*Technical Report*

UCAM-CL-TR-26
ISSN 1476-2986

Number 26

# Portable system software for personal computers on a network

Brian James Knight

This technical report is based on a dissertation submitted April 1982 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Churchill College.

ISSN 1476-2986

CONTENTS

## Preface

I wish to thank my supervisor, Dr Martin Richards, for his encouragement during my research, and to acknowledge my indebtedness to him and other members of the TRIPOS Group and Systems Research Group for many illuminating and constructive discussions.

During my research I received a grant from the Science Research Council: I am grateful for that support.

Except where otherwise stated in the text, this dissertation is the result of my own work, and is not the outcome of work done in collaboration.

I hereby declare that this dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.

I further state that no part of my dissertation has already been or is being currently submitted for any such degree, diploma, or other qualification.

## Summary

This dissertation is concerned with the design of the portable operating system TRIPOS, and its use as the basis for an operating system to run in 'single connection' computers - that is, computers whose only peripheral is an interface to a local area network.

TRIPOS is a lightweight, yet powerful, multi-tasking operating system aimed at personal minicomputers. It is designed to be relatively straightforward to transport to new hardware, providing an almost identical user interface and program environment on each machine. Particular emphasis has been placed on avoiding unnecessary complexity, in order to make it simple to understand, explain, and adapt for special purposes. The majority of the system and utilities are written in the language BCPL, and can be moved without change to different computers. They run on a kernel and device drivers written in assembly language for each particular machine. The user's view of the system is presented first, with samples of console dialogue, and then its internal structure is described.

The main part of the work described concerns the building of a portable operating system presenting user and program interfaces as similar as possible to ordinary TRIPOS, but running in processors connected only to a local area network - the Cambridge Ring. The system makes use of 'server' computers on the Ring in order to gain access to disc storage, terminals, and printers. Several methods are investigated for using the primitives provided by a universal file server to construct a filing system which can be shared by machines of different types. Some conclusions are drawn on the effects of distributing operating system functions in this way.

# CHAPTER 1

# INTRODUCTION

This chapter sets the scene for the rest of this dissertation by stating the motivation behind, and aims of, the work described below. It outlines the contents of subsequent chapters, and surveys other relevant research.

## 1.1 Motivation and Aims

The original motivation for the work described in this thesis was the desire to write a simple but powerful operating system for personal computers, which would be both easily portable to a variety of minicomputers, and would avoid features and restrictions which had been found annoying in manufacturers' operating systems.

As the implementation of this operating system, TRIPOS, was nearing completion, a local area network, the Cambridge Data Ring, became available in the laboratory. Work then began on conversion of TRIPOS into a system to run on minicomputers with a ring connection as their only peripheral, accessing terminal, disc, printers, etc., via other 'server' machines on the Ring. The resulting system was designed to retain the portability of the original, so that a user of the Ring could ask for 'a computer running TRIPOS', and not need to specify a particular type of machine.

## 1.2 Outline of Thesis

The work described in this thesis falls broadly into two parts. Firstly, the design and implementation of the portable operating system TRIPOS is described. Secondly, an account is given of the use of a local area network from TRIPOS, and the design of an operating system based on it for use in personal computers whose only peripheral is the network.

Chapter two mentions the history of TRIPOS, giving the ground rules which influenced its design. It then goes on to illustrate how the system looks externally - i.e. to a user sitting at a terminal. The internal structure of the operating system is explained in chapter three, which also includes some conclusions on the merits and defects of its design.

The remainder of the dissertation is concerned with the interfacing of TRIPOS to a local area network, the Cambridge Ring, and with the production of an operating system to run in computers which have the network as their only peripheral. Chapter four contains a brief description of the Data Ring and computer interfaces to it. The Ring interface software for TRIPOS is described in chapter five.

The next two chapters discuss the issues involved in using a network file server to replace a local disc. In the past, file servers have usually been designed specifically for an existing filing system. The Cambridge Fileserver aims to be more general than this, so the opportunity was taken to investigate different ways of implementing a filing system on an existing file server. The first experiment was to use a single large Fileserver file as an image of a real disc, and to retain the old file handler task; the results of this are given in chapter six. Chapter seven describes a more thorough approach, rewriting the file handler to take advantage of facilities offered by the Fileserver.

The final steps to an operating system for 'single connection' computers are described in chapter eight, which includes some conclusions on the effects of distributing parts of the system. Chapter nine contains a summary of the work done and some concluding remarks.

## 1.3 Extent of Collaboration

The original design and first few implementations of TRIPOS, described in chapters 2 and 3, was a group project, on which I worked with Adrian Aylward, Paul Bond, and Richard Evans, under the supervision of Martin Richards. My particular area of work at this time was in the design and implementation of the first full version of the TRIPOS kernel (for a PDP11), disc drivers, and numerous commands and utilities. The comments on the system are my own.

Chapter four, on the Ring hardware, interfaces and protocols, is included for completeness and to provide sufficient context for the following chapters; the work described is not mine.

All of the Ring software for TRIPOS, the Fileserver-based filing systems, and the creation of the operating system for single-connection computers, are my own work. Inevitably, it relies on the work of others who have built the Ring and interface hardware, and written servers for the Ring; they are referenced in the text. The discussions of the Ring 'world', and of the facilities provided by various servers, are my own.

## 1.4 Related Work

There are two main areas of research related to the work described in this dissertation - the design of operating systems (especially portable ones) for personal computers, and investigations into the distribution of operating system functions over local networks.

### 1.4.1 Operating Systems

Many operating systems have been written for minicomputers in recent years. This section concentrates on those which have some aims in common with TRIPOS, in that they were designed to be portable, have been transported, or are specifically for personal computers. Portable operating systems is a fairly recent area of study. Most of the systems described below were designed or first transported during the period that TRIPOS was being developed. Thus they are presented for retrospective comparison with TRIPOS, rather than because they strongly influenced its design.

OS6 [43] is an operating system designed by Stoy and Strachey at the University of Oxford. It is a single process system for a single user and is thus able to sidestep many problems which arise in more general systems. The designers claim that the design of a satisfactory single-user system should be the first stage in the production of a multi-user system, and that it presents several interesting problems which should be solved before the complexities of concurrent processing are faced. The original implementation was for a Modular One.

As a deliberate act of policy, OS6 was written entirely in one high-level language. The designers considered that the ideal language would be one that concentrated its resources around its control facilities, and left matters concerning storage and representation very much to the programmer. The language they chose was BCPL [37], which was compiled to an interpreted code. The use of an interpreter reduced the effective machine speed by a factor of 15, but left the designers free to concentrate on the structure of the operating system, without being bullied by the idiosyncrasies of the particular machine on which they implemented it. The operating system contains only 17 words of virtual machine code, and the interpreter consists of 250 intructions of Modular One machine code.

No special job control language was devised, BCPL being used at all levels. The top level is not special: any program may load, obey, edit or compile any other, to any depth.

OS6 is a very open system, with no privileged routines. This offers great flexibility to the programmer, but means that absolute protection of the system from its user is impossible, since anything may be overwritten at any time. The problem of dealing with malice on the part of the user is avoided altogether, as it is not possible in such an open system, and as the authors considered it unnecessary in their sort of environment. However, the system does check against the most common forms of accidental error. The store is divided into two segments, making accidental overwriting of program very unlikely.

OS6 is notable for its I/O streams, which are defined in a device-independent way. It has the concept of stream functions, which take one stream as argument and return a new one as result. This allows compound streams to be built, which do arbitrary processing on data as they pass. The system also aims for device independence by having its own internal code - basically ASCII with the concept of underlined characters, and a special code for '4 spaces'. Conversions to and from this code are done in the device handlers.

As so much of OS6 was written in a high-level language, it was likely to be fairly easy to transport. Snow [42] gives an account of moving it to a Burroughs B1726 computer. This was partly achieved by adapting the target machine to the operating system: it is a microcodable machine, and a microprogram was written to directly interpret OCODE [37], the intermediate code produced by the BCPL compiler.

The major changes needed to the operating system were those necessary to reconcile its view of I/O streams with the record-oriented Burroughs hardware. Changes were also made to the object module format, the load and unload routines, and the dump program. OS6 internal code was abandoned and replaced by EBCDIC, forcing minor modifications to a few programs.

Snow concludes that BCPL is particularly suitable for achieving portability because it produces code for a well-defined virtual machine. The fact that OS6 was written in a highly structured way made it easy to identify those places requiring modification, and to be confident that changes would not have unfortunate side-effects. He also notes that since OS6 is a rather rudimentary operating system, it does not require much support from the underlying machine, making it simpler to move.

Another interpreted single-user system is SOLO [4], written by Brinch Hansen primarily as an exercise in producing a reasonable-sized Concurrent Pascal program. It is a system with a fixed number of processes, and was implemented on a PDP11/45. Like OS6 it uses its implementation language as a job control language, and a program called from the console is not in a special environment. Calling a command without arguments causes it to print a list showing what it expects.

The Pascal compiled code is interpreted, and the system rests on a machine code kernel of 4K words. The rest is written in Sequential and Concurrent Pascal. System protection is achieved largely by compile-time checking of access rights, so run-time checking is minimal and not supported by hardware.

The filing system is flat, but each user has his own disc. Files are typed, and are protected only against <u>accidental</u> overwriting or deletion. There is a restriction that only one new file at once can be created.

It is possible for more than one program to want to read from the terminal. Hence programs requesting console input identify themselves by means of a prompt, and gain exclusive access for one line.

Brinch Hansen claims that it is worthwhile to write even a small concurrent program in a high-level language that enables a compiler to check data types and access rights, as this helps to eliminate time-dependent errors.

Powell [32] has transported SOLO to a Modular One. He found three main areas in which differences between that machine and the PDP11 caused problems. Firstly, the Modular One had slightly less store. Secondly, it had a different and less sophisticated memory management scheme. Thirdly, it had no floating-point hardware, so some of the Pascal virtual code had to be patched to use integer arithmetic. He concludes that, although in a system supported by a special-purpose virtual machine it should be possible to hide all the details of the real machine from the virtual code, in practice such things as small word lengths or lack of floating-point hardware can prove very costly to hide.

One of the first systems to be designed with portability in mind is MUSS [12], which is a family of compatible operating systems for both large computers and mini-computers. It supports multiple users, giving them both batch and interactive working. The designers directed their efforts not towards producing an operating system which does "everything for everyone", but towards making one which could be adapted to suit the requirements of any individual user or installation.

MUSS is a multi-process, message-based system. The designers argue that message-passing offers a significant number of advantages over inter-process communication by data sharing, as it preserves the independence of modules, and is more amenable to distribution over a network. MUSS consists of a matrix of modules, with several versions of each one, so that a range of compatible systems can be built. The designers found that the requirement for several different versions of each module to exist and be maintained simultaneously led to much cleaner specifications of the modules and the interfaces between them. In fact, system specification became the primary objective of the project, with a working implementation as the second. To this end, the process of documenting each module was largely automated.

MUSS is intended to support a potentially large number of user processes (possibly several hundred). It has been implemented on a wide variety of computers, including the MU5 and mu5 machines at Manchester University, ICL 1905E, ICL 2900 series, PDP11/40, and MEMBRAIN MB7700.

MUSS consists of a set of processes executing on virtual machines implemented by a central kernel. The designers considered two levels at which machine independence could be attempted: at the virtual machine / kernel interface or at the kernel / hardware interface. The first is sufficient to lead to user programs and most of the operating system being portable. The second is impractical because the kernel is inherently machine-dependent anyway. However, the difficulty of producing and maintaining a kernel for each machine is extremely high because of its complexity, so they had to attempt some level of machine independence within the kernel. This lead to the specification of a MUSS 'ideal machine'.

The designers were well aware that the effectiveness of this technique in achieving portability is critically dependent on the ideal machine interface. If this interface is set at too low a level, then machine-dependent features start to permeate the logic of the operating system, thus restricting the range of machines covered by the design. On the other hand, if it is set too remote from the hardware, then the amount of software required to emulate the ideal machine will be unduly large.

Peripheral control in the ideal machine is achieved by the use of dedicated control registers (similar to those in PDP11 hardware). These are emulated by a set of procedures to access the registers, and a set to handle real interrupts and force virtual machine interrupts where necessary. The ideal machine assumes that hardware support is available for two processor states: user and kernel (supervisor).

Since real devices vary so much, it was decided to design 'ideal peripherals' according to what was most convenient for the operating system, leading to only two sorts of device: input and output. It later became necessary to formalize several device-dependent functions and include them in the ideal device specifications.

The designers of MUSS found that the address translation method was one thing which could not sensibly be included in a single ideal machine specification. It applies to every instruction, so is very expensive to emulate on hardware which does not match the model. Instead, a family of ideal machine interfaces was produced, including at first support for paged and multiple base register machines.

Probably the nearest system to TRIPOS in aims is Thoth [5, 6], which is a portable real-time system for mini-computers. It supports multiple processes, multiple users, virtual memory, and swapping. The original objectives were to investigate the feasibility of portable operating systems for a specified class of machines, and to provide a tool for teaching real-time programming of mini-computers. Further aims were to create an environment which encouraged the structuring of programs as many small concurrent processes, and a system which was adaptable to a variety of real-time applications. It has been implemented on TI 990 and Data General Nova computers.

The language chosen for Thoth is Eh, a derivative of BCPL which is intended to conceal hardware idiosyncrasies while avoiding being a barrier between the programmer and the hardware. The language includes statements for enabling and disabling interrupts, and a "twit" statement for inclusion of in-line assembly code. A function may be invoked as a subroutine or a separate process.

Inter-process communication in Thoth is by means of messages with a fixed length of 8 words. Primitive functions exist to send and receive messages, reply to them, and forward them to other processes. The message-sending primitive blocks execution of the calling process until the message returns, meaning that a program which wants to continue executing while it has messages outstanding must consist of several processes.

It is possible to set up a team of processes which share the same address space, and thus may share data. Processes on different teams may not share data. The unit of swapping is a team.

Thoth aims to provide a reasonably uniform interface with peripheral devices and files. The filing system has a tree structure, and has a facility for mounting new disc volumes by grafting references to them into the tree. There is the concept of a "mark" within each file, which generalizes the notion of "end-of-file".

Programs running under Thoth can enquire about particular features of the host machine. This environment information is provided in three ways. Manifest constants are used for things known at compile-time, such as the number of bits per word. Parameters known only at execution time are made available as global variables, or through system function calls.

The authors of Thoth observed that it seems impractical to design system software to be portable over all computers, and so aimed for portability only over a restricted range of machines - the "Thoth domain". Description of this domain documents assumptions made about the machine architecture in the implementation language and in the machine-independent parts of the operating system. The machine should have a word length of at least 16 bits, it should be possible to store any word pointer in a word, and to address consecutive words with consecutive integers. The machine should be a single processor with interrupts and the means of disabling them, and should permit implementation of a stack using a dedicated index register.

The small amount of assembly code (a few hundred instructions) in Thoth is used for setting up interrupt vectors, handling interrupts, providing intrinsic functions called by compiled code, and efficient versions of frequently-used functions. Assembly code also occurs embedded in Eh programs as "twit" statements. The designers refrain from estimating the time taken to port Thoth, as their experience is limited to ports done by the implementors themselves while the system was still evolving.

Thoth supports only one language. The authors consider that a major deficiency of the system is due to the untyped nature of this language. Since an integer is just a word, the maximum (assumed) value is 32767, which leads to problems, for example, with the position pointer in large files. The lack of a special pointer type leads to inefficiency on byte-addressed machines, as a pointer is defined to be a word pointer. They plan to introduce types to the language, and note that their desire for types is based entirely on considerations of efficiency and portability over diverse machines. They also intend to attempt to devise portable abstractions for memory management and protection hardware.

By far the most widely used of the operating systems considered here is UNIX [39, 40]. This is a multi-user operating system designed by Ritchie and Thompson at Bell Labs, and originally implemented on a PDP11. They did not set out to produce a portable system, but rather to provide a congenial programming environment. Because UNIX is written in a high level language (C - another derivative of BCPL), and has a concise and elegant structure, it has proved feasible to transfer it to other machines. It has become very popular, and positive feedback is likely to make it become more so - the more installations there are, the greater the amount of useful software that will become available to run under it, and so the greater the benefits of moving it to a new machine. Perhaps one should be careful when concluding that wide use of a system means that it is inherently good; the merit of the most universally used computer languages seems to be that everyone else uses them rather than that they represent the best designs available. However, UNIX has spread by demand rather than because it has been pushed by its originators.

The designers of UNIX wanted to build a system that was simple, elegant, and easy to use. They considered it important to make each program do one thing well, and to make it straightforward for the output of one program to become the input of another. Thus programs are viewed as tools, and a new job can often be done by coordinating a suitable set of existing tools. The use of textual formats for both input and output is strongly encouraged.

UNIX is a multi-process system which makes use of memory mapping and swapping. New processes are created by fork operations to split existing ones. Inter-process communication is by means of <u>pipes</u> - channels for passing arbitrary streams of bytes. This mechanism is not completely general, as a pipe must be set up by a common ancestor of the processes involved.

There is a powerful command line interpreter (called the "shell"). Every command runs in a new process, and the user chooses whether or not he wishes to wait for it to finish before starting a new command. The shell provides three standard I/O streams: a standard input stream, standard output stream, and console output stream. It is possible to cascade commands so that the output stream of one becomes the input for the next, and they both run in parallel. Hence it is sensible to write commands as "filters" - to take text as input, transform it in some way, and produce text as output.

Files and devices are handled in a uniform way. The unconventional approach is taken of regarding devices as special kinds of file and allowing them to be retained in directories. A mounted disc volume can be grafted into the filing system as in Thoth.

UNIX contains about 1000 lines of assembly code, of which about 200 are for efficiency only. The system calls are subroutines. There is no general inter-process communication or synchronization scheme: the designers admit this to be a weakness, but do not feel it to be important.

In his retrospective look at UNIX [40], Ritchie suggests a number of reasons for its success. It is simple enough to be comprehended, yet powerful enough to do most of the things its users want. The user interface is relatively clean and surprise-free (but terse to the point of being cryptic). It runs on a machine (PDP11) that is popular in its own right, and a good deal of software is available to run under it.

He also lists some things it is not good at. It is not a real-time system, as processes cannot be locked in memory, and cannot connect directly to I/O devices. Although pipes are sufficient for related cooperating processes, they are not much use for multi-event processes. I/O appears to be synchronous, but read-ahead and write-behind occur invisibly. In some situations (e.g. network control) one would like to start several I/O transfers, and wait for the first to complete.

Ritchie makes several recommendations to operating system designers. He says that there is no excuse for not providing a hierarchical filing system as it is useful, efficient, and easy to implement. He considers that a file should consist of a sequence of bytes - the notion of a record being an obsolete remnant of the days of punched cards - and that there should be only one format for text files. Finally, he recommends that systems should be written in a high-level language that encourages portability.

UNIX was ported from the PDP11 to Interdata machines independently by Miller [25], and Johnson and Ritchie [40]. Johnson and Ritchie observed that even programs in high-level languages tend to make assumptions about word and character sizes, the character code, filing system structure and organization, and peripheral device handling. They conclude that portable programs tend to be good programs for reasons other than their portability, and that although making programs portable costs some intellectual effort, it need not degrade their performance.

They decided to refine and extend the C language to make most C programs portable, and to restructure the compiler so that it could be changed comparatively easily to generate code for different machines. The changes they made to the language were to allow types to be unions of other types, and to add a TYPEDEF facility enabling types to be parameterized. They wrote a program to check C programs, reporting on type rule violations, dubious coding practices, and such things as uninitialized and unused variables. Keeping type-checking separate from compiling is in keeping with the UNIX philosophy of making each program do one thing well.

In porting UNIX to the Interdata, Johnson and Ritchie found that the main problems were that their original routines for multi-programming turned out to be unimplementable and had to be redesigned, that the stacks on the two machines grew in opposite directions, the memory mapping hardware was different, and so was the handling of processor traps. They also encountered problems of byte-ordering within words when transferring files between machines, and had to write code to reverse the order in those words representing integers. They found that programs tended to assume an integral number of bytes per word, and at least 8 bits per byte. They felt inclined to make ASCII the standard character code for UNIX, as many programs assume it. They point out that a problem when porting between machines of very different size is that algorithms built into the system may not scale well (just as suitable algorithms for sorting 10, 1000, or 1000000 things are different). Finally, they note that a system can be easy to transport by virtue of being easy to change.

Miller's approach to porting UNIX to an Interdata machine was rather unconventional, as he did it top-down in three stages. The first stage was to transfer the C compiler and run-time library to run under an Interdata operating system. He then moved the UNIX kernel and ran it as a privileged user program on that system. Finally, he wrote the assembly code for the interrupt routines, etc., allowing UNIX to be run on the bare machine. The advantages of the top-down approach lie in the ease of extensively testing each level as it is moved. The first stage allowed bugs in the C compiler to be found. It also meant that several UNIX utilities (such as editors) could be made available under the Interdata operating system, to assist during the rest of the project. The second stage allowed the main logic of the kernel to be tested without being concerned with the detailed characteristics of the hardware. Finally, the low-level routines could be tested using the upper two layers, in the knowledge that the latter were already working correctly.

Miller too found that most of the changes resulted from implicit assumptions about the C implementation, and recommended more rigorous type-checking in the C compiler, and parameterization of machine-dependent constants. He expressed some reservations about the general portability of UNIX, as the PDP11 and Interdata are similar in some important respects, such as being byte-addressed, having several general purpose registers, interrupt-driven I/O and segmented memory management, but concluded that the project showed the value of the kind of careful and consistent design effort that went into UNIX.

An unusual single-user operating system is described by Lampson and Sproull [20]. Whereas most operating systems provide a kind of womb (virtual machine) to insulate the user and her program from the harsh realities of the outside world, this one is very "open", establishing no sharp boundaries between itself and the user's programs. It offers a variety of facilities which the user can reject, accept, modify or extend. Both low-level and high-level components of the system are accessible.

The most important part of the specification of such a system is the standardization of the disc representation of files and of the network representation of packets. The price paid for flexibility is that any changes to these representations require changes to several pieces of code, possibly in different languages and/or maintained by different people. In fact, the authors do not recommend standardization at this level when processor speed and memory are ample.

This operating system was written for the Xerox Alto computer [47] - a machine with 64K 16-bit words of memory and one or two discs, display, and keyboard. The instruction set supports BCPL, which was used as the implementation language of this system. The system provides only two processes: one for keyboard input, and one for everything else. There is no scheduling or synchronization, as the keyboard process is interrupt driven.

This operating system can be viewed as a collection of procedures which implement various potentially useful abstract objects. It is arranged as a set of subroutine packages placed from the top of store downwards in decreasing order of expected usefulness, so it is easy to selectively remove different layers of the system. The kinds of abstract object available include I/O streams, files, storage zones and physical discs; each abstract object can be represented in one BCPL word.

In such an open operating system, all communication between programs must take place via the disc. A convention is defined for restoring the entire state of the machine from a disc file, allowing an arbitrary program to take control of the machine. Such programs are called "juntas". This leads to an unusual coroutine structure, in which each coroutine switch consists of saving the entire machine state, and then reloading the machine with another saved state. The fundamental routines of the operating system are those which do this saving and restoring. They live at the very top of store, but should ideally be in read-only memory. The junta mechanism has found many uses, such as bootstrapping, debugging, checkpointing a running program, and activity switching. For example, a printing server could be split into a printer and a spooler as coroutines.

The disadvantages of a very open system are that it is difficult to change the representation or functionality of the filing system or communications. It is also impossible to intercept all accesses to the filing system or display - in order, for example, to direct them to a remote system.

Another single-user operating system written at Xerox PARC is PILOT [34]. It is a single language (Mesa) system for rather powerful personal computers with an interface to a local network, and supports virtual memory, a large "flat" filing system, multiple processes, and network streams. The Mesa language contains support for coroutines, processes, and monitors.

Pilot provides a basic set of services within which higher-level programs can more easily serve the user and/or communicate with other programs on other machines. It can be thought of as very powerful run-time support for Mesa. It omits such things as character-string naming of files and interpretation of user commands; these are provided as needed by higher-level software. The user interface is via a bitmap display with keyboard and pointing device.

The protection mechanisms in Pilot are defensive rather than absolute, since in a single-user system errors are a more serious problem than maliciousness. All the protection ultimately depends on Mesa type-checking. The resource allocation features are not oriented towards enforcing fair distribution of scarce resources. The virtual memory system gives a very large simple linear memory, and everything runs in the same address space.

The filing system is flat in the sense that there is no relationship between files. Files have 64-bit names that are unique in both time and space over all incarnations of Pilot. A file is accessed by mapping one or more pages of it into virtual memory. The file structures are such that the filing system can be completely rebuilt even if the maps are lost. Devices can be accessed through the Pilot stream facility or directly via a low-level device driver exported from Pilot. Most I/O devices (except discs) are available directly to clients. Streams have similar facilities to those of OS6 and UNIX.

The Mesa language supports communication between tightly coupled processes by means of shared memory. Pilot has a communications facility for loosely coupled processes (perhaps on different machines) consisting of a hierarchical family of packet communication protocols. These protocols are designed to be suitable for communication across multiple interconnected networks. Communication software is an integral part of Pilot because it is intended to be a suitable foundation for network-based distributed systems. If both ends of a communication are on the same computer, then the software will recognize this and avoid using the network. This means that the communication facilities can be used within multi-process programs on an isolated computer. Such programs are easily split over several computers at a later date.

Pilot supports I/O streams over the network. Usually, such streams are asymmetric, with one end being considered a client and the other end a server. Deleting such a stream causes no network traffic, but just removes the local record of it; it is left to the clients to have agreed termination at a higher level.

Pilot was implemented by a team of 6-8 people in 18 months. They attribute their speed to the use of the strongly-typed Mesa language, and the use of small modules.

The authors of Pilot found that the context of a large personal computer led to a reevaluation of many design decisions which characterize systems designed for more familiar situations. This has led to a system which provides sophisticated features but only minimal protection, which accepts advice from client programs, and even boot-loads the machine periodically in the normal course of execution. It provides an environment with relatively few artificial limitations on the size and complexity of the client programs which can be supported.

In a review of the development of Pilot [22], Lauer comments on several of the design decisions. He says that an assumption behind the way virtual memory was implemented was that disc accesses were expensive, leading to an implementation making heavy use of queues and multiprogramming. In fact, on two of the machines on which Pilot runs, disc accesses are cheap and it would almost be more efficient to treat the disc as a synchronous device. The stream facility has not been widely used by either clients or implementors. This is not because it does not work satisfactorily, but because the Mesa language makes it more convenient to bind programs with procedural interfaces, even over a network.

Typical of local networks are Ethernet [24], developed at Xerox PARC, and the Cambridge Ring [17, 49]. The Ethernet consists of coaxial cable structured as an unrooted tree connecting all the computers on it. Any computer wishing to send a message listens to the ether for a while to check that it is not being used, then transmits its message (or packet). It is possible that more than one station will decide to transmit at once. Both transmitters and receivers can detect that this has occurred: transmitters abandon the packet, wait a random time and try again, while receivers simply discard any garbled packets. The maximum packet length is conventionally limited to 500 bytes, to keep the latency of the network down. A packet may be addressed to a particular station or broadcast to all stations. Packets are delivered only with high probability, so applications which want an error rate better than that of the raw network must employ protocols which detect errors.

The Cambridge Ring is a loop of two twisted pairs of wire, around which a few packets circulate. Each packet can hold 16 bits of data. A transmitter sends data a word at a time by waiting for an empty packet to come past and filling it. Each packet is addressed to a particular destination station; there is no broadcast facility. A transfer unit of 16 bits is too small for most purposes, and addressing to just a station is usually too coarse, so data are usually packaged (by software) into basic blocks, containing between 2 and 2048 bytes of data, and addressed to a software port of the destination machine. The Ring is described in more detail in chapter 4.

Both of these networks have been used to build distributed systems based on the "user - server" model [27], in which some machines are dedicated to providing generally useful services such as printing or disc storage. Other computers requiring these services may call on the server machines as clients.

In most of these systems, users are provided with "workstations" consisting of a mini- or micro- computer of moderate size, with its own keyboard and display.

### 1.4.2 Local Networks

The linking of individual computers so that they can directly communicate can be achieved in a variety of ways. The method used and the tightness of the coupling tend to be dictated by the distances involved.

At one extreme, when the computers are miles or thousands of miles apart, they will usually be linked by telephone lines to form a 'wide area' network. Typically the bandwidth of such lines is low compared with the rate at which a computer can transfer data, the error rate is significant, and parts of the network may fail. This necessitates an elaborate protocol and routing mechanism to ensure that data gets through intact and in the right order, and to provide flow control. A lot of the technology is concerned with making the network as reliable as the computers connected to it. Thus communication is expensive and is preferably done in fairly large units, such as transferring a whole file or job.

At the other end of the scale are very closely coupled computers, in the same room or even the same box, sharing main memory and peripherals. Communication between machines is very fast and reliable, meaning that protocols can be very simple. In such a setup, the computers are often all of the same type, running the same software.

In between comes the type of network of interest here, the <u>local area network</u> linking computers within a building or small group of buildings, up to a maximum distance of a few hundred metres. The properties are intermediate between those of the extremes mentioned above. The bandwidth is of the same order as rate at which computers can supply data (e.g. 100,000 bytes/second), and the error rate is low. The principal cause of communication failure is likely to be the failure of one of the computers involved rather than the network. The overhead on communication is fairly small, so it is reasonable to converse in blocks tens or hundreds of bytes in size. There may be machines of several different types connected to the network.

For example, on the Ethernet at Xerox PARC, most of the workstations are Alto minicomputers [20, 47] with keyboard, high performance display, pointing device, and one or two local discs. Each station can thus be used as an independent personal computer, and is used in this way, calling on the many services available from the network only when they are required. This approach ensures that a network failure does not render all the workstations useless, but makes the cost per station high. Some of the workstations are built round the very powerful Dorado computer, running the Pilot operating system which supports both a filing system on the local disc, and network streams (see "Operating systems" above, and [34]).

In commercial networks, the tendency has been not to include a local disc at each workstation, but to employ one or more central file servers for all disc storage, thus reducing the cost of each station. Examples are NESTAR [33], based on Ethernet, and Logica's Cambridge Ring system [45]. NESTAR is a network of Apple II microcomputers. The network is connected to each user station through the machine's disc interface. Disc storage is provided by a file server which is an Apple with floppy or Winchester discs. Printing servers are also standard Apples, so can be used as user workstations when not needed for printing. The Logica network is broadly similar, except that it is based on a ring, and makes use of different types of computer for different functions.

The organization of the Ring in the University Computer Laboratory in Cambridge differs from all the above in that it does not have workstations with significant processing power. Instead, terminals are connected to the Ring via small multiplexing computers, and can be used to access computers across the Ring. One can log in to any of several multi-user computers in this way. In place of the personal computer in each workstation is a pool of uncommitted processors, which one may obtain exclusive use of, load, and connect a terminal to, all via the Ring. This system is described more fully in chapter 8.

An important way in which the various locally distributed systems differ is in the abstractions provided by their file servers, or in other words, how the job of implementing the filing system is split between client and server. The literature contains little information about how filing systems are built on file server primitives, but concentrates on the design of the servers themselves. This is probably because most file servers are built to meet the needs of existing filing systems.

The simplest file servers provide facilities at a level quite close to that of a local disc controller. They allow reading and writing of information in units of pages. Any high-level filing system must be implemented by code in each client, or by a separate "filing system server" which clients use as an intermediary when talking to the file server.

An example of this simple kind of file server is WFS [46], built at Xerox. WFS was written for the Woodstock office system, which had previously run with local discs, and included all the software necessary to transform access to physical disc pages into higher-level functions. The abstraction provided by WFS was of files consisting of pages of 492 bytes each. Each file had a 32-bit file identifier (FID). Each FID could be regarded as the name of a virtual disc. There were commands for creating and deleting files, and reading, writing and deallocating pages of them. Writing a page was done atomically with high probability.

All communication with WFS was connectionless, consisting of a single request packet and a single reply packet. All write actions were idempotent, so could safely be repeated. WFS handled commands serially in a single process, completing each before replying to the client, and thus needed to maintain no state between requests that could not be regenerated from the disc. This strategy also eliminated the possibility of deadlocks.

A simple facility for locking a file was provided, to permit a client to make changes to several pages without others being able to see a half-modified state.

A similar but slightly higher-level system is the Felix file server [13]. This provides similar abstractions to WFS, but also allows atomic updates over several blocks of a file, or over several files (a feature of more use to databases than normal filing systems). It also allows more sophisticated protection of files. The FIDs are chosen from a large name space, making them very difficult to guess. There can be several FIDs for one file, each allowing different access to it.

At the other end of the spectrum of file servers are those which implement the whole of a particular filing system and provide an interface allowing text-string naming of files, directory operations, and sequential access to files. They essentially replace the filing system process in an operating system. Most file servers in commercially available network systems (such as NESTAR and the Logica Ring mentioned above) are of this kind. Such a file server is usually also responsible for user authentication: the user logs in to it, quoting his password, before being allowed access to any files.

At an intermediate level come the "universal" file servers such as the Cambridge file server (CFS) [3, 9] and DFS [44] at Xerox. Both allow random access to arbitrary regions of files, support atomic updates to files, and use nearly connectionless protocols. The CFS is described in chapter 7. It is intended to provide an interface and structure at the highest level common to typical operating system filing systems. It supports two sorts of object:- the file - a vector of 16-bit words, and the index - a vector of pointers to files and indexes. These exist to aid clients in the construction of filing systems, without imposing any particular structure.

The DFS was intended as a basis for database research, so includes support for atomic update of several files. It also differs from the CFS in that it is distributed over several machines. DFS differs from all the other file servers mentioned here in that it is not entirely passive: it will send an unsolicited message to a client if it finds it necessary to break an interlock that client holds, in order to avoid deadlock. Other systems also break locks, but rely on the client discovering this when he finds that his "key" no longer works. A comparison of the DFS and the CFS has been published [26].

Much of the research on file servers is concerned with mechanisms for atomic updates, transactions involving several objects, security, maintaining integrity over crashes of client or server, and implementing a server as several computers on the network. The aspect of file servers which is of most interest in this thesis is the relative advantages of splitting the work of the filing system at different points. To this end, filing systems were built using the CFS in three different ways:- as a virtual disc (chapter 6), as a provider of a files only, and thirdly making full use of its facilities (chapter 7).

A very simple server like WFS makes each client contain all the code needed to implement its filing system. However, the code to communicate with the file server is likely to be small, as the possible requests are simple and few. The code of the file server can also be relatively small and simple, allowing more space for a cache of recently used disc blocks. Much of the processing required is done in the multiple clients rather than in the single file server, making good use of network resources.

A file server which provides a high level interface allows all of the filing system code to be removed from each client, but the amount of communication code is likely to increase substantially due to the large number of different requests which can be made to the file server. The amount of processing done in the server is increased, and so it is more likely that it will be busy while all the clients idly wait for it.

A universal file server such as the CFS is a compromise between these two. The client still has to build his particular filing system on top of the rudimentary one provided, and the interface to the file server has a fairly large number of operations available (about 20 for the CFS). Thus the expected price for flexibility might be an increase in client code over that for either the very simple or the very high-level file server.

McLellan's thesis [23] presents a detailed study of the issues involved in designing a file server. He points out that a disadvantage of a medium or low level file server is that it encourages development of various incompatible filing systems upon it, leading to the unfortunate situation where the file server provides mechanisms for sharing files between systems, but the contents of files cannot easily be shared. He also suggests that it is unclear whether or not the file server's disc page size should be visible to the client. Even if it is not visible, client programs will still tend to be written to use this page size for reasons of efficiency. This was exactly what was done when using the CFS as a virtual disc (see chapter 6).

