A RANGE OF OPERATING SYSTEMS WRITTEN IN A PURELY FUNCTIONAL STYLE

Simon B. Jones

Technical Monograph PRG-42

September 1984
(published as a monograph February 1985)

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD

Department of Computer Science
University of Stirling
Stirling FK9 4LA

Also published as:

Computing Science Technical Report TR.16

Department of Computing Science
University of Stirling
Stirling FK9 4LA

A Range of Operating Systems Written in a Purely Functional Style

Project Report

ABSTRACT

From February 1982 to September 1983 the author was investigating the design and implementation of operating systems in a purely functional style. The research was carried out at the Programming Research Group, Oxford University.

This report explores a variety of different designs of operating system, and the aspects of functional programming which lend themselves to these designs. The various techniques illustrated and exploited include the construction of simple interactive systems (through lazy evaluation and recursive functions on streams), streams functions as processes and networks of processes (clear modularisation of systems), sharing of resources and time dependent systems (through the use of a non-determinate choice operator), and physically distributed systems (streams are associated with remote communication lines). Functional programming provides a powerful tool in the design and implementation of such systems.

A companion report describes the underlying abstract machine support required for the implementation of functional operating systems.

Simon B. Jones
September 1984

A Range of Operating Systems

Written in a Purely Functional Style

Project Report

CONTENTS

A Range of Operating Systems

Written in a Purely Functional Style

Project Report

FOREWORD

This document is one of a pair reporting the results of the Functional Operating Systems project commenced at the Programming Research Group, Oxford University, in February 1982.

The report is divided into two parts: The development of an abstract machine to support a purely functional systems programming language [6], and the exploration of a spectrum of functional, distributed operating systems (this document).

The two aspects of the work progressed together, driving and supporting each other. So a certain amount of the narrative text is common to both reports (in particular the Introduction), and the reports may be read independently. Nevertheless, the reports must be taken together to provide a full record of the project, as the technical details are complementary.

I would like to acknowledge the financial support of the Distributed Computing Systems Panel of the Science and Engineering Research Council.

Thanks are due, in particular, to Peter Henderson and Geraint Jones, and also to others at the Programming Research Group, for valuable ideas and feedback during the project.

Simon B. Jones
September 1984

Chapter 1

INTRODUCTION

Motivation

The project is motivated by three general observations of contemporary hardware and software developments:

1. As has often been pointed out by manufacturers and researchers, the cost of computer hardware has been falling rapidly in recent years, and may continue to do so for some years yet. This has been due to improving integrated circuit technology. For example, the Hewlett Packard HP9000 series of microprocessors pack nearly half a million switching elements onto a silicon chip approximately 6mm square. Thus, not only costs but also sizes have been decreasing. These developments make it look sensible to attempt to harness the potential of many processors working in co-operation in order to construct more powerful computers. In addition, hardware experts assert that improvements in chip technology (greater density of switching elements, reduction in power consumption, etc) are approaching their foreseeable limits. This lends even greater urgency to the investigation of multiple processor computer architectures as a means of achieving greater computer power.

2. In the field of programming there is increasing interest in the role of purely functional programming languages as a major weapon in the software engineer's armoury against the problem of complexity. Although the first purely functional programming language was invented in about 1960 [8], the functional style of programming has remained simply an intellectual curiosity for most of the intervening period. More recently, with growing maturity of functional programming, and partly as a result of research on novel computer architectures (e.g. data flow machines [3, 10], reduction machines [2, 9], functional programming is being more widely accepted as one direction towards advanced programming tools. In Britian ICL and GEC are both examining how functional programming relates to their needs for systems and applications programming.

3. One of the natural roles for functional programming seems to be its use in describing and implementing computer programs or systems conceived as collections of concurrently executing independent processes. (Note that there is no implication here that independent processes must be executed on independent processors). The processes communicate via fixed channels and are thus configured as a static network determined by the channel connections. This approach leads to very clear programs in many rather sophisticated toy problems (e.g. the sieve of Eratosthenes [4]), and well modularised programs in larger, practical applications.

Taken together, these three observations suggest a rather exciting programme of research: To use some functional programming language as the systems programming language for implementing applications which are to be executed as a network of processes distributed over a network of processors. The results of such an investigation would be to extend our understanding of the potential of functional programming as a systems programming tool, to realise this potential in the form of an implementation, and to exhibit the practical value of such an approach by building useful multiprocessor systems. We would hope to demonstrate that in large practical applications the technique leads to easily managed, easily reconfigured, well modularised implementations.

Programme of Research

The starting point for the investigation had to be a small, uncomplicated implementation of a small, uncomplicated functional programming language. This simplicity was desirable since extending the language, and its implementation, would be easier, and the fundamental properties of the extensions would not be obscured. Extending a sophisticated functional language with a complex (and probably cumbersome) implementation would be neither easy nor illuminating. Thus we chose Lispkit Lisp, and its implementation as a high level abstract SECD machine [4]. Lispkit Lisp will henceforth be referred to as simply Lispkit.

Lispkit and its implementation have been modified and extended to provide a full systems programming environment when executing on a single processor. This extended system will ultimately enable a Lispkit program to run interactively, to receive input from the keyboard and serial lines, to produce output on the screen and serial lines, and to interact with a disk based file store.

A small collection of such extended Lispkit machines will be connected via their serial line ports to give some particular network. A single Lispkit program, comprising a collection of concurrent processes, will then be distributed statically over the network to execute in a true multiprocessing fashion. A single processor in the network may support one or more processes, as may be convenient for the particular application. Communication between processes running on the same processor will occur within the machine rather than via external serial lines. The physical network of serial lines will be determined by the application, and will be reconfigured quite easily for different applications.

A typical application would be a small operating system providing a single user workstation. For example, one processor can be running an intelligent file service, another can be handling the terminal, interpreting commands and editing, and a third can be executing background jobs requested by the user. By exploiting the network of processors in this way such a system could be expected to sustain a considerable workload from the user.

Alternatively, given a collection of Lispkit machines, a programmer could construct a stand-alone Lispkit program for some application, and could connect the machines in a network appropriate to that particular application. In this way the extended Lispkit implementation could provide better performance for particular applications, as well as a powerful component in a general purpose work-station.

Functional operating systems

The progress of the project is largely driven by the requirements of the different designs of operating systems which we wish to try out. As extensions to Lispkit and its implementation become necessary, they are modified, after some deliberation, by as little as possible to maintain simplicity and cleanliness.

Many styles of operating systems may be devised within the functional framework - imagination, as usual, is the limiting factor! We have tried several distinct varieties of systems so far, but other important approaches are being investigated elsewhere [1, 7]. However, the experiments reported here demonstrate the power which is available, through functional programming, to the professional designer of personal workstations, distributed systems, intelligent business systems, and so on.

One approach is to simply try to code a fairly conventional uniprocessing operating system (e.g. in the style of CP/M or Unix) as a single monolithic program to be run on a single functional programming computer. This would not exploit concurrency at all. Nevertheless, experiments have shown that extremely powerful operating systems can be provided in this way.

The first step to exploiting concurrency is to devise systems comprising several stream processing functions connected in a network. An input stream is received from the keyboard (the user's commands) and a result stream is sent to the screen (the system's responses). Unfortunately the components of such systems tend to work in synchronisation, and there is no large scale concurrent activity.

The potential for large scale concurrent activity is conveniently introduced by using a stream merging (interleaving) operator [5]. The output of such a merging operator is some unpredictable (non-determinate) mixing of the elements of the two streams. This suggests an implementation in which the producers of the streams to be merged beaver away continuously (and concurrently), presenting stream elements to the merge operator for selection.

The use of the non-determinate choice operator in this work, and its implementation in the Lispkit machine, are quite straightforward, but the mechanism has a controversial background from the theoretical point of view [1b].

Although non-determinacy (in the guise of merge) permits the construction of systems exhibiting useful concurrency, it is by no means obvious how to exploit this potential on the user's behalf in the best way. We have started exploring designs for more sophisticated operating systems which could assist a productive user in exploiting the power available in the collection of processors at his disposal.

These experiments in system design are reported in the later chapters of this document.

The Lispkit language and SECD machine architecture

As mentioned above, Lispkit Lisp and its SECD machine implementation were chosen as the starting point for the investigation. This is a clean and simple base from which to work. The language and implementation as described in [4], provide a mechanism for executing "one shot" programs which receive all the input data, perform some computation, and produce the result, in three strictly sequential steps. The outline of a mechanism for "lazy evaluation" ("demand driven computation") is also discussed.

Thus the base language and SECD machine fall short of the requirements of the operating systems research in a number of ways:

1. A detailed mechanism for lazy evaluation is the first essential addition. The machine must be extended. The Lispkit language is not altered syntactically, but the range of programs which can be expressed in the language is considerably widened.

2. The restriction to "one shot" program execution must be removed, and a program must be allowed to work interactively between its input and output streams (typically the keyboard and screen).

3. An operator for non-determinate choice must be introduced into the language and implementation. This involves the pseudo-parallel execution of concurrent processes on a single SECD machine.

4. Finally, in order to enable the programmer to access a range of input and output devices, the SECD machine must be extended to provide a mechanism for multiple input and multiple output streams. Most of the apparatus required is already available from the previous extensions.

The modified Lispkit language and SECD machine are reported fully in [6].

The development of the extended SECD machine is closely related to similar work by Abramsky at QMC [1a].

Hardware

Detailed arguments about the hardware to be used for running distributed systems are not a major concern of the project. However that is no excuse for not considering the matter at all.

We wish to attempt to exploit concurrency at a macroscopic level in a system. That is, substantial subsystems will be allocated statically to each processor in the network. This is in contrast to the exploitation of concurrency at a microscopic level, where there is dynamic allocation of simple tasks to processors. Examples of the latter approach are data flow machines [3, 10], and reduction machines Alice [2], ZAPP [9].

Thus we require a small collection of reasonably powerful processors (e.g. half a dozen Perqs) connected in some simple, easily reconfigured way. The distribution of parallelism at the microscopic level necessitates a large collection of small processors (e.g. 10s, 100s or 1000s of transputers) connected by a complex, general purpose communications network.

There are many groups attempting valiantly to develop and assess the latter approach in various ways, and with varying results. We have decided to opt for the former, more immediate approach.

However, beyond the intention to use a small number of powerful processors, the precise hardware techniques are not under consideration. For experimental purposes we have used "off the shelf" microcomputers, such as RML 380Z, SuperBrain, Sirius, Perq and so on, as available. These machines have either one or two serial lines.

A future option could be to support all the processors and memory on a single bus. The abstraction of a collection of processors communicating via fixed channels could be provided on such hardware without the expense of bulk data transfers along serial lines. That is, perhaps, a task for someone else in the future.

Chapter 2

EXPLOITING INTERACTIVE, LAZY EVALUATION

Introduction

Chapters Two and Three of [6] describe how the implementation of Lispkit Lisp can be extended to permit the construction and execution of stream processing programs. Such programs are written as pure functions producing an output stream of s-expressions by consuming an input stream of s-expressions. Usually the input stream is associated with a computer's keyboard, and s-expressions are read only when required by the Lispkit program. Similarly, the output stream is usually associated with a computer's screen, and the Lispkit program is driven by demanding it to produce the successive s-expressions of the output stream.

