalErr (ERR_OPENFILE x) (ERR_OPENFILE y) = True
>equalErr (ERR_NOENCLOSING x) (ERR_NOENCLOSING y) = True
>equalErr (ERR_NOTEMPTY x) (ERR_NOTEMPTY y) = True
>equalErr (ERR_DEVICEERROR x) (ERR_DEVICEERROR y) = True
>equalErr (ERR_EOF x) (ERR_EOF y) = True
>equalErr (ERR_USERERROR x) (ERR_USERERROR y) = True
>equalErr x y = False


>showsysErr :: sysErr -> [char]
>showsysErr (ERR_INUSE x) = "in use" ++ showstr x
>showsysErr (ERR_INAPPROPRIATE x) = "inappropriate message" ++ showstr x
>showsysErr (ERR_NOPROCESS x) = "no such process" ++ showstr x
>showsysErr (ERR_BADNAME x) = "bad name" ++ showstr x
>showsysErr (ERR_NOTFOUND x) = "not found" ++ showstr x
>showsysErr (ERR_ISFILE x) = "object is file" ++ showstr x
>showsysErr (ERR_NOSPACE x) = "no space" ++ showstr x
>showsysErr (ERR_NOTIMPLEMENTED x) = "not implemented" ++ showstr x
>showsysErr (ERR_OUTOFRANGE x) = "out of range" ++ showstr x
>showsysErr (ERR_OPENFILE x) = "file is open" ++ showstr x
>showsysErr (ERR_NOENCLOSING x) = "no enclosing directory" ++ showstr x
>showsysErr (ERR_NOTEMPTY x) = "directory not empty" ++ showstr x
>showsysErr (ERR_DEVICEERROR x) = "device error" ++ showstr x
>showsysErr (ERR_EOF x) = "end of file" ++ showstr x
>showsysErr (ERR_USERERROR x) = "user error" ++ showstr x


>showstr "" = ""
>showstr x = ": " ++ x
```

Brief explaination:

`ERR_INUSE` You get this if you try to access some non-shareable resource, like writing to the terminal while someone else is using it.

`ERR_INAPPROPRIATE` Things like sending a `FILE_DELETE` message to the terminal.

`ERR_NOPROCESS` Attempt to reference a process which has been killed ...this can happen with old wrappers.

`ERR_BADNAME` Bad format for a pathName ...eg. trying to open a terminal as if it were a disc. Also things like file name too long.

`ERR_NOTFOUND` Not found. Returned for things like file not found etc.

`ERR_ISFILE` Returned if (eg.) a path contains something which is not a directory.

`ERR_NOSPACE` Things like disc full etc.

`ERR_NOTIMPLEMENTED` Returned for requests for facilities not there yet.

`ERR_OUTOFRANGE` Some kind of bad argument ...eg. trying to format a disc with a silly number of sectors etc.

`ERR_OPENFILE` Returned by the file system for things like trying to delete an open file.

`ERR_NOENCLOSING` Returned for things like trying to delete the root directory.

`ERR_NOTEMPTY` Returned for (perhaps implicit) attempts to delete non-empty directories.

`ERR_DEVICEERROR` Returned (for example) by the disc device driver when a read/write error occurs.

`ERR_EOF` End of file.

`ERR_USERERROR` Applications are supposed to use this if nothing else fits. The attached string should obviously be used to give more specific information!

B.7 `collection.m` — Manage hetrogenous lists

```
>|| collection.m - hetrogeneous lists
```

This file provides a simple structure for holding objects of any type — rather like `keyedList *`. It is intended to hide `changetype`.

```
>%export
>       collection handle
>       emptyCollection addCollection indexCollection eindexCollection
>       updateCollection removeCollection
```

```
>%include "os/lib/lift"
>%include "os/lib/misc"
>%include "os/lib/key"
>%include <local/disgusting>
```

A `collection` is a set of objects of varying type. Associated with each stored element is a handle of the corresponding type. This can be used as an argument to the index/update/remove functions.

```
>abstype collection, handle *
>with    emptyCollection :: collection
>        addCollection :: * -> collection -> (handle *, collection)
>        indexCollection :: handle * -> collection -> *
>        eindexCollection :: handle * -> collection -> lift *
>        updateCollection :: handle * -> * -> collection -> collection
>        removeCollection :: handle * -> collection -> collection
```

`collections` are implemented on top of `keyedLists` for efficiency. We are careful to use the error trapping versions of the key functions, as errors with `changetype` involved can be hard to track!