A study of the effects of distributing the functions of a multi-user operating system has been made by Dellar [7], who wrote a file server based filing system for the CAP machine at Cambridge. He found that in this machine, the amount of filing system code and workspace was slightly reduced, and that as a consequence light-load performance was better than with a local disc, because more free memory led to less swapping. The reduction in storage requirement can be attributed to the fact that the CFS was designed specifically with the CAP in mind [26].

A considerable amount of other work related to the work described below has been done on the Ring in the Computer Laboratory in Cambridge. References to this will be found in the text where the various servers, etc., are described.

# CHAPTER 2

# A PORTABLE OPERATING SYSTEM

This chapter describes the design of TRIPOS, the operating system which was used as the basis for the work described in later chapters. An external view of the system is given - i.e. that seen by a user at a console.

## 2.1 History of TRIPOS

The original idea to write TRIPOS was due to Martin Richards, who began work on it in late 1976. In the spring of 1977, he and Alasdair Scott (a student on the Diploma course at Cambridge) wrote a prototype kernel for a PDP11 computer.

The project gained new momentum in October of that year when four research students joined it: Adrian Aylward, Paul Bond, Richard Evans and the present author. Over the following months, the kernel was redesigned somewhat, and a new version was running (on a PDP11 again) by the end of the year.

Progress in 1978 was very rapid, with the four system tasks all working by March. From this time, work began on implementations for two more computers - the Data General Nova, and the Computer Automation LSI4. During that year, numerous refinements and improvements were made, and many commands and utilities were written.

During 1979, the members of the team began to concentrate on particular aspects of TRIPOS, such as running languages other than BCPL on it [11], and producing a version to run on memory mapped machines [2]. The author investigated the interface of TRIPOS to the Cambridge Data Ring, and designed and implemented the version which uses no peripherals other than the Ring.

TRIPOS has been put to several uses both at Cambridge and elsewhere. As well as employment in its usual form as a general purpose operating system, it has appeared in specialized forms for particular applications - e.g. process control, data collection, and the File Server on the Ring [9]. Implementations now exist for PDP11, Nova, LSI4, General Automation 16/220, IBM Series 1, and Motorola 68000.

## 2.2 Design Aims

The following points are those which the designers of TRIPOS set out to achieve. They arose from the desire to create a system which was both portable, and which did not suffer from features which had caused use of manufacturers' operating systems to be difficult or annoying.

The type of machine for which the system was intended was the sort of minicomputer which could be expected to be the personal computer of the near future: typically with 28K to 64K 16-bit words, and floppy or hard discs with a capacity of one half to twenty megabytes. The main design aims were as follows; they are elaborated in subsequent sections:

- Portability
- Simplicity
- Friendliness
- Single User System
- Multiple Tasks

### Portability

The prime influence on the design of TRIPOS was the desire that it should be easy to move to a new machine. This suggested BCPL [35, 37] as a very suitable language, as it had already been shown to be highly portable, and was available on all the computers in the Laboratory.

BCPL is a language of the Algol family, with good facilities for controlling flow of execution, such as recursive procedures, conditional commands and expressions, and cycle commands. It is unusual in that it has only one data type - the machine word. A BCPL word may be conceptually of any data type, such as integer, character or truth value, and operators exist for all the normal operations on these conceptual types. BCPL's model of the computer store is as a linear array of words with addresses which are consecutive integers. There is an operator ('@') for obtaining the address of the storage location represented by a BCPL variable. A word may hold an address, and an indirection operator ('!') exists for accessing the contents of the storage cell thus referenced. Structures requiring more than one word of storage, such as text strings, may be represented in a single word as a pointer to their real location.

Separately compiled BCPL programs communicate through a region of store called the global vector, which is addressable by all of them. It is similar to a FORTRAN COMMON block. Variables and routine addresses are held in this vector at offsets known at compile time, so linking of segments is a particularly simple process which can be done at loading time; no linkage editor is necessary. In TRIPOS, this property is exploited to make the BCPL library routines resident and available to all programs.

An important aid to making programs easy to move, even when all the machine details cannot be hidden from the high-level language programmer is that machine-dependent quantities should be easily parameterized. BCPL has the concept of manifest constants to achieve this: these are named constants with values known at compile time. Thus machine dependencies can often be isolated into a number of manifest declarations at the top of a program, and the symbolic names used throughout it, rather than writing explicit values in the code.

BCPL is thus a language which provides many aids for the the programmer in organizing his program, but does not get in his way. It enables him to do almost anything he could do in assembly language. It is rather unsuitable for the novice programmer, as its flexibility means that it can do little to stop him writing nonsensical programs, but it is a powerful tool for the more experienced programmer.

The BCPL compiler is more portable than most because it is itself written in BCPL and produces code (OCODE) for a simple hypothetical computer. The only part of the compiler which need be rewritten for a new machine is the code generator which translates OCODE to a particular machine code.

Because the BCPL compiler is straightforward to understand and change, many installations run extended versions of the language. Alterations (even small ones) to an existing language are not usually in the interests of making programs more portable, even though that is surprisingly often presented as the motivation for changes. Hence, the temptation to include extensions for TRIPOS was resisted, and almost unadorned BCPL has been used.* A result of this is that programs written under TRIPOS tend to be relatively easy to move to other BCPL installations.

The characteristics of BCPL make it very attractive to the implementor of operating systems. This is one of the few applications where the linear mode of addressing used by most computer memories must be accessible in the language; BCPL can treat addresses as data objects and do calculations on them. The typelessness simplifies the treatment of various objects in the operating system which are of uncommitted type, such as fields in inter-process messages, and arguments passed between coroutines.

Stoy and Strachey [43] conclude that the ideal language for an operating system should concentrate its resources around the control facilities, and leave matters concerning storage and representation very much up to the programmer. Thus they chose BCPL for OS6. In fact, four of the operating systems mentioned in chapter 1 (OS6, Thoth, UNIX and the Alto system) were written in BCPL or closely related languages.

--------------------

* The exception was the inclusion of the '%' operator for accessing individual bytes in a vector, because its use can simplify and speed up so many programs. However, this language extension is not widely available, and has made necessary tedious changes when moving some programs to non-TRIPOS environments.

As far as possible, the TRIPOS system data structures both in store and on disc are identical on different machines. In particular, all pointers are stored as word addresses. Thus, BCPL code to handle these structures is machine independent.

Making an operating system portable involves designing a virtual machine on which the system is to run, and then implementing it on real computers. There are two principal levels on which this can be done. Firstly, the supervisor and library calls available to user programs should be the same on all machines. However, it is possible to standardize the virtual machine at the machine code level, and run an interpreter to execute compiled code. This has the effect of reducing the size of compiled code, and the amount of assembly language required for a new implementation, but seriously reduces execution speed (by a factor of 10 or 15). OS6 and SOLO use this approach, arguing that the loss of speed is not very serious in a single user system. TRIPOS was planned to be capable of real-time device control, so wants to use the full machine speed and does not use interpreted code.

The area in which machines tend to differ most is in the methods of driving peripherals and handling interrupts. TRIPOS hides these differences in machine code device drivers, which present a standard interface to the rest of the operating system in terms of packets (see chapter 3).

## Simplicity

TRIPOS aims to be easy to explain - both at the external "how to use it" level, and in internal details of how it works. A related intention is that modification should be straightforward. Several features of early versions were later abandoned because they were over-elaborate without offering any great advantage over the simple approach.

Thus fundamental operations such as task scheduling, message passing and store allocation use simple algorithms, and most commands are not packed with complex and rarely-used options.

The whole system runs in one address space, making it simple to pass around references to data structures. No attempt was made to use memory protection, as it is not necessary in a single-user system, it complicates the operating system and slows it down, most of the expected target machines lacked hardware for it, and it does not appear to be possible to find a portable abstraction for the mechanisms found on different machines [12, 5].

**Friendliness**

This is perhaps less easy to quantify than the other objectives. It includes a variety of things, small in themselves, but which together can strongly influence how pleasant a system is to use. Examples are:-

- A natural command language - based on English words and phrases rather than incantations requiring odd characters.
- Type-ahead.
  The console handler is always ready to accept and reflect input from the keyboard, buffering it until requested by the currently selected task. (Others systems will accept typed-ahead lines only sometimes, or only if preceded by a special character, or accept it and direct it to a random task, or accept but do not reflect.)
- Equation of letter cases wherever possible and sensible.
  For example in commands and their arguments, filenames, program source, and editor search strings.

  Most people find lower case more pleasant to use when the terminal supports it. It should not be difficult to work with lower case text if one is forced to use a teletype sometimes.
- Long filenames (30 characters) and a hierarchical filing system.
  This facilitates partitioning of work and storing it in files with names descriptive of their contents. Filenames are recorded in the case used when they were created - this can improve the legibility of directory listings.
- A low tendency to corrupt the disc after a system crash.
  The authors of TRIPOS found this not to be the case with manufacturers' systems they used. The possibility of corruption was kept very low by recreating the block allocation bitmap on each system restart (so it could not suffer cumulative corruption), and by checksumming blocks before writing, and the bitmap before allocating.

**Single User, Multiple Tasks**

TRIPOS was planned as a multi-tasking system for a single user. This influenced several areas, and particularly led to the simple method used to handle errors, with 'aborts' (see chapter 3) freezing the whole system. This places it in the middle of the spectrum of portable operating systems mentioned in chapter 1. Some of these (e.g. MUSS and UNIX) are systems for machines with many users. At the other extreme are the simple single user systems with a fixed number of processes and running on interpreters (e.g. OS6 and SOLO). TRIPOS runs compiled code, permits fast response to external events, and allows dynamic creation and deletion of processes, while taking advantage of the simplifications possible in a system not intended to support more than one user.

The image was always of a single user at the console of his own machine, aware of what he was running within it. The machines used for the original implementations had no memory protection, so the operating system could not be protected from the rampages of an untested program writing all over memory. Hence, it was accepted that rebooting the machine would not be an uncommon event during program development, and that this should be an easy operation.*

With only one user, there was no need for elaborate task scheduling to share out the machine evenly, hence the simple algorithm chosen.

## 2.2.1 The dependence of TRIPOS on BCPL

TRIPOS was designed as an operating system for developing and running programs written in BCPL. Some manifestations of this are mild, such as system data structures being vectors of words with pointers which are word addresses. In other areas, the implementation language is more firmly built-in. Its libraries and run-time system are resident, a global vector and stack are part of the structure of each task, kernel primitives look like

---

* Unfortunately, quick rebooting is to some extent in conflict with the reconstruction of the disc allocation bitmap on each start.

BCPL functions, and DEBUG (see chapter 3) has knowledge of the layout of BCPL code and stack frames.

The simplicity gained from this has probably been worthwhile, and the language chosen is less restricting than most, though less kind to the naive user. BCPL has been shown to be suitable for a wide variety of systems and applications programs.

Other languages such as FORTRAN, Algol68C, Basic and Pascal have been implemented successfully under TRIPOS, though each does of course need its own library, and some contortions are necessary to make them run in a BCPL environment. An account of this work is given by Evans in [11].

### 2.2.2 Use of assembly code

An early decision which had to be taken was that of how much of TRIPOS was to be written in machine code. Superficially, it seems that machine code is something to be avoided as far as possible; the more of it there is, the more there is to be done when moving the system to a new computer. Some portable systems have prided themselves on containing only a very small number of assembler instructions written as such. However, this can be deceptive. Machines are not the same, and the work has to be done somewhere; if it is not in explicit assembler, then it is likely to be in extensions to the high level language, consequently increasing the complexity of the compiler and code generator.

For TRIPOS, it was agreed that a reasonable amount of assembly code would be no great barrier to portability, and would be worthwhile in terms of modularity, speed and size.* Thus, the system has a kernel and device drivers written in machine code. These provide an environment which is defined at the function-call and message-passing levels, and is machine independent. For efficiency, more of the kernel than is strictly necessary has been written in assembly language.

---

* In practice, the total amount of assembly code has been about 1500 words.

## 2.3 TRIPOS as seen by the user

This section tries to indicate how TRIPOS looks to a user at a terminal. An account presented in the form of an annotated console session can be found in [36]. Both that paper and the text below describe TRIPOS on a machine with its own disc and terminal; a corresponding example is given later of the system which runs on single-connection computers on the Ring, where all the peripherals are remote.

TRIPOS is loaded from disc using the normal bootstrap mechanism for the machine. On starting, it issues some messages:-

```
TRIPOS starting
*** Mounting "Alpha-0" for updating on unit 0
User:
```

The first message merely indicates that the system has started to run. The second gives the name of the system disc, the number of the drive on which it is running, and whether it is available for updating or just for reading. The third line is requesting the name of the user's directory. The response might be:-

```
User: brian
>
```

(User input is distinguished from machine output by being in **bold** type.) The directory BRIAN has now been set as the current working directory.* The command language interpreter (CLI) issues its standard prompt "> ", indicating that it is ready to accept a command.

--------------------

* The current working directory is that in which unqualified filenames are looked up.

### 2.3.1 Commands

The format of a command is the command name, possibly followed by some arguments, which may be positional or keyed. There are no built-in commands; a command name is the name of the file from which the compiled code of that command should be loaded. The name is looked up first in the current directory, and if that fails, in the system commands directory. Items within a command are separated by spaces; commands are separated by being on separate lines or by semicolons (;).

For example, the DATE command can be used either with an argument to set the system date, or without one, to print the date:-

```
> date 27-mar-81
> date
Friday 27-Mar-81
>
```

### 2.3.2 Command argument decoding

The majority of commands use the library procedure RDARGS to read their arguments. RDARGS is called with a format string describing the expected arguments. The string consists of a series of keywords, possibly qualified, and separated by commas. Each keyword takes a single argument, delimited either by spaces or double quotes (") - used if the argument contains spaces or semicolons. The argument is separated from the keyword by spaces or an equals sign (=).

Keyword qualifiers come after the keyword, and have the following meanings:

| | |
|---|---|
| /A | This argument must be given (but the keyword is optional) |
| /K | If the argument to this keyword is given, then the keyword must be quoted. |
| /S | The keyword is a switch; it takes no argument. |

These qualifiers may be used in combination: e.g. KEY/A/K means that both keyword and argument are compulsory. If a keyword is unqualified, then both argument and keyword are optional.

Synonyms for a keyword may be included using equals signs; any qualifier applies to all the names for the keyword.

For example, the form of a command ABC could be given as:

```
ABC  "FROM=SOURCE/A,TO,OPT/K,QUIET/S"
```

The command has a compulsory argument which may be positional, or keyed by FROM or SOURCE. It has an optional argument which may be positional or keyed by TO. It has another optional argument OPT, which must be keyed if it is included, and a switch QUIET.

Thus possible ways of using the command ABC would be as follows:

```
ABC FILEA

ABC FILEA FILEB OPT X

ABC SOURCE FILEA TO=FILEB QUIET
```

A useful feature of RDARGS is that if a question mark is typed instead of the command arguments, then it prints out the format string, and waits for the real arguments to be typed. Thus, most commands can be made to indicate what parameters they are expecting. For example, using the assembler:-

```
> asm ?
PROG=FROM/A,CODE=TO,VER/K,LIST/S: from asmprog to asmcode ver log
>
```

### 2.3.3 The system tasks

For simple use of TRIPOS, the user need not be aware that it is a multi-tasking system. By default, all his input is directed to the CLI, which interprets it as commands. In fact, the standard system has four resident tasks, plus one which is there immediately after booting, but which kills itself soon afterwards.

The STATUS command is used to find out what tasks exist, and what they are. If this command is executed soon after starting, the result is as follows:-

```
> status
Task 1: running CLI     Loaded as command: STATUS
Task 2: waiting DEBUG
Task 3: waiting COHAND
Task 4: waiting DISC
Task 5: waiting RESTART
>
```

The name printed for each task is that of its principal code section.

| | |
|---|---|
| CLI | is the command language interpreter |
| DEBUG | is the interactive debugging aid |
| COHAND | is the console handler |
| DISC | is the file handler |
| RESTART | is the filing system's restart task |

After a few tens of seconds (depending on the speed of the disc and the number of files and directories on it), the fifth task finishes and deletes itself. A call of STATUS then gives only the first four lines of the output above.

The user can direct typed input to tasks other than the CLI. This facility is described in the section on the console handler.

## Console Handler

The console handler task is described first, as it is the one which the user encounters first, and because some of its features should be mentioned before showing how CLIs and DEBUG are used.

The console handler provides input line reflection and editing, input line buffering, management of output lines, the ability to set task attention flags, and input characters not available directly from the keyboard.

It is always ready to accept input, and a partially-typed input line holds up output. If typing of an input line is started while an output line is being printed, then reflection is delayed until the output line is complete. Corrections to an input line may be made by rubbing out characters, or by deleting the whole line. A line is not transmitted to the selected task until it is terminated by one of the characters 'carriage return' or 'escape'.*

Input lines may be typed before the current task is ready to read them. In this case, they are buffered until requested.

Output lines are normally accepted from any task, and are printed in the order received.

The character '@' is treated specially. Sequences starting with this character are used for two purposes. Firstly, some such sequences are used for input of characters that may not be available on all terminals - e.g. '@v' for '¦' (vertical bar). An '@' character is put in as '@@'.

Secondly, other sequences are used to give commands to the console handler itself: e.g. '@L' to rub out the current line, '@F' to free all typed-ahead input lines.

--------------------

* 'Escape' sends the line, but nothing is reflected to the terminal. Hence, the cursor remains in the same position.

Input lines are directed to the currently selected task. This is initially task 1, the CLI, but can be altered by means of the escape sequence @Snn, where 'nn' is the number of the new target task. For example, @S02 selects task 2, the debug task.

Whichever task is selected for input in the above manner, output is allowed from any task. An alternative method of task selection is by means of the escape sequence @Tnn; this acts like @Snn, except that only task nn is allowed to write to the console. Output from other tasks is held up until it is explicitly permitted again.

Some of the task attention flags (see chapter 3) may be set in the current task directly from the console, providing a quick way of signalling to that task. The characters control-B to control-E set flags 1, 2, 4 and 8 respectively. Most commands respond to control-B by quitting immediately; the CLI tests the control-C flag between commands, giving a clean way of aborting a command sequence.

## Command Language Interpreter

The command language interpreter is the task to which console input is normally sent. It executes commands serially, with each command running as a coroutine (see chapter 3) of the CLI. This means that the command has its own stack, with dynamically chosen size. If the CLI were to call commands as subroutines, its root stack would have to be large enough to accomodate the command with the largest stack requirement, and this space would remain allocated even when no command was running.

Since coroutines within a task share the same global vector, commands run in the environment of the CLI, and can use the input and output streams which it provides.

The command name (i.e. the first word in a command) is treated as a filename, which is looked up first in the current working directory. If the name is not found there, or is found but does not refer to a file containing an object module*, then the name is looked up in the system command directory (SYS:C by default; it can be altered).

The CLI can run interactively or non-interactively (e.g. after the RUN and C commands described below). In non-interactive mode, commands come from a file or store buffer; when that is exhausted, the CLI reverts to interactive console input (if a stream is present), or commits suicide.

## DEBUG

DEBUG is a general debugger for use under TRIPOS. It allows inspection and alteration of any store location, and has knowledge of TRIPOS structures and BCPL routine and stack frame layout, so can be used to investigate TRIPOS and programs running under it. Facilities include printing a backtrace of routine calls within a task, giving the source names of the routines, setting breakpoints, and selectively holding and releasing tasks.

DEBUG has two modes of operation, called <u>task mode</u> and <u>stand-alone mode</u>.

Task mode is the normal one, when the rest of TRIPOS is running, and DEBUG is just a task within it (conventionally task 2). The user can elect to talk to DEBUG at any time by using the console escape sequence @S02; he can then inspect other tasks while they are running, perhaps to see how variables within a task are changing. Any task may be held, so that it can be seen in a static state, with the rest of the system left running.

There are four ways in which DEBUG can be entered in stand-alone mode:-

(i) After an abort

(ii) When a breakpoint is encountered

--------------------

* This strategy is included because the names of several standard commands clash with names which may be desirable for sub-directories or text files. For example, many directories contain sub-directories called "bcpl" and "asm".

(iii) If a task abort is forced by typing control-A

(iv) By jumping to its stand-alone entry point using the machine's handswitches or console.

In stand-alone mode, the rest of the system is frozen, interrupts are disabled, and all communication with the terminal is performed using the routines SARDCH and SAWRCH in MLIB. All the normal DEBUG commands are available, and, unless entry was by method (iv), running of TRIPOS can be continued with or without holding the task which aborted or encountered a breakpoint.

Calling stand-alone DEBUG from the handswitches is usually used only after a severe crash, when some vital area of store has been overwritten and TRIPOS cannot run. It relies on very little of the system being intact in order to work, so can often be used to gain some post-mortem information on what happened.

To save on use of store by resident tasks, a small version of DEBUG can be installed instead of the full one. This provides only basic handling of aborts, but does have a command for loading the full version, if required. DEBUG is also available as a command.

## File handler; Filing system

The file handler task provides an interface to the disc in terms of named directories and files. Each directory may contain an arbitrary number of files and other directories, giving a tree-structured filing system. It is a strict tree - for simplicity, a file or directory may not be retained in more than one directory, so there are no loops or shared subtrees. There is a single root directory for the disc; conventionally, this holds the main system directories, and users' personal directories.

Filenames consist of a series of components separated by dots describing the route to be taken from the current working directory to reach the file. Each component is a sequence of letters, digits and/or '-' characters, and may be quite long (up to 30 characters). Thus, the file "prog" in the sub-directory "bcpl" of the current directory could be typed with the following command:-

```
type bcpl.prog
```

It is possible to indicate that a path name should start from the root directory of the disc, rather than the current directory, by starting it with the device name ':'. For example, the standard BCPL library header is kept in the file "libhdr" in the directory "g" in the root directory. The command

```
type :g.libhdr
```

would type it regardless of what directory happened to be set at the time.

From BCPL, files are opened in the conventional way with library routines FINDINPUT and FINDOUTPUT. Data may be transferred on the resulting streams either a character at a time, using RDCH or WRCH, or in blocks of words, using READWORDS or WRITEWORDS.

The file handler provides directory operations to delete and rename files and directories, and to create new directories.

The filing system combines simplicity with a general freedom from irritating restrictions. There is only one file type - essentially a vector of bytes or words which are written and read back without interpretation, meaning that there is no need to treat binary data specially. There is no limit on the length of a file, or on the number of entries in a directory (other, of course, than that imposed by the capacity of the disc). Although the length of filenames is limited, the limit is sufficiently large that it does not discourage the use of filenames which describe the contents. The ability to have an arbitrary tree of directories means that related files can be tidily grouped.

## File Handler Restart

The file handler restart task starts running when the system is booted. It serves two purposes: to create a bitmap for the file handler, showing which disc blocks are allocated, and to do some checking of the integrity of the disc.*

While restart is running, the file handler does not allow any operations which would involve writing to the disc. There are two reasons for this. Firstly, it cannot write to files because it does not yet have the bitmap to tell it which disc blocks can be allocated. Secondly, it should not alter any directory until all of them have been checked. However, it will allow anything which only reads from the disc. Thus, when the system first starts up, the user does not have to wait for restart to finish before he can start to use it. He can load commands and other programs, set the date and time, type files, and examine directories. When restart finishes, it issues messages to announce this and to indicate the amount of disc used:

```
*** "ALPHA-0" validated
*** 22 directories, 157 files, 77% utilization
```

The restart task then kills itself. The system is now fully started, files can be written and directories altered.

An extra job done by restart is that of setting the system date and time if they are not set explicitly. As it scans all the files and directories, it notes the latest date and time of creation of any object on the disc. If, when it is about to finish, the date and time have not been set, it sets them to this value. This means that time does tend to move forward monotonically when the machine is rebooted. Even though the time will not be accurate, it will be possible to see in which order files were created, for instance, and that is a property which usually matters more than the exact absolute times.

--------------------

* It is not an exhaustive check of the disc; each directory and the first and last blocks of every file are inspected. This will catch most forms of disc corruption, and is much cheaper than a full scan (in both store and time).

This does introduce a potential problem: if the system date accidentally gets set to a future value, then there is a danger of this getting recorded as the creation date of many objects, and so being set again on the next restart. In practice, this has rarely caused difficulty, and the convenience of the automatic date setting is worth the potential danger.

### 2.3.4 Command sequences; the C command

As well as reading commands from a console stream, a CLI can take commands from a file. This enables a sequence of commands to be stored, and then executed without typing them all in again. The CLI is in non-interactive mode whilst reading from a file; in this mode it does not issue prompts. The mode can be tested by commands which may chose to behave differently according to whether or not they can expect interactive input.

The normal means of executing a command sequence file is to give the filename as argument to the C command. For example:-

```
> input to seq
echo hello
echo there!
/*
> c seq
hello
there!
>
```

(ECHO is a simple command which just prints its argument.)

A further feature of the C command is that it will perform parameter substitutions in the command stream before execution, making it possible to create more general command sequences. This facility is used by beginning the file with a directive line starting with ".K" and containing a string in RDARGS format describing the expected arguments. The value of an argument can be substituted for the name by enclosing the name in angle brackets '<' and '>'. A default value may also be given inside the brackets, separated from the name by a '$'.

For example, the following is a command sequence file which compiles a specified program from the sub-directory "bcpl", putting the object code in a file of the same name in sub-directory "obj" by default, or elsewhere if a name is given. It also prints a message, verifying what it is doing.

```
.k name/a,to/k
echo "Compiling <name>"
bcpl bcpl.<name> to <to$obj.<name>>
```

Suppose this were kept in the file "comp"; it could be used as follows:-

```
c comp prog

c comp from prog

c comp prog to :c.prog
```

Several commands exist designed for use within command sequences - e.g. to execute commands conditionally on the value of parameters or other conditions, and to terminate the sequence if any command gives a return code exceeding a certain level.

The "control-C" break flag (= task flag 2) is tested between commands, and can be used to stop a command sequence cleanly at the next command boundary.

### 2.3.5 The RUN command

An alternative method of using a CLI non-interactively is provided by the RUN command. This creates a new CLI task which then executes the remainder of the the RUN command line (so a RUN command cannot be terminated by semicolon). After executing that line, the extra CLI task goes away.

This is an easy method of running any command, or group of commands, asynchronously to the main CLI, leaving that free to accept further commands from the user. The created CLI runs at low priority, so RUN effectively creates a background job.

For example, printing a file in the background, and using STATUS to look at the tasks:-

```
> run print :g.libhdr; echo "printing finished"
> status
Task 1: running CLI     Loaded as command: STATUS
Task 2: waiting DEBUG
Task 3: waiting COHAND
Task 4: waiting DISC
Task 5: interrupted CLI     Loaded as command: PRINT
>
```

After a while, printing finishes, the echo message comes out, and STATUS shows that the task has gone away again:-

```
> printing finished
status
Task 1: running CLI     Loaded as command: STATUS
Task 2: waiting DEBUG
Task 3: waiting COHAND
Task 4: waiting DISC
>
```

RUN copies some environment from the current CLI into the new one. In particular, it uses the same current and command directories, the same command stacksize, and the same console.

## 2.3.6 Multiple CLIs; the NEWCLI command

It is possible to have more than one interactive command program available under TRIPOS. The command NEWCLI creates another CLI task, which announces that it has started, then waits for input from the console. The user chooses which CLI he wishes to talk to by using the console handler's "@Snn" or "@Tnn" task selection mechanism.

This facility provides very simply much of the convenience of being able to log on more than once to a multi-user system. It means that while one CLI is tied up in a long interactive command, such as the editor, others can still be available for inspecting files and directories, printing, and so on. Other operating systems which allow the running of several interactive commands at once do not allow the user to direct his input (e.g. SOLO).

Instead, each time an input line is expected, a prompt is printed to indicate which program will receive it. It is thus unsafe ever to type ahead in such systems, because one cannot be sure to which program the input will go. It is also impossible to talk exclusively to one of the running programs and ignore the others.

For instance, consider referring to a header file while editing a program. The user decides after starting that he wants to work with two CLIs. He makes use of the fact that CLI prompts are printed out by the BCPL routine WRITEF, and that the first argument is the task number, to give his two CLIs distinct prompts:-

```
> prompt "%n> "
1> newcli
1> New CLI task 5
5>
```

He carries on working in CLI 1, and later edits his file "myprog". While editing, he goes to the other CLI to type another file, in order to check the name of a manifest constant.

```
1> edit myprog                    [issue EDIT command to CLI 1
Edit ready                        [message from editor
l/factor/                         [EDIT command to locate string
127.
    x := size * wadgetfactor      [found on line 127
@s05                              [direct input to task 5
5> type myhdr                     [type a file in CLI 5
MANIFEST
    $(
    blocksize    = 10
    maxchars     = 19
    widgetfactor= 42
    $)
5> @s01                           [back to task 1 (still in EDIT)
e/wa/wi/                          [exchange strings
127.
    x := size * widgetfactor
w                                 [windup EDIT
1>
```

NEWCLI copies some CLI environment in the same way as RUN. An extra CLI created by NEWCLI can be removed by causing it to execute the ENDCLI command.

# CHAPTER 3

# THE STRUCTURE OF A PORTABLE OPERATING SYSTEM

The previous chapter gave the design aims and history of TRIPOS, with a description of some of the facilities offered, and what it is like to use. This chapter gives an account of the internal structure of the operating system, and the reasons behind the design decisions made. The system is described in some detail. Portability of an operating system is achieved by constructing it in such a way as to conceal low-level hardware features, and it is necessary to go into some detail to explain properly how this has been done. The description also provides the background necessary for later chapters which describe how the same structures and primitives were used to build a network-based version of the operating system. The simplicity of TRIPOS makes it possible to include a fairly complete account of its structure.

As many of the areas considered below are interrelated, it has proved necessary to mention some things before they have been fully explained. The diagram gives an overview of the components of TRIPOS, indicating with arrows which modules call which others.

The linking together of a system is explained, and there is a discussion of what is involved in producing an implementation of TRIPOS for a new computer, and the portability problems which have been experienced. Finally, some retrospective comments are made on the design of TRIPOS in the light of several years of use, and it is compared with other portable and single-user operating systems.

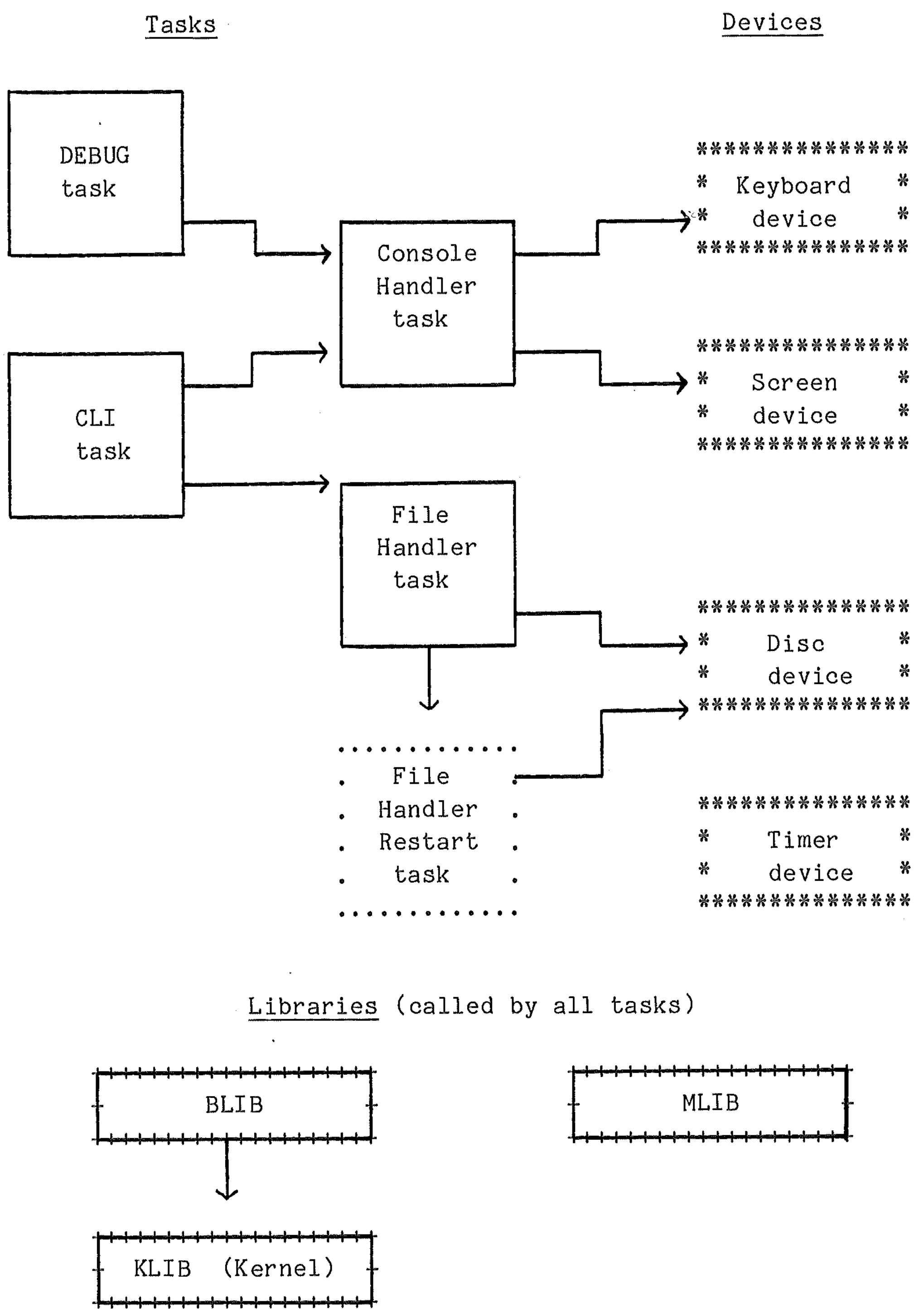


Fig. 1: TRIPOS Tasks, Devices, and Libraries

## 3.1 The Kernel

The TRIPOS kernel is a collection of BCPL-callable routines and other pieces of code which provide a machine-independent interface to the BCPL-written parts of the system. It is written in assembler, and is the largest assembler component of the system. It contains both code which is inherently very machine dependent and so is most conveniently written in assembler (e.g. code to start up the system, and to save and restore machine state on task switches and interrupts), and also code which runs outside a BCPL environment and/or is executed very frequently so needs the speed of machine code (e.g. the task scheduler). The kernel also contains some system data structures, such as the root node.

The kernel contains the following components:-

(i) The Root Node.
This is the base of all the system data structures.

(ii) The Task Selector.
The low-level scheduler, which decides which task to run next whenever one ceases executing.

(iii) The library of kernel primitives.
This is a set of BCPL-callable functions - the supervisor calls of TRIPOS.

(iv) The Idle Task. This is a dummy task (it does not have a BCPL global vector or stack) with lowest priority. Its TCB and code (usually only one instruction) are built into the kernel.

(v) The Clock Interrupt Routine.
This both maintains the date and time fields in the root node, and implements the timer device.

(vi) Entries to stand-alone DEBUG. These include the lowest level handling of aborts and breakpoints, and the entry available from the handswitches.

(vii) System start-up code.
TRIPOS starts executing in the kernel. Most of the actions performed are machine-dependent, but include resetting all peripherals, initializing the TRIPOS devices, starting the clock, and starting the initial task.

(viii) Interrupt and trap vectors.
On machines with vectored interrupts, the vectors are all initially set to point to kernel code which will enter stand-alone DEBUG to produce an error message.

The kernel usually consists of one absolute and one relocatable code section. The absolute section contains the parts which must be loaded into particular locations (such as interrupt vectors), and things which are put at fixed addresses so that they are easy to find (e.g. the root node, and the handswitch entry to DEBUG). The relocatable section contains all the rest of the kernel.