A very simple example of an interactive program is the following function, which doubles, increments and outputs each number read from the input stream:

```
λ(kb).inc(double(kb))
whererec inc(s)    ≡ cons(1+head(s),inc(tail(s)))
         double(s) ≡ cons(2*head(s),double(tail(s)))
```

where the dummy identifier kb has been used simply to suggest that the numbers are read from a keyboard. When this small Lispkit program, which is complete as it stands, is compiled and executed on a lazy, interactive Lispkit system (such as one of those described in [11]) it behaves as might be expected: The program will wait until a number is typed, the incremented double of the typed number is displayed on the screen, and the program waits for another input.

A complete suite of program development tools has been developed to enable construction of Lispkit programs. Each tool is itself a purely functional Lispkit program operating on the stream input/stream output principle, and will execute on the modified SECD machine. The suite of tools, which includes a structure editor, compiler, syntax checker and source code librarian and various significant applications programs, are fully documented in [11].

A simple interactive operating system

The availability of a lazy, interactive, stream input and output Lispkit system prompts the following challenge: Is it possible to construct a purely functional, interactive program which will provide a "traditional" operating system environment in which the user may store and retrieve files, and execute programs (themselves files) which read and write files? In this context "traditional" should be taken to mean that the user types a sequence of program execution commands, possibly separated by keyboard input to an executing program, in much the same way as would be done to a Unix or CP/M system.

What follows is a very simple instance of such an operating system. For historical reasons it is called OS7 (no relation to any other which may have that name).

OS7 maintains a file store of programs and data. Each file has a name which is a single atomic value (a number or symbol), and its contents is any s-expression. Since the SECD machine has no access to a disk based file store, OS7 manages its file store as an association list in the cell heap. Programs to be executed are fetched from the file store. Input data files are fetched from the file store, and output result files are placed in the file store.

Each command which the user may type to OS7 is a 3-list specifying a program to be executed and its input and output files:

```
    (programname  dataname  resultname)
```

programname must be the name of a file currently in the file store. dataname may be either the name of a file in the store, or the special name "kb" indicating that the input is to be a sequence of items taken from the keyboard, or the special name "none" indicating that a dummy input consisting of the single atom "NONE" should be supplied to the program. "kb" and "none" have precedence over files of the same name. If a file is named as input but is not present in the file store then the atom "MISSING" is supplied to the program as input. resultname may be either a file name, in which case the output is placed in that file, or the special name "screen" indicating that the output is a sequence of items which should be sent to the screen. "screen" takes precedence over a file name - a fact that makes it impossible to create a file called "screen".

In OS7 the program execution mechanism is a source level interpreter, and so the program file named in a command must contain the source code for a Lispkit function. The function is expected to return a single result computed from a single input. Details of the interpreter will not be given here - there is no novelty in the interpreter, and similar interpreters are described in [4] and [11]. We will simply assume the availability of a function

```
    execute(f,a) ≡ ...
```

which will compute the value resulting when the function valued program f is applied to the single argument a.

When a command specifies that "kb" should be used for input, the single argument supplied to the program is a list formed from the subsequent s-expressions typed at the keyboard up to, but excluding, the first s-expression which is the atom "end". When a command specifies that "screen" should be used for output then the result from execution must be a list of items and each is placed in sequence in the output stream which is being sent to the screen.

With these preliminaries over, we can tackle the recursive functions which control the interactive kernel of the system:

```
system (kb,fs) ≡
     if prog = "MISSING"
     then cons ("ERROR", system (tail(kb),fs))
     else systemstep(prog,source,sink,tail(kb),fs)
     whererec prog   ≡ get(fs,first(head(kb)))
              source ≡ second(head(kb))
              sink   ≡ third(head(kb))
```

The system function has as arguments kb, the stream of s-expressions typed at the keyboard, and fs, the current state of the file store (an association list). Its result is a stream of outputs which will be sent to the screen. The head of kb will be a command to be obeyed, and the operations first, second and third extract the components of the command. get(fs,n) looks up the file named n in the file store fs (this is defined later) - returning "MISSING" if the file is not found.

The responsibility for executing the program and continuing the interaction is delegated to the function systemstep, which has the output stream as its result:

```
systemstep (prog,source,sink,kb,fs) ≡

     append(scr,system(kb',fs'))
     whererec
        input   ≡ if source = "kb" then untilend(kb)
                  else  if source = "none" then "NONE"
                  else  get(fs,source)

        output  ≡ execute(prog,input)
        scr     ≡ if sink = "screen" then output else "NIL"
        kb'     ≡ if source = "kb" then afterend(kb) else kb
        fs'     ≡ if sink = "screen" then fs else put(fs,sink,output)
```

Systemstep receives as arguments the text of the program to be executed, the source and sink fields of the command being obeyed, the input stream with the command removed from its head, and the current file store. The resultant output stream is formed by appending any screen output from the current command onto the output stream from continuing system interaction. The system continues interacting with a possibly shortened keyboard stream, and a possibly modified file store. This gives the illusion to the user of a system and file store which are undergoing state transitions, although the implementation is purely functional. In the local definitions of input and kb' we see how the input data for execution of the program is taken from a file or from some prefix of the keyboard stream. In the local definitions of scr and fs' we see how the output from the executed program is directed either to the screen or to a file in the file store. put(fs,n,c) generates a new file store state from fs in which the file named n has contents c, and any previous file named n is inaccessible.

Various simple functions have been used and must now be defined.

For extracting data files from the keyboard stream:

```
    untilend(kb)  ≡ if head(kb) = "end" then "NIL"
                    else cons(head(kb),untilend(tail(kb)))

    afterend(kb)  ≡ if head(kb) = "end" then tail(kb) else afterend(tail(kb))
```

Fetching files from the file store, and updating the file store:

```
    get(fs,n)     ≡ if fs = "NIL" then "MISSING" else
                    if head(head(fs)) = n then tail(head(fs))
                    else get(tail(fs),n)

    put(fs,n,c)   ≡ if fs = "NIL" then cons(cons(n,c),"NIL") else
                    if head(head(fs)) = n then cons(cons(n,c),tail(fs))
                    else cons(head(fs),put(tail(fs),n,c))
```

Note that put(fs,n,c) has no side effect. Instead it simply recreates as much of the association list as is necessary and returns the new list.

With the addition of the trivial standard functions which are missing, this completes the design of OS7, with exception of the invocation of the system function to connect the input and output streams. When invoking the system function it must be supplied with an initial file store. One possibility is to build a quoted constant file store into the program:

```
    λ(kb).system(kb,initialfs)
    whererec
          initialfs     ≡ "(...   ...)"
          system(kb,fs) ≡ ...
          ...
```

Alternatively the "input stream redirection" facility, of the Lispkit system described in [11], can be exploited in order to read an initial file store as the first s-expression of the input stream:

```
    λ(kb).system(tail(kb),head(kb))
    whererec
          system(kb,fs) ≡ ...
          ...
```

Using OS7

There is now the small matter of deciding what should be placed in an initial file store, and looking at example programs for execution under OS7.

The first point to notice is that OS7 is not capable of obeying any command unless the program named in that command is already present in the file store. Thus if the file store starts empty then it must always remain so. A minimal initial file store will contain a program which will enter keyboard input into the file store - this is the bootstrap which will enable us to program anything that is possible with OS7.

A suitable bootstrap program to include in the file store is the program:

```
    λ(kb).head(kb)
```

or, in the concrete s-expression syntax required by OS7:

```
    (lambda(kb)(head kb))
```

and the initial file store containing it as a file named put:

```
    ((put.(lambda(kb)(head kb))))
```

When this program is executed by the command

```
    (put kb newname)
```

followed by keyboard input

```
    (contents of a new file)
    end
```

the program put is applied to the single argument

```
    ((contents of a new file))
```

The output from the program is the head of the input list

```
    (contents of a new file)
```

which is then placed in a file called newname.

A complementary program, called get, could be added to the initial file store, or added to the running system by executing put. get is:

```
    λ(f).cons(f,"NIL")
```

When executed by a command of the form

```
    (get filename screen)
```

the program get will fetch the named file and will send the contents of the file to the screen in a list of items containing only the file contents.

Note that put expects to receive a list of items (of which it only processes the first), and get produces a list of items. Hence they are only suitable for their designed roles as transfers between the terminal and file store.

Other example programs:

To copy one file into another:

```
    λ(f).f                          (the identity function)
```

Example execution:

```
    (copy get newget)
```

copy can also be used to echo the keyboard to the screen until end is typed:
(Lines with * are typed by the user)

```
    (copy kb screen)               *
    abc                            *
    abc
    1 2 3 4                        *
    1 2 3 4
    (x y z) end                    *
    (x y z)

    next command ...
```

A program, intsfromto, which takes two numbers from its input argument and produces the list of numbers from the first to the second:

```
    λ(in).fromto(head(in),head(tail(in)))
    whererec
         fromto(m,n) ≡ if m=n then cons(m,"NIL")
                       else cons(m,fromto(m+1,n))
```

Hence generating numbers on the screen from keyboard input:

```
    (intsfromto kb screen)         *
    1 5 end                        *
    1 2 3 4 5
```

Reading the input from a file, sending the list to a file, then displaying the result file on the screen:

```
    (put kb data)                  *
    (1 5) end                      *
    (intsfromto data result)       *
    (get result screen)            *
    (1 2 3 4 5)
```

(Notice the difference between dumping the result list directly to the screen (which could be simulated by (copy result screen)), and "getting" the result file, where get is careful to preserve the parentheses!)

An interesting effect in the last example, which cannot be observed in this written presentation, is that the lazy evaluation causes the generation of the list of integers to be delayed until it is required to be displayed on the screen. However large its result file, intsfromto will apparently generate that file in negligible time. The price is paid, though, when the file is displayed since its contents will be computed at that time. Fortunately, the lazy evaluation mechanism ensures that subsequent accesses to the result file find that it has already been evaluated, and it will be displayed "instantly". Note that none of this subtlety has been coded explicitly into the operating system - it is a natural consequence of lazy evaluation.

This effect is clearly related to Unix "pipes". The example is a pipeline with three program stages and two pipes (the files data and result). Taking the second pipe only, the result file is a temporary file between intsfromto and get, and these two programs will execute in co-routine fashion when the output from get is displayed. In fact OS7 implements a generalisation of pipes since <u>every</u> result file from <u>any</u> program execution is delayed and may subsequently become the intermediate file in the pipe between executing programs. Hence pipelines do not need to be entered as consecutive commands. It is easy to see that the command processing in system and systemstep could be modified to recognise a special syntax for a pipeline, and could expand such a command to a series of normal commands, with no alteration to the underlying execution mechanism. For example, a pipeline could be entered (somewhat unoriginally) as

```
     (put kb | intsfromto | get screen)
```

and could be expanded to the sequence of commands

```
     (put kb temp1)
     (intsfromto temp1 temp2)
     (get temp2 screen)
```

In this way it should be possible to implement much more sophisticated "shell" languages, such as that described by Shultis [12].