```
>collection == keyedList item
>item == num || changetyped later!
>handle * == listKey
```

We store `items` in the `keyedList` — an `item` is intended to be some untyped fragment of graph. We use `changetype` to turn the objects we store into `items` and back. The typed handles guarantee that playing with types in this way is safe: we always `changetype` something back into its original form.

The `collection` functions just call the appropriate functions in `key.m`.

```
>|| emptyCollection :: collection
>emptyCollection = emptyKey


>|| addCollection :: * -> collection -> (handle *, collection)
>addCollection x coll
>     = addKey (changetype x) coll
```

```
>|| indexCollection :: handle * -> collection -> *
>indexCollection h coll
>      = changetype x
>        where
>        (P x) = eindexKey h coll


>|| eindexCollection :: handle * -> collection -> lift *
>eindexCollection h coll
>      = changetype (eindexKey h coll)


>|| updateCollection :: handle * -> * -> collection -> collection
>updateCollection h x coll
>      = newc
>        where
>        (P newc) = eupdateKey h (changetype x) coll


>|| deleteCollection :: handle * -> collection -> collection
>removeCollection h col
>      = newc
>        where
>        (P newc) = eremoveKey h col
```

Appendix C

Interactions

This appendix contains the complete source to the `interact` library described in chapter 5.

C.1 Introduction

This script provides some simple routines to help the handling of input and output streams. Using this package, one can write Mirandaish code that side-effects I/O in an intuitive sort of way. It is based havily on Simon Thompson's interactive input/output package. The implementation of exception handling arose from discussions with Mark Longley, who uses something similar in his version of interacts[13].

`interacts` do five main things

- hide the continuations that the kernel uses. `interacts` are functions that can be composed etc.
- keep track of the input wrapper and the kill wrapper. User just calls `get` for input and just exits to kill himself.
- fake multiple inputs. `get` takes as an argument a predicate on `inputType` and returns the next input value that satisfies the predicate. Incoming values which fail are saved in a queue which is searched by the next call to `get` before the input is read again.
- provide a simple exception handling system. `catch` marks a point which errors should be caught at, `raise` raises an error.

- provide a state which interacts can side effect. This is important if you use the exception system: you need a model for your history which is reliable even after disasters.

C.2 lowinter.m — The core of interact

```
>|| lowinter.m - version 5
```

This is supposed to provide the minimum functionality. inter.m builds on this abstype to provide a nicer interface.

```
>%export
>       interact
>       get execute run comp return raise catch
>       getState putState applyState

>%free {
>       inputType :: type;
>       userState :: type;
>}

>%include "os/sys/ker/kernel"
>%include "os/lib/lift"
>%include "os/lib/misc"
```

The abstype. See below for explainations of all the functions.

```
>abstype interact *
>with   get :: (inputType -> bool) -> interact inputType
>       scaninp :: (inputType -> bool) -> interact inputType
>       execute :: ((* -> process) -> process) -> interact *
>       run :: interact * -> receive inputType -> kill -> process
>       comp :: (* -> interact **) -> interact * -> interact **
>       return :: * -> interact *
>       raise :: sysErr -> interact *
>       catch :: (sysErr -> interact *) -> interact * -> interact *
>       getState :: interact userState
>       putState :: userState -> interact ()
>       applyState :: (userState -> userState) -> interact ()
```

A `contination` is the most basic part of an `interact` — it's a function taking a `*`, a list of pending input items, a catch point, the receive wrapper for this process, the kill wrapper for this process and the user's state. Catch points are continuations saved up by `catch` ready for `raise` to return to; see the (brief!) discussion at the top of this file.

`catchPoint` is defined as a algebraic type to stop Miranda complaining about circularity. Annoying!