### 3.1.1 Kernel Primitives

The kernel primitives are a set of routines which can be called from BCPL, and provide the supervisor calls of TRIPOS, thus implementing the portable virtual machine environment. They are written in assembly code for a mixture of reasons. Some need close contact with the underlying hardware, to access the registers, to call machine code subroutines in the device drivers, or to save a BCPL environment and call the task scheduler. Others are in machine code because they are called very frequently and the overall efficiency of the system depends on them. Nearly all of the primitives run with interrupts disabled, as they manipulate system data structures, though none disables interrupts for very long at a time. It was a policy decision not to provide BCPL-callable routines to disable and enable interrupts, as their careless use could cause havoc in the rest of the system (deadlock, or loss of device interrupts), and as they are unnecessary anyway. A task can execute indivisibly with respect to other tasks simply by changing its priority to the maximum possible.

The kernel primitive functions have been carefully chosen, as the whole character of the operating system depends upon them. TRIPOS is an "open" system in the sense that above the level of the kernel there is no distinction between user and system programs; this is in keeping with the recommendation of Lampson and Sproull [20] that both high and low level abstractions should be made available to the user. Thus the primitives reflect the design aims listed in chapter 2 by being simple to explain and use, and capable of straightforward implementation on a variety of machines. The sixteen kernel primitives fall into six groups, concerned with storage allocation, tasks, devices, messages, task flags, and program linking.

A summary is given here; most are described in more detail in the appropriate sections on storage allocation, tasks, devices and packets.

The convention for results of primitive calls is that the value passed back is zero if the function fails, non-zero otherwise. When zero is returned, the global variable RESULT2 holds a fault code giving the reason for failure. In some cases, a non-zero result contains further information from the call (e.g. GETVEC returns the address of the vector it has allocated).

## Store Allocation Primitives

The form of these is the same as corresponding routines in existing BCPL libraries.

```
VECTOR   := GETVEC(UPPERBOUND)
            Allocate a block of store

            FREEVEC(VECTOR)
            Free a block of store obtained by GETVEC
```

## Task Primitives

The fundamental routines required here are those to create and delete tasks. A new task is characterized by the list of segments which comprise its code, the size allocated to its BCPL stack, and its priority. A new task has no special relationship with its creator. This is in keeping with the lack of distinction between user and system. A hierarchical scheme (e.g. as in Thoth) is unnecessary when there is only one address space and accounting for CPU time is not of interest.

It is useful to be able to alter task priorities dynamically, though this facility is principally used by a task wishing to claim the maximum priority in order to run indivisibly. The facility for holding and releasing tasks is primarily an aid to debugging; these primitives are usually called in direct response to user commands. An abort routine is a standard feature of BCPL run-time systems. In TRIPOS, its action is to enter the debugger in stand-alone mode.

```
TASKID   := CREATETASK(SEGMENTLIST, STACKSIZE, PRIORITY)
            Create a new task

RESULT   := DELETETASK(TASKID)
            Delete a task

RESULT   := CHANGEPRI(TASKID, NEWPRIORITY)
            Change the priority of a task

RESULT   := HOLD(TASKID)
            Hold a task (prevent it from running)

RESULT   := RELEASE(TASKID)
            Release a held task

            ABORT(CODE, ARG)
            Abort the current task with error code CODE.
            ARG can be used to pass more information.
```

**Device Primitives**

The only primitives required specifically for device drivers are those to create and destroy them. All other communication with devices is done via the message system. The only information required to create a device driver is a pointer to its control block (and implicitly to its code).

```
DEVID    := CREATEDEV(DCB)
            Create a new device

RESULT   := DELETEDEV(DEVID)
            Delete a device
```

## Message Primitives

TRIPOS is a system based on message passing rather than sharing of data under monitors [16]. Lauer and Needham [21] suggest that the two schemes are duals of each other, and that the resulting complexity is independent of which is chosen. However, Frank and Theaker [12] argue that message passing is better for preserving independence of modules. No justification will be given for choosing to base TRIPOS on messages, except that it seems more amenable to distribution over a network. Note, however, that the kernel primitives are themselves monitors, as they disable interrupts to make their execution indivisible. Thus both schemes are represented in TRIPOS.

The important thing was to design a simple, flexible communication scheme. Thus messages are unrestricted in length and format, except for two words used by the system. (A conventional format for the first few words is defined, and almost universally used.) The message sending primitive does not block execution of the calling task, so the programmer has the choice of whether or not immediately to await the reply. There is a uniform message interface to tasks, peripheral devices and the system clock. A third routine allows a previously sent message to be forcibly retrieved.

```
RESULT   := QPKT(PACKET)
            Send a packet (to a task, device or clock)

PACKET   := TASKWAIT()
            Receive the head packet on the work queue, or
            suspend the task to wait for one to arrive.

RESULTID := DQPKT(ID, PACKET)
            Dequeue a packet from the work queue of a task,
            device, or clock (i.e. cancel a message)
```

**Task Flag Primitives**

These provide a cheap signalling system between tasks, and are usually used in direct response to user input to signal to running programs.

```
RESULT   := SETFLAGS(TASKID, MASK)
            Set one or more task flags

RESULT   := TESTFLAGS(MASK)
            Test and clear specified flags for this task
```

**Global Vector Initialization Primitive**

BCPL compiled code contains information which enables the run-time system to initialize global vector slots to the entry point addresses of routines. This is the only "link editing" required to combine separately compiled programs. The routine to do this is used by the kernel when a new task starts up but is made generally available for programs which wish to overlay code segments.

```
RESULT   := GLOBIN(SEGMENT)
            Write into the global vector the addresses
            of global routines and labels defined in SEGMENT
```

### 3.1.2 Use of machine facilities

TRIPOS is designed for minicomputers of a size such that they are suitable for use by one person - typically with memory sizes of from 32K to 64K 16-bit words. Most of these machines are broadly similar, in that their store appears as a single vector in which words or bytes can be addressed, they have a few central registers, a device interrupt mechanism, and a clock. However, the low-level details can differ considerably. This section discusses the use made by TRIPOS of the features peculiar to some of the machines on which it has been implemented. This includes processor priority levels, processor states, register types, hardware stacks, and address mapping.

**Priority Levels**

Some minicomputers allow selective enabling of interrupts of different priorities. For example, the PDP11 has 8 priority levels [0 -> 7], and each device has an associated priority. The processor is always in one of the eight levels when running; only interrupts from devices of higher levels are permitted. This allows the writing of interrupt routines which can themselves be interrupted by devices requiring more rapid attention.

The PDP11 version of TRIPOS makes use of only the two extreme levels:- 0 (all interrupts enabled) and 7 (all interrupts disabled), for the following reasons:-

- No piece of code runs with interrupts disabled for long anyway, so allowing nesting of interrupt routines would not give substantial speed increases. Most device interrupt routines consist of little more than returning one packet to a task, and initiating the action requested by the next. Kernel primitives generally need to disable interrupts completely, and most are fairly short.*
- Full use of the levels would need rather more complex state saving to determine what to do when returning from the outermost of a set of nested interrupts - i.e. which was the highest priority task to which a packet had been sent.
- The pieces of code which disable interrupts for longest are in general kernel primitives rather than interrupt routines. In most cases it is not permissible to allow a task swap, or indeed any other code to run, during execution of a primitive. It would also be awkward to return from an interrupt routine nested within a primitive, complete execution of the latter, and then enter the scheduler rather than return to the caller.

  If most of the disabling of interrupts is in kernel primitives, and they must disable all, then elaborate use of priority levels is unlikely to improve performance.

--------------------

* Some kernel primitives could run for a long time if there were a large number of tasks, or a long packet queue - e.g. CHANGEPRI, DQPKT. This is not normally the case. GETVEC can spend a long time scanning the block list, but deliberately enables interrupts briefly in the search loop. It seems reasonable to treat GETVEC as a special case and to employ this trick. The alternative method would be to have a special task for managing free store, which would not need to disable interrupts at all. However, this would make getting a vector rather more expensive, slowing the whole system. It would also make it difficult to obtain vectors when starting up a task (before its stack and global vector exist).

- Many minicomputers lack this feature; it is desirable to keep various kernels as similar as possible.

## Processor States

Of the machines on which TRIPOS was originally implemented, only the PDP11s had special processor states - e.g. a supervisor state in which privileged instructions are available, and in which memory protection may be overridden.

Supervisor state can be used to protect an operating system from user programs, by making it impossible for them to write to its instruction or data store - i.e. to damage it in any way. Instructions to control I/O operations are available only in supervisor state, forcing all handling of peripherals to be done via the operating system, which can check them.

It was decided not to attempt to use such facilities in TRIPOS, as the gain did not justify the increased complexity:

- Many minicomputers have only one state anyway
- Complete protection of the operating system is not usually necessary in a single-user system (though it can be, of course, a nice thing to have)

## Memory Protection / Memory Management

The simplicity of TRIPOS is in part due to the fact that the whole memory is available to every task and device, and that addresses are global. Thus, packets, I/O buffers, etc., can all be passed by reference, which is much cheaper than copying them. The assumption that this can be done is built into TRIPOS fairly firmly in a number of places, making it very difficult to make a modified version of the system to take advantage of any memory mapping or protection features that may be available on a particular computer. Most of the computers on which TRIPOS has been implemented have a global address space, and no way of restricting access to any regions of it.

The justification for ignoring these facilities is (as with other simplifications described above) that the gain is not worth the loss of simplicity and efficiency in a single-user system. It is also worth noting that those portable systems which do make use of memory management hardware (see chapter 1) have not been able to find a generally portable abstraction of it, so have had to have several versions of the relevant code.

A system based on the original TRIPOS, but which is able to take advantage of memory mapping hardware, has been implemented by Aylward [2].

## 3.2 Tasks

TRIPOS tasks are described here in terms of their components, scheduling rule, and how they are activated. The creation and deletion of tasks, and the inter-task communication mechanism are explained elsewhere in this chapter.

### 3.2.1 Components of a Task

A TRIPOS task consists of the following components:-

(i) A Task Control Block (see below)

(ii) A Stack.
This is the run-time stack for the task. If the task contains more than one coroutine (see "Coroutines in TRIPOS" below), it is the stack of the root coroutine.

(iii) A BCPL Global Vector.
In TRIPOS, there is one global vector per task. Coroutines within a task share the global vector.

(iv) A Segment List.
This is a small vector, each entry of which points to the start of a code segment (a linked list of one or more BCPL code sections). It defines the code of the task, and enables sharing of code modules by different tasks. By convention, the first two entries in each segment list point to the standard libraries MLIB and KLIB (linked together), and BLIB.

### 3.2.2 Task Control Blocks

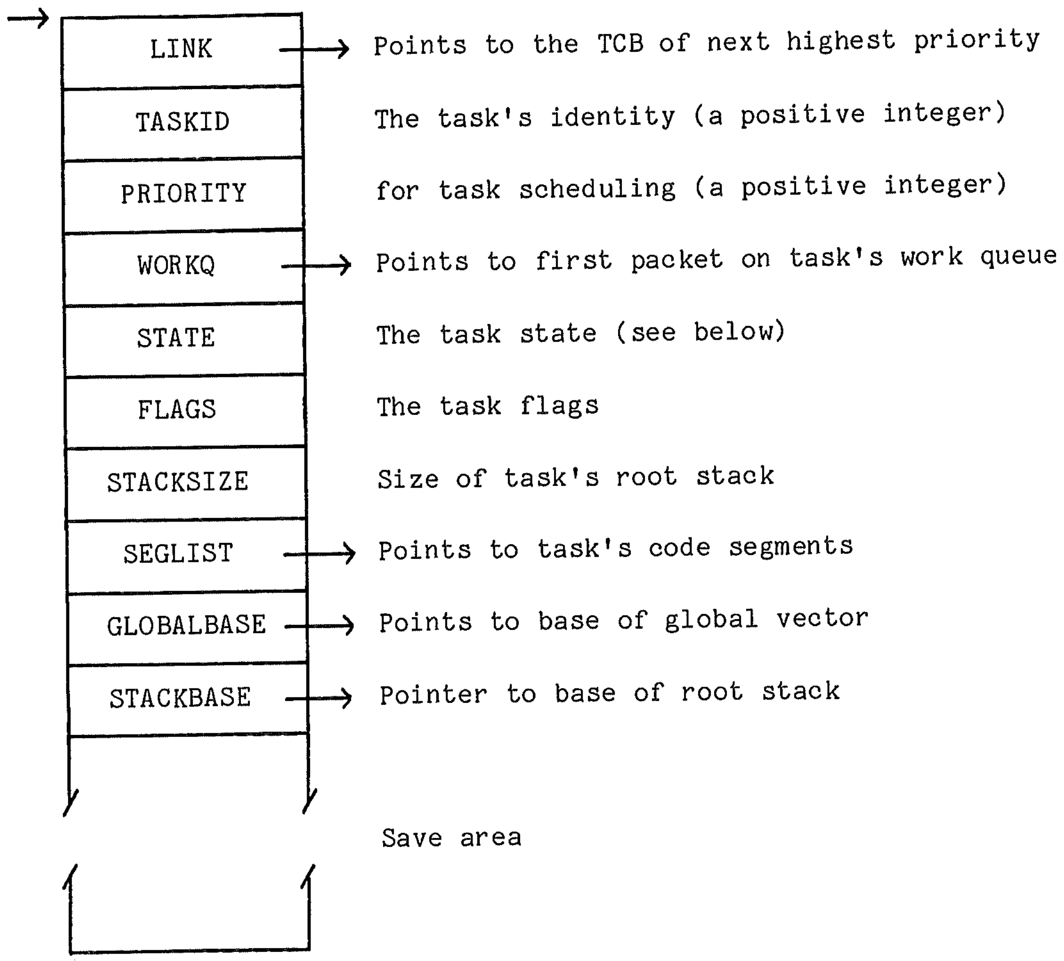

LINK
Points to the TCB of next highest priority
TASKID
The task's identity (a positive integer)
PRIORITY
for task scheduling (a positive integer)
WORKQ
Points to first packet on task's work queue
STATE
The task state (see below)
FLAGS
The task flags
STACKSIZE
Size of task's root stack
SEGLIST
Points to task's code segments
GLOBALBASE
Points to base of global vector
STACKBASE
Pointer to base of root stack
Save area

Fig. 2: Task Control Block

Each task in the system has a task control block (TCB) which contains information relating to the task. There are two parts to a TCB. The first part is machine independent, and contains information used by the operating system for controlling the task. The second part contains the save area used to hold the machine registers, program counter, and processor status when a task suspends itself or is interrupted. Its format is necessarily machine dependent.

**The Task State**

This is held in the least significant 4 bits of the state field. All the remaining bits are zero. The significance of each bit is as follows:-

0001: Packet bit.
If this bit is set, the task has at least one packet on its work queue. If clear, then the work queue is empty.

0010: Held bit.
This bit is set when the task is in held state. It means that the task will not be selected for running, even though it might otherwise be eligible. Its primary purpose is as a debugging aid.

0100: Wait bit.
This is set when the task has called TASKWAIT, and is waiting for a packet to arrive. The task will not run while its work queue is empty.

1000: Interrupted bit.
When this bit is set, the task has been interrupted. The task will run again when the interrupt service routine is complete, and any higher priority tasks it may have activated are once again held up.

1100: Dead state.
Note that this bit pattern would otherwise be invalid, as a waiting task could not be interrupted, and an interrupted task could not call TASKWAIT. It indicates that the task is dead or dormant, with no BCPL stack or global vector. The only other TCB fields which are valid are the LINK, PRIORITY, STACKSIZE, and SEGLIST.

All of the 16 possible bit patterns are valid. This means that the the task selector can rapidly decide how to deal with a task, by using the state to index a table of routine addresses.

**Flags**

The bits of this word are available as flags which may be set by other tasks. They are set and tested by using the kernel primitives SETFLAGS and TESTFLAGS. They are useful as a cheap signal between tasks, and are used to implement console 'break'.

### 3.2.3 Task Scheduling; The TCB list

As well as being addressed by the task table, the TCBs are linked into a chain, for the benefit of the scheduler. The TCBLIST field of the root node points to the TCB of highest priority, whose link field points to the TCB of next highest priority. The chain links all the TCBs in order of decreasing priority, ending with that of the idle task, which has a priority of zero, and a link of zero. No two tasks may have the same priority, so the correct chain order is well defined.

The task scheduler is a kernel routine which is entered with interrupts disabled whenever something occurs which may require a change of currently running task. The reasons this can occur are:-

(i) A task calls TASKWAIT when its work queue is empty. It will be suspended until a packet arrives for it.

(ii) A task sends a packet to another task of higher priority. If the other task had stopped to wait for a packet, then this makes it free to run.

(iii) A device interrupt causes a packet to be returned to a task whose priority is higher than that of the one running when the interrupt occurred.

(iv) A task HOLDs itself.

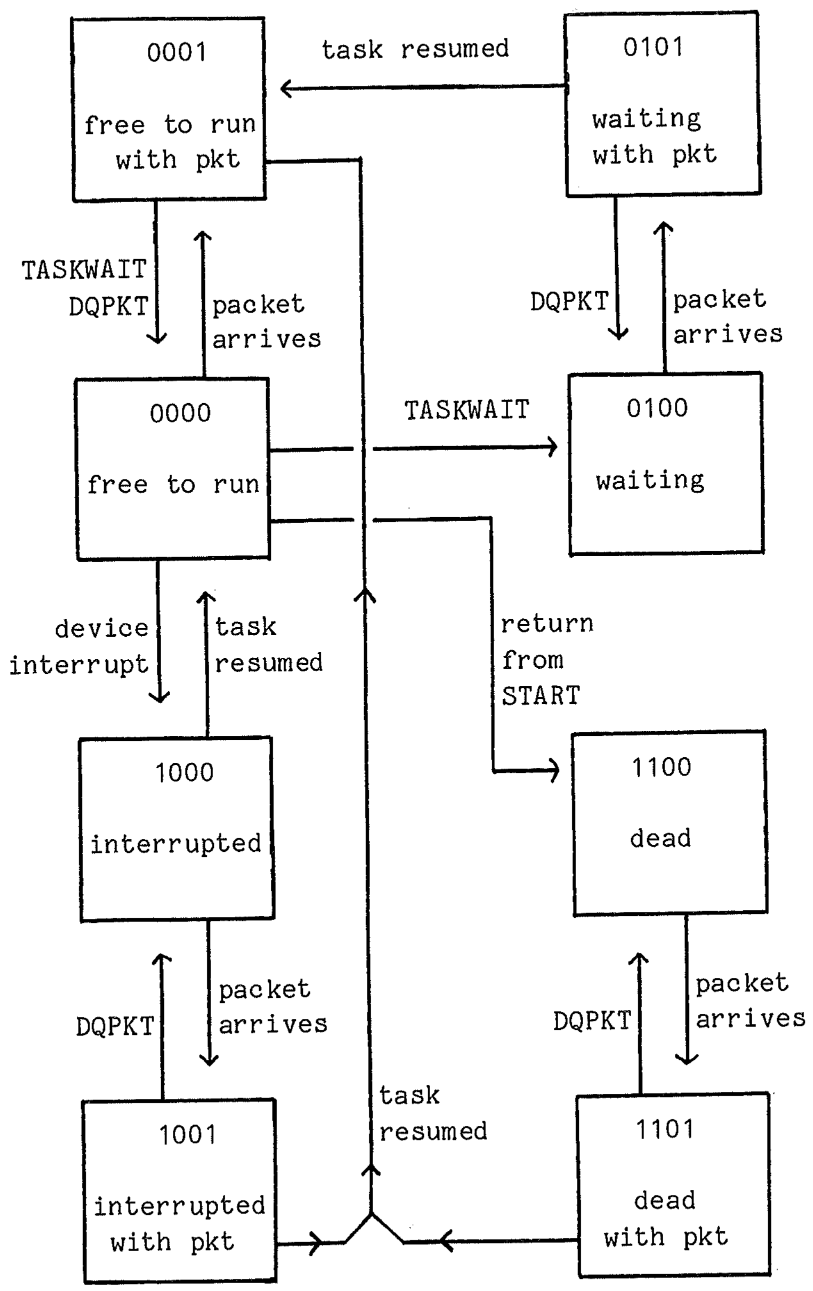
(v) A task RELEASEs one of higher priority than itself.

(vi) A task calls CHANGEPRI to reduce its own priority, or to cause a task which had lower priority than itself to have higher priority.

(vii) A task deletes itself.

The diagram shows the task states and the possible transitions between them.

The scheduling rule is very simple: the task currently running is the one of highest priority which is free to run. After any of the events above, the scheduler is told which is the highest priority task which might be eligible to run. It starts at that position in the chain and works its way down until it finds one which is able to run. The presence of the idle task ensures that there will always be at least one able to run.



Only half of the possible task states are shown: corresponding to each state in this diagram is a state with the 'held' bit (0010) set. Entry to these states is caused by HOLD, exit by RELEASE. A task will not run while in any state with the 'held' bit set.

Fig. 3: Task State Transitions

The search for a task to run is done very efficiently. From each TCB in turn, the task state is extracted. This 4-bit number is used to index into a 16-way jump table, which either causes the scheduler to be re-entered to try the next TCB down, or enters the correct code to resume or activate the task, according to the state it was in.

It is important that switching tasks should be cheap, as TRIPOS relies on it being so.

### 3.2.4 Task activation

Until a task receives its first packet, it is in 'dead' state, and has no space allocated for its stack or global vector. When the first packet arrives, the kernel activates the task as follows:-

(i) It gets an area of store for the stack (of size given in the TCB), and records its address in the TCB.

(ii) It scans the global initialization tables (at the end of each code section of the task), in order to find out the highest offset in the global vector referenced by any section. It then gets space for a global vector of this size, records its address in the TCB, and applies GLOBIN to each segment in turn to initialize the addresses of all routines and labels declared as global.

(iii) Finally, it starts the task by calling global 1 (START).

## 3.3 Devices

A device consists of a device control block (DCB) which points to a device driver containing the code. In practice, the assembled code of devices is often stored as two concatenated object modules. The standard loading routine LOADSEG (in BLIB) scatter loads concatenated modules as a linked list, so the required link between DCB and driver is established automatically. As there are no backward pointers, a driver may be shared by several DCBs.

As the way in which peripherals are controlled varies considerably between machines and types of device, only the first few words of DCBs and drivers have a machine independent format. Also, as most of the fields are accessed only from the machine code written drivers and kernel primitives, all pointers except the link and work queue are machine addresses.

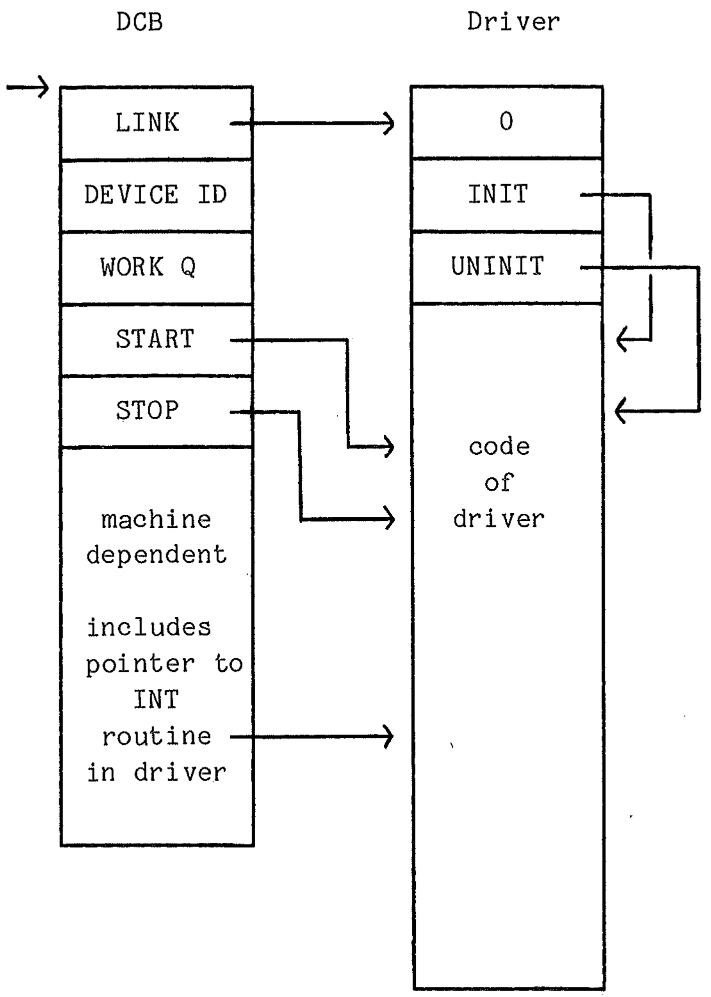


Fig. 4: Device Control Block and Driver

A driver contains five machine code subroutines called from outside:

INIT: Called by CREATEDEV when the device is created. It sets up the pointers to the START, STOP and INT(errupt) routines in the DCB, and performs any action necessary to initialize the device (such as setting up the interrupt vector).

UNINIT: Called by DELETEDEV. Performs any action necessary to uninitialize the device (often nothing).

START: Called by QPKT when putting a packet on a previously empty work queue. Should initiate the action requested in the packet.

STOP: Called by DQPKT when removing the head packet from the work queue. Should cancel the action currently in progress.

INT: The interrupt routine. It is called in a way which makes the address of the corresponding DCB available. Usually, the interrupt indicates that the last action is complete or has failed, so the INT routine puts a return code in the head packet on the work queue, sends it back, and initiates the action requested by the next packet (if any).

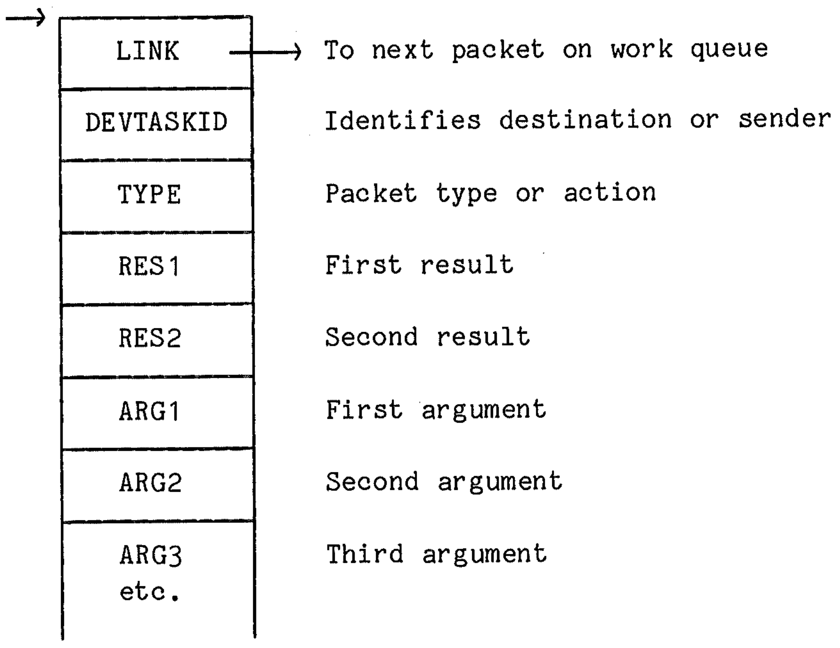
### 3.3.1 The Timer Device

The kernel contains a special device used for timing purposes. It has the identifier -1, which is recognized by QPKT, as the 'device' has no DCB, and does not appear in the device table.

Packets to the timer contain in their ARG1 field a time given as a number of 'ticks'. The action of the device is to return the packet to the sender after this number of clock interrupts has occurred. The work queue of the timer starts from the root node, and is unusual in that packets on it are stored in order of expiry rather than order of arrival. This saves time in the clock interrupt routine, as in the normal case of no packets expiring, only the head packet need be inspected. A result field of each packet is used to hold the number of ticks between the expiry of the packet in front and this one; QPKT and DQPKT maintain this queue structure.

## 3.4 Inter-task communication; Packets

Under TRIPOS, all communication between tasks, device drivers, and the timer, is performed by sending packets. A packet is a vector of at least two words. The first two words are used by the system; any further words are available for data. The conventional format is as follows:-



The LINK field is used when the packet is on a work queue. It points to the first packet to arrive after the current one, or contains zero if this one is on the end of the queue. Whenever a packet is not queued, the link should contain the value of the manifest constant NOTINUSE.

Three kernel primitives - QPKT, TASKWAIT and DQPKT - exist to handle packets. BLIB contains some useful, slightly higher level, routines - SENDPKT, RETURNPKT, PKTWAIT and DELAY.

Before QPKT is called to send a packet, the DEVTASKID field should be set to indicate the destination. Values less than or equal to -2 indicate devices, -1 means the clock, and values greater than zero refer to tasks. The value zero is invalid. As QPKT sends the packet, it overwrites this field with the identity of the sender. Not only does this automatically mark the packet with its sender's identity, but also leaves it in the correct form to be returned by a call of QPKT in the receiving task.

By convention, the third word is used to specify the type or requested action of the packet. The two result fields are in general not looked at by the receiver of the packet, but are used to return results or error codes. The number and format of the arguments are entirely up to the users of the packet. The arguments and type field should not be overwritten by the receiver, so that the packet can be reused without modification by the sender.

As the message-sending primitive, QPKT, does not cause the task which sends it to wait, it is possible to write tasks which are either sequential or multi-event. A sequential task follows each call of QPKT with one of TASKWAIT, and so never runs with a packet outstanding. A multi-event task may send out several packets, and then act on each one as it returns; it calls TASKWAIT only when it has nothing left to do.

The message system in TRIPOS is fairly efficient. One way to measure the speed is to create a pair of tasks which just bounce a packet between themselves. On an LSI4/30 computer, the number of bounces (and hence the number of task changes) per second is about 5000.

### 3.4.1 Work queues

Packets are not moved in store; when they are "sent", they are simply linked into a chain called a work queue. Tasks and devices process packets in the order in which they arrive, so a new packet is linked on the end of the work queue. Packets are removed from the head of the queue, which is addressed by a field in the TCB or DCB.

The timer's work queue is different, as described above. The head of this queue is in the root node.

## 3.5 The machine code library MLIB

MLIB is a library of BCPL-callable routines written in assembly code. It contains those routines which are found in most BCPL run-time systems, rather than those specific to TRIPOS, but which are best written in machine code for speed, size, or because of the low-level nature of the things they do.

A brief list of the functions and their purposes is given here:-

| | |
|---|---|
| LEVEL | Return current stack level |
| LONGJUMP | Jump out of current procedure activation |
| APTOVEC | Call a function with a workspace area allocated from the stack |
| SARDCH | Stand-alone read character from console |
| SAWRCH | Stand-alone write character to console |
| CREATECO | Create coroutine |
| CALLCO | Call coroutine |
| COWAIT | Suspend coroutine |
| RESUMECO | Resume other coroutine |
| DELETECO | Delete coroutine |
| GETBYTE | Extract byte from vector |
| PUTBYTE | Insert byte in vector |
| MULDIV | Evaluate (a*b)/c, holding the intermediate product in double length |
| STOP | Exit from current command or task. |

In some implementations, MLIB also contains routines to which calls are generated by the BCPL code generator. This technique is used to reduce the total code size when functions that would normally be code-generated in-line (e.g. multiply, byte access) are not well supported by the hardware.

### 3.5.1 The action of STOP

BCPL run-time libraries usually contain a routine STOP, which terminates the calling program (in a machine-dependent way). The STOP routine in TRIPOS is more complicated in its action than it would be in a run-time library for a single-task BCPL system, as it needs to observe the environment from which it is called.

STOP is expected to finish the program which calls it, passing back the returncode given as its argument. Programs in TRIPOS which may call STOP run in one of two environments:- either as the main body of a task, or as a subsidiary coroutine of a task. In particular, commands run by a CLI are executed as coroutines.

Thus, if STOP is called in the root coroutine of a task, it returns control to the task activation code in the kernel that started the task, and the task becomes 'dead' - i.e. it is the same as returning from START. The returncode is discarded, there being no higher level to pass it to.

STOP(N) called from any other coroutine is equivalent to returning from the main procedure of the coroutine after setting the global RETURNCODE to N. In commands called from a CLI, this is the same as setting RETURNCODE and returning from START. Control goes back to the CLI, which receives the returncode and deletes the code of the command.

## 3.6 Coroutines in BCPL

Some of the routines in MLIB form a coroutine package for BCPL programs. This package is described in the paper [38]. TRIPOS uses coroutines in its CLI, console handler and file handler tasks in the standard system, in some commands, and in much of the Ring interface software mentioned in later chapters.

A brief description of the coroutine system is given here, with some examples of how coroutines can be used to simplify multi-event tasks.

A coroutine is created by a call of CREATECO with arguments giving the procedure which is to be the main body of the coroutine, and the stack size required. This returns a coroutine pointer - an object which identifies the coroutine:-

```
coptr := createco(routine, stacksize)
```

The coroutine is called using CALLCO, which allows one argument to be passed. On the first call, this argument appears as the argument to the main body routine. The called coroutine retains control until it makes a call of COWAIT, which suspends it, and returns control to its caller. COWAIT also takes an argument, which comes through as the result of CALLCO in the calling level.

A further CALLCO causes the coroutine to resume by returning from its call of COWAIT; the argument to CALLCO appears as the result of COWAIT. Thus, the root level will contain lines of the form:-

```
result := callco(coptr, arg)
```

and the body of the daughter coroutine may be of the form:-

```
LET routine(res1) BE
    $(
    ....
    res2 := cowait(arg1)
    ....
    res3 := cowait(arg2)
    ....
    $)
```

RESUMECO takes a coroutine pointer and a value to be passed as arguments; it directly resumes that coroutine as if control had been passed back to the caller of the first, which had then in turn called the second.

DELETECO takes a coroutine pointer as argument, removes the coroutine, and frees its stack space.

### 3.6.1 Coroutines in TRIPOS

Coroutines are extensively used in TRIPOS for several reasons:-

- They can greatly simplify the programming of a multi-event task - that is one which may have several of its own packets outstanding at once, as well as being ready to receive unsolicited request packets from other tasks. This application is discussed below.

- They can be used to implicitly provide interlocks on shared data structures. When one coroutine within a task is running, no other coroutine of that task will run until the first explicitly relinquishes control. Hence, a running coroutine can make updates to a structure belonging to the task, knowing that no other coroutine will see it in an inconsistent state.
- Coroutines enable operations to be interleaved, giving some of the power of separate tasks, but at lower cost. The store cost of a coroutine is little more than its stack; calls are faster than task switches.

### 3.6.2 Use of coroutines in multi-event tasks

Many TRIPOS tasks are multi-event in nature, particularly device handlers such as the console and file handlers. One way to write such tasks is to remember state information in various global variables and flags, which are inspected when a packet arrives to discover the point reached in any multi-stage operations in progress.

However, the task code becomes much simpler to write and understand if there is a separate coroutine for each multi-stage operation. Then, although the task as a whole is interleaving the stages of different processes, each coroutine is proceeding sequentially through its own operation. Much of the state can be remembered implicitly, by the position reached in the code. A scheme for writing tasks in this way is presented below; it was used successfully in the Ring handler, virtual terminal handler, and Fileserver filing system tasks, as well as in some commands - notably that used for logging on to other Ring machines, which handles asynchronous input and output. It was also used used in a remote debugging system [1].

A multi-event task receives packets of two kinds: those which it sent out and are now returning, and those which originate from other tasks, and are requesting services. The former can be recognized by remembering their addresses as they are sent. Any not recognized in this way are of the second kind; the service they want is indicated by their type field.