Interactive programs may also be written to run under OS7, using the familiar "stream in/stream out" paradigm. The only precaution to be taken is to remember that the input will be a finite list if "end" is typed, and the program should take care to terminate its output list cleanly when the input stream is exhausted.

Remarks

Despite the remarkable simplicity of OS7, it nevertheless provides the basis for an extremely powerful operating system. OS7 has been fully implemented and will execute on the "distribution" Lispkit system [11]. In its own limited way it is quite satisfying to use.

Without introducing undue complexity, a number of significant improvements can be made to OS7:

The system can be altered to execute programs which are present in the file store as compiled SECD machine object code. If the object code is a closure for a function of one argument then the interpreter invoked by execute(f,a) can be dispensed with and function application used instead:

```
    execute(f,a) ≡ f(a)
```

In this case the minimal file store must contain the object code for the put program. The object code for a compiler, which can convert a Lispkit source file into a file containing the object code as a closure, would be an essential requirement and could usefully be included in the initial file store. Advantages gained from this would be the full speed execution of programs, and a smaller operating system (since now a part of its function, the interpreter, has been moved into the file store as the compiler).

The input/output interface for each executing program may be extended to allow multiple input files and multiple output files. Thus each program would be a function of several arguments, producing several results. For example, an editor program could be written which processes a sequence of edit commands and a file to produce a sequence of responses and an output file. So to edit file1, giving commands at the keyboard, receiving responses on the screen, and placing the edited file in file2 the following command would be entered:

```
    (edit (kb file1) (screen file2))
```

Alternatively, the input/output interface could be altered more drastically in order to give a program control over its own file input and output. One scheme to achieve this is to construct programs so that they generate a single output sequence of requests for operating system services (such as "get this file", "get that file", "put this in this file", "fetch keyboard input", "send this to the screen", "get a directory of the file store", "delete this file", and so on), and receive as input a single sequence of replies to those requests for system services which need replies. The operating system then services the requests, replying to the program, reading keyboard input, sending output to the screen and updating the file store as necessary:

```
    requests ≡ execute (prog,replies)

     scr,kb',fs',replies  ≡  servicerequests(requests,kb,fs)
```

(where the tuple "scr,kb',fs',replies" indicates that servicerequests returns a compound result - probably a four-list).

Each of these extensions has been implemented (yielding various operating systems OS8 - OS11) - providing a range of experimental operating systems, each quite rewarding to play with. Unfortunately there is no room to explore each of them in detail here.

A system with each of the extensions incorporated is very powerful. In such a system it becomes possible to see how the entire command interpretation strategy and program execution mechanism could itself be placed in a program file in the file store. Thus the core of the operating system itself would become even smaller and simpler. This would be a move very close to the Unix principle of a "shell" program. This is a challenge remaining for the future, as it has not yet been implemented.

Chapter 3

NETWORKS OF PROCESSES

Introduction

A function which consumes one or more input streams and produces one or more streams as output is referred to as a process. Such a process can execute autonomously, independently of any other processes, in the sense that it is constrained only by the demand for values on its output stream(s) and the availability of values on its input stream(s).

Thus the interactive doubling function given in the introduction to Chapter Two is a single process.

OS7 is also constructed as a single, monolithic process. Although OS7 is broken down into several distinct groups of functions (the system kernel, the file store and the program execution mechanism), nevertheless there are only two discernible streams in the system (the keyboard input and the screen output) and the system kernel has explicit control of the file store data structure and of the submission of programs to be executed.

The design of complex systems can be simplified if major subsystems are implemented as autonomous processes. Then a further modularisation step has been achieved - not only will the functions handling the filestore have been isolated syntactically from the rest of the system, but they will also execute as a unit, without interference or external control. The subsystem processes will be linked to form a static network by associating the output streams of processes with the input streams of other processes - and the processes will communicate their requirements to each other by the transmission of messages in the streams.

The purpose of this chapter is to explore some systems of this kind.

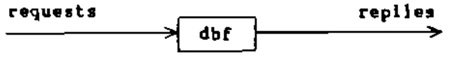
A major benefit that results from this style of modular design is that the individual processes (modules) can be reused in different networks, with only minimal reprogramming being necessary within each process (for example to extend message vocabularies, or to adjust message formats). The only significant difference between networks (in terms of the program text) would be in the program "glue" specifying the network linkage - this can all be given within the basic functional programming framework.

A simple determinate system:

As we shall see, by using non-determinate components very powerful systems can be constructed. However the general principles of networks of processes can possibly be seen best in a simple determinate network. The example to be presented here is a determinate version of "sys5" in [5].

The system comprises two interacting components, a database which acts as a file store for the system, and an editor which controls operation of the system.

The database is a simple process with a single input stream composed of messages requesting database actions, and a single output stream composed of replies to the requests.



Requests are of two forms:

(GET f)	in which case the reply is the contents of file f, or "MISSING" if there is no such file.
(PUT f s)	in which case file f is updated to have contents s, and the reply is "DONE".

dbf is easily programmed as a recursive function from a single input stream of requests c to a single output stream:

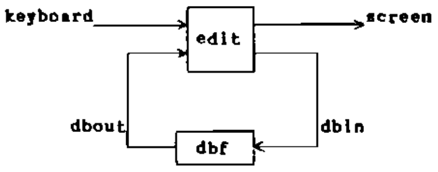
```
dbf(c)     ≡    dbf1(c,"NIL")    where "NIL" represents an initially empty
                                          database

dbf1(c,db) ≡    cons(m,dbf1(tail(c),db'))
                       where m,db' ≡ dbstep(head(c),db)
```

dbstep interprets the different requests, generating a reply and new database, as appropriate:

```
dbstep(("PUT" f s),db)  ≡  "DONE",put(db,f,s)
dbstep(("GET" f),db)    ≡  get(db,f),db
```

where we can adopt the definitions of put and get as applied to filestores in OS7.

The editor process accepts two input streams, commands from the user's keyboard and replies from the database, and generates two output streams, responses to the user's screen and requests to the database. Hence the overall system structure is



Given definitions of the process components edit and db, the whole system may be constructed and executed by the following definition:

```
    system(keyboard) ≡ screen
    whererec screen, dbin ≡ edit(keyboard,dbout)
             dbout  ≡  dbf(dbin)
```

This simply describes the linkage of the streams keyboard, screen, dbin, dbout between the processes.

It remains to define the edit process function.

We imagine the editor to maintain a copy of the file that is currently being edited. The editor will recognise a simple set of commands from the keyboard, which is adequate for the purpose of illustration:

(GET f)	Replace the current file with the contents of file f from the database.
(PUT f)	Place the current file in the database with name f.
(CHANGE t1 t2)	Edit the current file in some way determined by t1 and t2.
PRINT	Display the contents of the current file on the screen.

Again, edit may be programmed easily as a recursive function; starting the editor with an initial current file "NIL":

```
    edit(kb,dbin) ≡ editstep(kb,dbin,"NIL")
```

editstep(c,dbin,x) must decode commands that arrive on c from the keyboard, generate responses on the two output streams, and edit the current file x where necessary. There are four cases. The first deals with fetching a new file to be edited:

```
    editstep((("GET"f).c),(x'.dbin),x) ≡
                          cons("NEWFILE",screen),cons(("GET" f),dbout)
                          where screen,dbout ≡ editstep(c,dbin,x')
```

This requires some explanation. The first command to be executed is (GET f), and so the first request to be output to the database is (GET f). Since db works in a synchronous fashion (exactly one reply for each request) the first item on the input stream from the database is thus the file contents requested, x'. The response NEWFILE is sent to the screen to indicate that the file has been fetched. To determine the continuation of the screen and dbout streams, editstep is called recursively to execute the subsequent commands c, with subsequent replies from the database dbin, and the new current file x.

The availability of the new file contents x' in the current recursive step seems somewhat miraculous, but it is quite valid and healthy, and is a natural consequence of lazy evaluation!

The remaining cases are similar:

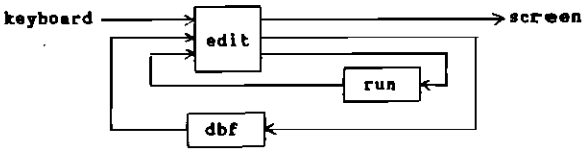
```
    editstep((("PUT" f).c),("DONE".dbin),x) ≡
                              cons("DONE",screen),cons(("PUT" f x),dbout)
                              where screen,dbout ≡ editstep(c,dbin,x)

    editstep((("CHANGE"t1 t2).c),dbio,x) ≡
                              editstep(c,dbin,change(x,t1,t2))
    (in this case no screen response is generated)

    editstep(("PRINT".c),dbin,x) ≡
                              cons(x,screen),dbout
                              where screen,dbout ≡ editstep(c,dbin,x)
```

Details of the editing function "change" are not required here. With this exception, we have a complete program for the editor system.

Here is an example session with the system (lines marked with an asterisk are typed by the user):

```
    (system starts)
    (GET FRED)                                              *
    NEWFILE
    PRINT                                                   *
    MISSING
    (CHANGE x (HELLO FRED))                                 *
    PRINT                                                   *
    (HELLO FRED)
    (PUT FRED)                                              *
    DONE
    .
    .
    .
```

If we wish to introduce a program execution facility into the system then the following network is a simple solution:



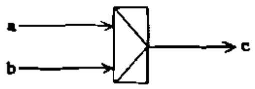
The editor component now has three input and three output streams. One pair is used to transmit the current file to the run component when the command RUN is entered at the keyboard, and to receive the result of evaluating the program (it could replace the current file in the editor).

This system is straightforward to construct. However if we wish the result of program evaluation to be entered directly into the database then a more intricate solution is required - since the database must be prepared to receive its next input from either the editor or run component. In this particular system it is possible to solve the problem by placing a switch on the input to the database, and operating the switch by a separate stream of control signals from the editor. This is essentially the use of an "oracle", to be discussed shortly, and in general becomes too intricate or impossible.

Sharing, asynchronous activity and non-determinacy

The simple network systems exhibited so far execute in a purely synchronous fashion, although there is certainly scope for some parallel evaluation given a machine of appropriate architecture. However, in more advanced systems it is desirable to be able to express that certain computations do not need to wait for completion of other computations. For example, where several programs are being evaluated concurrently, we may wish that the file store is updated with the results in the order that the evaluations are completed. For this, some kind of non-determinate program component is required.

In [5] Henderson proposes the use of a non-determinate stream interleaving operator, which I shall refer to as "merge". In a network of processes the merge operator is represented as follows



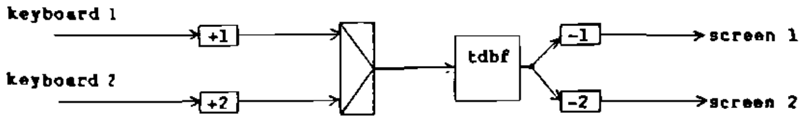
and we would write

$c \equiv merge(a,b)$

The intended meaning of this equation is that the stream c comprises all the elements of streams a and b, with the ordering of elements from a preserved, and the ordering of elements from b preserved, but with the elements of a and b interleaved in some arbitrary fashion. Operationally, we might expect that the interleaving of elements is in some way determined by the availability of the elements; the availability would itself by determined by, for example, the arrival of a value from an external device such as a keyboard, or by the completion of the computation of a value.