```
>continuation *
>     == * -> [inputType] -> catchPoint -> userState ->
>        receive inputType -> kill -> process
>catchPoint
>     ::= CATCH (continuation sysErr)
>interact *
>     == continuation (continuation *)
```

Explaination: The first argument an `interact` sees is the continuation it must pass it's result on to. If this `interact` is a `raise`, it can call the `catchPoint` instead. These continuations will already have had *their* continuations wired into them by either `comp` or `catch`.

`get` returns the next input item satisfying the predicate passed. Inputs that fail the test are saved in a queue — the next call to `get` scans this queue of previously rejected messages before reading from the input again.

First step: look through the queue of saved messages for one that satifies the predicate. If we find one, return it. Otherwise we hop into `scaninp` below.

```
>|| get :: (inputType -> bool) -> interact inputType
>get pred cont q
>      = cont x q', isproper pos
>      = scaninp pred cont q, otherwise
>        where
>        pos = findIndex pred q
>        (P n) = pos
>        x = q!n
>        q' = remove n q
```

This subfunction of `get` scans the input stream looking for a message statisfying the predicate. Rejected messages are queued, ready for the next call to `get`.

```
>|| scaninp :: (inputType -> bool) -> interact inputType
```

```
>scaninp pred cont q catchp st rx k
>     = rx (scan q)
>       where
>       scan q' (I err)
>             = raise err cont q' catchp st rx k
>       scan q' (P x)
>             = cont x q' catchp st rx k, pred x
>             = rx (scan (postfix x q')), otherwise
```

`execute` does some kind of kernel-type continuation based function as if it were an `interact`. Easy to do `put`/`spawn`/`kill`/`sys` etc. in terms of this.

```
>|| execute :: ((* -> process) -> process) -> interact *
>execute thing cont q catchp st rx k
>     = thing grab
>       where
>       grab x = cont x q catchp st rx k
```

`run` runs an `interaction`. Print an error message if it fails. Set the start state to `undef`.

```
>|| run :: interact * -> receive inputType -> kill -> process
>run inter
>     = inter cont [] (CATCH wicketkeeper) undef
>       where
>       cont y (a:x) catchp st rx k
>             = sys (Stdout helpful) (k undef)
>               where
>               helpful = "inter exits with unread messages\n"
>       cont x [] catchp st rx k
>             = k undef
>       wicketkeeper err q catchp st rx k
>             = sys (Stdout helpful) (k undef)
>               where
>               helpful
>                     = "unexpected exit with: " ++
>                       showsysErr err ++ "\n"
```

`comp` composes two `interactions`.

```
>|| comp :: (* -> interact **) -> interact * -> interact **
>comp inter1 inter2 exit
>      = inter2 cont
>        where
>        cont x = inter1 x exit
```

return makes an interact — it's result is simply the argument passed.

```
>|| return :: * -> interact *
>return x cont
>      = cont x
```

raise hops back to the last catch. We leave the catch point on the error handler as undef — catch will fix this up for us.

```
>|| raise :: sysErr -> interact *
>raise err cont inp (CATCH catchp)
>      = catchp err inp undef
```

And catch installs a new catch point before calling an interact. It carefully restores the old catchpoint afterwards — hence the undef attached to raise above.

```
>|| catch :: (sysErr -> interact *) -> interact * -> interact *
>catch catchinter inter cont inp catchp
>      = inter tidycont inp (CATCH catchcont)
>        where
>        catchcont err inp' newcatchp
>              = catchinter err cont inp' catchp
>        tidycont x inp' newcatchp
>              = cont x inp' catchp
```

Accessing the state — the bare minimum!

```
>|| getState :: interact userState
>getState cont q catchp st
>      = cont st q catchp st
```

```
>|| putState :: userState -> interact ()
>putState newst cont q catchp oldst
>      = cont () q catchp newst
```

Slightly fancier — transform the state. Saves a lot of getting and putting if you know what you want to do.

```
>|| applyState (userState -> userState) -> interact ()
>applyState f cont q catchp oldst
>     = cont () q catchp (f oldst)
```

C.3 inter.m — Useful functions over lowinter.m

```
>|| inter.m - nice interface to lowinter.m
```

This script builds on `lowinter.m` providing a nice set of functions for progs.

```
>%export
>       then hideLift revealLift sendSys fork put debug getNext handle
>       ignore careful imap ifoldl ifoldr interFold sput
>       "os/sys/lib/lowinter"

>%free {
>       inputType :: type;
>       userState :: type;
>       debugging :: bool;                || Turn on debugging output.
>}

>%include "os/lib/lift"
>%include "os/sys/ker/kernel"
>%include
>       "os/sys/lib/lowinter"
>       {       inputType == inputType;
>               userState == userState;
>       }
```

`then` is rather like ',' in LISP — the first `interaction` is done, the result thrown away and then the second is done. We only allow `()` to be chucked though! To discard `interact` returns, use const yourself.