The Fileserver file handler task will be used as an example. Its root coroutine has three daughter coroutines - two equivalent ones which process requests (called REQUEST.CO.1 and REQUEST.CO.2), and one which wakes up from time to time to do periodic operations (called CLOCK.CO). There are corresponding global variables with names such as REQUEST.CO.1.PKT, which hold the addresses of the packets for which the coroutines are waiting. The request coroutines can also be in an idle state, in which they are ready to process a request packet. This is indicated by the expected packet address being set to zero.

The main loop of the task has the following structure:-

```
$(
LET packet   = taskwait() // The only TASKWAIT in the program

IF packet=request.co.1.pkt
THEN $( request.co.1.pkt := callco(request.co.1, packet); LOOP $)

IF packet=request.co.2.pkt
THEN $( request.co.2.pkt := callco(request.co.2, packet); LOOP $)

IF packet=clock.co.pkt
THEN $( clock.co.pkt := callco(clock.co, packet); LOOP $)

// If we reach here, then the packet is not one expected
// by any coroutine, but is a request packet.
// Pass it to a request coroutine if one is free,
// otherwise save it until one becomes free.

IF request.co.1.pkt=0
THEN $( request.co.1.pkt := callco(request.co.1, packet); LOOP $)

IF request.co.2.pkt=0
THEN $( request.co.2.pkt := callco(request.co.2, packet); LOOP $)

// Put the request packet on an internal queue from which
// the next request coroutine to become free can extract it

save.request.pkt(packet)
$) REPEAT
```

For this to work, each time one of the coroutines sends out a packet, it must call COWAIT with the packet address as argument. This is most easily achieved by redefining the BLIB routine PKTWAIT (which exists to be redefined - see the section "BLIB") as follows:-

```
LET pktwait(dest, pkt) = cowait(pkt)
```

This works because the routines in BLIB which send packets (such as RDCH, WRCH, FINDINPUT, LOADSEG and DELAY) do it by means of SENDPKT. As SENDPKT uses PKTWAIT to wait for the returning packet, calls of SENDPKT now have exactly the right effect: the coroutine suspends itself, delivering the packet address; when it is resumed, it expects the same packet address as argument and checks it. The coroutine bodies can be written as simple sequential code using the normal library calls, with little regard for the fact that they are part of a multi-event task.

## 3.7 The BCPL-written library BLIB

The majority of the resident library routines are written in BCPL, and are identical in versions of TRIPOS for different machines. Most of these are concerned with input/output and file operations; others are for loading and calling programs, sending packets, comparing and manipulating strings and characters, and converting fault codes into messages. About half of the routines perform actions specific to TRIPOS; the others are found in most BCPL run-time systems.

A list of all the BLIB routines, with a brief indication of what each one does, is given in Appendix 1.

The remainder of this section describes aspects of TRIPOS implemented by functions in BLIB, including input/output streams.

### 3.7.1 Streams

Streams under TRIPOS reflect BCPL's view of input and output. That is, streams are opened using FINDINPUT or FINDOUTPUT (in BLIB) with a stream name as a string argument; these return a stream identifier, a value which is used as argument to SELECTINPUT or SELECTOUTPUT. These set up the current input and output streams respectively (recorded in globals), and routines which read or write data, or close streams, always apply to those currently selected. Reading and writing are normally done one byte at a time, by means of RDCH and WRCH. Blocks of words may be read and written with READWORDS and WRITEWORDS.

Most TRIPOS streams eventually talk to a peripheral. The low-level operation of this is achieved by a device driver, and that in turn is usually driven by a handler task, which provides higher level formatting and management functions (e.g. console handler, file handler). The stream identifier is the address of a stream control block (SCB), which includes:-

(i) The task number of the handler task

(ii) The address of the current buffer

(iii) Pointers to the current character position and the end of the buffer

(iv) The address of a routine to be called when the buffer is empty and should be refilled (input stream), or when it is full and should be transmitted (output stream).

(v) The address of a routine to be called to close the stream.

RDCH and WRCH use the buffer until full/empty, and then call REPLENISH or DEPLETE (in BLIB), which in turn call the appropriate routine from the SCB. This normally sends a packet to the handler task to deal with the buffer appropriately. READWORDS and WRITEWORDS bypass the SCB buffer, transferring instead the client's buffer address directly to the handler.

An output stream may be marked interactive in the SCB. This means that the WRCH buffer is transmitted on the end of every line, (i.e. on '*N', '*C', etc.) rather than when it is full.

**Stream Names**

The string supplied to FINDINPUT or FINDOUTPUT has the general form:-

device:path

"Device" may be a mounted disc, the system disc, or a <u>pseudo-device</u> (see below). "Path" is of the form "filename", "dirname.filename", "dirname.dirname.filename", etc. for a disc device; for other devices, it specifies the route of the stream in an appropriate way.

Some special device names are:-

| | |
|---|---|
| SYS: | the system disc |
| : | the root of the disc on which the currently selected directory resides |
| * | the console. This stream name is treated specially by BLIB. |

### 3.7.2 Pseudo-Devices

A powerful feature of the stream opening mechanism is that it is easy to add extra devices to those known to the operating system, and that these need not correspond to real peripherals, but can be 'pseudo-devices'. Examples are:-

| | |
|---|---|
| NIL: | the dummy device: behaves as an infinite sink on output, and gives immediate end-of-stream on input. |
| LP: | loads the devices and handler task for the line printer |
| PIPE: | loads the pipe handler for inter-process stream communication |
| BSP: | Loads the Ring byte stream handler (see chapter 5) |

When FINDINPUT or FINDOUTPUT is called with a stream name containing a device part, it first looks for the device name in the assignments list in store (see "File Handler Interface Changes" in chapter 7). If it is not there, then an overlay is called (by the BLIB routine CALLSEG), and this in turn attempts to open the file "SYS:H.devicename" and call the program therein, passing it the string argument to FINDINPUT or FINDOUTPUT, and a blank SCB. This program is expected to return a completed SCB for the stream.

The effect is that arbitrary new devices may be included, without rebuilding the system, simply by including files containing appropriate programs in the directory ":H".

### 3.7.3 PKTWAIT

The routine PKTWAIT in BLIB is a dummy, with the following definition:-

```
LET pktwait(destination, packet) = taskwait()
```

It exists only so that any task can redefine it in its own global vector.

Most packets are constructed and sent using the BLIB function SENDPKT - in particular, all those sent by BLIB routines such as RDCH, WRCH, DELAY, etc. SENDPKT sends the packet composed of its arguments, then waits for it to return by calling PKTWAIT (rather than TASKWAIT).

Thus, by replacing the standard PKTWAIT, a task may regain control inside SENDPKT (which would otherwise block execution of the task). This can greatly simplify the coding of a multi-event task involving several coroutines, as described in the section on coroutines in this chapter. It can also be used to introduce some multi-event character into an otherwise sequential task.

A common use of this is in applying a timeout to a device operation. SENDPKT is used to send the packet to the device, and PKTWAIT is replaced by a routine such as that below, which issues a packet to the timer, and cancels the device operation if the timer packet comes back first:-

```
LET pktwait(destination, packet) = VALOF
    $(
    LET timer.pkt       = TABLE notinuse, -1, ?, ?, ?, timeout.ticks
    LET received.pkt = ?

    qpkt(timer.pkt)                     // Start the timeout

    received.pkt := taskwait()          // Wait for the timer packet
                                        // or the expected one
    TEST received.pkt = timer.pkt
    THEN
      $( // Other packet has been timed out
      dqpkt(destination, packet)          // Retrieve the packet
      packet ! pkt.res1 := failure.code   // Set its result field
      RESULTIS packet
      $)
    ELSE
      $( // Correct packet (or unexpected one!)
      dqpkt(-1, timer.pkt)                // Cancel timeout
      RESULTIS received.pkt
      $)
    $)
```

## 3.8 System data structures

Most of the system data structures within TRIPOS have a format which is machine-independent. In particular, pointers are held as word addresses.

### 3.8.1 The Root Node

The root node is the central point from which all the system structures in store can be found. It is a vector containing pointers to the main chains and tables. The position in store of the root node is fixed on any given machine. Unfortunately, it is not practical to use the same address on all machines, as each machine has some special store locations used for interrupt vectors, device registers, etc. Instead, the address is given as the manifest constant ROOTNODE in the standard BCPL library header for each machine.

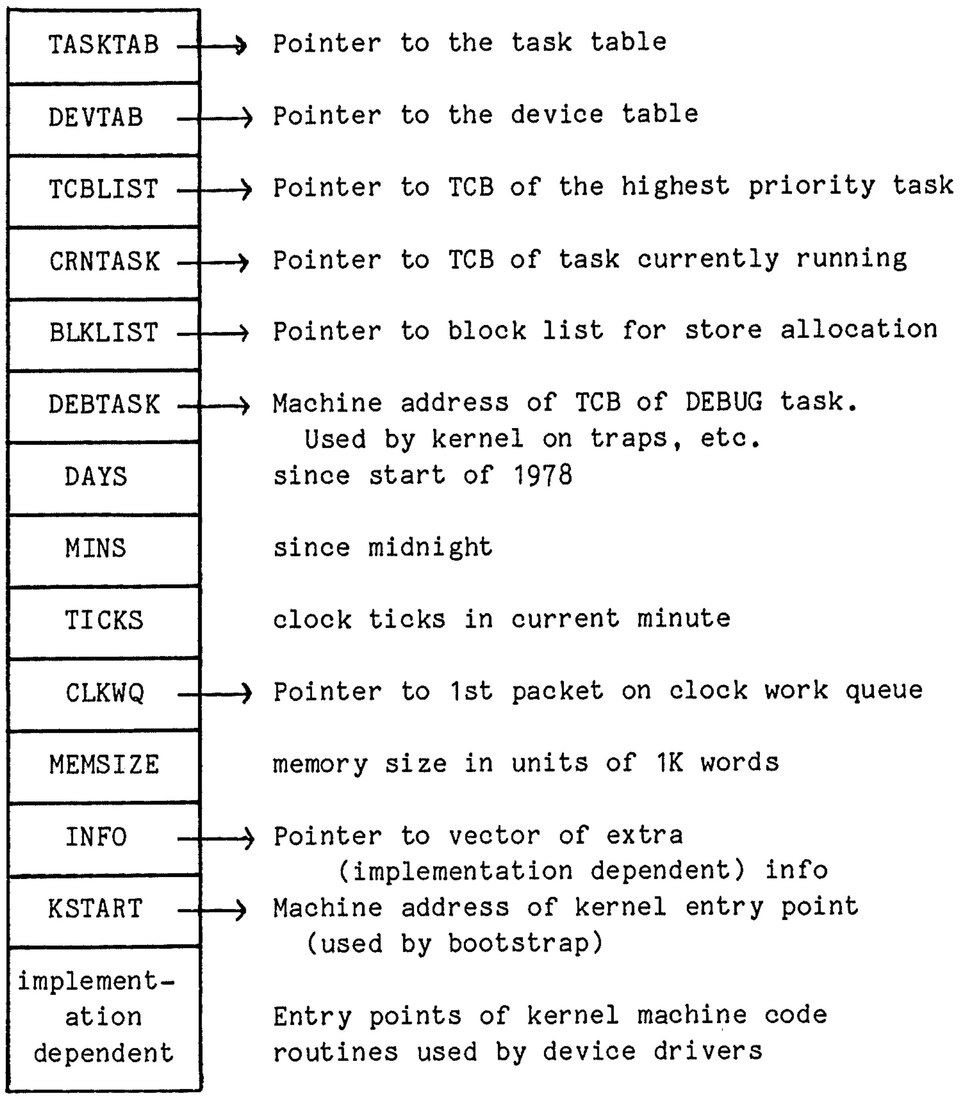

TASKTAB
Pointer to the task table
DEVTAB
Pointer to the device table
TCBLIST
Pointer to TCB of the highest priority task
CRNTASK
Pointer to TCB of task currently running
BLKLIST
Pointer to block list for store allocation
DEBTASK
Machine address of TCB of DEBUG task. Used by kernel on traps, etc.
DAYS
since start of 1978
MINS
since midnight
TICKS
clock ticks in current minute
CLKWQ
Pointer to 1st packet on clock work queue
MEMSIZE
memory size in units of 1K words
INFO
Pointer to vector of extra (implementation dependent) info
KSTART
Machine address of kernel entry point (used by bootstrap)
implement-ation dependent
Entry points of kernel machine code routines used by device drivers

Fig. 5: The Root Node

The first part of the root node has a machine independent format. It is followed by some addresses of kernel entry points for such operations as saving and restoring registers on interrupts, and entering the task selector. These are needed by device drivers which may be dynamically loaded, so must be in fixed store locations.

**BLKLIST**

This points to the start of the area from which store is allocated by GETVEC. For details of the format of store blocks, see the section "Store allocation" below.

**DAYS, MINS and TICKS**

The clock interrupt routine maintains the date and time in these three words. DAYS is the number of days since the start of 1978 (i.e. 1st Jan 1978 is day 0). MINS is the number of minutes since midnight. TICKS is the number of clock ticks since the last minute boundary. The time is updated at an implementation dependent frequency given by the manifest constant TICKSPERSECOND.

**INFO**

The INFO field is provided so that extra information may be made available from the root node without making any change to root node format. For example, it may point to a vector containing details of machine type, the machine name as a string, and/or the identities of particular non-resident tasks.

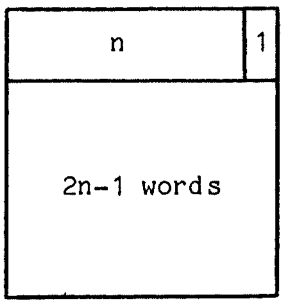
**KSTART**

This value is assembled into the root node, and is the means by which the bootstrap finds the kernel entry point after loading the system into store.

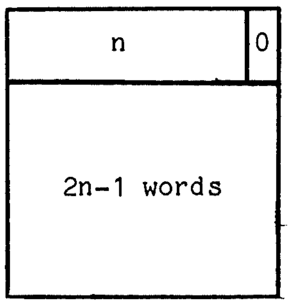
### 3.8.2 Store Allocation

Blocks of store are allocated and freed by the kernel primitives GETVEC and FREEVEC. Store is allocated from an area which starts at the address given by the BLKLIST field of the root node. This area is divided into contiguous blocks, which each consist of an even number of words. The first word of each block is used both to indicate the length of the block (and hence the start of the next), and to record whether or not the block is allocated. As all block lengths are even, the least significant bit of the length is not needed, and so this is used to indicate allocation:

Free block:



Allocated block:



The end of the block list is marked by a word containing zero.

## 3.9 The Resident System Tasks

The standard TRIPOS system has the four resident tasks mentioned in the previous chapter. Some aspects of their internal design are presented here.

### 3.9.1 Command Language Interpreter

The CLI is the simplest of the system tasks. On starting, it calls CLI.INIT, which in the initial load is the routine that starts all in other tasks. In other environments (e.g. a CLI created by the RUN command), a different version is used, which performs appropriate CLI initialization.

The main body is a loop which reads the name of a command from the input stream, loads the file of that name from the current directory or command directory, and calls START in the command code as a coroutine. It can run interactively, giving prompts and reading from a console, or non-interactively, reading commands from a file (see the C command). On reaching the end of a command file, it reverts to its standard input stream.

## 3.9.2 DEBUG

The majority of the code in the DEBUG task is concerned with its use as an interactive debugging aid. It reads command lines from the terminal and acts on them.

The peculiarity of this program is that it can run either as a normal task, or in a 'stand-alone' mode after an abort, breakpoint, or stand-alone entry from the handswitches. It remembers the mode in which it was called, as a few commands behave differently in the different situations, and when in stand-alone modes it must use SARDCH and SAWRCH to talk to the terminal, as the console handler is not running.

When the DEBUG task is activated, it records its identity in the root node, so that the kernel can find it after an abort. Its initial packet contains no information, and is passed straight back. When the kernel calls DEBUG in stand-alone mode, it calls START with a four-word vector as argument, giving the calling reason, task number, abort code, and abort argument (if relevant). The global vector used is that of the DEBUG task; the stack is a specially reserved area in the kernel (actually, the space where the kernel initialization code was).

On stand-alone entry, DEBUG outputs a message, announcing entry, abort or breakpoint as appropriate. The commands whose action is affected by the mode are 'C' (continue) and 'H' (hold). In task mode, these simply continue or hold DEBUG's currently selected task. After entry from the handswitches, neither of these has any useful function (- the system cannot be resumed, so it does not matter if tasks are held or not). After an abort, 'C' and 'H' both cause exit from stand-alone mode and resume the full system, with the aborted task running or held respectively. The action is similar after a breakpoint, except that 'C' takes a numerical argument. The breakpoint is not taken again until it has been encountered that number of times.

The kernel has a final mechanism to fall back on if an abort occurs in a system which does not include DEBUG. It halts the machine, with the program counter pointing at the 4-word vector which would have been passed to DEBUG, so that information about the abort can be found from the handswitches.

### 3.9.3 Console Handler

The console handler task provides a line-by-line interface to the single character console devices. It maintains two packets on the keyboard device, so as to always be able to accept typed characters. Input is reflected as soon as possible after it is received; if a line is being output when an input line is started, then reflection is delayed.

The current line is read into a fixed buffer; when it is complete, a block of store large enough to hold the line is obtained from the heap, and the line copied into it. If there is a read request packet outstanding from the currently selected task, then the buffer is returned in the packet; otherwise, the buffer is added to a chain of waiting input lines. When a packet arrives requesting the next input line, this chain is inspected to see if any lines have been typed ahead for that task. If so, then the first is returned; if not, then the request packet is appended to an internal queue.

Request packets containing lines to be printed are similarly queued, if there is already a line being printed, an input line is being typed, or output from the requesting task is inhibited (by '@Tnn'). Output packets pass the line in a buffer which is FREEVECed by the console handler. Thus, the handler is able to return the packet when it starts to print the line, allowing the client task to produce the next line while the current one is coming out. Returning the packet before commencing output of the line would mean that the flow control would be lost, and store could rapidly become filled with output buffers.

### 3.9.4 File Handler; disc format

The file handler task is the interface between the disc device and operations in terms of files and directories. It treats its disc as being composed of blocks of equal size (256 words for most discs) numbered from zero upwards. These block numbers are used as disc addresses, and are translated into cylinder, surface, and sector only in the routine which sends packets to the disc device.

When it starts, it receives a packet describing the disc (blocksize, number of cylinders, etc.). As none of the disc characteristics is built into the program, its code can be shared by handler tasks when more than one disc is mounted. After starting, it enters a mode whereby the only operations that are allowed are those which do not involve writing to the disc. When the restart task finishes, it passes over a block allocation bitmap to the handler; all operations are then permitted.

All disc blocks have a similar format: there is a six word header including the block type and a checksum of all the words in the block. The checksum is checked just before writing each block, and just after reading one, as a safeguard against overwriting in store and disc errors.

The directory structure is unusual in that directories are of a fixed size (1 block), and do not contain conventional directory entries giving details of the objects held in the directory. Instead, the majority of a directory consists of a hash table. The table is indexed by a value derived from the name of the object to be looked up in the directory. Each hash table slot points to the first object on a chain of objects with the same hash value. Thus, all information about an object, such as its name, date of creation, etc., is held within the object itself.

A file consists of a header block and some data blocks. A header block is very similar to a directory block, except that the hash table is replaced by a list of the block numbers of data blocks. The data blocks are also chained, to improve the efficiency of serial reading. Each data block contains the standard 6-word header; the rest of the block is used for data.

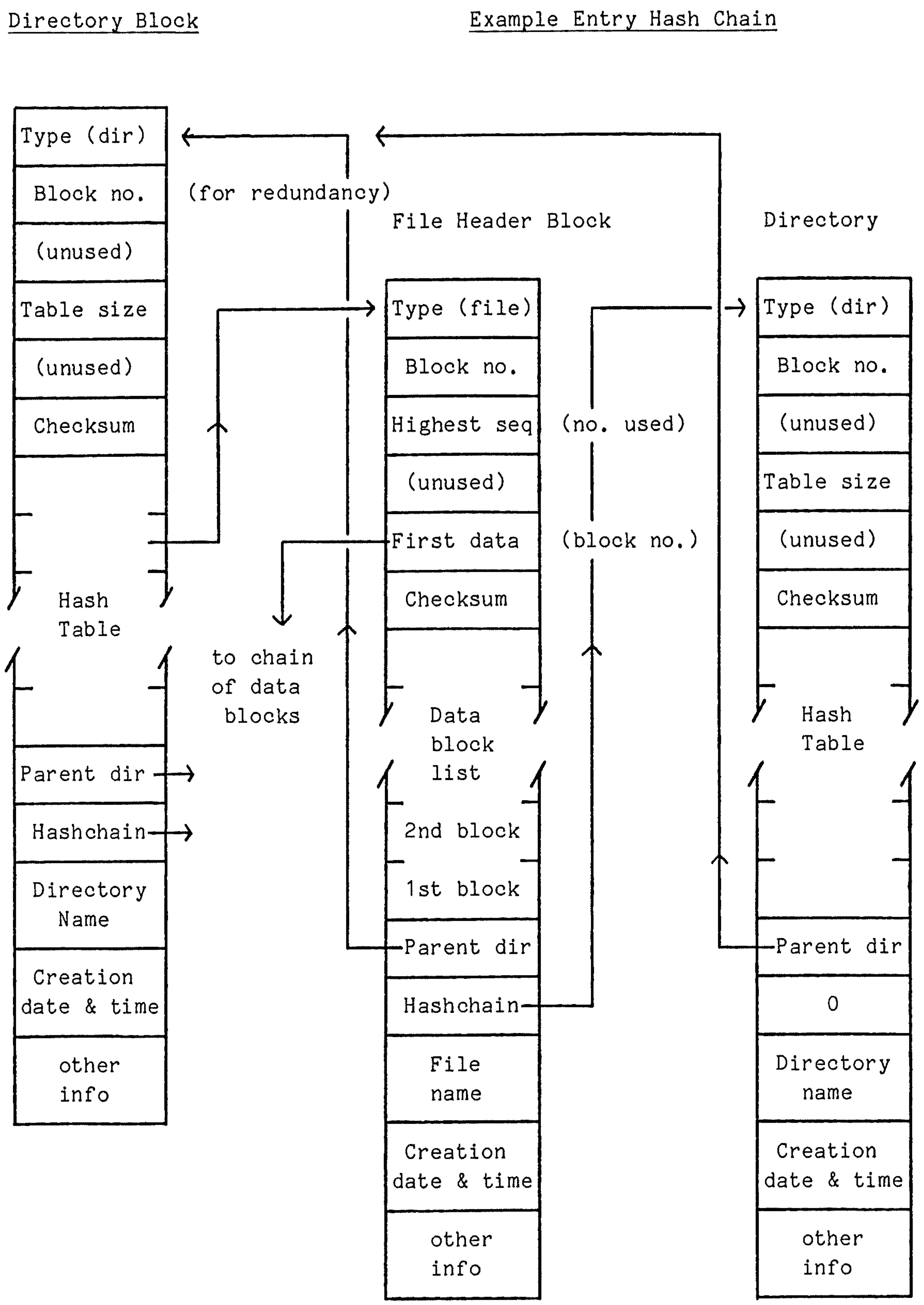


Fig. 6: Directory Structure

Each disc has a root directory, which differs from an ordinary directory only in its type field. The root directory is positioned half-way through the disc; thus, the file handler can always calculate the block number of the root from the description of the disc.

The file handler maintains a cache of a few disc blocks, helping to reduce the number of disc transfers when, for instance, the same directory is used several times.

The file handler restart task builds the disc block allocation bitmap which the file handler needs before it can write to the disc. To do this, it reads every directory block and every file header block. No data blocks need to be read, as each file header contains a list of all the data blocks allocated to the file. In fact, the last data block of each file is read as a consistency check; this is the block which is most likely to disagree with the header if a crash occurred while a file was being written.

A second bitmap is used to record which blocks are still to be inspected. This enables restart to scan the disc (several times), reading blocks as they are passed, keeping disc seek time to a minimum.

## 3.10 Program Environments in TRIPOS

There are three main environments in which a normal BCPL program (i.e. one defining the routine START, and expecting to be entered by a call of that routine) may be run under TRIPOS. These are:-

(i) As the root coroutine of a task.

The argument to START is the packet which caused task activation; its arguments should be read, and the packet returned, as soon as possible. The global vector is that belonging to the task; the only values in it are addresses of global routines and labels defined in the code segments of the task, plus a few special variables (e.g. TASKID, TCB - set by the system to the task's identity number and the address of its TCB). The stack is the task's root stack. The effect of returning from START is to free the root stack and global vector, putting the task into dead state; any result from START is lost. A call of STOP is the same as returning from START. No streams are open at the time of entry, and INITIO should be called before any streams are opened.

(ii) As a command called from a CLI.

The argument to START is zero. The global vector is that of the CLI task. Its contents are as in (i), with the addition of some variables used by the CLI - e.g. whether running interactively, return code from last command. Routine addresses in the global vector may be redefined by the command, as the CLI will restore them before executing the next command. The stack is that of a coroutine called from the CLI's root coroutine. The effect of returning from START is to end the command, freeing its stack, and unloading its code. A call of STOP(N) returns from START after setting the global RETURNCODE to N. The value N indicates whether the command worked (0) or how severely it failed (> 0). If the CLI is taking commands from a file, then a return code greater than a (settable) threshold causes processing of the whole file to be abandoned. The value of RESULT2 on exit should give the reason for failure in the form of a fault code (which can be translated into a text message by the BLIB routine FAULT). The current input and output streams of the CLI are open and selected on entry; they should not be closed by the command.

(iii) As a subroutine called from a disc file by the BLIB routine CALLSEG.

CALLSEG loads the code contained in the specified file, initializes the globals defined therein, and calls START. START receives up to 4 arguments, being the 2nd to 5th arguments to CALLSEG. The global vector and stack are as for the calling program. Returning from START causes the loaded code to be unloaded again. CALLSEG returns to its caller, passing back the result from START and RESULT2. STOP should not be called from a CALLSEGed routine (nor should LONGJUMP), as it will exit without unloading the code. The selected streams will be as for the caller; in general, they should not be closed or unselected.

Programs may be written so that they may be run in any of the above environments by making them inspect the first argument to START, and specifying that, when they are CALLSEGed, this argument has a value which is neither zero nor a packet address (e.g. -1).

### 3.10.1 Environment of Device Driver Code

The assembler code of device drivers runs in a simpler environment than the rest of the system. All of this code runs with interrupts disabled, so is not allowed to call any kernel primitives (which may enable interrupts again). The INIT, UNINIT, START and STOP routines of the device are called as machine code subroutines from kernel primitives (CREATEDEV, DELETEDEV, DQPKT or QPKT, and DQPKT, respectively) - i.e. the full BCPL calling sequence is not used. The INT routine is called from the kernel on an interrupt, so no stack or global vector is available to it. Devices use special kernel routines to perform operations such as sending packets back to tasks, and returning from interrupts.

## 3.11 Multi-User TRIPOS

TRIPOS was specifically designed as a single-user system. However, it can easily be used by more that one person, and has been used in such a way.

The command RUN CLI2 creates a new CLI which then makes a new console handler task and new devices attached to the machine's second terminal port, and adopts this as its console. The second console then becomes virtually equivalent to the first.

Multi-user use is only worthwhile between cooperative users who are editing, etc., rather than testing dubious programs. As there is no memory protection, any crash may affect both users, and of course there is no protection against malicious attack by one on the other. Response for the second user can be badly degraded if the first is using a lot of the CPU time, as he is at a strictly lower priority.

Nevertheless, multi-user working is simple to provide, and has proved very useful on occasions.

## 3.12 System generation; SYSLINK

A new version of TRIPOS is linked together as follows:-

- All the programs comprising the libraries, tasks, device drivers and DCBs are compiled or assembled.
- A file describing the system to be constructed is created (see below).
- The command SYSLINK is run on this description file to produce a system object module.

The driving file for SYSLINK is a text file containing the following information:-

(i) Some parameters of the target machine: e.g. its store size, addressing unit, and some details of the system which depend on the underlying machine - e.g. the root node address, size of TCBs.

(ii) The layout and initial contents of the root node info field.

(iii) A list of filenames of files containing code of system components. Code which is used only for system start-up is marked so that it can be positioned contiguous with the free store, to reduce store fragmentation when it is unloaded.

(iv) A description of each task, including its number, root stack size, priority, and code segments. The task which is to be started first is flagged.

(v) A description of each device, including its number and code segments.

The output from SYSLINK is an object module containing the complete initial core load of the system. It is usually processed further, by a command such as SYSIMAGE, to create a core-image on disc for use by a bootstrap loader.

Linking a new system is thus a simple and rapid process once all the components have been compiled, taking a few tens of seconds.

## 3.13 Transferring TRIPOS to a new computer

The following work is required to transport TRIPOS to a new machine:

- Writing of a BCPL code generator for that machine
- Coding of the libraries KLIB and MLIB in assembler
- Writing of device drivers for disc and console
- Modification of DEBUG for new function entry sequences, address units, and possibly stack layout.
- Production of an assembler

**BCPL code generator**

The front end of the compiler does not need to be changed, as it is itself written in BCPL and produces a machine-independent intermediate code [37]. This code is processed by a code generator particular to the target machine.

The writing of the code generator is probably the most difficult part of the porting process. It involves decisions on how to make use of the features of the machine: which central registers to use for argument passing, holding pointers to global vector and current stack frame, and as general work registers; whether to make use of any hardware stack(s); design of function calling, entry and exit sequences.

A detailed account of several code generators written for TRIPOS can be found in [11].

**KLIB and MLIB**

Although the amount of code is not insignificant, the writing of it is not usually too difficult. These libraries consist of a collection of fairly small routines with clear specifications. The process is usually aided by reference to an existing kernel, and can approach instruction-for-instruction translation. (I.e. it is coding which does not involve many decisions.) A detailed guide to one particular implementation of TRIPOS (for PDP11s) has been written [19] as an aid to this kind of translation.

**Device drivers**

The operations coded here will be highly machine dependent, but drivers are not very long, and are each split into five routines.

**DEBUG**

The bulk of DEBUG will be unchanged; the alterations are just a few minor ones concerned with such things as recognizing BCPL function entries.

**An Assembler**

An assembler to run under TRIPOS is desirable, in that it is necessary in order to make the new system self-supporting. It may not be essential for bringing up the system if a manufacturer's assembler is available.

Note that the assembler is needed only for a few components of the system, and so need not be very elaborate - features such as macros, conditional assembly, multiple sections, and external references probably are not needed. Typically, a fair portion of the assembler is stolen from an existing one.

### 3.13.1 Portability Problems

Experience with transporting TRIPOS has shown up several areas in which programs tend to make assumptions about underlying hardware. Differences in word length have caused very few problems, perhaps because the first few implementations were all on 16-bit machines, so moving to machines with longer word lengths meant only that some store was wasted rather than program logic destroyed. The few programs which were incorrect either assumed that the bit 8000 (hexadecimal) could be inspected by testing for the word being a negative number, or set up logical masks incorrectly. For example, the expression **a & #XFFF0** should be written **a & (NOT #XF)** to be independent of word length. Similarly, very few programs proved sensitive to the number of characters in a BCPL word, or their ordering within the word.

Some trouble was experienced when moving from the PDP11 to the LSI4 because for the first time BCPL addresses could be negative numbers (the latter machine having 64K words of store). This meant that programs which compared addresses (e.g. to implement a circular buffer) would fail if their workspace fell in the top half of store. It also meant that the memory size could not be directly expressed as an integer; hence the MEMSIZE field of the root node holds the size as a multiple of 1K words.

A programming mistake which more than once caused programs not to work when moved was the inadvertent assumption of the contents of location zero of memory, usually from an incorrect test for the end of a linked list.

In order to ensure that programs written for one machine will work on others, the sizes of BCPL stacks have been kept the same on different machines, regardless of the amount of memory available. Unfortunately, some hardware forces the overhead per procedure stack frame to be higher than others, so the effective stack size cannot be made identical on different computers.

A few pieces of code assume that integers are held as twos-complement binary: the system has not yet been moved to a machine with a different representation. Also, in a few places there are implicit assumptions that the character code is ASCII (e.g. tests for a letter as a character between 'a' and 'z', or a printable character as one greater than space). However, most programs which are driven by the values of input characters make use of the BCPL SWITCHON statement with the cases labelled by character constants, so would still compile correctly with any character code.

A problem faced in the PDP11 version of TRIPOS is that the machine is available in a range of models with slightly different instruction sets. Particularly infuriating is that, although there are several ways of disabling interrupts on any particular model, there is no way which works on all models. Several slight variants of the kernel had to be produced. Some PDP11s lack certain instructions, such as multiply, divide and multiple shifts. To cope with these an option was added to the code generator to make it generate calls to a supplementary machine code library instead of the missing instructions.

## 3.14 Comments on the Design of TRIPOS

The original design aims have been realized to a large degree, in that TRIPOS has proved relatively straightforward to transport, pleasant to use, with good response, and simple to understand and modify. Perhaps a good demonstration of the latter points is that several people unconnected with the project have used TRIPOS as the basis for their own real-time systems, for process control, data entry, automatic data collection, and 'servers' on the Cambridge Ring. The latter include the Fileserver, its asynchronous garbage collector (which runs in a separate machine), a database server, and a server to drive a "pointing machine" (an aid to producing wire-wrapped circuit boards). In some ways, making a system too easy to understand creates problems. Almost every user suggests changes that he regards as essential, and one has to resist strongly to stop the system growing randomly under user pressure, destroying its original portability and elegance.

This section reviews various aspects of the design of TRIPOS in the light of extensive use of the system, and experiences with transporting and modifying it.

### 3.14.1 Tasks

Tasks in TRIPOS are inexpensive in terms of time (i.e. task switches are fast), but do have a significant store overhead, as each has its own global vector and stack. The simple scheduling rule has been quite satisfactory for a single-user system.

The reversion of a task to 'dead' state when it returns from start is a facility which has been hardly used in practice. It would enable a task which wakes up only rarely to have a small store overhead while it was idle (as a dead task has no stack or global vector). In order to remember anything between activations, it would have to use static variables.

The use of the task creation primitive, CREATETASK, always involves calling it in a loop in order to find an available priority. It would be neater if the priority specified in the call were a maximum, and the primitive itself found an available one.

### 3.14.2 Devices

Device drivers contain the minimum amount of machine code necessary to provide a packet interface to a peripheral. This strategy has made it relatively easy to write device driving code when moving TRIPOS to a new computer, as all the complicated work is done in BCPL in handler tasks.

The model of a device as something which serially processes the request packets which arrive on its work queue, initiating a peripheral action for each, and getting an interrupt for each, has proved an over-simplification for multiplexers, multi-drive disc controllers, and intelligent Ring interfaces. These devices have to process each packet as it arrives, holding them on a private queue. This means that DQPKT cannot be used to cancel a device request; another packet must be sent to request cancellation. (See discussion of DQPKT in "Packets" below.)

Another situation where a real device does not conform to the model is when the terminal has a half-duplex connection (as was the case on the Laboratory's LSI4s). TRIPOS expects the terminal to be two independent devices, so some subtle coding is needed to create this illusion by having a single device driver shared by two DCBs.

### 3.14.3 Packets

The conventional packet layout, and the sending and receiving primitives QPKT and TASKWAIT, seem to have been good choices, being easy to use in programs. The convention that the same packet as made a request is used for the reply, and the way QPKT flips the ID field both simplify programming. The BLIB routine SENDPKT is widely used by sequential code which wants to temporarily create, send, and receive, a packet.

The fact that tasks and devices have only a single message channel (work queue) has not been a great restriction, particularly as PKTWAIT can be used to simulate the effect of multiple channels.