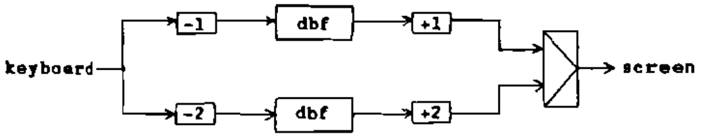
Leaving aside, for the moment, the implementation of merge, Henderson shows how a number of different systems may be constructed where independent users may access shared databases concurrently. Three examples follow:

"sys2"



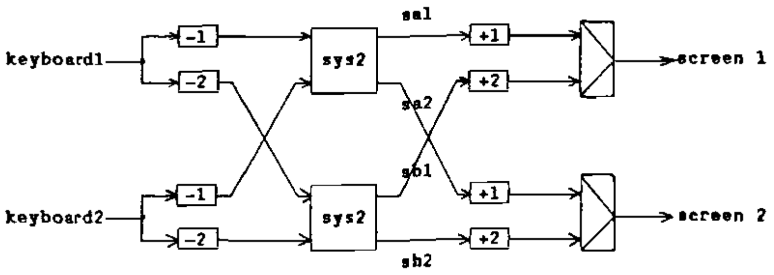
Here a single database is shared by two users, each with their own keyboard and screen. The process $\xrightarrow{}\boxed{+n}\xrightarrow{}$ tags each message on a stream with the number n. The process $\xrightarrow{}\boxed{-n}\xrightarrow{}$ passes only those messages with tag n, and removes the tag. tdbf is like dbf, but it tags each reply to a request with the tag from the request. The commands from each keyboard are passed to the database as they are available, and each user sees their own replies. Of course, since the database is shared, as in a conventional shared filestore, each user will observe changes to the database as the result of the other user's activity at the keyboard.

"sys3"



Here a single user may make simultaneous use of two independent databases - the user must tag commands appropriately to indicate which database is to be used. Replies will appear on the screen as they become available, and hence short requests sent to one of the databases might easily overtake long requests sent to the other database.

"sys4"



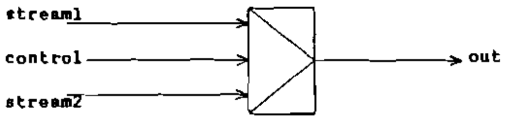
Here two independent users share two independent databases. The system contains two embedded copies of sys2. Each user may issue concurrent requests to both databases, and each database merges the requests from each user. The equations for this network are easy to write down. In [5] Henderson gives the following system definition (with minor adjustment of tagging conventions):

```
    sys4(keyboard1,keyboard2) ≡ screen1,screen2
         whererec  screen1 ≡ merge(tag(1,sa1),tag(2,sb1))
                   screen2 ≡ merge(tag(1,sa2),tag(2,sb2))
                   sa1,sa2 ≡ sys2(untag(1,keyboard1),untag(1,keyboard2))
                   sb1,sb2 ≡ sys2(untag(2,keyboard1),untag(2,keyboard2))
```

We have seen how some simple systems exhibiting sharing and asynchronous activity can be constructed using the non-determinate merge operator. But how is merge to be implemented? There are various approaches. Abramsky [1a] has chosen to make merge a primitive operator in his functional programming language - this places the burden of implementation entirely on the systems programmer, who can ensure that the input streams are evaluated and inspected concurrently, and also can ensure that a "fair" choice is made between elements of the input streams. Friedman and Wise [13] introduce a non-determinate list construction operator, "frons", which may be used to program merge and other operations. I chose to look briefly at the prospects for using "oracles", since this did not require any language extensions, and then at more length at the provision of an "ambiguous choice operator" (McCarthy [14]) which, again, can be used to program merge and other operations.

Oracles

As hinted in a previous section, in some cases it is possible to supply to merge a stream of control signals indicating from which input stream the next transmitted element must be taken. Hence we may define an "oracular merge" operator which has three stream inputs:

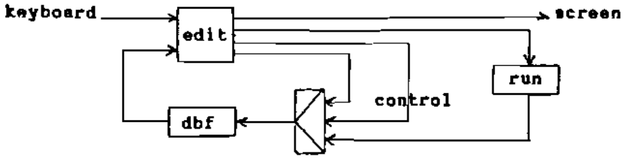


If an element "1" on the control stream indicates that the next element must come from stream 1, and "2" indicates stream 2, then we have the following simple recursive definition:

```
    merge(s1,s2,c) ≡ if head(c) = 1
                       then cons(head(s1),merge(tail(s1),s2,tail(c)))
                       else cons(head(s2),merge(s1,tail(s2),tail(c)))
```

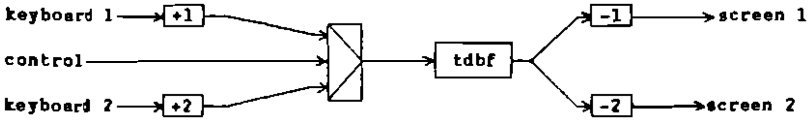
The remaining problem is then the origin of the control signal stream.

In the case of the editor system proposed earlier, where both the edit and run components needed to send requests to the database, then the edit component itself could generate the control signals to route the correct next request to the database:

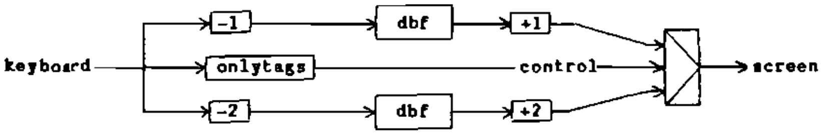


Where we are trying to merge stream elements from two keyboards, then we need to assume the existance of external hardware which generates an input stream of oracle signals by observing, for example, the key depressions at the two keyboards.

Thus "sys2" simply uses the external oracle to control interleaving:

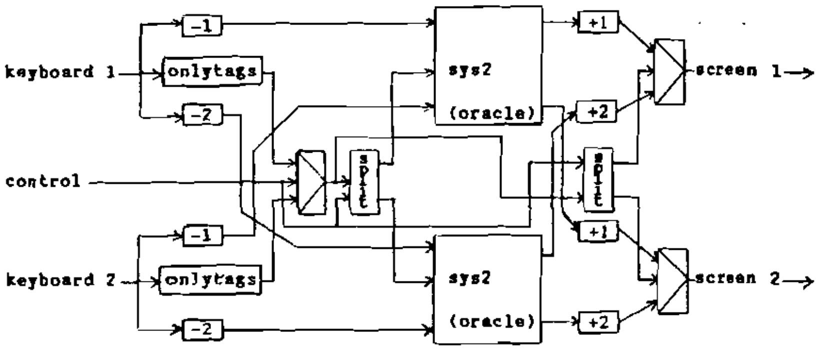


To employ the oracular merge in "sys3", the control signals can only be generated by inspection of the tags typed by the user indicating which database is to be used:

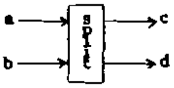


Here the keyboard is passed through a process, "onlytags", which discards the database request part of each message. These residual tags are used as the oracle signal controlling the merge operator. This gives a functioning system, but unfortunately the interesting asynchronous behaviour, whereby requests passed to one database may overtake requests passed to the other, has been eliminated from the system. The effect of generating oracle signals in this way is to produce a strictly synchronous system (though in principle it is still possible for both databases to be active concurrently). Since the entire system is programmed in a determinate functional programming language this is hardly surprising.

Similarly "sys4" can be implemented using oracular merge, but it took considerable effort to design the following solution:



Where "split" is a special purpose switching process:



Each element of b is sent to either c or d. It is sent to c if the corresponding element of a is 1, and to d if the corresponding element is 2:

```
    split((1.a),(x.b)) ≡ cons(x,c),d
                         where c,d ≡ split(a,h)

    split((2.a),(x.b)) ≡ c,cons(x,d)
                         where c,d ≡ split(a,b)
```

This implementation of sys4 is very obscure, and is very good evidence that the topic of explicitly programmed oracles should not be pursued any further.

<u>The ambiguous choice operator</u>

In [14] McCarthy discusses the properties of a new primitive operation <u>amb</u>, in order to allow some controlled measure of non-determinacy into a functional programming language. In the work reported in [6] I adopted this operator, renaming it <u>or</u> (a practice since dropped), as a suitable tool for experimenting with non-determinacy in functional operating systems. Details of the implementation of <u>or</u> are given in [6]. <u>or</u> is a binary operator, and an expression of the form

    e1 <u>or</u> e2

takes the value either of e1 or of e2. The expressions e1 and e2 are in fact evaluated concurrently (in a time sliced fashion), and the first to return a value determines the choice of value for the overall expression. If either expression computes indefinitely, or waits for some external input, then the computation of the other expression is not held up. This is a simple choice between two values based on the availability of those values. It is a committed choice, and involves no backtracking. We can use <u>or</u> to program the stream merging operator, but some care is needed to obtain an operationally correct solution. Various solutions are considered in [6]. For the purposes of this report I shall assume the following definition of merge:

```
merge(S1,S2) ≡ alt1 or alt2
        where alt1 ≡ if here(head(S1))
                     then cons(head(S1),merge(tail(S1),S2))
                     else UNDEFINED

              alt2 ≡ if here(head(S2))
                     then cons(head(S2),merge(S1,tail(S2)))
                     else UNDEFINED
```

where "here" is some test of the availability of a stream element, for example

```
here(x) ≡ if atom(x) then true else true
```

Note that this definition of merge does nothing to guarantee fairness - either stream might accidentally be ignored indefinitely. However, neither is it intrinsically unfair, and in practice turns out to behave quite well with the implementation of <u>or</u> given in [6].

Note also that the indiscriminate use of <u>or</u> can lead to great confusion, as the properties of functional programs become less flexible, and programs must be rearranged and transformed with care. For example, the expression

```
if x=x then YES else NO
where x = e
```

will always yield the result YES, for any expression e, since e is evaluated once and its value is bound to x. On the other hand, if we substitute e through for occurrences of x

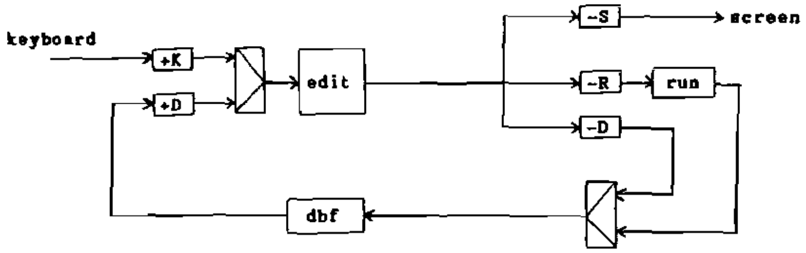
```
if e = e then YES else NO
```

then the result could also be NO if e contains any non-determinate components.

Thus we now have a technique for implementing the non-determinate merging of streams of messages in a network of processes. The shared and asynchronous database systems operate as desired with this technique of merging, with no other special programming.