```
>then :: interact () -> interact ** -> interact **
>then inter1 inter2
>     = const inter2 $comp inter1
```

Get rid of a `lift` — kernel functions return lifted results, we want exceptions instead.

```
>hideLift :: (lift *) -> interact *
>hideLift (I err) = raise err
>hideLift (P x) = return x
```

We sometimes want to return a lifted result, rather than an exception.

```
>revealLift :: interact * -> interact (lift *)
>revealLift inter = catch (return . I) ((return . P) $comp inter)
```

`sendSys/fork/put` in terms of execute.

```
>sendSys :: sys_message -> interact ()
>sendSys mess
>      = execute thing $then return ()
>        where
>        thing cont = sys mess (cont ())
```

We could use a couple of dots and get rid of the arguments — but it's far too confusing.

```
>fork :: [char] -> job * -> interact ()
>fork name newjob = hideLift $comp execute (spawn name newjob)


>put :: send * -> * -> interact ()
>put wrap x = hideLift $comp execute (wrap x)
```

Debugging — print a message on the console.

```
>debug :: [char] -> interact ()
>debug msg
>       = sendSys (Stdout ("debug: " ++ msg ++ "\n")), debugging
>       = return (), otherwise
```

A useful function — often want just the next message. `getNext` does this.

```
>getNext :: interact inputType
>getNext = get (const True)
```

`catch` is a bit longwinded for some operations — often we just want to handle a specific error from an `interact`.

```
>handle
>     :: ([char] -> sysErr) -> (sysErr -> interact *) -> interact * ->
>        interact *
>handle handleerr recover
>      = catch lookout
>        where
>        lookout err
>              = recover err, equalErr err (handleerr "")
>              = raise err, otherwise
```

```
>ignore :: ([char] -> sysErr) -> interact () -> interact ()
>ignore err
>      = handle err (const (return ()))
```

Perform some `interaction` on a `*`, doing a cleanup on the `*` even if the `interaction` fails. Makes state changing operations safe.

```
>careful :: (* -> interact ()) -> (* -> interact **) -> * -> interact **
>careful clean int x
>      = doclean $comp catch (((.) . comp) raise doclean) (int x)
>        where
>        doclean y = clean x $then return y
```

Versions of `map` and `fold` that work for `interacts`.

```
>ifoldr :: (* -> ** -> interact **) -> ** -> [*] -> interact **
>ifoldr inter start = foldr (comp . inter) (return start)
```

```
>ifoldl :: (** -> * -> interact **) -> ** -> [*] -> interact **
>ifoldl inter start
>      = foldl (converse (comp . (converse inter))) (return start)
```

```
>imap :: (* -> interact **) -> [*] -> interact [**]
>imap inter
>      = ifoldr f []
>        where
>        f a sofar = (return . (:sofar)) $comp inter a
```

Another useful function — use a 'process' `interact` to `foldl` up a stream of items coming from an 'input' `interact`. The input `interact` might be `getNext`, for example, and the process `interact` might be some kind of 'process event' function. This is used as the main loop for many of the system processes. Return a new state when the input `interact` fails with `ERR_EOF`.

We wrap the input `interact` up with `revealLift`, then switch on what we see. If it's `EOF`, we can exit. If its some unexpected error, we raise it.

```
>interFold :: (* -> ** -> interact *) -> * -> interact ** -> interact *
>interFold trans start inp
>      = loop start
>        where
>        loop oldst
>               = test $comp revealLift inp
>                 where
>                 test (P x) = loop $comp trans oldst x
>                 test (I (ERR_EOF x)) = return oldst
>                 test (I err) = raise err
```

A useful function — when replying to 'untrusted' messages, we want to be careful to avoid `ERR_NOPROCESS` errors from put. This version of put just returns silently if the send fails.

```
>sput :: send * -> * -> interact ()
>sput wrap x
>      = ignore ERR_NOPROCESS (put wrap x)
```

KENT
LIBRARY
UNIVERSITY