The primitive DQPKT was not such a good choice; it has rarely been used, and tends to be the most tricky primitive to write for a new machine. It is almost useless for retrieving a packet from a task, as multi-event tasks receive packets quickly, and hold them on private queues. Most devices process packets serially, so all unprocessed ones are still on the work queue, and the one currently being serviced is at the head. However, some device drivers need to store packets internally. DQPKT is most useful for cancelling timer packets.

A better mechanism might have been to define a 'cancel' packet type, and require all devices, and tasks providing services, to accept it. This would rationalize a facility already provided in some tasks and devices, and would mean that any device could be replaced by (or hidden behind) a task with the same interface.

### 3.14.4 Tidying up After Tasks and Commands

A significant deficiency in TRIPOS is that it has no way of keeping track of the resources claimed by a task or command, and so cannot tidy up when a task or command finishes. Programs written for use under TRIPOS need to exit cleanly, freeing all the store they have GETVECed, closing their streams, sending back all packets they have received, and reclaiming all outstanding packets. Although this tidy programming practice should be encouraged, it can be tedious for it to be always necessary.

To improve this situation, it would be necessary to hold in each TCB a record of store blocks claimed, streams open, packets held and outstanding. Problems occur when buffer space is obtained in one task, and then passed to another for processing; to avoid copying data, the ownership of the buffer would need to be passed over as well.

It is doubtful whether it is worthwhile going to elaborate lengths to solve this problem. Keeping a record of a task's resources would slow the system down, but would still not be effective unless programs obeyed a variety of conventions on how they should use store blocks, etc. TRIPOS has always been planned as a single-user system which can rapidly be rebooted if necessary.

Other systems seem to adopt one of two methods to solve the tidying-up problem. Systems running in machines without memory management are usually single-threaded monitors, which load a program, allocate most of the memory to it, and them reclaim that memory when it finishes. The majority of operating systems are able to take advantage of memory management hardware in order to completely remove all traces of a program. Most personal microcomputers combine the two techniques by having all the monitor code in read-only memory.

### 3.14.5 Attention Mechanisms

The only attention (or 'break') mechanisms provided in TRIPOS are the task flags set by the console handler when control-B, C, D, or E is typed. These rely on the target program polling the flags periodically. This gives a clean method of signalling to a program that it should tidy up and stop; no input or output is lost.

However, it is inadequate for killing a program which is not inspecting the flags because it is stuck in a tight loop, waiting for a packet, or was not written with the tests included. The best that can be done is to use DEBUG to hold the task.

It would be desirable to have a system 'break' facility as well as the flags. For the reasons given in the previous section, it would be very difficult to provide a method of completely removing a task at any time it was running. However, it would be possible to interrupt it and cause it to call a particular global routine, which it could redefine to handle its own tidying up. There remains the problem of where the stack frame for this call should be placed; the highest location in use on the stack cannot be determined, and in a multi-coroutine task it is not clear even which stack frame should be used.

### 3.14.6 Store Allocation

All tasks in TRIPOS allocate store from a single heap, which covers most of the machine's memory. This allows maximum use to be made of the available store, but can also lead to fragmentation if, for instance, lines are typed ahead while a large program is running, and are allocated buffer space in the middle of memory.

This could be improved by making system tasks, at least, obtain the store they would need when they start, and allocate privately from that. However, this would lead to restrictions, such as a small limit on the number of lines that could be typed ahead.

The attempt to use only the store actually necessary means that commands and tasks are usually run with a small stack, and expected to GETVEC any extra store they need. In most other BCPL systems, the bulk of the available store is given to the stack. A common change that has to be made when importing programs to TRIPOS is that calls of APTOVEC (obtaining store from the stack) have to be replaced by calls of GETVEC.

## 3.15 Comparison with Other Operating Systems

Research into portability of operating systems is a fairly new area of study, and most of the portable systems mentioned in chapter 1 were being designed or first ported at about the same time as TRIPOS was being developed. Hence their influence on it was small, and this is a retrospective comparison.

The other operating systems mentioned in the first chapter seem to fall at the ends of a spectrum in terms of complexity and facilities provided. At one end are OS6 and SOLO, both single-user systems with just one process or a fixed number of processes, and running interpreted code. Most of the others (such as MUSS and UNIX) are much more elaborate, allowing multiple users, and providing virtual memory, swapping and protection. Thoth is one of these, but is aimed at similar machines to TRIPOS.

TRIPOS comes in the middle of this range, being explicitly a single-user system and deliberately simple, but aiming to provide powerful facilities and to be able to support real-time control of external devices. To this end it runs compiled code, and allows dynamic creation of tasks and coroutines, but has only minimal and defensive protection, and simple fault handling.

In the power it aims to provide for a single user, and in its lack of distinction between user and system, TRIPOS resembles Pilot and the Alto operating system from Xerox (though the latter is really only a single-process system). Both of these run all programs in a single address space, and avoid all protection except that designed to trap common errors. Both accept that boot-loading the machine is a fairly common occurrence. Pilot does not consider such things as command line decoding or text-string naming of files to be part of the operating system (but provides programs which can be used to do these things). The Alto operating system is open to a far greater extent than the majority of systems. Since any or all of the system code can be discarded, the operating system is really defined only by its disc representation of files and its network representation of messages.

TRIPOS is like Thoth in the sort of target machines it is intended for, though the latter is rather more elaborate, catering for multiple users and supporting virtual memory, multiple address spaces and swapping. Both are based on message-passing, but differ significantly in the mechanism provided. The length of Thoth messages must be known to the system, as they have to be copied between address spaces. Thus all messages are the same size (8 words). As TRIPOS messages are not copied, they can be of any size and the size need not be known by the system. The message-sending primitive in Thoth blocks execution of the task calling it, whereas that in TRIPOS (QPKT) does not. This means that a multi-event process in Thoth must be constructed from a team of tasks, with at least one for each outstanding message. The TRIPOS primitives permit a multi-event process to be written either as a single task (possibly with several coroutines, as explained above), or as several tasks in the Thoth style.

UNIX is another minicomputer system rather more complex than TRIPOS, and lacking a general inter-process communication facility, so not amenable to direct comparison. However, Ritchie's recommendations to system writers based on his retrospective look at UNIX [40] show that the designers of the two systems had some common criteria. He says that there is no excuse for not providing a hierarchical filing system, because it is useful for grouping files, and efficient. He considers the notion of a "record" to be obsolete, and supports the idea that all files should consist of a sequence of bytes, with only one format for text files. He also recommends that systems should be written in a high-level language which encourages portability. The design of TRIPOS fits in with all these guidelines.

The suitability of BCPL as a language for portable systems has been convincingly demonstrated, both from the fact that the language has been implemented on over 25 different machines, and from the successful use of it and close derivatives for writing at least four systems apart from TRIPOS. Its complete lack of types allows the programmer great freedom, and forces him to take care when coding (encouraging good programming practices), but means that even a small error can cause a program to fail catastrophically. Having no types can also lead to run-time inefficiency, particularly on a byte-addressed machine where the lack of a special pointer type means that all pointer values must be converted from BCPL to machine addresses before being used. The lack of an integer type means that portable programs should not assume more than 16 bits in an integer, giving a restrictive maximum value.

For these sorts of reasons, the designers of Thoth plan to introduce some types into their language Eh, but point out that their desire to do this is based entirely on considerations of efficiency and portability. The language C was developed from BCPL for UNIX by adding a few types. However, the UNIX philosophy of making each program do one job has been adhered to, by separating the functions of type-checking and compiling. The compiler has to know about those types included for efficiency or portability reasons, but its job is compiling, so type-checking is left to another program, which also reports on dubious coding practices likely to produce non-portable software. Thus the type rules may be broken if necessary, without preventing the program from being compiled. The authors of Pilot say that all of the protection in that system depends ultimately on type-checking in the Mesa language.

The authors of TRIPOS rarely found the lack of types to cause trouble, so do not plan to extend the language. Producing variants of a language has the effect of reducing portability, and it is useful for TRIPOS to be able to run BCPL programs written on other systems, and for programs written under TRIPOS to be easily transferred to other BCPL installations.

# CHAPTER 4

# THE CAMBRIDGE DATA RING

## 4.1 Introduction

This section gives a brief description of the Cambridge Ring hardware, and mentions the interfaces to the machines on which TRIPOS runs in the Cambridge Computer Laboratory. It is included in order to provide sufficient context for the following chapters and does not describe work done by the author. A more detailed description can be found in [17]. It goes on to outline the protocols in general use at Cambridge; their use from TRIPOS is described in chapter 5.

## 4.2 Overview

The Data Ring is a high bandwidth local area communications medium - i.e. it is intended to connect computers within a single building or small group of buildings near each other. The service Ring in the Laboratory joins over thirty machines of various types, including Z80s, Novas, PDP11s, CA LSI4s and the University's IBM 370. Many of these computers are dedicated Ring 'servers', providing generally useful services such as disc storage, terminal connection, machine allocation, accurate date and time, etc.

Data is transferred in units of 16 bits. The point-to-point data rate is of the order of one megabit per second.

## 4.3 Hardware

The Ring consists of a loop made of two twisted pairs of wires, containing a number of repeaters. Data is carried in a fixed number of packets, which continuously circulate. The number of packets is limited by the electrical length of the Ring, and the clocking frequency used - this determines how many bits are stored in the wires and in the repeaters. In fact, what is circulating is an integral number of packets (typically one to three) followed by a gap shorter than a packet.

The packets are 38 bits long and contain 8 bits of source address, 8 bits of destination address, 16 bits of data, a 2-bit transmission returncode, and four bits used for framing, control and error detection.

Each computer is connected to the Ring via a station, which is in turn connected via its own repeater. Stations have a transmission and a reception half, and the two halves appear independent to the host computer. In particular, a station can transmit to itself - a facility which has proved very useful while developing Ring programs. Every station has a unique 8-bit address in the range 1 to 254.

### 4.3.1 Transmission

The basic transmission operation provided by a station is to send one 16-bit data packet to a specified destination station. It achieves this by waiting for the first empty Ring packet to pass its repeater, claiming this, and loading into it the data word and the addresses of the source (itself) and destination. It then waits until the same packet has been all the way round the Ring. When it returns, it is compared with the packet sent (as an error detection measure), and the 2-bit transmission returncode is extracted and made available to the host computer.

The four possible transmission returncodes are as follows:

- Accepted - the packet was received by the destination station.
- Busy - the destination was not yet ready to receive another packet.
- Unselected - the 'source acceptable' register (see below) was set to a value other than our address or 255 ('anywhere').
- Ignored - No station received the packet: that Ring address either does not exist, or its station is turned off.

Two hardware features are included to limit the load that any one station can impose on the network:

(i) It is not possible to reuse the same Ring slot for consecutive transmissions. Therefore, every station gets an equal chance to grab empty slots, even when there is a lot of traffic.

(ii) Artificial delays are introduced to limit the retry rate after a 'busy' or 'unselected' transmission failure.

### 4.3.2 Reception

The reception operation provided by a station is to receive one packet from a source controlled by the source acceptable register. If this register is set to a value from 1 to 254, then only packets from the station of that address will be accepted; transmissions from anywhere else will be rejected 'unselected'. The value zero means 'receive from no-one'; it is used at times when the software is not ready to take anything more. The value 255 enables reception from any station. The actual source can be read from another register within the station.

## 4.4 Interfaces

The nature of the Ring interface is rather different on each of the machines on which TRIPOS runs. A brief description of each follows:

PDP11: one interrupt per 16-bit hardware packet on both transmission and reception. It is also possible to poll a status flag to see when an operation is complete, rather than using interrupts.

Nova: one interrupt per hardware packet.

LSI4 with picoprocessors: Direct Memory Access (DMA) transfer of vectors of 16-bit words to/from store, with an interrupt at the end of the vector. Automatic (and indefinite) retry on transmission 'busy'. No processing of transferred data. This was the original interface used, making use of CA's "intelligent cables".

LSI4 with 'Type 2': An intelligent interface incorporating an 8X300 bipolar microprocessor, the design of which is described in [14]. (The name is historical - it was the second design of Ring interface to include a microprocessor.) The 'Type 2' provides full handling of the Basic Ring Transport Protocol, including checksum calculation, holding several outstanding reception requests, retrying on transmission errors, and the ability to receive data split over more than one block into a single buffer. The data part only of each block is transferred to/from store using DMA.

General Automation 16/220: This computer will also use the 'Type 2' interface.

## 4.5 Protocols

This section describes the three main protocols in use on the Cambridge Ring - the fundamental one, and two others based on it:

- Basic Ring Transport Protocol
- Single Shot Protocol (based on BRTP)
- Byte Stream Protocol (based on BRTP)

### 4.5.1 Basic Ring Transport Protocol

Most Ring traffic is sent in the form of basic blocks* [48], each of which is a sequence of 16-bit hardware packets in the format shown in the diagram.

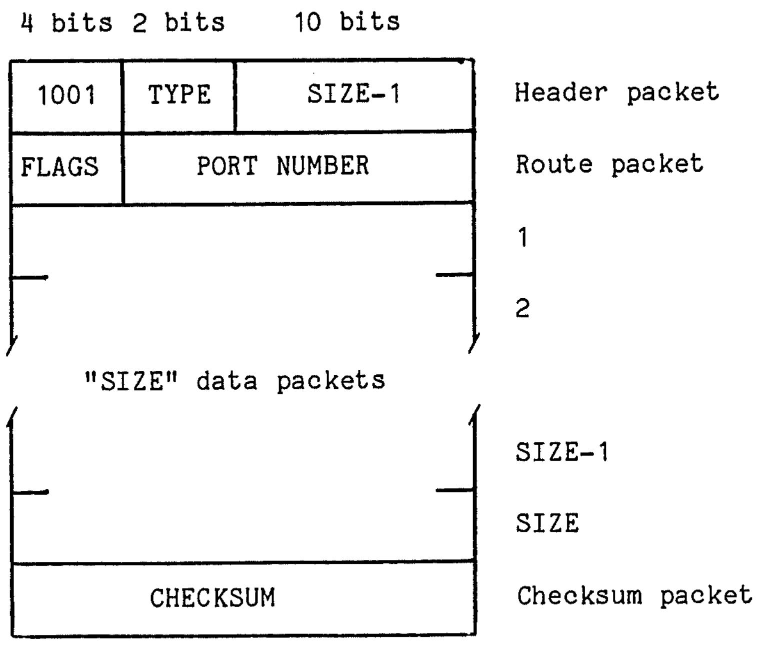


Fig. 7: Ring Basic Block

--------------------

* Terminology: two alternative naming schemes for the units of Ring transmission are in use. In the first, the 16-bit quantity is called a hardware packet (or just packet), and the structure shown above is called a basic block. In the second, they are called mini-packet and packet respectively (by analogy with other networks, where the transmission block has the term "packet"). As the basic block is the unit mentioned most, the former scheme has been used in this dissertation, to avoid over-use of the word "packet", which has yet another meaning in TRIPOS.

The header packet contains 4 bits of fixed pattern (as a weak form of identification), a 2-bit type code, and the length of the data part of the block (1 - 1024 stored as 0 - 1023).

The type codes have the following meanings:-

| | |
|---|---|
| 00 | block with calculated checksum |
| 01 | block with zero checksum |
| 10 | immediate data: the header packet is itself the whole block, and the COUNT field contains the 10 bits of data. |
| 11 | unassigned |

The route packet contains a 12-bit port number, used to direct the block to the right process within a machine. The four flag bits are at present unused. (A possible use for one might be to indicate that the block is addressed to an intelligent Ring interface, rather than the host - e.g. to instruct it to reload the host.)

The checksum is a 16-bit end-around-carry sum of all the other packets in the block, or zero for blocks of type 01. (Note that a calculated checksum cannot be zero - that could only happen if all the other packets were zero, and the header is never zero.) The checksum is there more to aid detection of block framing errors (e.g. because the wrong packet was identified as header, or because transmission was abandoned part-way through a block and restarted), than to protect against Ring data errors (which are very rare).

In practice almost all blocks have a calculated checksum, though most systems will also accept those with zero checksum. The latter block type would be useful only in situations where the transmission rate was so high that calculation of checksums were a serious overhead (on most machines, it takes a time comparable with that to send the block), and the occasional error would not matter - for example, something highly interactive like transmission of graphical display files or voice.

The immediate data block type is also used for only a few specialized applications. It was once envisaged that this might have been the vehicle for character traffic to and from unintelligent terminals connected directly to the Ring, as it has room for an 8-bit byte with 2 bits left for control information; things did not turn out that way.

There are some conventions about how systems should behave when sending and receiving basic blocks:-

- Once transmission of a block has started, the rest should be sent as fast as possible.
- Conversely, once a system has received header and port, it should accept the rest as fast as it can (or set its source acceptable register to zero briefly as an indication of rejection).

These conventions tend to rule out the possibility of interleaving transmission or reception of blocks at the hardware packet level because most hosts could not achieve this at full Ring speed.

Basic blocks are not acknowledged. Protocols using them should be prepared to cope with an occasional block disappearing without trace, as it is legitimate to receive all blocks and simply discard those which have bad checksum, are too long, or are not currently expected.

The fact that a sender is not reliably informed whether or not his block was received is one of the most serious deficiencies of this protocol, and stems from it being implemented in software on hardware which provides return codes only at the packet level. The lack of an immediate indication of rejection is often the cause of programs running very slowly, because they spend most of their time waiting for replies to blocks which appeared to be received but were not (e.g. because the destination was not quite ready at the time of transmission). Ring driving software tends to be carefully tuned to give reliable transmission of basic blocks. It has several times been the case that a change in the timing characteristics of the Ring has made it necessary to re-tune driving software in many machines.

### 4.5.2 Single Shot Protocol

This is a protocol used for making remote procedure calls to services when the amount of data involved in the call and the reply is limited. It is defined in [30].

SSP request

| word | | | |
|---|---|---|---|
| 0 | type | flags | block type indication |
| 1 | REPLY PORT | | reply expected on this port |
| 2 | FUNC CODE | | part of service's Ring address |
| 3+ | service dependent arguments | | |

SSP reply

| word | | | |
|---|---|---|---|
| 0 | type | flags | block type |
| 1 | PORT or 0 | | port for rest of conversation (if any) |
| 2 | RETURN CODE | | zero iff successful |
| 3+ | reply data | | |

One basic block is sent to request a service, and one is received in reply. The first three data packets of each block are treated specially; the size and meaning of the rest depends on the service. The diagrams show the data parts only of the blocks.

## 4.5.3 Byte Stream Protocol

The byte stream protocol (BSP) [18] is used to provide pairs of streams (one input, one output) across the Ring, giving an error-free channel with flow control. Facilities are defined for forcing transmission of data, and for resetting the stream pair to a known state.

BSP is built on the Basic Ring Transport Protocol by reserving the first two data words of each block for commands and sequence numbers relating to the two streams. Each block is acknowledged, thus making the protocol immune to blocks being lost, and data is not sent until requested by the destination, giving flow control. Each end may, if it wishes, periodically repeat its last block to the other at times when no data is flowing, providing confirmation that the other end is still alive.

Byte stream connections are initiated by a special OPEN block, which is similar in format to an SSP request block (but with a distinct type code). The data passed is in two parts: that relating to the byte streams, and user data relating to the particular service called. The only BSP data included at present are two blocksizes: the largest that this end will send, and the largest it is prepared to receive.

The reply to OPEN is an OPENACK block, which is like an SSP reply - indeed the type codes are the same. It includes a new port to be used for further transmission, and the called end's ideas on suitable blocksizes.

There is another method by which a new byte stream may be formed, from two existing ones. If a machine has two byte streams open, it may perform a 'replug' operation to connect them together, removing its own association with both. The Session Manager makes use of this mechanism to connect a terminal to a machine it has just booted (see chapter 8).

BSP guarantees to provide reliable communication, but because basic blocks may be discarded without warning, it will not necessarily get information through quickly. It is thus not completely ideal for interactive uses such as terminal streams, though works reasonably well in practice.

## 4.6 The Name Lookup Server

Machines and services on the Ring are known by names which are text strings; in general, it is these names which are used to refer to them and which may be bound into programs. The station number of a particular machine, or the location of a service should not be expected to remain constant from day to day.

In order to make the use of these names practical, there is a machine on the Ring called the Name Lookup Server (or just Nameserver), whose job it is to convert names into Ring addresses. A Ring address is the set of three numbers needed to call an SSP or BSP service: the station number, port number, and function code. The station number of the Nameserver is "written on the wall" (i.e. well known), and is the only station number guaranteed not to change. The primary service provided is on a well-known port and function code. It is an SSP service, in which the argument is a name to be looked up, and the reply is the Ring address corresponding to that name, together with some flags indicating what sort of thing it is the name of (machine or service), and what protocol should be used to talk to it (SSP, BSP, or non-standard). Another flag indicates that the function code has not been supplied; this is used by a service which provides a family of related functions, the codes for which are published in the definition of the service. A further flag is used to warn that the service may be slow to reply.*

The Nameserver runs in a dedicated Z80 microprocessor, for reliability and speed. A subset of the name table is blown into PROM. The full and up-to-date version is kept in volatile store, but is dumped to the Fileserver every time it changes, giving a very high degree of immunity to

--------------------

* There is no precise definition of what 'slow' means. A reasonable interpretation is that a service without this flag will respond in 5 seconds; one with it might take up to 30. In practice, there is a fairly clear separation between those services provided by dedicated microprocessors, which respond in a matter of milliseconds, and those provided under an operating system, where the response time may be measured in seconds.

crashes, which themselves are very rare. Looking up names is an operation which is both simple and fast, so the use of this indirection is not a major overhead.

The Nameserver provides several other services in addition to lookup. These services have their own names, so their ports and function codes need not be well-known. They are:-

- Reverse lookup
  This takes a station number as argument and yields the name of the machine as a string. It is primarily used in order to produce informatory messages.

- Own name
  This returns the calling machine's own name, and requires no arguments.

- Name table listing
  A list of the entire contents of the name table can be obtained by a series of SSP calls, each asking for one entry by its number in the table.

# CHAPTER 5

# RING INTERFACE SOFTWARE

## 5.1 Introduction

This chapter describes the way in which the operating system primitives and structures described above were used to build portable Ring interface software. It discusses the issues involved in handling the basic protocol, arranging for a newly-loaded machine to avoid confusion from Ring traffic meant for earlier incarnations, and in providing network streams in a uniform way to local streams. A mechanism is presented for allowing arbitrary programs to be executed in response to calls from other machines on the network (enabling the operating system to act as a "server" as well as a "client"). The effects of using a microprocessor-based Ring interface to handle the lowest protocol level are reported.

The Ring interface software components fall into five main groups:-

(i) Ring transmitter and receiver device drivers

(ii) Ring system tasks:- Ring Handler
BSP Handler
Ring Services

(iii) Libraries of commonly used routines

(iv) Services:- programs loaded by the Ring services task to provide particular services

(v) Commands:- to start and stop system tasks, transfer files, provide information on the Ring, and access remote Ring services.

The libraries, services and commands are described in Appendix 2. All components except the device drivers are written in BCPL and are identical on different machines.

## 5.2 Ring Device Drivers

All communication with the Ring is done through two device drivers - an independent transmitter and receiver. These are written in assembler to provide a machine-independent interface at the TRIPOS packet level. The transmitter function is to send a vector of 16-bit words to a given Ring station. The receiver function is to receive a specified number of 16-bit words from a particular (or any) station into a supplied buffer.

As the only routing information is that provided by the hardware station addresses, these drivers are intended to be used directly (for reception, at least) by only one task at once, otherwise confusion will result. In general, only the Ring handler task talks to these devices, and user programs and other system tasks never send packets to them.

## 5.3 Ring Handler Task

The Ring handler task (RHT) provides a 'basic block' (see chapter 4) interface to the Data Ring. It acts as a multiplexer, allowing several tasks to use the Ring independently. It is normally the only task in the system to send packets to the Ring device drivers, and is itself used by all other Ring programs. This follows the normal strategy in TRIPOS of hiding device drivers behind a handler task.

Other facilities provided by the RHT are timeouts on both reception and transmission of blocks, retry on transmission errors, and allocation and reservation of ports.

### 5.3.1 Transmission

To transmit data in basic block(s), a packet is sent to the handler giving the address of a buffer containing just the data, the number of data words, and the station and route code to which it should be sent. The supplied data is transmitted in basic blocks with full checksum; no functions are provided for transmission without checksum or as immediate data packets, as the need for these block types has not arisen.

There is no special packet type for sending a large amount of data in more than one basic block - this is done automatically if the buffer size exceeds 1024 words.*

A transmission is abandoned if the destination ignores any Ring packet, or goes busy for more than a timeout period (a few hundred milliseconds). If the receiving station is 'unselected' for the header packet of a block, then the transmission request is put to the back of the queue (so as not to block any others), and retried several times. The most common reason for a station to be unselected at the beginning of a block is simply that it happened to be receiving from someone else at the time transmission started. Since reception of a block can be expected to be over in a short time (a few tens of milliseconds), fairly rapid retrying is sensible for that sort of period. Note that 'unselected for header' can also indicate that the destination machine has crashed, so the number of retries has to be limited.

The situation where the receiver goes unselected during reception of a block is treated as an error not worth retrying. Some machines on the Ring have the strategy of deciding whether they want a basic block immediately after reading the header and route packets. If there is no reception request outstanding which matches the block, then they set their select register to zero for a short period (either timed, or until one packet has

--------------------

* The ability to send data in a chain of blocks was included particularly for the Fileserver 'write' operation, which expects the material to be written in this form.

been rejected 'unselected').[#] Thus the usual meaning is "Go away - I'm not expecting anything from you or on that port", and it is pointless to retry.

However, from the Fileserver, this effect can occur on its command port, or during a multi-block write operation, and then the meaning is "I'm busy - try again". (The number of command processes is fixed: if all are in use, then there will be no reception request on the command port. During a write, the delay can be due to waiting for a disc write to complete in order to free a buffer.) Thus, in practice, the cases when a retry is not expected are the rare ones, so there is some reason for automatic retry in the handler.

This was put in, but was restricted to the occurrence of 'unselected in block' during a multi-block transmission (e.g. a Fileserver write) in blocks other than the first. It proved preferable to retry single block transmissions more slowly from the next level of software, which can be expected to know some of the characteristics of the machine it is talking to. In particular, it is decidedly anti-social to transmit to the Fileserver hardest when it is already very busy.

The TRIPOS packet requesting transmission is sent back only when the whole buffer has been sent, or when transmission has been abandoned (after retries if necessary), with a returncode to indicate what has happened.

### 5.3.2 Reception

The Ring handler task provides two reception modes:-

(i) Reception of a single basic block.

(ii) Reception of a specified amount of data in one or more basic blocks.

--------------------

# Some systems accept all blocks, and simply discard any they do not understand, so 'unselected in block' cannot be <u>expected</u> to happen: it is just a hint.

In both cases the parameters given are:-

- the address of a buffer for the data part of the block(s)
- the size of the buffer
- the station from which to receive (with 255 meaning 'any station')
- the reception port
- a timeout period (in 'ticks')

Requests of mode (i) are satisfied by any basic block matching both station and port and whose data part will fit into the supplied buffer. Those of mode (ii) are satisfied only if the total data size of appropriate received blocks is exactly the buffer size.*

All reception requests have a lifetime. If no suitable block arrives during this period, then the request packet is returned with a result indicating this. For the majority of applications, this is precisely the action required, and often saves other programs needing to issue their own timeout packets to the clock.

For example, consider the operations needed to do a 'single shot' request (simplified):

(i) Set up reception request for reply

(ii) Transmit request block

(iii) Wait for the reception packet to return:
either with the reply,
or having timed out

Only one system program needs an indefinite timeout: the Ring services task, which maintains a request on its port all the time. It did not seem worth providing the function for just this program; instead, the RST uses a long timeout, and bounces the packet each time it expires.

--------------------

* This mode was provided primarily for the Fileserver 'read' operation.

Both basic blocks with checksum and those with zero checksum are accepted. All blocks are fully received, but are discarded if they match no request, match but are too long, or have a bad checksum. A timer is started on reading a block header: the rest of the block has to arrive within a short time (a few hundred milliseconds), otherwise reception is abandoned, and a new header expected.

The third reception action available is to cancel a previously issued reception request or requests (specified by station and port). On being cancelled, packets containing reception requests are simply 'dropped' rather than sent back, thus saving messy loops to collect them in the client program. The link field is set to NOTINUSE, and the identity field to point to the RHT, so that the packet is in the same state as if it had been through QPKT and TASKWAIT, and can be resubmitted without further modification.

Use of this cancel facility has turned out to be rather rare; almost always the port releasing function (see below) does the job instead.

### 5.3.3 Port Allocation and Reservation

Ports for transmission are always chosen elsewhere (either given in a Nameserver address, or in a message as a reply port), so no problems arise over allocation or use in the transmitting machine.

Ports for reception are used in two ways:-

(i) Some are "well known" - used for requesting services and opening byte streams - and usually recorded in Nameserver entries. The prime example of this type is the port used by the Ring services task.

(ii) Usually, the requirement is for a port number which is known not to be in use for any other reception in the same machine. Apart from this constraint, its value has no significance.*

Thus, the following port functions were provided in the RHT, which is the logical central manager of ports:-

(i) Reserve port N.
The Boolean result indicates whether or not it was already reserved.

(ii) Allocate and reserve a new port.
The result is the chosen port number.

(iii) Release port N.
As well as freeing port N, this cancels all outstanding reception requests on that port:

(a) they are no longer sensible
(b) this simplifies client programs, as they always want both operations done

Dynamically allocated ports are chosen from only a part of the range offered by 12-bit numbers. This leaves the rest free for use as "well known" ports (of which there are only a very few), and for programs which prefer to do their own allocation from a private range of ports for reasons of speed (e.g. the Fileserver file handler).

Care is taken to try to choose a random starting value for port allocation on each run of the RHT. This is necessary to avoid trouble when rebooting a machine immediately after it has crashed.

---

* Reception requests specify a (station, port) pair, so it usually would not matter if two concurrent receptions happened to use the same port, as long as they were from different stations. However, in a multi-tasking environment, there is always the possibility of two parallel Ring transactions with the same station.

Consider the following:-

- Machine is booted
- A virtual terminal is connected to it via a byte stream
- An untested program is run, crashing the machine
- Machine rebooted
- New terminal connection made.
  With simple choice of a starting port, the byte stream will get the same reception port as last time.
- Terminal concentrator decides that the first byte stream has worn out, so sends a BSP CLOSE.
- The CLOSE is received by the second stream!

The new connection is killed soon after it has been made. As typical times allowed before abandoning byte streams are a minute or two, there would have been plenty of time to have started editing, say, before the connection was lost.

This problem can be made very unlikely by choosing a random starting value for port allocation. The RHT originally used the time-of-day in the root node to generate this quantity. However, this technique was useless when the RHT was loaded as part of the resident system, as the task started before the time-of-day had been set, and so the value read was always the same. Neither could a random number be obtained by such ploys as adding up words of store, as the same value would be obtained each time.

The solution finally adopted is, on starting, to count in a tight loop until the TICKS field in the root node first changes, and to use the count value. Hence the first port depends on the phase of the real time clock at which loading finishes, and is likely to be different each time.

## 5.4 Byte Stream Protocol Handler

The BSP handler task was written to implement the byte stream protocol defined in [18].

BSP provides pairs of error-free channels (one in each direction), which will transfer data as a stream of 8-bit bytes. The streams have flow control - i.e. if the reading end cannot take information at the rate the writing end can generate it, then "back pressure" is produced in the stream, which will control the rate of writing. Mechanisms are defined for resetting the stream pair to a known state, flushing data through all the levels of buffering, and closing streams down either tidily or forcibly.

The BSP handler is a single task which can handle one or more streams. It is loaded when the first byte stream is opened, and deletes itself when the last stream is closed.

The definition of the protocol presents it as a state table indicating what action to take and what new state to enter, as a function of the current state and the event which has just occurred. Events are such things as a block arriving from the Ring, a timeout occurring, or a buffer becoming full or empty. The handler task reflects this model, and contains a large BCPL SWITCHON statement to direct flow of control according to the state/event pair. Some thought was given to structuring it as a set of coroutines (one for each open stream) as described in chapter 3, but it was decided that this would not simplify the structure, as each stream has some multi-event character. Instead, there is a control block for each open stream. All events correspond to packets arriving at the task. The stream associated with the packet is identified and its control block passed as a parameter to the routine which processes the packet.

### 5.4.1 Stream Interface

The BSP handler was designed to use the normal TRIPOS stream interface, so that existing programs would be able, without alteration, to use byte streams. This is similar to the uniform treatment of local and network streams in Pilot [34]. Thus, each half of the stream pair has its own SCB, which can be used as an argument to SELECTINPUT/SELECTOUTPUT. Bytes are written and read with the standard routines WRCH and RDCH. This proved to pose significant problems, mainly because byte streams come in mutually dependent pairs, whereas BCPL regards all its streams as independent.

However, it does mean that byte streams can be used by programs which have no special knowledge of them.

If a BSP connection is closed by the other end, then the local streams become as if connected to the pseudo-device "NIL:". I.e. the output stream is an infinite sink, and the input stream is an infinite source of ENDSTREAMCH.

### 5.4.2 Opening byte streams

A pair of byte streams is opened by calling one of the normal library routines FINDINPUT and FINDOUTPUT to open a stream to the pseudo-device "BSP:". The input or output stream (as appropriate to the call) is returned, and the other stream of the pair is placed in the global variable RESULT2. If the stream cannot be opened, then the result is zero, and RESULT2 contains a fault code, in the usual way.

The service to which the stream is to be connected is specified by its service name - a string which can be looked up in the Nameserver to yield the Ring address of a BSP service. The name is included in the "BSP:" string as follows:

Either:

```
        instream  := findinput("BSP:servicename")
        outstream := result2
```

or:

```
        outstream := findoutput("BSP:servicename")
        instream  := result2
```

### 5.4.3 Closing streams

BSP streams are closed with the normal library routines ENDREAD and ENDWRITE. Calling either of these breaks the connection, and kills both sides of the stream pair. Hence, only one of then should be called.

ENDREAD causes a BSP CLOSE to be transmitted, which forcibly closes down the streams (destructively).

ENDWRITE closes down tidily by sending the last buffer (even if empty) with the 'close request' and 'force transmission' flags set. This results in the byte streams dying only when the other end has processed all the data sent.

### 5.4.4 Streams to remote files and printers

The BSP: device allows remote files and printers to be accessed by programs with no knowledge of the network, making them available to ordinary commands. A string separated by a slash from the ring service name will be included in the user parameter area of the BSP OPEN block, so can be used to convey a filename or document title.

For example, if the name of a printing service is "TITANPRINT", this might be used as follows to produce hard copy of the output from a command:-

```
EX :C TO BSP:TITANPRINT/Commands
```

[Examine the directory ":C", sending the list of filenames to the printer, with document title "Commands".]

TRIPOS provides Ring services enabling byte streams to read and write files. Suppose the machine called NOVA provides such services with the names "READ-NOVA" and "WRITE-NOVA": other machines on the Ring can then directly access its files. For example:-

```
type bsp:read-nova/:g.libhdr

ex :c to bsp:write-nova/:brian.commandlist
```

## 5.5 Ring Services Task

The early communications programs in TRIPOS were all rather specialized; in particular, a special program had to be ready running in the destination machine, expecting something to arrive on a fixed port number (built into that program). This was adequate for early experiments, but was obviously no good for long term use. For example, transferring a file between two machines involved typing at the consoles of both.

The 'Ring services' task (RST) was written as a general mechanism for dynamically firing up other tasks in response to a request received from the Ring. On starting up, it read a driving file containing on each line a port number and a filename. From this it built a corresponding list in core, and issued a reception request to the Ring handler task on each of the ports. Whenever something arrived on one of these ports the program was loaded from the corresponding file, a task made from it, and a packet sent to the task to start it and to give it the received basic block which was its reason for existing. The reception request on that port was immediately reissued.