We can now return to the editor/database/run component systems discussed earlier, and look briefly at the use of non-determinacy in the asynchronous network given by Henderson as "sys6" in [5]:

"sys6"



The editor component now has only one input stream and one output stream. The input stream is an interleaving of the commands entered at the keyboard and replies from the database - each tagged suitably with K or D to identify their origin in order that the editor may handle them appropriately. The editor generates a single stream containing messages for all destinations - again they are tagged by the editor and the network directs each message to its appropriate destination.

Messages passing directly from the editor to the database, and database messages resulting from program evaluations are merged into a single stream of requests to the database. Hence the editor can make unrestricted access to the database while a program is being evaluated (the user may have to exercise restraint in inspecting the database in the hope of finding the results of the evaluation!)

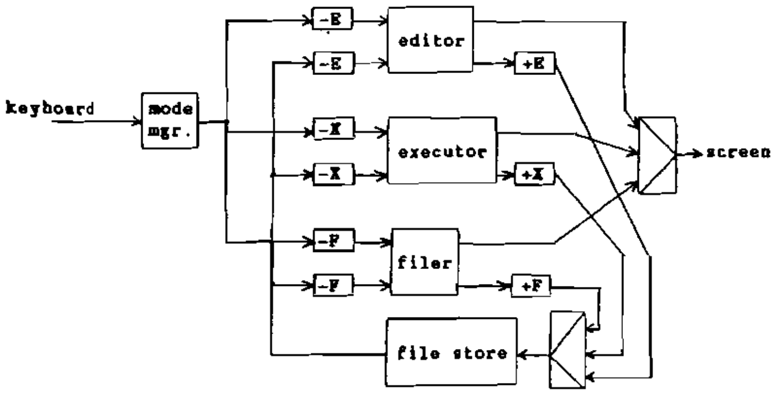
The merging of keyboard and database streams, before input to the editor, enables the user to continue issuing commands to the editor, and to have them acted upon, while activity is occurring asynchronously in the run and/or database components. Again the user will have to exercise restraint, since there will be a period, after requesting a new file to edit and before that file has been noted by the editor, during which any edit commands entered will be applied to the old file!

"sys6" is thus one step towards the design of a powerful single user operating system. The system is an integrated network of functional components (in both senses) which can absorb the workload provided by the user through asynchronous activity.

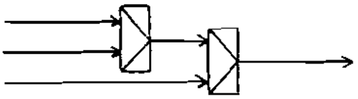
Without delving any more deeply into the programming details, it is now possible to imagine a great range of more general operating systems. Each system will be a network of asynchronously operating component processes. I will show three possible system structures. In each case a user at a keyboard and screen maintains simultaneous conversations with three subsystems - an editor, a program executor, and a "filer" (not the file store, but a component specialising in listing, renaming, copying, deleting files, and so on). The user enters commands sometimes to one subsystem and sometimes to another, switching the conversation by explicit commands to a "mode manager" - this is exactly analogous to the use of windows and a window manager at a personal workstation.

The mode manager keeps track of the intended destination of keyboard commands, and tags messages appropriately to ensure their correct routing in the network. The editor can store and retrieve files. The program executor fetches program and data files, and stores result files. Similarly the filer must have access to the file store. Thus a quite general communications network is required.

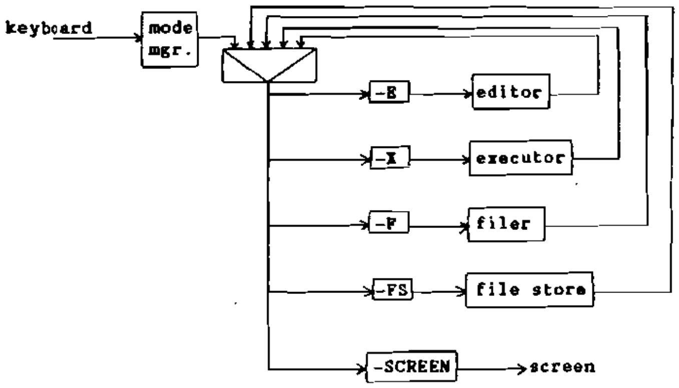
The first system provides an independent path through each subsystem from the keyboard to the screen, and each subsystem has a largely distinct path to and from the filestore:



In this system I have taken the liberty of using the merge symbol to merge 3 streams - this is easily implemented by cascading 2-way merge operators:

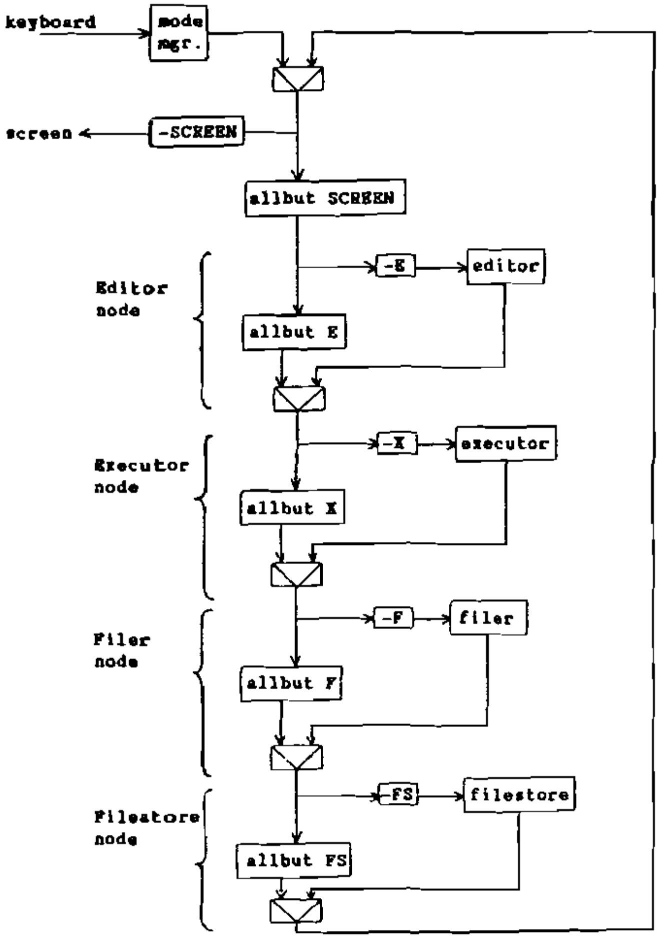


A second possible system uses a single centralised "bus" to distribute all messages from source to destination. In this case we also need tags to identify the file store and screen as destinations:



Here a more sophisticated tagging system is required. The main tag on a message must indicate the destination of the message. In addition however, the message part must contain the tag of the sending component, to be used as the "return address" tag on any reply evoked from the destination component. This is a typical communications problem, and it is not a fundamental issue here. This system also enables a more flexible communications strategy, if it is required. For example, the editor could make use of services provided by the filer.

The final offering in this chapter is a rearrangement of the components of the previous system. The new system has a ring architecture, somewhat reminiscent of the Cambridge Ring. Each component is attached at a node on the ring - passing messages with the correct tag are extracted from the ring, and any responses are merged with the residual message stream:



where allbut x eliminates those messages tagged with x, but passes all others unchanged.

Chapter 4

PERIPHERAL HARDWARE AND DISTRIBUTED SYSTEMS

Introduction

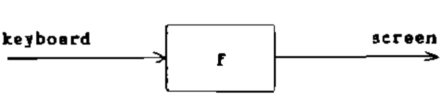
In the previous chapter we were concerned simply with the logical architecture of operating systems conceived as networks of communicating processes. It was implicitly assumed that, if required, each entire network could be run as a single functional program (which is precisely what it is) on a single processor. The only external services required would be the provision of an interface connecting the single input and output streams to the user's keyboard and screen. Where a system required a file store we implicitly simulated this by using the main memory of the processor.

Clearly, more realistic operating systems must also make some provision for access to the wide range of peripheral devices which may be connected to computers. In particular I would like to consider access to external file stores, and communication between computers via remote lines. The former is necessary for systems with large storage capacity, and the latter will enable the construction of true distributed systems in which networks of processes are also networks of processors.

The standard Lispkit machine, described in [11], provides an abstract machine for the execution of Lispkit programs in which the programmer's view of the physical world is restricted to a single input stream of s-expressions and a single output stream of s-expressions. Usually these streams are interfaced to a terminal by the abstract machine implementation. It is possible for the streams to be temporarily reconnected to files held on disk, but this is outside the logical architecture of the Lispkit abstract machine. The reconnection cannot be controlled by an executing Lispkit program since it is entirely dependent on the user typing special control codes.

In [6] the abstract Lispkit machine is extended to support many physical devices, each being associated with its own stream(s) of input to and output from the machine. Thus the user's terminal is still supported by one input and one output stream, a printer or graph plotter could be supported by a single output stream each, and each remote port provided by the hardware can be supported by one input and one output stream (for received data and transmitted data, respectively). A file store can similarly be supported by a single output stream (carrying command messages to a disk based file store driver) and a single input stream (receiving replies to commands where required). This view of a filestore corresponds more or less precisely to the re-implementation of dbf as a process running outside the Lispkit machine, and storing data on a disk in a conventional way. In fact the semantic properties of both file store implementations are identical as far as any application program is concerned.

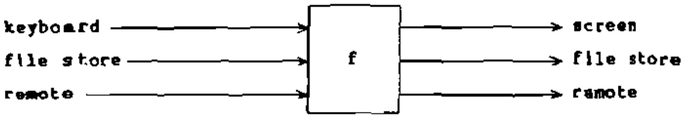
The standard machine has a simple logical interface:



The *program* to be executed is a function f which must be defined as mapping a single argument stream to a single result stream:

$$f(\mathit{instream}) \equiv \text{outstream} \quad \underline{\text{where}} \ldots$$

The extended multi-stream machine simply generalises this interface:



and the program function *f* now must be defined as mapping a single argument which is a tuple (or list) of input streams, to a single result which is a tuple of output streams:

$$f(\langle\text{keyboard},\text{fstorein},\text{remin}\rangle) \equiv \langle\text{screen},\text{fstoreout},\text{remout}\rangle \quad \underline{\text{where}} \ldots$$

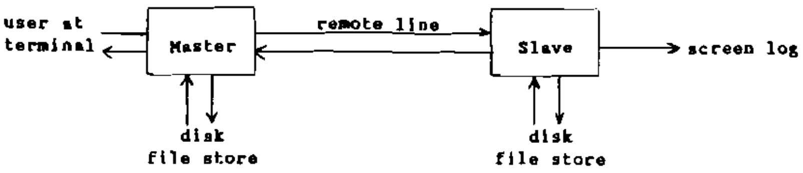
It is the job of the implementer of any particular instance of the abstract multi-stream machine to establish a correspondence between physical devices and particular streams in the input and output tuples. Each program must respect the correspondence for the machine on which it is to execute. There is no restriction in the logical architecture on the number of devices which may be represented in the tuples - there may be any number of terminals, file stores, remote lines, etc. There need not be a one to one pairing of input and output streams - for example joysticks or digitising tablets could be supported by unpaired input streams, and printers and plotters by unpaired output streams.

Further details of the logical properties and implementation of file store and remote line interfaces are contained in [6].

Two example systems follow. The first is a file transfer system between two separate computers connected by a single remote line - this illustrates the use of external file stores and remote line communication. The second example is another single user network operating system - this will be used to illustrate the physical distribution of operating system components over a network of computers, and the ease of reconfiguring such a system with changes in hardware availability.

<u>A simple file transfer system</u>

The hardware configuration that we will be working with in the design of this system comprises two processors each with a terminal and disk based file store, and connected by a single remote line cable. One of the processors will be designated as the Master, and all requests for file transfers will be entered at its keyboard. The Master controls all file transfers. The other processor is designated as the Slave, its terminal is used only to log the activity of the slave machine, and hence the keyboard is ignored entirely. The Slave carries out actions in response to requests from the Master.



The user can enter two types of command:

(SEND A B)	"Copy the contents of file named A on the Master machine to a file named B on the Slave machine"
(RECEIVE A B)	"Copy the contents of file named A on the Slave machine to a file named B on the Master machine"

A very simple strategy is to implement the Slave so thet it acts as a transparent communication channel between the Master and the Slave's disk. The Master is than implemented as if it had direct access to two file stores. Thus messages arriving et the Slave along tbe remote line are directed immediately to the disk, and replies from the disk are sent on the remote line. The screen log is obtained by monitoring the messages which pass from the Master to the file store. The functional program to be executed on the Slave machine is easily designed:

```
Slave(<keyboard,filein,remin>) ≡ <screen,fileout,remout>
        wharerec    fileout ≡ remin
                    remout  ≡ filein
                    screen  ≡ cons(heading,monitor(remin))
                    heading ≡ "(File transfer slave)"
```

The function "monitor" cannot be implemented without knowledge of the format of messages passed from the Master to the Slave's file store. Requests to the file store have exactly the same form as requests to dbf in the previous chapter.

(GET A)	"fetch contents of file A"
(PUT B C)	"write a file B with contents C"

A simple log would note the action and file name from each request:

```
monitor((message.more)) ≡
     cons(list(head(message),head(tail(message))),
          monitor(more))
```

so the sequence of requests

```
(GET fred)(GET bill)(PUT joe(new contents))
```

will be logged on the screen as

```
(GET fred)(GET bill)(PUT joe)
```

Hence we require that the Master generates file store requests for both the Master machine and the Slave machine.

To satisfy a send command (SEND A B) the Master must issue the request (GET A) to its local file store, obtain the contents C as reply (which will be the atom MISSING if there is no such file), and transmit the request (PUT B C) to the Slave. To satisfy a receive command (RECEIVE A B) the Master must transmit the request (GET A) to the Slave, obtain the contents C as the reply on the remote line, and issue the request (PUT B C) to the local file store.

Now we can write down the recursive function defining the Master program, including a heading line for the screen and a trap for invalid commands from the keyboard:

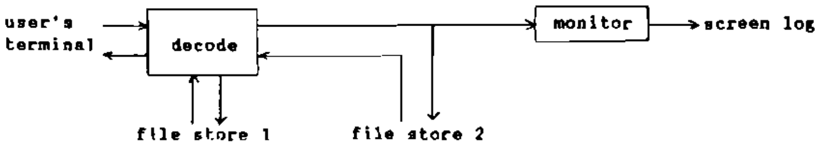
```
    Master(<keyboard,filein,remin>)  ≡
           <cons(heading,screen),fileout,remout>
            where  screen,fileout,remout ≡ decode(keyboard,filein,remin)
                   heading ≡ "(File transfer master)"


    decode((("SEND" a b).commands),(c.filein),remin)  ≡
             errors,cons(("GET"a),fileout),cons(("PUT"b c),remout)
             where errors,fileout,remout ≡ decode(commands,filein,remin)

    decode((("RECEIVE" a b).commands),filein,(c.remin))  ≡
             errors,cons(("PUT" b c),fileout),cons((GET a),remout)
             where     errors,fileout,remout ≡ decode(commands,filein,remin)


    decode((x.commands),filein,remin)  ≡
             cons("ERROR",errors),fileout,remout
             where     errors,fileout,remout ≡ decode(commands,filein,remin)
```

Note that, as implemented in [6], external file stores do not return any response to a PUT request (unlike dbf which returns DONE). In common with the programming of the editor component in the previous chapter, the cases for SEND and RECEIVE both assume that the requested file contents from local or remote disks are available in the same recursion step. Again this is quite healthy. Conventions on the representation of file names have not been considered here, though clearly some specific decision will be required in practice.

Thus the two parts of the file transfer system have been represented quite easily as recursive functional programs. When loaded onto suitable multi-stream abstract Lispkit machines, connected by a remote line cable, a rudimentary file transfer system is available.

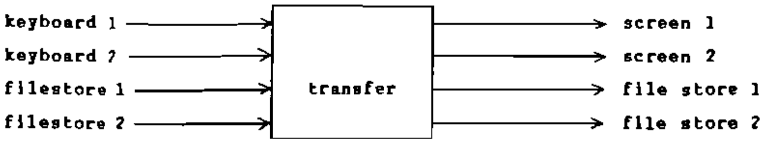
A note on methodology

In the design of the file transfer system above, the precise configuration of the hardware was almost irrelevant. The system is conceived as essntially a single functional program for the transfer of files from one disk to another, taking commands from one terminal and displaying a log of activity on the second disk on the screen of a second terminal.



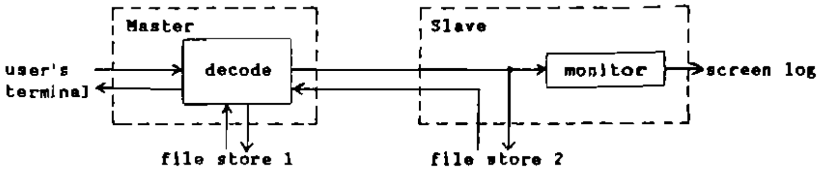
This diagram shows the logical architecture of the file transfer system.

Given a single computer with two terminals and two disks, this logical architecture can easily be configured to run on the single physical machine. For example:



```
    transfer(<kb1,kb2,fsin1,fsin2>)  ≡
         <cons(heading1,scr1),cons(heading2,scr2);fsout1,fsout2>
         wherereec   scr1,fsout1,fsout2 ≡ decode(kb,fsin1,fsin2)
                     scr2 ≡ monitor(fsout2)
                     heading1 ≡ "(File transfer master)"
                     heading2 ≡ "(File transfer slave)"
```

Alternatively we could notice that the logical architecture can be partitioned to give two subsystems, each of which could be configured to run on a single machine. The criterion for partitioning the system is to discover subsystems connected to each other by streams (preferably a forward and reverse pair) which can correspond to remote lines in a physically distributed system. Thus the logical architecture of the file transfer system can be partitioned into the Master and Slave subsystems as implemented earlier:



A distributed operating system

Now we can proceed to the design of a general purpose, single user, distributed operating system. The design will be described in two stages. Firstly we shall consider a network of processes which implement the desired facilities. Secondly, we shall consider configuring the logical system architecture to various physical architectures.

Here is a verbal specification for the system:

The system should provide facilities for storing program files (source code and compiled code) and data files, editing those files, executing compiled program files with given data files and placing resultant files in permanent storage, and pretty printing files to some output device. The user will be able to issue commands of three types:

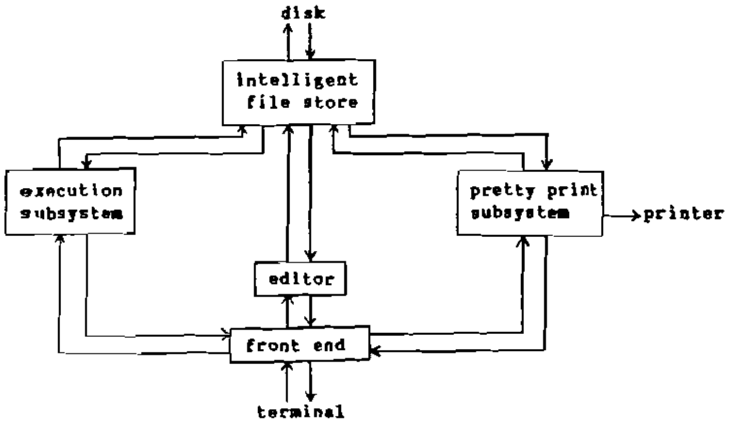
(EXEC A B C)	"Execute object code program in file A, taking contents of file B as data, and placing result in file C"
(PRETTY A)	"Pretty print the contents of file A"

Any atom, or list where first element is not EXEC or PRETTY:

"Perform an edit operation, possibly a transfer from file store to editor or vice versa"

In addition we would like, firstly, that program source files be compiled automatically as they enter the file store, and secondly, that actions invoked by the three classes of command take place concurrently, as far as possible. A file store which ensures that all source programs also exist in an up-to-date compiled form, is one example of an "intelligent" subsystem which automatically performs important housekeeping duties, etc on the user's behalf.

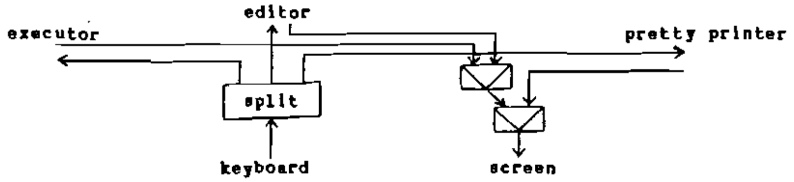
The diagram that follows shows the logical network architecture of one possible solution to the problem specification:



Input from the terminal passes immediately to a front end process which directs each command to the appropriate subsystem. The front end also collects replies from each subsystem and displays them on the screen - this involves a non-determinate merge of three streams of messages. While any one or more of the three command processing subsystems is unoccupied and ready to receive a command, the front end process will be ready to accept keyboard input. Any input for a subsystem that is currently occupied will be queued - this happens incidentally through the properties of streams and lazy evaluation. Note that, individually, the executor, editor, and pretty printer are not required to service many commands concurrently, (and in the implementations which follow, they will not do so), and hence queuing will occur - on the other hand, if these subsystems were re-implemented in such a way as to provide concurrent service then the queues would disappear without any reprogramming of the front end becoming necessary.

The front end process is most easily programmed as a small network of processes. This is an illustration of the methodological point that any process in a network can itself be refined as a network of processes. Process networks in the domain of purely functional programming are beautifully modular in that processes are semantically entirely independent - no process can have hidden side effects on the working of any other process. The only interactions between processes are manifest quite explicitly in the streams of messages that pass between them. This remains true even when non-determinacy is present in the network - despite the fact that non-determinacy does compromise some of the hallowed properties of purely functional programs.