Installing a program to provide a new Ring service then became an easy matter, as all that had to be done was to include an extra port-to-filename mapping in the driving file (a text file which could be edited in the normal way). The only overhead in running was the store required to hold the new map entry.

The Ring services task was relatively small, so could usually be left running. It meant that the machine could offer a variety of Ring services, but each was actually loaded only when required.

Consideration was given to running the called service other than as a new task. It could have been executed as a subroutine of the RST (e.g. using CALLSEG), giving some advantage in the total store required (as a new task requires its own global vector and stack), and slightly simplifying the code of the RST. This would have meant providing the RST with a reasonably large stack, which would be sitting round unused most of the time, so calling the service as a coroutine (thus giving it a dynamically allocated stack) would have been a better alternative. Running within the RST is only suitable for services which can generate their reply quickly, send it, and finish. In practice, very few turn out to be of this nature: most common are things like file transfer and remote logging-on which are going to take some time to complete, and so ought to run in a task independent of the RST. Thus, it was decided to make the RST always start up a new task for every service call.

The original RST performed no checks on a received block, but merely passed it on to the program corresponding to the port on which it was received. Two other changes brought about a need for an alteration in the method of operation:-

(i) The 'SSP request' and 'BSP open' block formats came into general use. These each contained a type byte, enabling them to be recognized, and a 'function code' field.

(ii) The Ring handler interface was modified so that a buffer was supplied by the user with each reception request, rather than a buffer being obtained from free store when a basic block arrived from the Ring.

The definition of the two block types expected to initiate a Ring service meant that the RST could refuse to respond to anything which did not resemble one of them. It was originally intended to use the function code field to specify various forms of each service (e.g. for a date service, whether the answer was expected as strings or numbers). However, a result of (ii) was that it became very expensive in store usage to maintain several reception requests, as each required a buffer big enough for the largest expected block. Thus, in the revised version, all services provided by a machine had Ring addresses with the same port, and were distinguished by function code only. Only one reception request was needed, on this fixed port.

At the same time, the reading of the driving file was delayed until a block was actually received, thus saving the resident store previously used to hold a copy of this information.

Commands were provided to run and stop the RST. In the processor bank machines, it became a resident task (though it could be killed, as it was not an essential one).

## 5.6 Driving an Intelligent Ring Interface

During 1981, all the LSI4 computers in the Processor Bank were equipped with 'Type 2' microprocessor Ring interfaces [14] (see also chapter 4). These provide handling of the Basic Ring Transport Protocol, and the Ring devices and handler task were rewritten to take advantage of this. The interface to the Ring handler seen by other tasks remained identical.

The advantages of having this lowest level of protocol performed outside the main CPU are:-

(i) The CPU is relieved of the burden of checksum calculation on both transmission and reception. The speed of the Ring is such that the time taken to calculate a block checksum is comparable with the transmission time of the block.

(ii) The interface to the 'Type 2' transmits from and receives into buffers containing just the data part of the basic block, meaning that it is possible to place buffers end-to-end in store without the need for copying or multiple device requests, to insert or strip off the header, port and checksum.

(iii) Transmission retries on 'busy' and 'unselected' are automatic, as is timing out of incomplete received blocks.

(iv) The Ring interface will hold several outstanding reception requests, interrupting only when a valid and wanted block has arrived. It will also receive data of a specified length which arrives in more than one basic block (as used in the Fileserver read operation).

### 5.6.1 Device Drivers

The 'Type 2' interface is controlled by presenting to it codewords, which are small vectors of store describing the required transmission or reception, and providing slots for return codes to be passed back.

The transmitter device remained broadly similar in operation to that for simpler interfaces. Each TRIPOS packet to it requests the transmission of one basic block. The order of arguments in this packet is chosen so that part of the packet itself can be used as the codeword.

The reception device has become rather more complex, due to the 'Type 2's ability to have more than one reception codeword outstanding at once. This means that the device driver must process each TRIPOS packet as it arrives, presenting the corresponding codeword to the 'Type 2' (again formed from part of the packet to relieve the driver of any store allocation problems), and transfer the packet to an internal queue.

On an interrupt, the corresponding codeword address is passed back by the interface. Thus, the packet which initiated the reception can be found, removed from the internal queue, its result fields can be filled in from the codeword return codes, and the packet sent back.

As the device's work queue is always empty (interrupts are disabled for the transient period inside QPKT when it is not), a reception cannot be cancelled by a call of DQPKT. Instead, an extra packet type is accepted by the driver, to cancel a previously issued reception request. It first tells the interface to forget the codeword, and then 'drops' the original request packet.

### 5.6.2 Ring Handler Task

The job of the RHT was simplified by the introduction of the 'Type 2' interface, as it no longer had to construct and dismantle basic blocks. However, some parts of the basic Ring transport protocol not handled by the new interface remain:-

- Transmitting buffers of more than 1024 words in length as several basic blocks; retrying on 'unselected in block' for blocks other than the first.
- Providing timeouts on reception requests:- i.e. if this request is not satisfied within N seconds, cancel it.

The overall reduction in run-time size of the Ring handler and devices was not as great as might have been hoped. The transmitter device driver remained much the same size, the receiver grew somewhat, the handler shrank by a few hundred words, but required some extra workspace to provide a pool of packets to send to the reception device.

The effect on the speed of Ring transmission and reception was a small increase. Although the real time taken for transmission or reception was not greatly altered, the amount of CPU time used decreased considerably. Heavy use of the Ring used to virtually stop all other tasks, but does not do so with the 'Type 2' interface.

# CHAPTER 6

# DISC IMAGES ON A FILESERVER

## 6.1 Introduction

This chapter describes the first method used to run TRIPOS with a non-local disc, employing a simple interface to the Fileserver* machine on the Data Ring. In effect, the Fileserver was treated as if it were a disc storage server of the simplest kind, providing just the ability to read and write disc pages. Chapter 7 contains an account of the structures provided by the Fileserver. For this chapter, all that need be known is that it can provide files seen as randomly addressable vectors of 16-bit words. The experiment showed both that it was feasible to use a remote disc by re-writing a comparatively small amount of code, and that the Fileserver could successfully be used in a simple-minded way.

## 6.2 Disc Images

The approach used was to replace only the lowest level of disc software in TRIPOS - the disc driver device. In its place was substituted a task, with exactly the same packet interface. This is possible because packets to tasks and devices in TRIPOS are sent in exactly the same way - the only difference is in whether the packet ID field is positive or negative. Thus, no changes were needed to the file handler task.

--------------------

* A convention throughout this thesis is that "Fileserver" is used to refer to the Cambridge File Server.

The 'disc driver' task made use of a single large Fileserver file, treating it as a physical image of a real disc, by considering it to be a series of blocks placed contiguously. The requests made to the driver by the file handler task are each to read or write a single disc block. These were translated into Fileserver reads or writes of the corresponding region of the disc image file.*

The original version ran on a PDP11, and the disc modelled was a 2.5 megabyte RK05 cartridge disc. A Fileserver file of this size was created, and a modified version of the normal disc copy program used to perform a physical copy of an existing filing system disc to the disc image file. The block size used on real RK05s was 256 16-bit words, so the layout of the disc image file was as 'blocks' of this size, written contiguously in block number order. Thus, the word offset of a particular block within the file was 256 times its block number. This layout took advantage, for speed, of the knowledge that the file handler tries to allocate blocks with consecutive numbers to a file, and that the Fileserver's unit of disc allocation is 1024 words for a large object. Thus, no TRIPOS block spanned two Fileserver disc blocks, and reading a TRIPOS file serially would tend to benefit from disc block caching in the Fileserver.

The same disc image 'driver' task was then moved to LSI4 machines - first tests being done with images of floppy discs. At this time, there were two LSI4 computers available, one with floppy discs, and the other with two 80 megabyte discs. The file handler used 20 megabyte sections of these as its logical discs, providing a file system of considerable size and speed for a minicomputer. The machine with the large discs was due to become the Fileserver for the Ring, so TRIPOS would have to move from its luxurious environment. The straightforward solution was clear: copy the old logical disc to a large Fileserver file and use the disc-image driver, so that

--------------------

* A slight infelicity in this was that the packets that the file handler thought it was sending to a disc driver device contained block addresses in terms of cylinder, surface and sector. It produced these from the logical block numbers used within the filing system. However, the disc image driver wanted to know the block numbers in order to calculate file offsets, so had to immediately invert the mapping done by the file handler on each packet.

service could continue with the same filing system.

The doubt as to the validity of this approach came from the anticipated speed of the new system. The logical disc (as with all filing systems!) was nearly full, and the file handler would still need to validate the "disc", in order to make its allocation bitmap. On the fast disc, this process took over a minute. However, the prospect of doing the same operation via the Fileserver, which was, after all, using the same physical discs, led to fears that the restart might take half an hour.

Some trimming and tuning of code followed both in the disc-image driver task, and in the Fileserver itself. The main improvements came from the introduction of "single shot" read and write operations within the Fileserver. Previously, even a small read had required a 3-part interaction:

```
Client:     "I want to read X words from offset Y in file Z"
Fileserver: data (on one Ring port)
Fileserver: return code and other info (on a different port)
```

and a write had required a 4-part one:

```
Client:     "I want to write X words to offset Y in file Z"
Fileserver: "OK, ready"
Client:     data
Fileserver  returncode
```

The disc-image driver was always transferring data in chunks of 256 words, which the Fileserver considered a rather small amount. Thus, the above protocol, designed to allow very large transfers, was rather heavyweight. The new single shot operations reduced both read and write to single interactions, combining the data and returncode on read, and the request and the data on write:-

Single Shot Read:

```
  Client:     "I want to read X (<=256) words from offset Y in file Z"
  Fileserver: Returncode + data, all in one block
```

Single Shot Write:

```
  Client:     "I want to write X (<=256) words to offset Y in file Z"
              + data, all in one block
  Fileserver: Returncode
```

The introduction of these functions simplified the code necessary for small transfers. The time taken to write 256 words to a file fell from 105 ms by the full mechanism to 65 ms using the SSP method (measured on an LSI4/30, writing to scattered regions of the file to defeat the Fileserver's cache). The time to read 256 words stayed the same, at 64 ms. The gain from having one less Ring block is absorbed by the extra copy operation needed to extract the raw data from the reply.

The transition to disc images went very smoothly, and the new system proved quite reasonable to use - as fast as a PDP11 with local RK05, for instance. It remained in use as the main TRIPOS system for some months.

## 6.3 Multiple Discs; Virtual Disc Description

It soon became desirable to have more than one disc image, for different people and departments. These were created in various sizes, corresponding to the real discs from which they were copied. Sizes used were 300 Kbytes (floppy disc), 2.5 megabytes (RK05 cartridge disc), and 20 megabytes (logical pack area of CDC 80 megabyte disc).

The start of the Fileserver file for each image contained a description of the disc being simulated, so that the MOUNT command could pick up this information automatically, and pass it on to the file handler task for the mounted disc image. Thus, the MOUNT command could be used in the normal way, and users did not need to be aware of the changes in the underlying mechanism.

## 6.4 Sharing Disc Images

A disc image could not be fully shared for writing, as each file handler task would have had its own copy of the allocation bitmap. Sharing for reading was safe. Sharing where one machine was allowed to write, but all others only to read, was possible with the chance of occasional confusion for the readers. (An object could change while being read; also, local block caching could lead to use of an out-of-date directory block.) In practice, the main filing system image was shared in this way for some time, and little trouble was found.

Completely safe sharing of images would be both expensive and unsatisfactory. The allocation bitmap would have to be kept on the Fileserver, in a special file so that it could be indivisibly inspected and updated. Access to this map could become a serious bottleneck if there were many clients, and there would always be the danger that it could become corrupted over a long period of use. As the Fileserver's unit of interlock is a whole file, some private arrangement would be needed to lock TRIPOS directories and files, such as writing a flag to the header block of them. It would be difficult to arrange for these interlocks to time out if a client machine crashed leaving objects open.

## 6.5 Comparison with a Simple File Server

The style of use of the Fileserver described in this chapter is similar to the expected mode of use of a simple file server such as WFS (see chapter 1 and [46]). WFS provides files which resemble virtual discs, and the files are read and written in pages (with the somewhat bizarre size of 492 bytes). The principal advantage of WFS over the Cambridge Fileserver for "virtual disc" use is that it supports the explicit deallocation of pages, allowing disc space to be released.

Locking in WFS is at the file level, as in CFS. However, it does permit a small amount of client data to be stored with each page, and checked on every access to that page, so a client filing system could use this to provide interlocks in terms of its own files and directories. The main problem with this is that nothing would ever time out such interlocks if the client machine which set them crashed.

It is probable that a file server which provides only simple page access has an efficiency advantage over a more elaborate file server used in a simple way. There are two ways in which the existence of higher-level functions within the file server has an indirect degrading effect on the performance seen by clients making only simple use of it. Firstly, if there are any clients making use of the higher-level functions, then the file server is consuming processor and disc time for their benefit, slowing its response to other users. Secondly, the presence of code for these functions reduces the amount of cache space available in the file server, meaning it has to perform more disc transfers.

## 6.6 Use of Disc Image Files: Summary

The experiments in using disc image files proved worthwhile, both as a simple way of bringing up TRIPOS on a machine with no local disc, and in showing that the Fileserver worked well when used in a manner for which it was not specifically designed.

Some merits and disadvantages of the disc image approach are given below:-

**Merits**

- Only the disc driving code need be rewritten; the file handler stays the same.
- Copying an existing disc into an image file is an easy process (though this should not be a common event).
- The file handler knows how much disc space is still free.
- Utilities written to use the disc driver interface, such as the disc editor, and disc copy programs, will still work.
- It is possible to MOUNT a real disc (e.g. floppies) on a machine using a disc image, and share the filing system code.
- The amount of communications code required is fairly small, as only two Fileserver operations are used. Furthermore, it can be a simple sequential program because the file handler presents disc requests serially.
- More work is done in the (multiple) client machines, and less in the central Fileserver. This exploits the processing power available in a distributed system.

**Disadvantages**

- The TRIPOS machine still has to look after block allocation, so the code to do this, and the in-core bitmap, are retained.
- The file handler expects to be talking to a serial disc device, so does not take advantage of the parallelism offered by the Fileserver.
- Machines cannot fully share a disc image for writing, as the block allocation map is not held centrally.
- The RESTART task must still run after each booting, to check the filing system and build the allocation map.
- After a period of use, the whole file will have been written to, and thus disc space in the Fileserver will be allocated for all of it, regardless of how much is actually in use by filing system blocks.

The most serious of the disadvantages was the difficulty of sharing disc images. While there were only one or two machines running TRIPOS, then it was bearable for them to have different filing systems, but for the

Processor Bank (see chapter 8) a new filing system was written, which both allowed sharing by any number of machines, and took advantage of the operations and structures offered by the Fileserver. This is described in the next chapter.

# CHAPTER 7

# THE DESIGN OF A FILESERVER-BASED FILING SYSTEM

## 7.1 Introduction

The use of disc images, and a driver task simulating the disc driver device, described in the last chapter, was an interim measure to enable TRIPOS to use the Fileserver. This chapter describes the design of a replacement filing system, which overcomes the restrictions of disc images, and makes full use of Fileserver primitives. The literature concentrates on the design of file servers; the emphasis here is on the issues involved in designing a client filing system to work with an existing file server. A short description of the Cambridge Fileserver is included.

## 7.2 The Cambridge Fileserver

The Fileserver on the Cambridge Ring is a Computer Automation LSI4/30 computer with three 80-megabyte CDC discs. It provides a basic filing system, on top of which other computers can build their own filing systems. A detailed account of the design can be found in [9], and a summary in [10]. The Fileserver supports two kinds of object:- the file and the index. Each object has a unique name, or Permanent Unique IDentifier (PUID), which is a 64-bit value.*

--------------------

* There are no access controls on Fileserver objects, so protection is achieved by having PUIDs sufficiently long as to be almost impossible to guess. 32 bits of a PUID contain type and disc address information; the remaining 32 bits are random.

A file is a vector of 16-bit words, potentially of very large size (> 14 million words). Blocks of data of any length may be read from or written to any position in a file. Each file has a size recorded in the Fileserver, which is the maximum offset in it which may be used. Space is allocated on the disc only for those parts of the file which have actually been written.

An index is an object which holds references to other objects, and is seen as a vector of PUIDs. An index may contain pointers to both files and to other indexes (or itself), and a PUID may be stored in any number of index slots. Thus, the structure within the Fileserver is a full directed graph. Each disc pack has a root index, from which the whole structure on that pack hangs. Pointers between packs are not allowed, so that packs may be mounted independently. The Fileserver undertakes to preserve an object as long as it is reachable from a root index. Otherwise, its PUID becomes invalid, and the disc space allocated to the object is reclaimed.

Two sorts of file are available - normal files, and special files. The difference lies in the guarantees that the Fileserver gives about the consistency of the stored data in the event of a crash (of either client or Fileserver) during a write operation.

Normal files are intended for the majority of ordinary data storage. If a crash occurs while data is being written to a normal file, then the file may be left with the write only partially complete.

Special files are used for storing information that must always remain self-consistent, such as a filing system directory. Any change made to a special file will either happen completely, or the file will remain totally unaltered. Operations on special files are correspondingly more expensive than those on normal files.

It is possible to open a file (or index) for reading and writing, or for reading only. The Fileserver responds to the OPEN operation by returning a Temporary Unique IDentifier (TUID - another 64-bit value) for the object, which should be quoted instead of the PUID while the object remains open. A TUID is valid until either the object is explicitly closed, or until it times out. The Fileserver protects itself against an object staying open indefinitely because the machine which opened it has crashed, by cancelling a TUID if it is not used for some time (about 3 minutes). Normally, this timeout is long enough not to be a problem; however, if an object is to be kept open for an extended period, then the client must make sure to refresh this timeout by using the TUID periodically.

Opening a file has two useful consequences:-

(i) It gives the client an interlock on the file.

If it is opened for reading, then further requests (from the same, or other, machines) to open for reading will be granted, but no-one will be allowed to open for writing. Conversely, if it is opened for writing, all further open requests will be refused.

Thus, a client can make sure a file is not altered while being read, or is not accessed at all by others while being written.

(ii) For a special file, the operations of opening for writing, and closing, can be used to bracket a series of updates. The CLOSE operation takes a Boolean argument, saying whether the updates since the OPEN should be done, or whether the file should revert to its state before the OPEN. A crash while a special file is open - of either the Fileserver, or the client (detected by the TUID timing out) - has the effect of closing without updating.

An extra operation, ENSURE, is provided, which commits all of the changes made so far but without closing the file.

Most Fileserver operations are of a 'single shot' nature. The exceptions are the full READ and WRITE functions, which are outlined in the previous chapter. The Fileserver can process several (currently three) commands in parallel.

## 7.3 Filing System Design Considerations

The Fileserver-based file handler task for TRIPOS was designed with the following considerations in mind:-

- The filing system within the Fileserver should be capable of being used simultaneously by several machines.
- The code should be portable between machines of different makes, and such machines should be able to share the filestore. In particular, the byte order within words should be defined.*
- It should preserve the existing packet interface to other tasks.
- It should make full use of Fileserver structures and operations to simplify its own job.

Three possible ways of building the filing system with Fileserver files and indexes were investigated, and a prototype handler task written for each. The first two tried to restrict the number of Fileserver objects needed by having only a single, large, master index. The third design had a more complex structure, with one index per TRIPOS directory, but proved the most suitable to meet the above objectives.

The next section outlines the first two designs, and following sections describe the final version.

--------------------

* This was overlooked in the disc filing system: some byte fields were written using the BCPL operator ('%') for accessing bytes in a vector, which is usually code-generated to use the natural byte order for the machine. Thus a floppy disc, for example, written by an LSI4 was not directly readable on a PDP11. The same problem was encountered when UNIX was transported [40].

## 7.4 Filing Systems with a Master Index

### 7.4.1 Why Use A Master Index?

The attraction of organizing the filing system with only one Fileserver index arises from the fact that the objects provided by the Fileserver do not map exactly onto those required by the filing system.

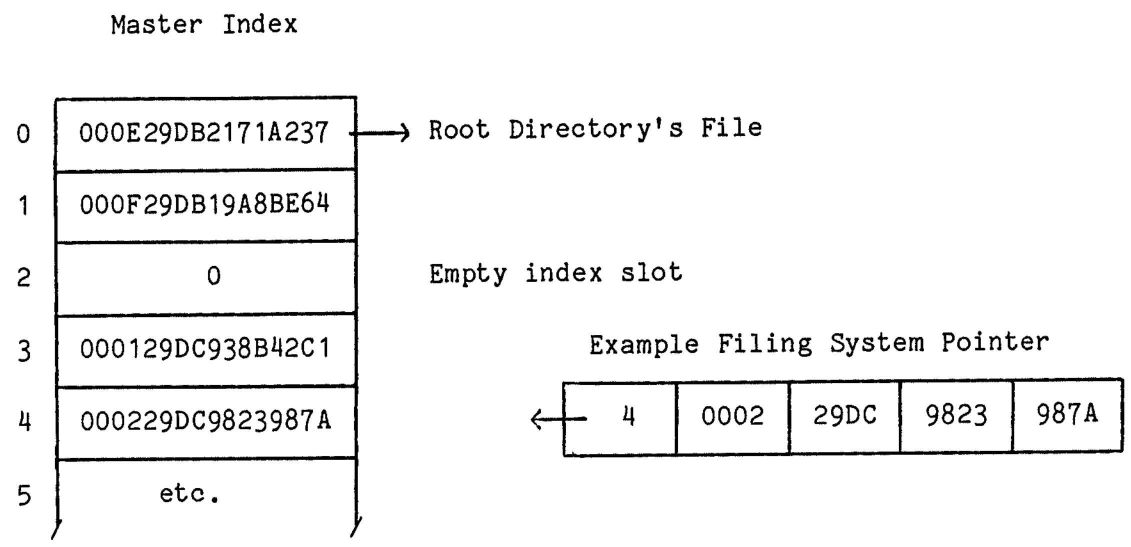


Fig. 8: Master Index and corresponding pointer structure

The Fileserver's idea of a file corresponds very closely with TRIPOS's. However, a directory is an object which needs to contain both text strings (and other data), and pointers to other objects. The data part of a directory must be represented as a Fileserver file, but pointers must be retained in an index, so that the Fileserver knows that the referenced objects are still wanted.

The idea of having a master index was to hide this problem from the majority of the filing system code, by retaining all filing system objects in this index. This was in effect treating the Fileserver as if it were one which provided just files, with no special relationships between them (e.g. WFS, Felix and DFS - see chapter 1). Pointers to objects would become 80-bit values: 64 bits of PUID, and a 16-bit number giving the offset in the index at which the PUID was retained (needed for deletion from the index, but also potentially useful as a consistency check). The low level software would manage the index, making sure that the index entry was created whenever an object pointer was stored in a directory, and deleted when the pointer was deleted. Thus, all higher levels would be able to use pointers freely, and not need to know about indexes. A secondary advantage of this approach is that it keeps the number of Fileserver objects to a minimum, saving some disc space.

### 7.4.2 First Version

The first version of this filing system closely modelled the data structures used by the disc filing system (see chapter 3). Each TRIPOS file and directory was represented by a Fileserver file. Directories were of fixed size, containing some header information, and a hash table. To find an object held in a directory, its name was hashed to give an offset in the hash table. The entry at this offset was a pointer to another Fileserver file, being the start of a chain of directories and files whose names had the same hash value.

As all the TRIPOS objects were represented by <u>files</u> in the Fileserver, they were just data to it, so it could not know where PUIDs were stored, and hence know the structure. All objects in the filing system had to be retained in the master index, in order that the Fileserver should not throw them away.

Allocation of master index slots was done in a first-fit fashion. The index was created with more than enough slots for the number of files anticipated on a full disc; the search for a free one commenced from the position after the last allocation and treated the index as if it were circular.

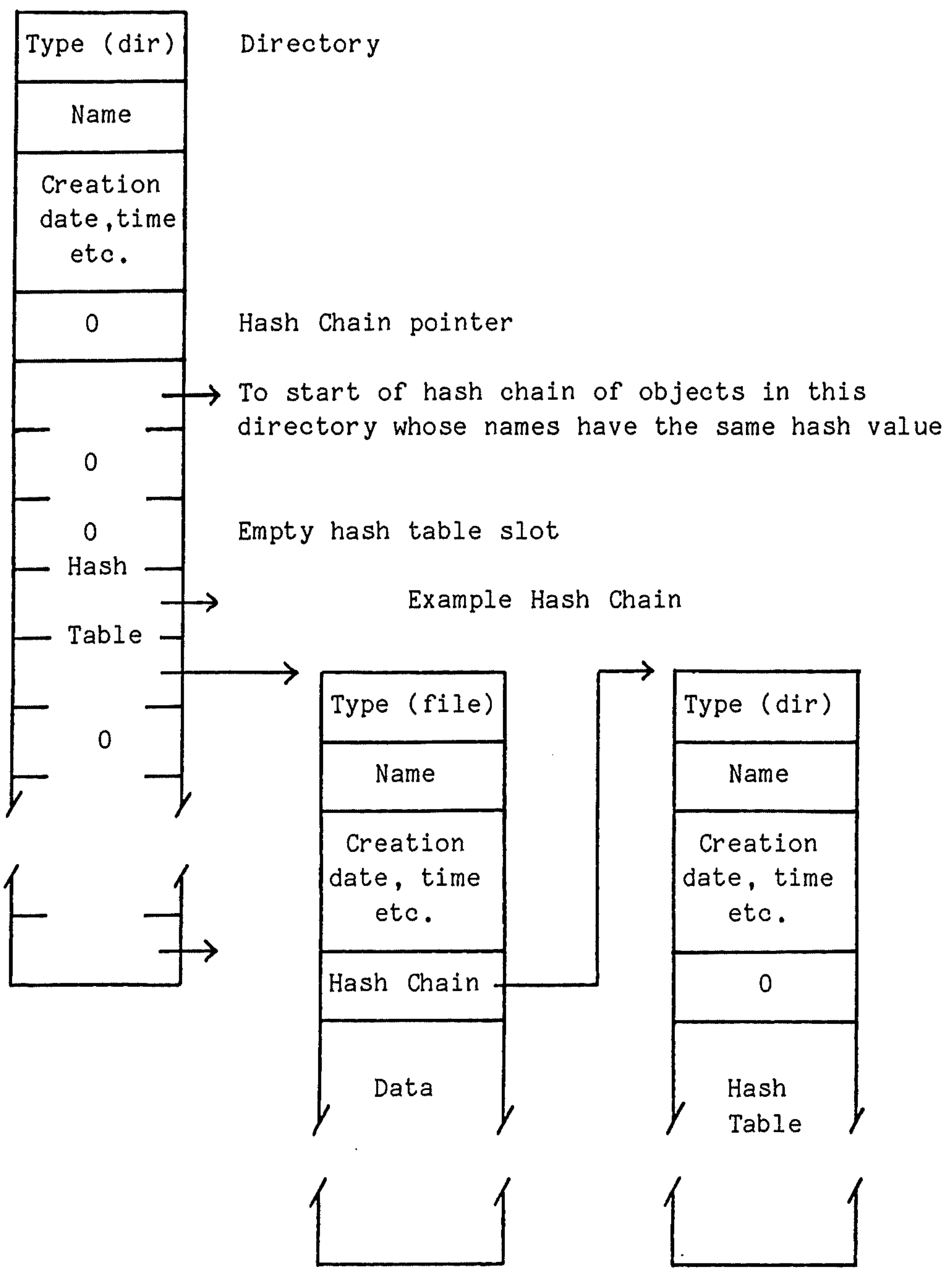


Fig. 9: Directory and File Structure in Version 1

### 7.4.3 Comments on the first version

This filing system has the advantage of being very close in logical structure to the disc filing system. As directories had a fixed size, it was possible to read a whole directory at once, find the first file on the hashchain, and read a reasonable chunk of that. Most hashchains would be of length 1, so two Fileserver reads would usually get some of the data part of the required file into store. The similarity in structure also helped in maintaining the interface to the EX (examine directory) command. It remained easy for it to print the full name of any directory examined.

The major problem with this structure came from the desire to share the filing system by holding interlocks within the Fileserver. Exclusive locks on the files representing TRIPOS directories were expected to be only transient - while altering a hash table entry. However, an exclusive lock would be needed on a file all the time it was open for writing, possibly for hours. As interlocks are enforced by the Fileserver, such a file would become unreadable to all clients except the one who opened it (and knows the TUID). Thus the header information would become inaccessible, hiding the name of the open file, and the rest of the hash chain beyond it. This could only be made to work if the Fileserver supported interlocks which could be tested, but which it did not enforce. The file handlers would respect these when attempting to open a file, but would still be able to look at the header of a file that was open for writing.

### 7.4.4 Second Version

In this version the master index was retained, but the directory structure was more conventional, with file description blocks within the directory itself. Now, each file or directory was a Fileserver file, and the hash chains were completely contained within the directory. Thus it was possible to have a file exclusively open for an extended period without losing access to other files.

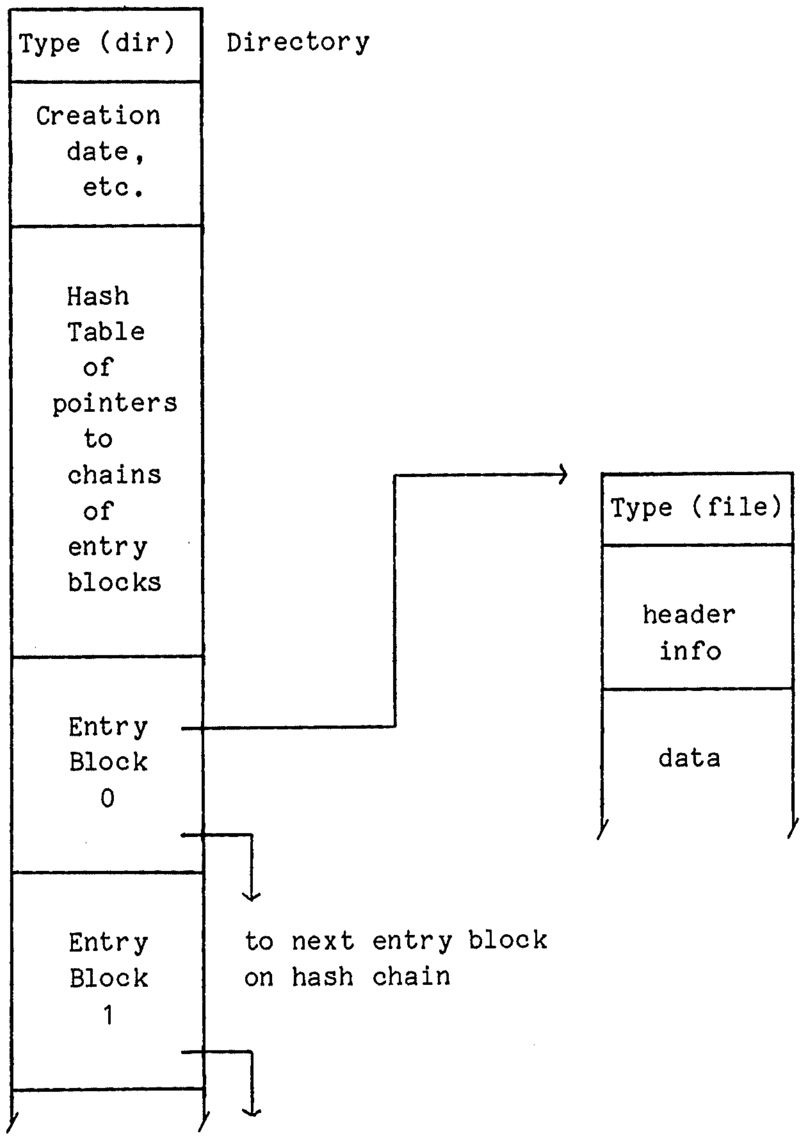


Fig. 10: Filing System Version 2

This structure allowed an object to be held in more than one directory - i.e. the original entry could have 'aliases' of equal status to itself. If a new slot in the master index were used for each entry (holding the same PUID), then there would be no need for the file handler to be aware that the aliases were there; each could be deleted independently, but the Fileserver would keep the file until the last reference to it had gone.

### 7.4.5 Thoughts on the use of a master index

Objects in a filing system tend to contain a mixture of data and pointers to other objects. The Fileserver provides files, containing only data, and indexes, containing only pointers; it would be difficult for it to have mixed objects in any clean fashion. The use of a master index attempts to solve this, by allowing the filing system structure to be built out of files only, and making sure all those files are kept by the Fileserver by retaining them in the single index. It keeps the number of Fileserver objects to a minimum: just one file for each TRIPOS object.

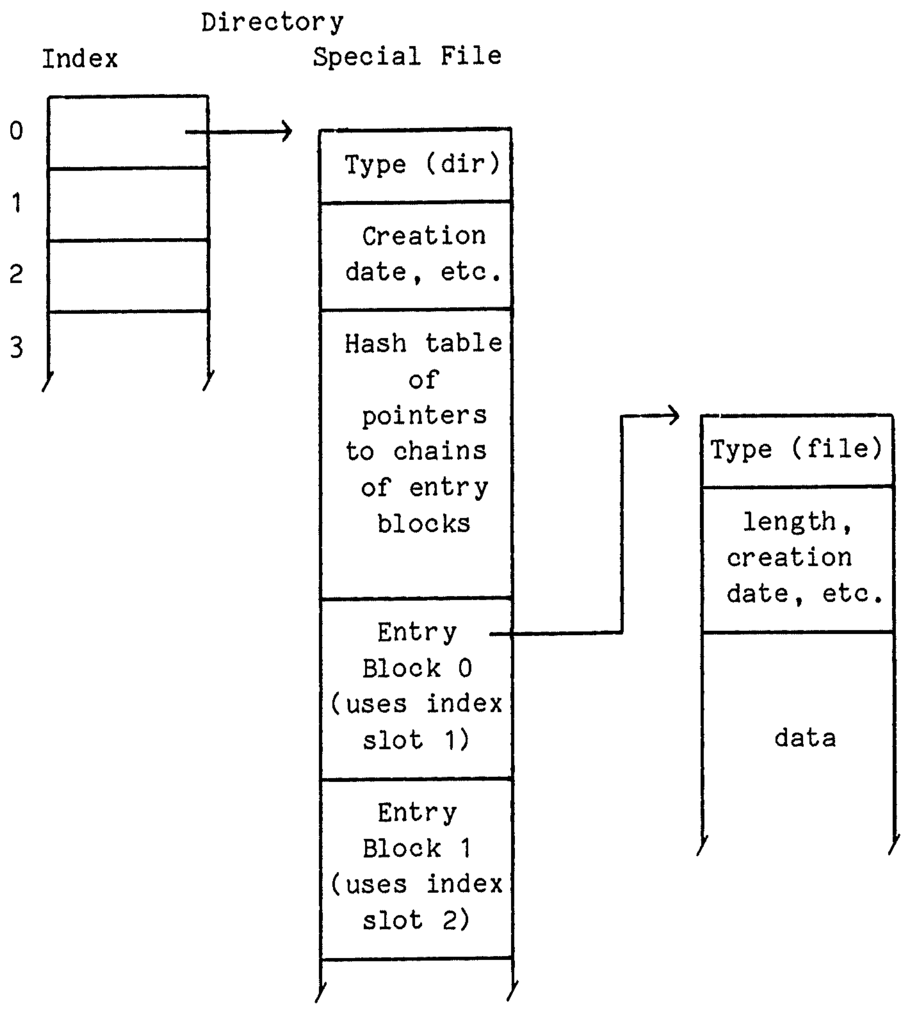
However, using only one index has two infelicities, both really caused by the fact that facilities provided by the Fileserver become unavailable.