The refined network for the front end is:

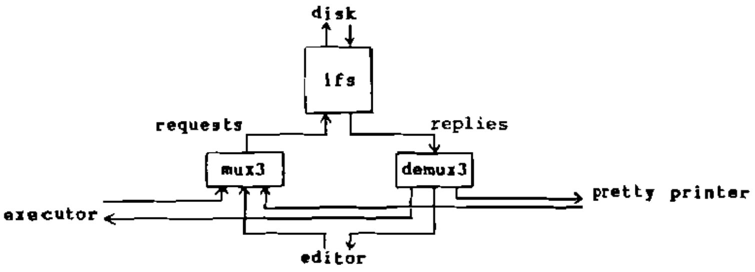


which can be programmed as:

```
    frontend(keyboard,execreplies,editreplies,prettyreplies)  ≡
         screen,execcommands,editcommands,prettycommands
         where screen ≡ merge(merge(execreplies,editreplies),prettyreplies)
                 execcommands,editcommands,prettycommands ≡ split(keyboard)

    split(keyboard) ≡      filter(isexec,keyboard),
                           filter(notexecorpretty,keyboard),
                           filter(ispretty,keyboard)

    isexec(c) ≡ (not atom(c)) and head(c) = "EXEC"
    ispretty(c) ≡ (not atom(c)) and head(c) = "PRETTY"
    notexecorpretty(c) ≡ (not isexec(c)) and (not ispretty(c))
```

The process which manages the intelligent file store is also conveniently refined as a network of processes. It must accept requests for file store actions from communication channels with the executor, editor and pretty printer. These requests will arrive asynchronously (in no predetermined order) and hence non-determinate merging is again necessary. Before merging, each request must be tagged to indicate its source so that the corresponding reply from the file store can be directed to the correct subsystem. This is a multiplexing operation. Clearly the disk cannot handle these tagged commands, so the file store manager must strip off the tags and reattach them to the replies from the disk (there is no reply to a PUT request, and in this case the tag is discarded). Fortunately, as implemented in [6], the disk is a synchronous external device, and the requests are serviced in precisely the order they appear in the output stream - thus the retagging is a sound operation. When the file store manager passes a PUT request to the disk it checks whether the file is a program source text (by applying a test to the file name), and, if it is, generates an extra PUT request with the compiled form of the program as contents (and with a modified file name). Replies from the file store are returned to the appropriate subsystem by a "demultiplexer", which is rather like "split" above.

Hence the intelligent file store can be refined to the following network:



which can be implemented by the following functions:

```
filestore(execreq,editreq,prettyreq,diskreplies)  ≡
     execreplies,editreplies,prettyreplies,diskrequests
     whererec replies,diskrequests ≡ ifs(requests,diskreplies)
         requests ≡ mux3(execreq,editreq,prettyreq)
         execreplies,editreplies,prettyreplies ≡ demux3(replies)

ifs(((tag  ("GET" a)).requests),(c.diskreplies))  ≡
     cons((tag c),replies),cons(("GET" a),diskrequests)
     where replies,diskrequests ≡ ifs(requests,diskreplies)

ifs(((tag  ("PUT" a c)).requests),diskreplies)
     if issource(a)  then  replies,dr2
                     else  replies,dr1
     whererec  dr1 = cons(("PUT" a c),diskrequests)
               dr2 = cons(("PUT" a c),
                     cons(("PUT" modify(a) compile(c)),diskrequests))
               replies,diskrequests ≡ ifs(requests,diskreplies)

compile(prog) ≡ ... not specified here ...
modify(name)  ≡ ... not specified here ...
mux3(a,b,c)   ≡ merge(tag(1,a),merge(tag(2,b),tag(3,c)))
demux3(s)     ≡ untag(1,s),untag(2,s),untag(3,s)
```

For this definition of mux3 we must use a version of merge which ensures that both the tag and the tagged message are present before selecting a stream element. The following definition of "here" will do the trick (see the definition of "merge" in Chapter 3):

```
    here(x) ≡ atom(head(x)) and if atom(head(tail(x))) then true else true
```

It remains to design the editor, executor, and pretty printing subsystems. These are suitable to be implemented as simple recursive functions - an attempt to refine them as networks produced rather contrived and inelegant solutions. For the editor we can adopt the synchronous editor described in Chapter 3. This had terminal input and output streams, and file store input and output streams, and hence will fit quite naturally into this network system. The only change required is to remove the check for DONE following a PUT command. The executor is quite straightforward. Each message it receives from the front end will have the form (EXEC a b c). For each such message the executor will request files a and b from the filestore and receive the contents of the files in reply (or MISSING for either of the files if not present in the file store). If either a or b is missing then suitable error messages are sent to the user's terminal (and the execution does not occur), otherwise the result is sent to the file store (and the screen remains quiet). Thus the execution component

$$\begin{tikzcd}
\text{front end} \arrow[r, shift left] & \boxed{\text{executor}} \arrow[l, shift left] \arrow[r, shift left] & \text{file store} \arrow[l, shift left]
\end{tikzcd}$$

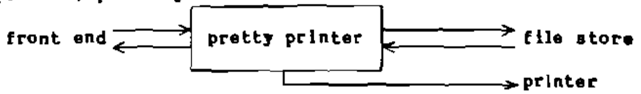
can be programmed:

```
    exec((("EXEC" a b c).commands),(prog data.filereplies)) ≡
          responses,cons(("GET"a),cons(("GET" b),filerequests))
          whererec  responses ≡ if errors
                                then append(errormessages,otherresponses)
                                else otherresponses
                    filerequests ≡ if errors
                                   then otherrequests
                                   else cons(("PUT" c result),otherrequests)
                    otherresponses,otherrequests ≡ exec(commands,filereplies)
                    result ≡ execute(prog,data)
                    errors ≡ (prog = "MISSING") or (data = "MISSING")           *
                    errmessages ≡ if prog = "MISSING"
                                      then cons("(Program missing)",em)
                                      else em
                    em  ≡ if data = MISSING
                                      then cons("(Data missing)","NIL")
                                      else "NIL"
```

(*The or used in this definition is a Boolean operator).

Here exec has two arguments - the stream of EXEC commands from the keyboard (via the front end) and the stream of replies from the file store. The function has a pair of streams as its result; a stream of error messages destined for the user's screen, and a stream of requests for file store operations.

The pretty printer is easily programmed in a similar way. It receives messages from the front end of the form (PRETTY a). For each such command it must request the contents of file a from the file store. If the file is missing then an error message is sent to the user, otherwise the contents is sent to the printer, prettily formatted:



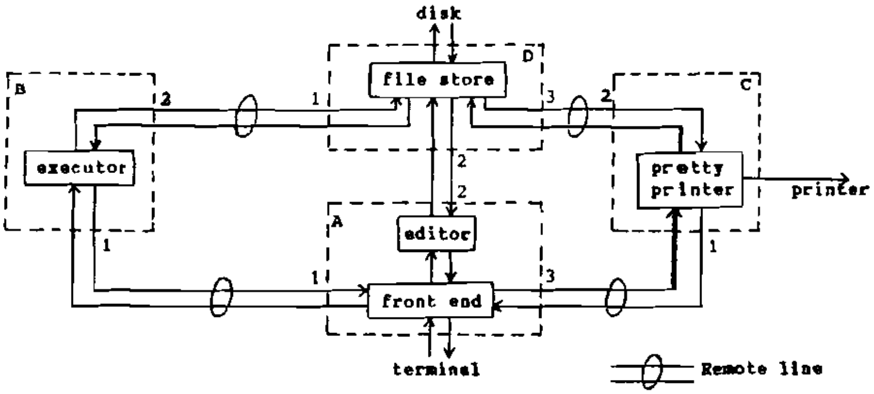
```
    pretty((("PRETTY" a).commands),(text.filereplies)) ≡
         responses,cons(("GET" a),filerequests),printing
         whererec  error     ≡ text = "MISSING"
                   responses ≡ if error
                               then cons("(Text missing)",otherresponses)
                               else otherresponses
                   printing  ≡ if error then otherprinting
                                       else cons(prettify(text),otherprinting)
                   otherresponses,filerequests,otherprinting ≡
                                       pretty(commands,filereplies)

    prettify(text)  ≡  ... not specified here ...
```

This completes the programming of the individual component processes of the distributed operating system. We must now look at the problem of configuring the logical architecture of the system to specific hardware. Clearly the simplest physical organisation for the system is to run the entire network as a single functional program on a single multiple stream computer. For this we require an abstract machine with, at least, terminal input and output streams, a disk based file store input and output streams, and a printer output stream. The function to be executed by the abstract machine is then given by the following definition:

```
    system(<keyboard,filein>)  ≡ <screen,fileout,printer>
         whererec  screen,fetoex,fetoed,fetopp ≡
                          frontend(keyboard,extofe,edtofe,pptofe)
                   edtofe,edtofs ≡ edit(fetoed,fstoed)
                   extofe,extofs ≡ exec(fetoex,fstoex)
                   pptofe,pptofs,printer ≡ pretty(fetopp,fstopp)
                   fstoex,fstoed,fstopp,fileout ≡
                          filestore(extofs,edtofs,pptofs,filein)
```

This function definition simply gives the connections which make up the network. Names have been allocated to each of the internal streams in the network, such as pptofs which stands for "pretty printer to file store".

When all the subsystems are executing on a single processor there will clearly need to be timesharing to achieve concurrent execution of the subsystems (the abstract machine will take care of this). For maximum speed we would like each subsystem to execute on a separate processor. If four processors are available then each major component can execute on its own processor, and the front end could share one of the processors. Bearing in mind that one remote line can support one stream in each direction, the front end and editor can be allocated to a processor A with a terminal and three remote lines, the executor can be allocated to a processor B with two remote lines, the pretty printer can be allocated to a processor C with two remote lines and a printer, and the file store can be allocated to a processor D with a disk and three remote lines:

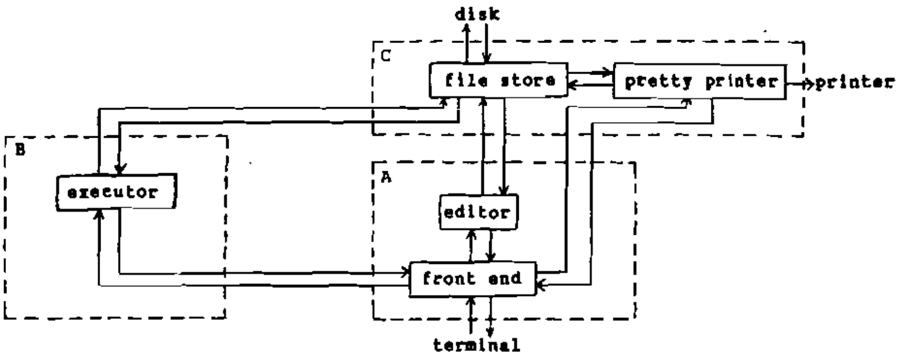


We require four functions, one to be executed by each machine:

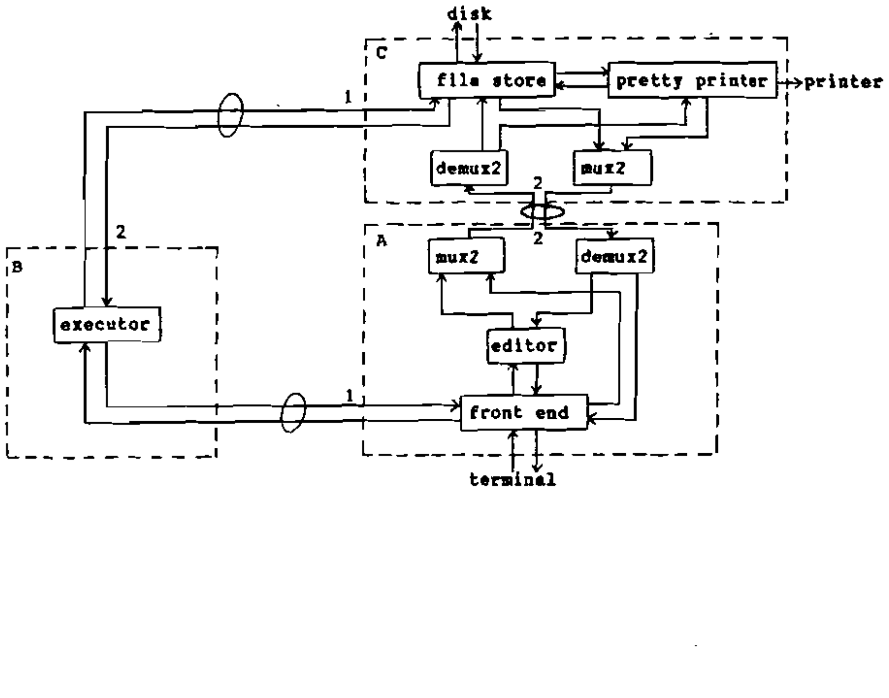
```
     sysA(<keyboard,rem1in,rem2in,rem3in>)  ≡
          <screen,rem1out,rem2out,rem3out>
          whererec    screen,rem1out,fetoed,rem3out  ≡
                            frontend(keyboard,rem1in,edtofe,rem3in)
                      edtofe,rem2out  ≡  edit(fetoed,rem2in)

     sysB(<rem1in,rem2in>)  ≡ <rem1out,rem2out>
          where  rem1out,rem2out ≡ exec(rem1in,rem2in)

     sysC(<rem1in,rem2in>)  ≡  <rem1out,rem2out,printer>
          where  rem1out,rem2out,printer  ≡  pretty(rem1in,rem2in)

     sysD(<filein,rem1in,rem2in,rem3in>)  ≡  <fileout,rem1out,rem2out,rem3out>
          where  rem1out,rem2out,rem3out,fileout  ≡
                            filestore(rem1in,rem2in,rem3in,filein)
```

Each pair of streams connecting the subsystems has been associated with one remote line port on each of the two processors involved. The appropriate physical connection must be made between each pair of corresponding ports. For example, the streams previously known as extofs and fstoex are now associated with remote port 2 on machine B and remote port 1 on machine D, and these ports must have an appropriate electrical connection made between them. In this way the network has been partitioned by associating communication streams directly with physical remote lines. No major software components need altering, only the functions describing the interfaces between streams and hardware ports need to be rewritten.

Many other configurations are possible. Suppose that the available hardware is rather more limited - three processors, A with two remote lines and a terminal, B with two remote lines, and C with a disk, a printer and two remote lines. Then the system can be partitioned as in the following diagram:



Communication between machines A and B, and between machines B and C can be supported by one remote line for each route. However, between machines A and C we have two pairs of streams but only one remote line available. To solve this problem we can multiplex several streams on a single remote line:



Now we can write down the function to be executed by each machine:

```
    sysA(<keyboard,rem1in,rem2in>) ≡
         <screen,rem1out,rem2out>
         whererec    screen,rem1out,fetoed,fetopp ≡
                          frontend(keyboard,rem1in,edtofe,pptofe)
                     edtofe,edtofs ≡ edit(fetoed,fstoed)
                     rem2out ≡ mux2(edtofs,fetopp)
                     fstoed,pptofe ≡ demux2(rem2in)

    sysB(<rem1in,rem2in>)  ≡  <rem1out,rem2out>
         where  rem1out,rem2out ≡ exec(rem1in,rem2in)

    sysC(<filein,rem1in,rem2in>)  ≡
         <fileout,rem1out,rem2out,printer>
         whererec  rem1out,fstoed,fstopp,fileout  ≡
                      filestore(rem1in,edtofs,pptofs,filein)
                   pptofe,pptofs,printer ≡ pretty(fetopp,fstopp)
                   edtofs,fstopp ≡ demux2(rem2in)
                   rem2out ≡ mux2(fstoed,pptofe)
```

Again, no subsystem has been altered, only the interfacing glue. Functions *mux2* and *demux2* can be constructed by analogy with mux3 and demux3.

If only two machines were available, with a single line between them, then it would be straightforward to allocate the executor to the same machine as the editor, and to multiplex three pairs of streams on the single remote line between the two machines.

Thus we have a methodology that produces systems which are very easy to lay out on the available hardware, and which can easily be reconfigured if some hardware components fail, or if more hardware becomes available. During the initial stages of system design it is not necessary to consider these eventualities in detail.

Chapter 5

SUMMARY AND CONCLUDING REMARKS

In the two year duration of the Functional Operating Systems project a considerable amount of progress was made. Achievements occurred in the development of the LispKit abstract machine to support the advanced features required of operating systems - this work was reported in [6]. In parallel, a range of operating systems was explored, exploiting the various advanced facilities offered by the enhanced abstract machines - a record of the major points in this exploration has been reported in this monograph.

The exploration can be seen in three stages:

Firstly, it was shown in Chapter Two how conventional styles of operating system can be programmed purely functionally in a quite straightforward way. The design of one particular, moderately conventional, system was considered in detail, and various extensions were discussed. The provision of a lazy, interactive implementation of a functional programming language makes such systems a realistic proposition.

Secondly, functional programming is an excellent medium for the construction of operating systems as networks of co-operating processes communicating via streams of messages. Considerable scope for interesting systems arises with the addition of non-determinacy to the language to allow merging of streams. These themes were explored in Chapter Three, with the consideration of a variety of simple shared database applications, and the design of a single user operating system with various different networks of processes.

Thirdly, the networks/processes/streams paradigm extends very nicely to operating systems which are distributed physically over a collection of processors. The architecture and major components of such operating systems are designed without consideration of the eventual hardware configuration. The network is then partitioned for the available hardware, and only small amounts of interfacing code need be written. The independance of the logical architecture of the operating system from the hardware means that the systems are easily reconfigured with changes in hardware availability. These ideas were illustrated in Chapter 4 by the design of a computer-to-computer file transfer system, and the design and configuration of a single user operating system for the exploitation of true physical concurrency.

Thus functional programming has brought to operating systems its power of expression, the modular design of networks (processes may be refined as sub-networks), and ease of distribution and reconfiguration. In the time available it has only been possible to make a broad feasibility study of these topics - many more sophisticated operating systems remain to be designed, other advanced techniques of functional programming remain to be discovered and exploited in the construction of operating systems, and there are surely other, probably more appropriate, enhanced versions of the LispKit abstract machine to be designed.

I believe that this project has shown that the functional approach to programming operating systems is indeed feasible. It is certainly worthy of further study in order to refine the implementations and techniques. Other groups are already pursuing these topics from their own viewpoints, and other approaches are coming to light.

CSP [15] and occam [16] provide an alternative approach to the construction of networks of processes, and they handle message streams in an elegant way. In fact, CSP and occam as they stand are more or less language skeletons - a framework for expressing processes and communications, with no advanced language component for expressing the computational part of each process (searching file stores, compiling, etc). It has been argued that, in functionally programmed processes, the expression of stream communications is the least elegant part. Perhaps CSP and occam have something to offer here, or perhaps there are equally elegant functional programming techniques waiting to be discovered (research employing functional programming does continue to discover elegant techniques with surprising frequency). CSP and occam also present their own solution to the non-determinacy problem from the outset.

Finally Clark and Gregory [17] report on work progressing concurrently with the research reported here. They describe the use of PARLOG, a parallel logic programming language, to tackle a number of the problems explored in the earlier chapters of this monograph. Their solutions are broadly similar, with some interesting differences where they exploit unification for the parsing of message streams, and where they use the "logical variable" to combine a stream of messages and a stream of replies into a single stream in a very elegant fashion. PARLOG is essentially a version of Prolog, extended with many features of operational significance. The language is quite intricate. It tackles the problem of non-determinacy by making a committed choice when a subgoal matches several alternative clauses in the program. Backtracking cannot occur once the choice has been made. This effectively provides the equivalent of <u>or</u> as described in Chapter Three.

This considerable interest clearly indicates the significance of the continuing study of functional operating systems.

References

[1a] ABRAMSKY, S SECD-M: A virtual machine for applicative multiprogramming.
Queen Mary College, Computer Systems Laboratory, 1982.

[1b] ABRAMSKY, S A simple proof theory for non-deterministic recursive programs.
Queen Mary College, Computer Systems Laboratory, 1982.

[2] DARLINGTON, J and REEVE, M Alice, a multiprocessor reduction machine for the parallel evaluation of applicative languages.
Internal report, Department of Computing, Imperial College, 1981.

[3] DENNIS, J B Varieties of data flow computers. Proc. of 1st Int. Conf. on Distributed Computer Systems, pp 430-439, October 1979.

[4] HENDERSON, P Functional programming: Application and implementation. Prentice Hall, London, 1980.

[5] HENDERSON, P Purely functional operating systems.
In Functional programming and its applications, Eds. Darlington, Henderson and Turner, CUP 1982.

[6] JONES, S B Abstract machine support for purely functional operating systems.
Programming Research Group Technical Monograph PRG-34, Oxford University, August 1983.

[7] KARLSSON, K Nebula: A functional operating system.
Internal Report, Laboratory for Programming Methodology, Chalmers University of Technology and University of Goteborg, 1981.

[8] McCARTHY, J et al. The Lisp 1.5 Programmer's Manual. MIT Press, 1962.

[9] BURTON, F W and SLEEP, M R Executing functional programs on a virtual tree of processors.
Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, October 1981.

[10] WATSON, I and GURD, J A prototype dataflow computer with token labelling.
Proc. Nat. Comp. Conf. Vol 48, pp 623-628, 1979.

[11] HENDERSON, P, JONES, G A and JONES, S B The Lispkit Manual.
Programming Research Group Technical Monograph PRG-32 (2 volumes), Oxford University, 1983.

[12] SHULTIS, J A functional shell.
ACM SIGPLAN Notices, Vol 18, No. 6, pp 202-211, June 1983.

[13] FRIEDMAN, D P and WISE, D S An indeterminate constructor for applicative programming.
In Conf. Record of 7th Annual ACM Symposium on Principles of Programming Languages, Las Vegas, 1980.

[14] McCARTHY, J A basis for a mathematical theory of computation.
In: Studies in logic: Computer programming and formal system.
Eds. Braffort and Hirschberg, North Holland, 1963.

[15] HOARE, C A R A model for communicating sequential processes.
Programming Research Group Technical Monograph PRG-22, Oxford University, 1981.

[16] INMOS Ltd Occam Programming Manual.
Prentice-Hall International, 1984.

[17] CLARK, K L and GREGORY, S PARLOG: A parallel logic programming language.
Department of Computing Research Report DOC 83/5, Imperial College, London, May 1983.