Firstly, garbage objects could be expected to accumulate within the index. There are many things that can go wrong with a directory operation carried out over a network. The use of 'special' Fileserver files should ensure that the directories themselves remain intact. The safe method of doing a deletion involves removing a directory entry, and deleting the PUID from the index only when that has successfully completed. If the index deletion fails, then an object is left behind. This is harmless to the integrity of the filing system, but, as the slot will never be reused, the Fileserver will keep the object indefinitely. This can be solved only by writing a special garbage collector for the filing system, and running it occasionally. As the Fileserver has its own garbage collector, it seems sensible to avoid writing another if at all possible.

The second infelicity is that deletion of a non-empty directory requires work by the file handler to remove the objects one by one, which may involve working down a large tree. The disc file handler in TRIPOS has always made the restriction that a directory may not be deleted if it is not empty - (a) because the code to implement arbitrary deletion would be of significant size and rarely used, and (b) as protection. It is something you want to do only occasionally, but disastrous if you do it by accident.

## 7.5 Final Version of the Fileserver Filing System

The third design of filing system, which was the one finally adopted, abandons the use of a master index and employs a structure in which the logical links between filing system objects are accurately reflected within the Fileserver.



Index
Directory Special File
0
1
2
3
Type (dir)
Creation date, etc.
Hash table of pointers to chains of entry blocks
Entry Block 0 (uses index slot 1)
Entry Block 1 (uses index slot 2)
Type (file)
length, creation date, etc.
data

Fig. 11: Final Version of the Filing System

As a directory must hold both data and PUIDs, it is represented by a pair of Fileserver objects - a file and an index. The file must be a 'special' one, as some directory operations require indivisible update of several regions of a directory at once.

A TRIPOS file is represented by a Fileserver 'normal' file. The first few words are used to hold information which relates to the file itself, as distinct from information which belongs in a directory entry pointing to the file.* This includes its length in bytes, when it was created, and who created it. The Fileserver's record of the file's length is not used, as it is in 16-bit words, so has insufficient resolution for character files. All files are created with the maximum size allowed, and the Fileserver's idea of the length is thereafter ignored.# Note that maintaining the length as data at the start of the file actually saves a Fileserver operation when opening a file for reading, as the header information can be read as part of the first read of the file.

In order that a directory may be referenced by a single pointer (i.e. PUID), its special file is retained in the first slot of the corresponding index; the PUID of the index is used as a handle on the whole directory. This also has the desirable property of tightly coupling the existence of the two objects within the Fileserver, in that deletion of all references to the index will also remove the file.

## 7.6 File Handler Interface Changes

Although the original intention was to retain the existing packet interface between the BCPL library (BLIB) and the file handler, it was decided that several changes would be desirable for future use of the Fileserver filing system, and that it was as well to make them when it was installed. The disc file handler was also modified for the new interface.

---

* This starts to matter only when aliases are allowed, i.e. when an object can be retained in more than one directory. The name of the object is a property of the route taken to reach it, so belongs in the directory entries, whereas its creation date and length (of a file) are properties of the object itself, and should be stored with it, or more conveniently, in it.

# The Fileserver's file length value could, in fact, harmlessly be set to the length of the file in bytes - i.e. double the correct value. However, this trick is a little devious, halves the maximum file size, and is anyway more expensive than recording the length within the file.

There were two main reasons for making changes. Firstly, there was the possibility of there being sufficient machines in the processor bank for it to be reasonable to take one to run the TRIPOS filing system. It should then be possible in each other machine to replace the file handler task with a much smaller task which translated each packet from BLIB into a call of the filing system machine. Previously, part of the mechanism of decoding filenames was in BLIB, which made multiple calls on the handler in order to open a file, for instance. The interface was altered so that all operations involving a filename became single packet calls, passing the complete filename to the handler. Also, a cleaner interface was provided for the EX command to obtain file and directory information, without having to know about the internal structure of such objects.

The second change was a generalization of the way in which the identity of a handler task was found from a disc name. This used to be done by a piece of code with a compiled-in table of names and task numbers. The generalization was to replace this by a chain of association blocks accessible from the root node, and to allow 'device' names to be assigned to directories as well as complete discs. (The subtree of filing system below a directory is logically similar to the root of a mounted disc.)

This change comprised the introduction of the routine DEVICETASK in BLIB and of the ASSIGN command. For an assignment to a device, all that is remembered with the name is the task number of the handler; with an assignment to a directory, a shared file handler lock is also held, providing quick access to the directory as it contains the PUID.

## 7.7 Implementation of the File Handler Task

The file handler task contains three coroutines, organized as in the example of coroutine use in chapter 3. There are two identical 'command' coroutines, each of which serially processes request packets, and one 'clock' coroutine, which wakes up at intervals to perform housekeeping operations. Two instances of the command coroutine were included as it is useful to have some parallelism when several client tasks are using the filing system.

However, logging has shown that a third instance would be called only rarely* and, currently, the Fileserver can process only three commands in parallel anyway.

The clock coroutine keeps all TUIDs alive for open files by touching them periodically. It also regularly calls the Fileserver's 'status' function, so it can send a warning message to the terminal when a planned Fileserver stop is imminent.

The file handler includes a cache to speed the sequential reading of files. The handler tries to keep both the current and next file buffers in the cache so that the client task will not have to wait for Fileserver reads. The cache slots are kept in order of least recent use, so that active ones tend not to be reused. This works well: using 4 cache slots of 400 words each, programs reading a file sequentially nearly always achieve a 100% cache hit rate. The typical overall hit rate is 80 - 90%, but is the result of combining two extremes - a very low rate while opening files and doing other directory operations, and a very high rate during sequential reading. Block reads larger than a cache slot are performed straight into the client's buffer in a single Fileserver operation, by-passing the cache completely.

Both SSP and full read and write Fileserver operations are used, depending on whether the transfer is of more or less than 256 words. The SSP versions are rather faster than the full ones, but do involve the CPU overhead of copying the bare data, while the full versions allow it to be transferred directly from or into the final buffer. A cache slot has to be used as a temporary buffer for SSP reads and writes. Logging shows that most use is made in practice of full read and SSP write; the other two are used considerably less often.

--------------------

* It would never be called when only one task was using files, and for less than 10% of requests when three tasks were using files.

Care is taken to ensure that the filing system is not corrupted if either client or Fileserver should crash at any moment. The Fileserver allows atomic updating of only one object at once, so a TRIPOS directory (2 Fileserver objects) cannot be altered indivisibly. The approach adopted is to update atomically the special file part of the directory, and make corresponding changes to the index part in a safe order. This means opening the file, and making the alterations to that and appropriate retentions in the index, being sure to use free directory slots so that no used index slot is overwritten. The file is then ensured, and only when the changes have been committed are any deletions done from the index. Thus, the worst that can happen if either end crashes is that an index slot which should be empty will contain a PUID. This does not affect the integrity of the filing system, and the slot will eventually be reused because it corresponds to a free directory entry in the file part. The holding of a Fileserver interlock on the file part ensures that only one client at once can update a directory, and that other clients cannot see it in an inconsistent state.

Fileserver operations are retried indefinitely if they fail through communication errors - i.e. if transmission to the Fileserver fails, no reply is received, or a Fileserver return code is received in the range allocated to communication faults. If transmission fails repeatedly for several seconds, then it is likely that the Fileserver has crashed, so a warning message is sent to the console.

Hard return codes from the Fileserver cause the operation to be abandoned immediately, except in two cases. The return code 'object in use' is treated specially if it is received while trying to access a directory. The operation is retried a few times, as directories should be held open only briefly. The other special situation is when an attempt to read or write to a file, using a TUID, fails. One try is made at re-opening the file to get an equivalent TUID before abandoning the operation (see "Interlocks" below).

Fileserver return codes occupy 16 bits. In order to map them into the set of TRIPOS return codes, use is made of the knowledge that the less significant 8 bits merely indicate which Fileserver module the code originated from. The other 8 bits are extracted, and added to a fixed base, to form a distinct range of TRIPOS return codes.

An 'editor' program was written as an aid to implementing the filing system; this allowed Fileserver operations to be performed by means of interactive commands. It was used to create the index and file forming the root of the filing system. The file handler was then MOUNTed as an extra 'disc' in the normal version of TRIPOS, a complete set of directories created in the new filing system using the normal CREATEDIR command, and all the files copied across using a (long!) command sequence of COPY commands. (In fact, a slightly modified COPY program was used, in order to preserve the creation dates in all the copied files.)

## 7.8 Sharing The Filing System Between Machines

This filing system is intended to be sharable between machines, both in the sense that the structure on the Fileserver may be used simultaneously by several computers, and that the code of the file handler is portable between different machines.

### 7.8.1 Portability of Code

Given the machine independent environment provided by TRIPOS, the writing of a portable file handler task presented only a few problems. A major one is that of machine word length. The file handler approaches this by considering files to be vectors of bytes. However, some of the information in directories and file headers is stored as 16-bit quantities, and the code was originally written using the BCPL indirection ('!') operator to manipulate these - hence worked only on 16-bit machines. In order to run on a machine with a different word length, the code was modified to use routines which were the 16-bit equivalents of PUTBYTE and GETBYTE (PUT2BYTES and GET2BYTES). An alternative solution would have been to add a new operator to the language to do 16-bit indexing. This was rejected, as language extensions have the effect of reducing portability of programs to other systems.

Within 16-bit machines, the problem reduces to one of byte order within words. The file handler uses a constant byte order in all words written to the Fileserver, regardless of what sort of machine they originate from.

### 7.8.2 Concurrent Filing System Access; Interlocks

If the filing system structure on the Fileserver is to be used concurrently by several machines running the file handler, interlocks on objects must be held in a central place, the obvious choice being the Fileserver itself.

When a file is opened for reading or writing, the file in the Fileserver is opened in the corresponding way, and all data transfers done under the resulting TUID. This automatically provides the required multiple-reader / single-writer interlock. The file handler still produces a local 'lock', returned in the SCB. This contains the file's PUID and TUID, plus other information, such as the current position in the file.

Whenever a directory is being read, its special file is opened for reading to prevent any changes being made to it while it is being inspected. When a directory is being updated, the file part is opened for writing over all the updates, both to prevent anyone else from reading it in an inconsistent state, and to allow the changes to be committed or completely cancelled on closing (because it is a special file).

However, a TRIPOS shared directory 'lock', used for such things as the currently set directory, cannot be reflected in the Fileserver. Taking out a shared Fileserver lock on your current directory would make it impossible to update it. Thus, these locks do not prevent deletion of the object referred to, which is unfortunate, but not often a problem in practice. What is needed is a form of Fileserver OPEN which allows all further read/write access, but forbids deletion.

The index part of a directory is never opened, as it is effectively protected by the lock on the corresponding file.

The Fileserver times out TUIDs which are not used for several minutes. The existence of this timeout is ignored for TUIDs on directories, as none should ever exist for more than a few seconds. However, files are commonly open for long periods (e.g. while editing). The file handler's clock coroutine deals with this problem by scanning the chain of local locks regularly, and using each TUID to keep it refreshed. A small read is done from each file open for reading, and the current size is written to each file open for writing.*

A disadvantage of keeping locks in the Fileserver is that all TUIDs are forgotten if the Fileserver crashes and restarts. If this happens during a directory operation, then that operation will fail, but can usually be retried easily. However, it is desirable that an ordinary file should be able to stay in use over a Fileserver restart (e.g. so that any compilation, editing, etc. should just carry on). To achieve this, if an attempt to read or write gets the returncode "invalid UID", then one try is made at re-opening the object to restore status quo. In practice, this is very successful, and saves inconvenience.

Initially, the file handler tested specially so that it could forbid deletion of an open file. However, it cannot reasonably test that an open object is not implicitly being deleted when a directory is deleted, so this test was removed. The Fileserver undertakes to retain any open object, even if it is not reachable from the root, so the holder of the TUID is unaffected.

---------------------

* This has the useful side-effect that if a machine crashes with a file open for writing, the size is quite likely to be up to date. Updating the size on every write operation would be very expensive.

### 7.8.3 Filing System Variants Per Machine

Although the filing system as a whole is shared, and should appear similar regardless of which machine one happens to be accessing it from, there each a need for each machine to have sections of the filing system for its own use.

An example of this is the directory used to hold temporary files. TRIPOS has never had the concept of anonymous files which exist only while they are open. Instead, all temporary files (e.g. editor workfiles) are created within the filing system, and deleted after use. This has the advantage that if a machine crashes during editing, very little work will be lost. Temporary files are created within a special directory - ":T" in original TRIPOS - with unique names, usually constructed from the name of the program generating them, and the number of the task in which it was running (to allow for multiple instances of a program).

Utility programs running independently on a network cannot so easily produce unique filenames, so this scheme could not continue to be used. The solution was to use the assigned device name "T:" as the directory for temporary files, and arrange that this pointed at a different directory in each machine.*

Thus, the assignment mechanism allows machines to share the bulk of the filing system, while retaining a portion for their own use, with exactly the same code running in each machine.

---

* In fact, each machine has a directory with the same name as the machine, (chosen because a machine can find out its own name from the Nameserver) containing a sub-directory called "T". E.g. the machine called Bravo runs with T: assigned to "SYS:BRAVO.T".

### 7.8.4 Filing System Variants Per Machine Type

If the group of computers sharing the filing system is heterogeneous, but running the same file handler, there is a need for a mechanism which allows only relevant parts to be shared, and others to be machine-type specific. This is vitally important for any directories containing compiled code, such as the commands directory :C, the overlays directory :L, etc. The method of assigning a logical device name to each (as used for T:, above) is not very practical here, as it would be messy to have the necessary large number of assignments required, and a very large number of programs would need to be changed to use the new names.

A solution is to take advantage of the ability of the filing system to support aliases, i.e. to allow an object to be retained in more than one directory. All the system directories used by standard software are at the top level of the tree, as sub-directories of the root. A different root directory can be created for each type of machine, with entries 'C', 'L', 'H', etc., pointing to private versions of those directories. The majority of entries, such as users' directories, are common in each root.

The system initialization code inspects the machine type (available in the root node info vector), and sets SYS: to refer to the appropriate root directory. Thereafter, references to 'SYS:C' or just ':C' will automatically pick up the right sort of code. No other programs need be changed.

There remains the problem of accidentally loading the wrong sort of code from a file outside one of these directories. The TRIPOS object module format does not specify what machine the code is for - hence LOADSEG will happily load the wrong code, and attempting to execute it will cause a crash.

One change that could alleviate this danger without requiring alteration of the object module format is to make the MLIB routine GLOBIN check that at least one of the globals defined in a segment appears to have an entry sequence for this machine.*

No BCPL program can be executed without GLOBIN being applied to it first to put the addresses of its global routines and labels into the global vector, so this trap should always work. As GLOBIN is written in machine code, it is inevitably different on each machine, and it can be expected to know entry sequences.

## 7.9 Protection

The filing system for TRIPOS with a local disc includes no file protection mechanisms. They would be impossible to enforce in such an open system, and are unnecessary in a single-user system. However, when a number of clients are sharing a network filing system, the situation is rather like that in a multi-user operating system, and file protection is desirable. As the Ring-based system is just as open as the ordinary one (and, anyway, a user can run any program he likes in his machine), any enforcable protection scheme must be supported by facilities of the Fileserver. However, a Fileserver PUID has no associated access controls, and allows full access to the corresponding object. For the file handler to operate, it must have the PUID of the root index of the filing system in store. This means it can be read by any user with a modicum of knowledge about the file handler program (there being no memory protection). The PUID could be made less easily accessible by holding it in encrypted form, but this does not remove the logical lack of security.

--------------------

* Note that labels as well as routines can be defined as globals, but it is reasonable to expect any segment to contain at least one routine definition.

Two file protection schemes have been incorporated in the filing system. The first is designed to prevent accidental overwriting or deletion by means of access flags in each directory entry, but is no defence against malicious attack. The second completely protects part of the filing system by making it unreachable from the normal root index. It works by having two instances of the root directory, almost identical in contents except that only one contains references to the protected part of the filing system. Part of the logon authentication process (see chapter 8) sets the root directory according to whether or not the user is privileged. This method has been used to protect the system sources, but is hardly ideal, as it makes it impossible for the unprivileged user even to read them. However, it does provide enforcable protection in unprotected machines.

## 7.10 Evaluation and Conclusions

In most distributed systems, the file server is designed to support a previously existing filing or database system. Hence the literature tends to concentrate on the design of file servers rather than on the design of filing systems to use them. This makes it difficult to compare the above work with that done elsewhere. However, it has indicated some features that are desirable in a file server.

In fact, the Cambridge Fileserver was originally conceived as a backing store server for the CAP computer, and its design has been influenced by the virtual memory and filing systems of that machine. Hence it supports a file and index structure which is a full directed graph, allows very large data transfers (for swapping), and maintains file sizes only to a resolution of whole 16-bit words (CAP files contain integral numbers of 32-bit words). The work of converting the CAP backing store system was done by Dellar [7, 8]. He too used a special file / index pair to represent a directory, but was able to map CAP files directly into bare Fileserver files, as the resolution of size is fine enough, and CAP associates no other information with a file (rather than with directory entries for it). He made no use of Fileserver interlocks; there is only a single client of the

CAP filing system, and it makes directory changes by reading the whole directory into memory, altering it, and writing it all back.

There seems to be no very good reason for the Cambridge Fileserver not to hold file sizes in terms of bytes, as does the Xerox DFS [44]. (Most other file servers are page oriented, so the question does not arise.) The majority of operating system designers seem to consider a vector of bytes to be the most sensible form for a file. It would also be useful if a small amount of information could be stored with a file. Client systems could then use this to hold items such as the time last updated (though ideally this would be maintained by the server), and the identity of the file's creator. This facility would perhaps be most neatly provided by allowing access to negative offsets within a file; a client could then choose whether or not to read the information with the first part of the file. WFS [46] carries the idea of associated information one stage further, by having system and client information attached to each <u>page</u> of a file.

Although nearly all of the Fileserver's functions are idempotent, and so can be harmlessly repeated when a network block is lost, two are not and both have caused trouble. These are the OPEN and CLOSE operations. If the reply to an OPEN is lost, then when the client retries it will find that the object is already open. As the corresponding TUID was lost, nothing can be done for several minutes until the Fileserver breaks the lock. A function meaning "break all locks which I hold on this object" would enable faster recovery.

The operation CLOSE (specifying that updates should be done) is essentially useless. If the reply to it should be lost, then a repeat will receive the response "invalid UID". There is then no general way of finding out whether or not the first one succeeded. The safest way to close a file is to call the ENSURE function, which is repeatable, first; there are then no changes for CLOSE to commit. However, even this does not work if the Fileserver crashes between committing the changes and replying to the ENSURE. Xerox DFS gets round this problem by maintaining a list of the fates of recent transactions, so that a client can enquire about how one ended.

The handling of directories consisting of a file/index pair would be simplified if the Fileserver allowed atomic transactions involving more than one object (as does, for example, DFS). There would then be no need to be careful about the order of updates, and no possibility of spurious entries being left in indexes.

The fact that the Fileserver allows the PUID of an object to be retained in more than one index slot makes the implementation of unrestricted aliases to files and directories trivial, because the server does all the work of maintaining use counts, deciding when objects can be deleted, and garbage collection. Aliases have proved to be a very convenient way of arranging sharing of the filing system between machines of different types, and of protecting some regions of the filing system, as described above.

If the Fileserver allowed several PUIDs for an object, giving different access rights, then it would be possible to implement more sophisticated file protection schemes in computers with unprotected memory. The Felix file server [13] has such a facility. A function exists which takes a file identifier (FID) and a list of access rights, and returns an equivalent FID having the requested access. The FID must be just an index into a private data structure of the file server's so that a client cannot manufacture a FID giving him more access than he is allowed.

The area in which the Fileserver filing system is much slower than the disc one is in locating files. The only safe way to follow a file's path name is in turn to open each directory for reading, so that the entry can be found without danger of the directory being altered from another machine. Finding a file thus requires a large number of interactions with the Fileserver, and can take several seconds.

In practice, it is rare for any directory to be in use by more than one client, so a great efficiency loss has been made to eliminate a small danger of incorrect operation. It would be of great benefit if clients could safely cache heavily used directories, and have some cheap method of finding out if a cached copy was out of date. One way of doing this would be for the Fileserver to provide "weak" interlocks, which it could break if another client wanted a "strong" (conventional) interlock on the object. The DFS file server has a similar facility (although there is not the concept of "weak" and "strong" interlocks). When DFS breaks a lock, it sends an unsolicited message to inform the client which obtained it.

A client could take out weak interlocks on all directories it had cached, and would be told by the Fileserver when any part of its cache became invalid. There is a small timing problem here, as the client may have made use of an invalid cache entry just before receiving the message. If the directory was changed in such a way as to add something, then the result is harmless - it is as if the new entry were inserted a little later than it actually was. If something was deleted from the directory, then the cached copy may point at something which no longer exists. As PUIDs are (theoretically) not reused, the entry is detectably invalid, so subsequent path name decoding will fail. However, this does mean that "object just deleted" is not necessarily distinguished from "filing system corrupted".

# CHAPTER 8

# AN OPERATING SYSTEM FOR 'SINGLE CONNECTION' COMPUTERS

Preceding chapters have described the design of a portable operating system for minicomputers with their own disc and console. This chapter gives an account of the version of the operating system written to run in the Cambridge Computer Laboratory's 'single connection' computers - those whose only peripheral is the Data Ring.

The new system was designed to appear to users to be very similar to the normal version, though some minor changes to the user interface were inevitable, and the method of access became quite different. The internal changes were quite substantial. Some comments are made on the benefits and disadvantages that resulted from the exporting of some of the functions of the file handler and console handler to other machines on the Ring.

## 8.1 The Processor Bank

The machines on which this work was done are Computer Automation LSI4 minicomputers with 64K 16-bit words of store. The Laboratory had just one of these machines initially, with two console lines and two 80 megabyte discs; it ran as the main TRIPOS machine for about two years, then went into dedicated use as the Fileserver.

Half a dozen more LSI4s were purchased for use as general computing engines. Their only peripheral was to be the Data Ring, and it was intended that ordinary users should need no physical access to the machines themselves. The model was of a pool of personal computers, which were available to be borrowed by any user with some computing to do.

This was an experiment in an alternative way of using personal computers attached to a network, forming part of the Cambridge Model Distributed System [29]. Other institutions (e.g. Xerox - see [47]) have used the approach of giving each programmer his own minicomputer in his office. It has its own console and a disc of modest capacity, so can be used independently from the network; however, the facilities of the latter are easily available when required.

This system has the advantage that people can still work when the network is not operating, but is expensive in the amount of equipment required (as only a fraction of the computers, terminals and discs will actually be in use at any moment), and makes maintenance awkward. The machines are liable to be widely distributed, and probably have to be physically wheeled away for servicing.

The advantages of a pool of single-connection computers are that the number of machines needed is only enough to cover the peak number expected to be simultaneously in use (hopefully less than the number of people), and that they can all be situated physically close together. This makes them simpler to maintain, and when one breaks down, it is not one particular person who is inconvenienced - it just means that there is one fewer machine in the pool until it is repaired. The idea of these machines forming a pool from which they can be lent out led to the name Processor Bank for them.

The LSI4s were all connected to the Ring via the 'Type 2' microprocessor interface described in chapter 5. During the development of the version of TRIPOS described below, the full power of this interface was not being used, as the program running in it was a rather simple one which did not handle the basic block protocol. The eventual introduction of basic block handling in the interface did lead to the hoped result of less code and less CPU time usage in the LSI4s.

There are two models of LSI4 in the Processor Bank, differing most significantly in their speeds. The LSI4/30 model has instruction times ranging from about 1.2 to 2.5 microseconds; the LSI4/10 is around three to four times as slow for most programs.

## 8.2 Access to the Processor Bank Machines

Use of the computers in the Processor Bank is by means of several server machines on the Ring. These are:-

- The Terminal Concentrators
  These each support several terminals, and allow each terminal to make byte stream connections to one or more other machines. They provide basic line-editing and escape facilities on input.

- The Resource Manager
  This is responsible for allocating Processor Bank machines. It maintains a table with an entry for each machine, indicating what attributes that machine possesses (e.g. whether it has floppy discs, whether it is an LSI4/30), whether it is allocated, and, if so, how long it has been allocated for.

- The Ancilla [41]
  The Ancilla performs low level operations on Processor Bank computers, such as loading bootstrap code into them from the Fileserver, and starting them. It is intended to hide the sordid details of these operations from the Resource Manager, which calls it to reset, load, and start machines.

- The Session Manager
  This works closely in conjunction with the Resource Manager. It manages the connection of terminal streams to machines, and provides an interactive interface for inspecting machine states, and allocating machines with particular attributes.

## 8.3 Towards a single-connection system

This section outlines the development of the modified TRIPOS which runs in the Processor Bank machines. The bank itself became available over a period of about a year. Initially, only one machine was in use, with its own console and dual floppy discs. A second was added, again with its own terminal, but with no discs.

The order in which dependence on real peripherals was removed from TRIPOS was as follows. When the first Terminal Concentrator appeared, a Ring service was included to allow 'logging in' from the Ring - i.e. by creation of a new CLI task and a virtual terminal handler task. This terminal handler formed the basis for the full version.

When the Fileserver came into full-time operation, the whole of the old TRIPOS disc was physically copied into a large Fileserver file. A task was added to the system which looked to the file handler as if it were the disc driver device, but which in fact transferred blocks to and from the large file (fully described in chapter 6). This served two purposes: it was a simple way to continue running the system, the main noticeable change being just that it ran more slowly; also it tested the viability of using the Fileserver in a simple-minded way rather different from that for which it was designed.

The next stage was to run TRIPOS with no real console, but without external resource management. The system was booted manually into a machine, either from the floppy discs, or by running a command on another machine to load the target computer via the Ring. The system could be used by connecting into the machine by name from the Terminal Concentrator. It allowed only one user at once, issuing an 'already in use' message to anyone else who attempted to connect in. Once the first user had disconnected his terminal stream, the machine was open to another connection.

This was a workable arrangement, given cooperative users, but far from satisfactory. It was awkward to find a free machine. If you did not reboot it, then you got a used system whose integrity could be suspect, as the LSI4s have no memory protection. However, unless the machine was the one with the floppy discs, rebooting it required a second computer to do it from.

The next change was the replacement of the disc file handler and its imitation disc driver task, by a completely rewritten file handler which made more use of the facilities provided by the Fileserver (see chapter 7). This meant that the whole filing system had to be copied again, using a file-by-file logical copy this time, as the old disc blocks no longer had any significance.

When the Resource and Session Managers, and the Ancilla, came into service, the old problem of choosing and booting a machine was removed. When TRIPOS found itself running, it could apply to the Resource Manager to connect itself to its user, and no longer required a Ring service for logging in.

The main change in the system after starting was that it had to talk to the Resource Manager periodically to ask for a further time allocation. This provided a "dead man's handle" mechanism whereby the Resource Manager would eventually reclaim any machine that crashed.

## 8.4 Design of the Processor Bank system

This section explains the internal organization of TRIPOS for the processor bank machines, and compares it with the normal system described in chapter 3. The diagram shows the tasks and devices which comprise the system, indicating with arrows which modules call which others.

Three of the four resident tasks (DEBUG, Terminal Handler, and File Handler - see below) had to be completely rewritten or modified, and two more had to be added to handle the Ring communication. In fact, a third task was added as well - not because it was essential to the working of the system, but because it would usually be desirable to have it running.

The three devices (apart from the timer) in the normal system were replaced by just two: the Ring transmitter and receiver. The initialization code had to be substantially extended, a few commands became redundant and a few were added, and some system overlays had to be replaced.

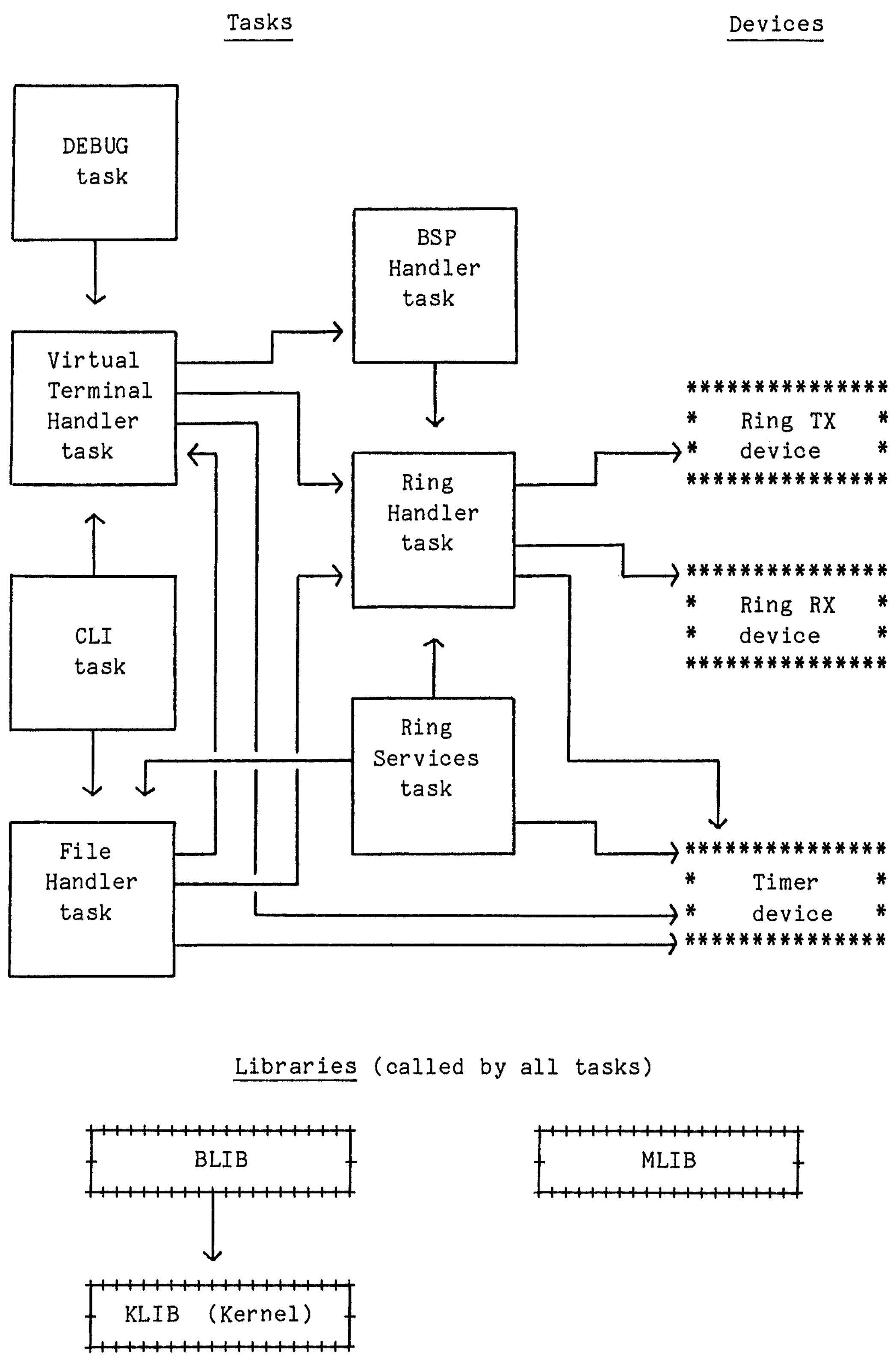


Fig. 12: Tasks, Devices, and Libraries in the Processor Bank System

However, the interface seen by programs at the system library level remained unchanged, so the majority of commands and other programs needed no alteration.

### 8.4.1 Ring communication

In the single connection systems, it is essential that the Ring handler and drivers are resident; without them, nothing else could be loaded, nor could anything be conveyed to the outside world.

The only changes needed to the Ring handler were minor ones. An extra field was added to its start-up packet, indicating whether it is resident or has been loaded dynamically. Only in the latter case will it go away in response to an "RHINFO KILL" command. The other modification was in the way the port number to be allocated first was chosen; this is explained in the section on the Ring handler task in chapter 5.

Thus, only one version of the Ring handler program is needed, and can run either in a single-connection or an ordinary system.

### 8.4.2 Byte Streams

One of the first actions of this TRIPOS after starting is to open a byte stream pair to the Session Manager, which is immediately 'replugged' (see chapter 5) to the terminal. Therefore, the BSP handler would always be loaded at a very early stage, and so might just as well be in the initial load. The extra time for boot loading is less than that to read the same code through the file handler.

However, there is no need to take precautions to make sure that a resident BSP handler cannot be unloaded. It does not go away until all streams are closed, anyway, and all the apparatus to reload it on demand is still available.

### 8.4.3 Ring Services

Although it is not a necessary component of the system, this task is also included in the initial load. The user of a Processor Bank machine is thus deemed to be willing to accept Ring service calls from other machines, unless he deliberately kills this task. This is useful mainly so that he can by default receive operator messages.

### 8.4.4 The four standard tasks

The resident tasks of normal TRIPOS - CLI, DEBUG, console handler, and file handler - all remain, albeit in different form, in this system. Thus, there are seven tasks in the initial load. (No Restart task is necessary with the Fileserver file handler.)

The changes made to these four tasks will now be described.

### 8.4.5 CLI

No modifications at all were needed to the CLI for the Ring system. However, CLI.INIT, the routine which it calls on starting to initialize it and the rest of the system, needed considerable extension (see below).

### 8.4.6 Terminal Handler

The terminal handler was completely rewritten for the Processor Bank system, as its character reflection and line editing functions were no longer relevant, and it had to interpret Virtual Terminal Protocol (VTP) [31] on byte streams, rather than driving a console device. It proved convenient to include some code connected with the Resource Manager and Active Object Table Manager (see "User Authentication" below) in this task, making it more of a session handler than just a terminal handler.

The terminal handler has two flows of data through it:- output lines from other tasks are written to the terminal, and input lines are sent to tasks, or held until wanted.

**Simplifications in VT handler compared to console handler**

- No character reflection
- Very little line editing - just deletion of whole lines
- No '@' escape handling. The terminal concentrators use this as an escape character, and provide most of the functions which it was used for in the console handler. The few operations which cannot be done by the concentrator (i.e. @Snn, @Tnn and @F) are done in the virtual terminal handler by corresponding control codes, to avoid the unpleasantness of interpreting '@' in two places (meaning you would have to type four '@'s to get one through in an input line).
- No 'system freeze' function (done by control-A in the console handler).
- No need to hold up output while an input line is being typed; this is done by the concentrator.
- No processing of output lines

**New facilities included in the VT handler**

The virtual terminal handler is largely compatible with the ordinary console handler in terms of its packet interface. However, it includes some facilities not available in the latter:

- Single Character Input Mode.
  Programs, such as screen editors, which need to receive each input character from the keyboard as soon as the key was pressed achieved this by driving the console devices directly in normal TRIPOS, by-passing the console handler. With the virtual terminal handler, the devices do not exist, so an interface was provided for doing single character input and uninterpreted byte output. A packet can be sent to the VT handler to switch it into 'single character' mode. It has to RESET the byte stream to the terminal concentrator in order to cancel the outstanding request for an input line (with reflection) and replace it with one for a single character (without reflection). Similarly, the byte stream must be reset on return to line mode.
- Extra 'break' mechanism. The Virtual Terminal Protocol specifies that byte stream reset initiated from the terminal end should be regarded as the equivalent of pressing the 'break' key on a direct terminal - i.e. it is an attention signal outside the character set. The VT handler responds to this by giving a prompt of "***-", and then reading a single character to find out what action is

required.* This enables task flags to be set, or the mode of the VT handler to be changed, whatever mode it was in before. The meanings of the various break keys are listed in the section on 'Console Characteristics' below.

### 8.4.7 File Handler

The File Handler task was also completely rewritten, to provide a similar filing system using the file server rather than a local disc. This is described in detail in chapter 7.

### 8.4.8 DEBUG

DEBUG in this TRIPOS version can be used only in task mode, as it depends on the terminal handler, BSP handler and Ring handler tasks running merely to talk to the console. The MLIB routines SARDCH and SAWRCH no longer have any effect, so stand-alone DEBUG cannot be used.

Normal DEBUG was modified to handle aborts, traps and breakpoints in a new way. After one of these occurs, it is entered in stand-alone mode by the kernel as usual. Instead of trying to output a message, it records details of the abort in a dummy packet#, which it puts 'by hand' on the work queue of the debug task. It then returns to the kernel as if the 'H' (hold) command had been typed. This causes the task which aborted to be held, and the rest of the system to resume running.

In task mode, DEBUG recognizes the arrival of the dummy packet, and responds by selecting the aborted task as the one currently being debugged, and issuing an abort, trap or breakpoint message as appropriate. The dummy packet is not sent back; it is just forgotten.

---

* The prompt "***-" requesting a single letter argument was chosen because it was already an established response to break on other machines in the Laboratory.

# 'dummy' in the sense that it is not sent by QPKT, and is not returned to the sender, as no task sent it.

Thus, DEBUG can handle aborts in commands, other user programs, and some system tasks. It is of no use when an abort occurs in a system task which is necessary for communication with the terminal, or when a crash has involved store corruption which has broken one of these tasks.

Provision of a fuller DEBUG could be achieved in two ways. Firstly, some very simple stand-alone Ring driving routines could be devised, enabling reading and writing of words of store, so that a DEBUG program in another machine could investigate the crashed one. This could be expected to be about as successful as stand-alone DEBUG in ordinary TRIPOS, as it would require only about the same (fairly small) amount of code to be intact.

A better approach is to provide debugging primitives in the intelligent Ring interface to the machine. This has the major advantage that no matter how serious the crash, the debugging system will still operate. The minimum set of primitives needed are those to read and write any store location. Desirable additional ones are those to start and stop the CPU, and to read the central registers. However, these are not needed if a small portion of the debugging code is retained in the target machine. A skeleton of stand-alone debug could be kept; on an abort, it would dump the registers to store, and then go into a tight loop. Thus, the remote debug program could read registers from the store dump, and could restart the system by temporarily overwriting one of the instructions in the tight loop with a jump out of the loop.

This approach would simplify the interface, but would once again mean that debugging would not necessarily be possible after a bad crash.

## 8.5 Other system components changed

### 8.5.1 System initialization; CLI.INIT

CLI.INIT is so called because it is the program which the CLI calls immediately after it starts running. One of its jobs is indeed to initialize the environment of the CLI; however, the version called from the standard CLI (task 1) is also responsible for starting up the other system tasks. It sends a packet to each task in turn, containing either nothing, or a small amount of information such as the identifiers of the devices to be used by that task.

In TRIPOS for a single-connection machine, the initialization process is considerably more complicated, and it is CLI.INIT which performs most of it. The Ring handler task must be started first, as everything else depends on it, then the BSP handler and terminal handler, as other tasks may want terminal streams.

The date and time are found from the Ring clock service, and used to set the system date and time. The name and station number of the machine are discovered from the name server, and recorded in a structure attached to the root node INFO vector, for easy access by any program which may want to use them.

Next to be started is the file handler, which is passed the PUID of its root index, and told how many cache slots to use, and of what size. When the file handler is running, the standard device and directory assignments are made to "FS0:" (the system device), "SYS:" (the root directory of the system device), and "T:" (the directory for temporary files).

CLI.INIT makes an SSP call to the Resource Manager to set up a further time allocation, and to discover whether the Session Manager has a byte stream waiting for it. If it does, then a BSP open call is made to the service specified in the reply to the SSP. This byte stream should be 'replugged' immediately after starting; CLI.INIT waits for this to happen before proceeding so that the initial messages and prompt are sent to the terminal, rather than going to the Session Manager or being lost in the 'replug' operation.

Once the byte streams exist, the terminal handler task is sent a packet passing them over, completing the terminal connection. Now, the "TRIPOS starting" message can at last be issued, and user authentication requested if necessary (see below). The DEBUG task is started at this point; if started earlier, it could have provided a means for the user to avoid the logon authentication mechanism. Finally, the CLI environment is set up, and control passed back to the CLI.

### 8.5.2 The Resource Manager's 'Dead Man's Handle'

Any system loaded by the Resource Manager should ask for an allocation of time soon after it starts running. Whoever caused the booting has (explicitly or implicitly) already specified two time limits: an absolute maximum time for which he wishes to claim the machine, and an initial period during which the loaded system should contact the Resource Manager. The latter time limit is usually set quite short, so that a failure to load and start properly will soon be detected. The loaded system may then request further allocation periods within the maximum.

For some programs loaded by the Resource Manager, it may be appropriate for them to request all the available time, and then not talk to the Resource Manager again. As TRIPOS is a general purpose operating system running in machines with no memory protection, there is no way it can protect itself from crashing when a program overwrites store, for instance. Therefore, it is better for it to request short time allocations at frequent intervals; in this way, the Resource Manager will notice fairly quickly when a crash occurs, and reclaim the machine.

In practice, a further 2 minutes is requested every 30 seconds, inside a default maximum period of 8 hours.

This function could be carried out in a new task, which just wakes up occasionally to do an SSP call to the Resource Manager. However, it was decided to do the refreshing by adding an extra coroutine to the virtual terminal handler task. This saves store: the cost of a new coroutine in a task which already employs several coroutines is little more than its stack space. Also, it is logically simpler to do the refreshing in the terminal handler. It should be carried on only while the user for which the machine was booted is still using it, and it is this task which knows whether or not he is still attached.*

In view of this, the terminal handler task is perhaps badly named; it is closer to being a session handler.

The introduction of an external debugging system for Processor Bank machines would make use of an extension of this mechanism, as the machine doing the debugging would have to be able to operate the dead man's handle on behalf of the crashed machine.

### 8.5.3 User Authentication

TRIPOS running in Processor Bank machines uses the Ring-wide authentication system [15] to restrict access to the system to known users, and to give those users access to other machines without needing to identify themselves again.

After establishing the terminal connection, CLI.INIT calls the command START (written by C.G. Girling), which requests a user identifier and password, and checks them with the Ring's User Authentication service. If a valid user / password pair is not typed in three attempts, or if nothing is typed for three minutes, then the machine is returned to the Resource Manager. To prevent by-passing of this program, the DEBUG task is not started, and the 'X' (create new CLI) break key (see

--------------------

* An explicit SSP call is made to the Resource Manager to free the machine when the user logs out; however, it is as well to stop refreshing too, in case this call fails.

"Console Characteristics" below) is not enabled, until the user has been authenticated.

Once a user's password has been verified, a UIDset is obtained from the Active Object Table (AOT) Manager to represent the validated user. This is a set of four 64-bit numbers containing the user's PUID (a synonym for his name), a TUID (indicating that he has proved his identity), an authentity (the authority identity under which he has been validated), and a TPUID (a capability for refreshing or revoking the TUID).

The PUID and TUID can be used as proof of the user's identity to any machine or service which trusts the authentity under which it was issued. A machine to which they have been presented can call the AOT Manager (which must of course also be trusted) to confirm that the TUID has been issued for the PUID by a particular authentity.

A special form of the BSP open block exists, including the PUID and TUID. This enables the setting up of authenticated byte streams, which can be used for such purposes as logging on to another machine without quoting one's user identifier and password again, or transferring files to or from one's protected directory on another machine. TRIPOS includes a pseudo-device called "AUTHBSP:", which is analogous to "BSP:", except that it opens an authenticated byte stream using the UIDset of the logged-on user.

The UIDset of the logged-on user, and any other capabilities he may obtain in the form of UIDsets, are kept in a chain hanging from the root node INFO vector. There is a timeout associated with the TUID and TPUID in a UIDset, so these must be refreshed periodically. The Virtual Terminal Handler task does this for all UIDsets in the chain, by making SSP calls to the AOT Manager. The name 'fridge' is often used for the chain of UIDsets; any set left in the fridge will remain fresh - if it is taken out, it will go bad.

### 8.5.4 Printers; The pseudo-device "LP:"

In normal TRIPOS, the printer (if available) is used by opening a stream to the pseudo-device "LP:" and then writing text to it. The code called to create this device loads the device driver for it, makes a handler task to look after buffering and low-level operation of the printer, and returns a stream to this task.

Ring-based TRIPOS must use the printing services provided on the Ring. These have a very simple interface: a byte stream is opened to a service and a stream of ASCII characters is written down it to be printed. There is no extra protocol level. If the BSP open block contains a string in the user parameters field, then this is printed as the document title.

This means that the printing services are directly available on TRIPOS, without any extra software being needed. For example:

```
type :g.libhdr to bsp:titanprint/LIBHDR
```

would print the BCPL library header file via the service "TITANPRINT" with title "LIBHDR".

For compatibility, a new version of the "LP:" device is provided, which also takes a document title, as in "LP:title". It has the advantage of elaborating the title before calling the printing service, by adding date and time, the user's name, the name of the machine, and the name "TRIPOS" to identify the source of the document. It also retries when the printer is busy.

As there is no extra protocol level superimposed on BSP, there is no need for a handler task to be created to handle the printer stream; the stream is just one to the BSP handler.

## 8.6 Use of the Processor Bank Machines

This section discusses the use of the Processor Bank machines, and mentions the ways in which this differs from use of TRIPOS on a normal machine.

### 8.6.1 Booting

The normal method of gaining access to a computer running TRIPOS in the Processor bank is to go to any free VDU Concentrator terminal, and type the following command to its "Monitor >" prompt:-

```
Monitor >c tripos
```

This establishes a connection to the Session Manager, which calls the Resource Manager, and that in turn calls the Ancilla; the user need know nothing of how it is done. In a few seconds, a machine has been chosen and booted. The Session Manager prints a message informing the user which machine he has been given (though he does not really need to know this). The newly-loaded machine opens a byte stream back to the Session Manager, which then performs a BSP 'replug' operation to connect the byte streams from the machine and from the terminal, and drops out of the proceedings. The terminal concentrator issues a local message indicating that a byte stream 'reset' has occurred.

TRIPOS then starts running, issues its start-up message, and asks the user's identity.

Suppose that the machine called Delta is allocated; the output to the console screen would be as follows:-

```
Monitor >c tripos
*** 1 allocated.          [from Terminal Conc.:  channel number
Delta allocated.          [from Session Manager: machine name
*** 1 RESET               [from Terminal Conc.:  stream 'replug'

TRIPOS starting           [from TRIPOS in Delta
User:
```

After identifying himself, the user gets a prompt from the CLI, and can proceed with what he wants to do. Note that the system is completely initialized by this time; there is no filing system restart to perform, so files can be written immediately. Also, the system date and time have already been set from the Ring clock service. Not only are they set each time a machine is booted, but they are set more accurately than would be likely had they been set by hand.

### 8.6.2 Console Characteristics

TRIPOS normally requests complete lines of input from the VDU concentrator. Hence all reflection and line editing, and most escapes, are handled there according to the concentrator's own conventions.

In TRIPOS with a real terminal, there are a few escape sequences which are used to give commands to the terminal handler, rather than to input unusual characters. It is useful to retain some of these when the terminal becomes remote, particularly @Snn, @Tnn and @F. As the concentrators in the Laboratory use the '@' character for their own escapes, it is not practical for the virtual terminal handler in TRIPOS to use it as well. Instead, the corresponding ASCII control characters are used, admittedly not with their standard meanings. The terminal handler reflects these characters as the '@' sequences which they replace. Control-F, control-S and control-T behave in this way.

The control characters used to set task flags (control-B to E) behave in the same way as before. However, the virtual terminal handler provides an alternative way of setting these flags: it responds to 'break' in the virtual terminal protocol (= RESET on the byte stream pair carrying the VTP) by giving a prompt:-

```
***_
```

This expects a single character response, and the replies B, C, D and E set the appropriate flags and continue.

Normally, the use of the control character is a preferable way of doing this, as it just sets the flag. Using VTP break will destroy any input or output buffered or in transit when it occurs, so it is not so easy to see exactly when it happened.

Other valid responses are:-

| | |
|---|---|
| L | Set 'line' mode for input. |
| N | Nothing - cancel the break. |
| S | Set 'single character' mode for input. |
| X | Create an eXtra CLI task, and select it as the current task. This is a powerful facility, allowing another CLI to be made available regardless of what the existing ones are doing. |

## 8.6.3 DEBUG

Using DEBUG as a task is the same as in normal TRIPOS. Stand-alone mode is not available, as so much of TRIPOS must be running simply to talk to a virtual terminal. Aborts, breakpoints and traps which are not fatal to the system as a whole cause the offending task to be held, and a message to be printed by the DEBUG task. This task may then be selected in the normal way to inspect and release the held task.

### 8.6.4 File Handler

The file handler provides a filing system which is very similar to the disc filing system from the user's point of view. The only noticeable differences should be in the file information printed by the EX (examine) command (e.g. file sizes given in bytes rather than blocks).

## 8.7 Comments on distributing parts of the operating system

This work has shown that TRIPOS provided a good basis for an operating system to run in 'single connection' computers. The new system presents user and program interfaces almost identical to that of the old.

The differences in the user interface inevitably include a new method of choosing a machine and booting it, and slight differences in the terminal characteristics.

### 8.7.1 Terminal Handling

Most of the old '@' escapes are made unnecessary by the Terminal Concentrator; the few which provide control functions in the Terminal Handler ('@Snn', '@Tnn', and '@F') have been implemented by means of control characters. The use of these control characters, and control-B to control-E for break is not entirely satisfactory. Their functions do not correspond to their significance as ASCII codes; the control characters used have been chosen on mnemonic grounds. A further infelicity is that the Terminal Handler has to request the Terminal Concentrator to force transmission of a partially typed input line on every control character (so that a special mode can be entered to read the arguments after '@S' and '@T'). This means that the control character, and anything preceding it on the line, cannot be rubbed out, as the Concentrator has passed the partial line on. (A special code in VTP is used to cancel the whole line.)

Unfortunately, the current Virtual Terminal Protocol provides no better way of achieving the required effect. The alternative to using control characters is to use byte stream 'reset' to signal the start of control information. However, 'reset' causes the loss of both input and output lines buffered and in transit. This is very undesirable: the usual uses of the TRIPOS control codes are to cleanly stop the current command, allowing the next to continue, or to type-ahead to one task, and then cleanly switch to typing to another one. Probably the neatest solution would be for the Terminal Protocol to allow for the sending of control information (or just a 'break' signal code) in the byte stream, so that the stealing of control characters could be avoided without the problems associated with resetting the byte stream.

The Virtual Terminal Handler does allow control functions to be signalled by byte stream reset followed by a single letter. This is included for compatibility with other machines on the Ring, but is really of use only when the Handler is in single-character input mode, so is not interpreting control characters. In line mode, this technique not only causes the loss of input and output lines, but also requires more keystrokes than using control characters.

As TRIPOS allows input to be typed to several tasks, it must always read and buffer input lines, so that one task is not held up waiting for another to read input. Thus, there must always be a line request outstanding at the Terminal Concentrator. This also gives the desirable property that even typed-ahead input is reflected as soon as it is typed, making correction of typing mistakes much simpler.

However, when single character mode is entered, the outstanding line request is of the wrong kind (asking for a whole line, with reflection). A byte stream reset must be issued to cancel it, and replace it with a request for a single character. Thus, typing ahead is not possible to a program which will use single character input.

The best way for the Terminal Handler to provide a more satisfactory user interface would be for it to always read characters from the Concentrator one at a time, doing all its own reflection and line editing. This indicates that the split made to distribute the terminal handling function has not been made in quite the right place. It would be better if the Terminal Concentrator was aware that it could be talking to more than one process down a single byte stream. Input requests and output lines would have to be labelled with a process identifier. The Terminal Handler task would be considerably simplified, as most of its function would have been exported.

### 8.7.2 CPU time usage; Response times

The Processor Bank system is both larger and slower than ordinary TRIPOS. Response time with a lightly loaded Fileserver is around a second for a small simple command, while it takes about three seconds to load a larger program, such as an editor. In ordinary TRIPOS with a fast disc, most commands take a time only just perceptible, while it takes about a second to load the editor. Programs which process files serially tend to be CPU-limited in both systems; reading ahead from the Fileserver helps greatly here.

Writing a line to a terminal takes 12 task changes, as the buffer goes from the client task via the Virtual Terminal Handler, BSP Handler, Ring Handler, and Ring devices, and the reply packets come back. (There are two interactions between the BSP Handler and the Ring Devices.) In normal TRIPOS, it takes 2, plus 2 for each character in the line (using the usual single-character device). It is observed that writing to a virtual terminal consumes a little less CPU time than writing to a real one.

Writing a line to a file involves 10 task changes, the buffer going from client task, to File Handler, Ring Handler, and Ring Device. In the normal system, this typically takes 4 (but it could be more if more than one disc block was involved, or relevant blocks were not in the cache). The lower speed of the Fileserver filing system is due mainly to the fact that calling the Fileserver takes rather longer (50 - 150 ms under light load, up to 3 seconds under heavy load) than reading or writing a block on a local disc.

It is the opening of files, and other directory operations, that are slowed particularly, as the claiming and freeing of interlocks within the Fileserver increases the number of calls required. Sequential reading of files is of similar speed in both systems.

### 8.7.3 Code Sizes

The Processor Bank system has considerably more resident code than ordinary TRIPOS. The code saved by exporting some of the work from the Terminal and File Handlers is outweighed by that required for Ring communication:-

(i) Distributing an operation over a network introduces numerous extra ways for it to fail, due to problems in communication. These include transmission failure, lack of reply, and congestion at the destination. Multi-part calls (such as a Fileserver write) have the possibility of errors at each stage; if any occurs, then the other stages must be cancelled, and the whole call abandoned or repeated. A lot of code throughout the Ring software is there to check for all the things that can go wrong.

This should be compared with driving a physical disc or terminal. In the former case, most of the error checking can be done at one place in the disc driving routine. On many machines, there are no errors that can be detected in driving a terminal.

(ii) A variety of different protocols has to be supported. Calling many Ring services uses the Single Shot Protocol. The Fileserver employs a slight variant of this. However, over a dozen different Fileserver functions are used, and a separate routine is needed to set up the transmission block and decode the reply block for each. The protocols for reading and writing are different again.

The (fairly large) BSP handler task is needed to drive the virtual terminal, but is also used for driving printers. The terminal stream needs another protocol layer on top of BSP.

(iii) Some extra code has to be included for refreshing state held elsewhere - e.g. applying for a new time allocation from the Resource Manager, regularly touching Fileserver TUIDs, and refreshing authentication UIDsets held in the 'fridge'.

(iv) When the primitives provided by a Ring server do not correspond directly with what is required, then it can take a lot of calls across the Ring to achieve the desired effect. This is illustrated by comparing the relative complexities of managing a directory block in the disc filing system, and of manipulating the corresponding object consisting of an index and a special file in the Fileserver.

Swinehart et al. [46] also make the observation that the size and computing requirement of network communication code often exceeds that needed to provide higher-level abstractions. However, Dellar [7] found that exporting backing store administration from the CAP computer to the Fileserver slightly reduced the amount of code and workspace required. This seems to be due to the fact that the CAP filing system is more complex than most, and that the Fileserver provides functions that fairly closely match CAP's needs, allowing more function to be exported.

# CHAPTER 9

# SUMMARY AND CONCLUSIONS

This dissertation has been concerned with portable operating system design for minicomputers in two different environments - firstly, for 'normal' computers with directly connected peripherals, and secondly, for computers whose only peripheral is a connection to a local area network.

In designing TRIPOS, the object was to produce a system which was easy to transport, pleasant to use, and simple to explain and understand both at the level of the user interface, and internally. These intentions have met with some success. It has been shown that TRIPOS can run on a variety of machines. Users outside the original group (and Laboratory) have successfully modified TRIPOS to suit their own particular applications.

The philosophy of TRIPOS can be compared with that of its implementation language, BCPL. Both are attractive to use because they offer many of the advantages of a very high level language, or elaborate operating system - that is, aids to organizing work, the means to achieve a lot with a small amount of typing, using abstractions to hide low-level machine peculiarities - but without carrying the abstractions so far that facilities offered by the hardware become difficult to get at. Both try to be an aid to the programmer, but not to get in his way. Both aim to provide abstractions, but make the right concessions to the underlying structure of present-day computers.

A feature of BCPL that distinguishes it from most other high-level languages is that many of the facilities available to the programmer are not bound into the language, but are provided as library routines. For example, the coroutine mechanism and all input and output operations are provided in this way. The analogy in TRIPOS is that many of its facilities are made available by dynamically loaded programs run as commands, rather than being built into the system. The command line interpreter is a small and simple program which executes commands serially. Nevertheless, it is possible to run programs in separate processes (via the RUN command), and to execute command sequences from files with parameter substitution, recursion, jumps and conditional execution (C, IF, SKIP commands). There are no built-in commands at all.

The principal way in which TRIPOS differs from other portable operating systems is that it offers much of the power and flexibility of a multi-user system, while exploiting the simplicity and efficiency possible in a single user system. The conscious decision not to make use of hardware features such as processor states, memory protection and virtual addressing has led to a simple system which makes few assumptions about the underlying machine, and is easily transported to a variety of computers. Other portable systems have either been rather restrictive interpreted single-user systems, or substantial multi-user systems which are rather more complicated and not capable of the same degree of portability without considerable rewriting of code. All levels of system have their value: simple interpreted ones are good for small machines or fast implementation, complex ones are needed to support many users running arbitrary programs. TRIPOS has demonstrated that there is a worthwhile compromise in the middle, providing power and flexibility for a single user without the overheads of protection that he rarely needs.

As it was designed with a particular style of use in mind, there are some things that TRIPOS is not good at. It is not suitable as a multi-user system, as it cannot protect the users from each other, or share the processor time fairly between them, and cannot extend the effective store size of the machine by using swapping or virtual memory. However, it can safely serve multiple users when they are allowed to run only tested programs with known behaviour. It has been used in this way for a multi-terminal data entry and editing system. The system cannot be recommended for the novice programmer as his mistakes can easily cause it to fail in a way which will not give a helpful diagnostic. Similarly, it is not suitable for anyone who is used to relying heavily on compiler type-checking or memory protection faults as a substitute for thought when getting programs working.

The internal structure and workings of TRIPOS are described in some detail in chapter three. The theoretical study of portability is of only limited value; it is important that techniques devised are implemented on a range of machines, and extensively used to show up their strengths and weaknesses. Because this system is simple, it has been possible to explain its design fairly completely in order to illustrate how the portability and power have been achieved. This also provides the necessary context for later chapters in which the same primitives and structures are employed to build an operating system with the same user and program interfaces as the original, but in a rather different hardware environment.

TRIPOS contains several features which (to the author's knowledge) are novel. The message system is particularly simple and unrestrictive, and provides a completely uniform interface to tasks, external devices, and the timer hardware. This uniformity enabled, for example, the disc device driver to be directly replaced by a task which provided the same functions by using a remote file server; no changes at all were needed to the filing system code which used the disc (chapter 6). The use of coroutines and the PKTWAIT routine to structure a multi-event task with only one message channel (chapter 3) gives an elegant and efficient technique for building a complex process on simple primitives. The provision of a debugging system as a separate task from the programs being inspected has often been found to be a valuable aid. The ability to have several interactive programs running simultaneously, and the mechanism for allowing the user to select which he wishes to talk to, and to type-ahead to each of them independently, has proved to be a useful facility and to make the system attractive to use. It is particularly powerful when using a local machine to debug code in another across a network. One task could be running a program that sends console input to the remote machine, another could be running a debugging program that enables memory locations in the remote machine to be inspected, and a third could be used for executing commands locally.

The fact that TRIPOS was running on several different types of computer provided good day-to-day experience in writing portable programs, and supported the view of Ritchie [40] that portable programs are good programs for reasons other than their portability. Most programs could be transferred between machines with no changes at all; only a few needed to take care of underlying machine differences such as natural address unit, or natural byte order within words.

Chapter five describes how the primitives described previously were used to construct portable software to drive a local network and its protocols, and to enable the operating system to act a server as well as a client. The mechanisms for pseudo-devices and streams provided by the operating system allowed the network stream protocol to be handled in the same way as a stream to a local terminal or disc file, giving simple access to remote files. The effects of exporting the lowest protocol level into a microprocessor-based interface are discussed, and it is observed that there is only a slight improvement in elapsed time for a data transfer, but a considerable reduction in the processor time consumed in the host.

A network environment introduces new constraints when writing code intended to be portable. The fact that otherwise identical systems are using different word lengths and byte orders becomes important when blocks of data are to be passed between them; it is usually not important when data is written and read on the same machine.

Since the basic data unit in BCPL is the machine word, code in that language has to be written very carefully to be independent of word length. This is best done by using procedures to put / get bytes and 16-bit quantities into / from buffers in a machine independent way. A tighter definition of the byte ('%') operator, would aid the writing of such programs; it should be specified that it treats bytes within a word as being in a machine-independent order (most significant byte in a word having lowest offset in vector). At the moment, the order is not defined, so implementors use the natural byte order for the host machine.

The work on distribution of the filing system in chapters 6 and 7 investigated different levels of interface possible between a client filing system and a medium-level file server. Three distinct levels were tried: treating the file server as if it were just a virtual disc server, using it as a provider of a flat filing system, and making full use of its index structure and locking primitives. Quantitative comparisons of efficiency were not made, as there was only a single client machine for much of the period of this work, and so it was not possible to investigate performance under load, which is what is ultimately important. Implementations of each design were made, to discover the complexity and quantity of the code needed, and to demonstrate the viability of each.

From the point of view of amount of code in the client, a medium-level interface to a file server is not very attractive, as the client still has most of the work of implementing his filing system, plus a substantial amount of communication code for calling all the file server functions. It seems likely that the best approach when there are many clients is to have a file server which provides only page-level access to virtual discs, and maintains a large cache. The client code is kept to a reasonable level, and the best use is made of the total processing power available on the network, but putting as little function as possible in the central server.

The modularity of TRIPOS meant that it was possible to use it as the basis for an operating system to go into 'single-connection' personal computers by taking some parts of the original completely unchanged, and joining them with a substantial amount of new code. Most programs can run without alteration in either system. The fact that the program and user environments could be moved to this unusual hardware environment justifies the level of abstraction incorporated into the system. TRIPOS has much of the unrestrictive nature of a very open system, in that there is little distinction between system and user programs, and that the facilities available to the operating system are available to other programs. However, it is not so open that many programs can see details of the peripheral hardware, so fundamental changes in that area can be hidden.

The work described above suggests some advice which can be offered to others embarking on related projects. A key to portability is to make as few assumptions as possible about hardware features. Any restrictive assumptions that are made will either reduce the range of possible target machines, or make it necessary to achieve portability by secondary techniques such as conditional compilation or multiple versions of programs. It is important in the design of an operating system to get a prototype version up and running at an early stage, so that practical experience can show the merits or otherwise of the primitives and structures chosen. For a portable operating system, it is important to have early versions on several machines, so that its design is not unduly influenced by particular hardware, and to give the implementors practice in writing portable programs. The designers must be prepared to make even quite far-reaching changes in the light of early experience with their system; actually implementing a design greatly increases the reluctance to alter it. TRIPOS was altered in many ways after early use. Examples are that the conventional packet format was changed to have two result fields (rather than one), and a primitive for finding the size of an allocated store block was abolished because it proved of little use. There should be a clear idea of what kind of system is being produced, because overall aims decided in advance help with detailed design decisions, and produce a more consistent result. In TRIPOS, this is illustrated by the determination to keep the system simple at all levels.

There are several areas for further research related to the topics discussed in this dissertation. Extensive development of TRIPOS itself is probably not a sensible undertaking, however. A major aim was to design a system which was simple throughout, so addition of many new features to it (such as elaborate command line interpreters, or complicated filing systems) would be likely to destroy much of the original elegance and adaptability.

The techniques of remotely debugging a machine across a network need further study. At the time of writing, two remote versions of the TRIPOS DEBUG program have been written (e.g. [1]), but neither allows the full facilities of a local debugger, such as inspecting a task's registers at any time. A particular problem with trying to debug a machine on a network is that it may have connections established with various other computers, which will rapidly time out if it is simply halted. For instance, a machine in the processor bank will normally be refreshing the Resource Manager's dead man's handle at frequent intervals, and will usually have a byte stream open to a terminal, files open in the Fileserver, and authentication UIDsets held in the Active Object Manager. A usable debugger needs to have some knowledge of which parts of the system should not be frozen in order to keep the rest of the world happy. It is not very good to have the facility to set breakpoints in a target machine, if it is essential to continue execution within, say, 30 seconds of a breakpoint being encountered.

A promising area of research is in investigating how to divide work between a file server and a client filing system, by varying both ends. This thesis describes experiments in varying just the client end, and employing different subsets of the facilities of a file server with a medium-level interface. Another way of distributing filing system function (which is being explored at the time of writing), is to dedicate a machine to providing the bulk of the client filing system (using the file server), and to have just the minimum communication code left in each client. This should certainly free some storage in each client, but it is not clear that it will be faster under heavy load. The filing system server can undertake intelligent caching because it knows a lot about the objects with which it is dealing; however, a considerable amount of work has been removed from the multiple clients and concentrated in a single server, so it could become a bottleneck.

The simplicity in the design of TRIPOS has meant that the distribution of operating system functions is probably not worthwhile in itself, as it can be in a more complex system. The use of a virtual terminal indeed simplifies, and reduces the size of, the terminal handler task, but the tiny console driver is replaced by a rather large handler task to drive the Byte Stream Protocol. The management of a filing system on the Fileserver is of comparable complexity to management of one on a local disc, but again, a great deal of code is needed to communicate with the Fileserver, replacing a very small amount to drive a disc. Some extra complexity is introduced into the file handler by the need to hold object interlocks externally, so the two filing systems are not directly comparable.

The advantages of this distribution of function must be viewed in wider terms. In any particular machine, it means that it runs slower and has less free store. However, a whole new personal computer can be added to the network for the cost of CPU, memory, and network interface only. The new machine immediately has disc storage, terminals and printers available, and a wide range of services at its disposal.

It seems likely that the widespread use of local networks will increase the demand for portable operating systems, as each user will have access to a greater variety of machines. At the time of writing, the Processor Bank contains fourteen computers, of two different types (eight LSI4s and six 68000s). The use of a portable operating system and shared central filing system has made it possible (and usual) to ask simply for "a computer running TRIPOS", and to be allocated any of the machines which happens to be available. Particular attributes of the required machine may be specified if necessary, but the availability of a portable operating system means that general properties such as memory size are now more significant than architecture or instruction set.

## References

[1] M.S. Atkins and B.J. Knight,
"Experiences with Coroutines in the Development of a Remote Interactive Debugger using BCPL", To appear in Software - Practice and Experience.

[2] A.R. Aylward,
"Memory Mapping in a Portable Operating System", Ph.D. Thesis, Cambridge University, November 1980

[3] A.D. Birrell and R.M. Needham,
"A Universal File Server",
IEEE Transactions on Software Engineering, Sept 1980, pp. 450-453.

[4] P. Brinch Hansen,
"The SOLO Operating System: A Concurrent Pascal Program",
"The SOLO Operating System: Job Interface",
"The SOLO Operating System: Processes, Monitors, and Classes", Software - Practice and Experience, Vol 6, No. 2 (1976), pp. 141 - 200.

[5] D.R. Cheriton, M.A. Malcolm, L.S. Melen, G.R. Sager,
"Thoth - A Portable Real-time Operating System", Communications of the A.C.M., February 1979, pp. 105 - 114.

[6] D.R. Cheriton,
"Multi-Process Structuring and the Thoth Operating System", Department of Computer Science, University of British Columbia, Vancouver.

[7] C.N.R. Dellar,
"The Distribution of Operating System Functions", Ph.D. Thesis, Cambridge University, September 1980

[8] C.N.R. Dellar,
"Removing Backing Store Administration from the CAP Operating System", ACM Operating Systems Review 14(4), pp. 41-49, October 1980.

[9] J. Dion,
"Reliable Storage in a Local Network", Ph.D. Thesis, Cambridge University, February 1980

[10] J. Dion,
"The Cambridge File Server", ACM Operating Systems Review 14(4), pp. 26-35, October 1980.

[11] R.D. Evans,
"Language Implementation in a Portable Operating System", Ph.D. Thesis, Cambridge University, March 1981

[12] G.R. Frank and C.J. Theaker,
"The Design of the MUSS Operating System", Software - Practice and Experience, August 1979, pp. 599 - 620.
(The same issue contains several other papers on aspects of MUSS.)

[13] M. Fridrich and W. Older,
"The Felix File Server", ACM Operating Systems Review 15(5), pp. 37-44, December 1981.

[14] J.J. Gibbons,
"The Design of Interfaces for the Cambridge Ring", Ph.D. Thesis, Cambridge University, September 1980

[15] C.G. Girling,
Ph.D. Thesis (in preparation), Cambridge University.

[16] C.A.R. Hoare,
"Monitors: An Operating System Structuring Concept", Communications of the ACM, 17, 10, pp. 549-557, October 1974.

[17] A. Hopper,
"Local Area Computer Communication Networks", Ph.D. Thesis, Cambridge University, April 1978

[18] M.A. Johnson,
"Byte Stream Protocol Specification", Systems Research Group Document, University of Cambridge Computer Laboratory, April 1980.

[19] B.J. Knight,
"The PDP11 Implementation of TRIPOS", internal document, University of Cambridge Computer Laboratory, 1978

[20] B.W. Lampson and R.F. Sproull,
"An Open Operating System for a Single-User Machine",
ACM Operating Systems Review, 13(5), (Nov 1979).

[21] H.C. Lauer and R.M. Needham,
"On the Duality of Operating System Structures", in Proc. Second International Symposium on Operating Systems, IRIA, October 1978, reprinted in ACM Operating Systems Review, 13(2), pp. 3-19, April 1979.

[22] Hugh Lauer,
"Observations on the Development of an Operating System", ACM Operating Systems Review 15(5), pp. 30-36, December 1981.

[23] P.M. McLellan,
"The Design of a Network Filing System", Ph.D. Thesis, Edinburgh University, November 1981.

[24] R.M. Metcalfe and D.R. Boggs,
"Ethernet: Distributed Packet Switching for Local Computer Networks",
Communications of the A.C.M., Vol. 19, pp. 395-404 (July 1976).

[25] R. Miller,
"UNIX - A Portable Operating System?", Operating Systems Review, Vol. 12, No. 3, pp. 32 - 37 (July 1978).

[26] James Mitchell and Jeremy Dion,
"A Comparison of Two Network-Based File Servers", to appear in Communications of the ACM. (Summary in ACM Operating Systems Review 15(5), pp. 45-46, December 1981.)

[27] R.M. Needham,
"User-Server Distributed Computing",
Proceedings of the Joint IBM / University of Newcastle Seminar on Distributed Computer Systems (Sept 1978), pp. 71-78.

[28] R.M. Needham,
"Systems Aspects of the Cambridge Ring", Proc. 7th Symposium on Operating System Principles, Pacific Grove, California, pp. 82-85, December 1979

[29] R.M. Needham and A.J. Herbert,
"The Cambridge Distributed System", In preparation; to be published by Addison-Wesley.

[30] N.J. Ody,
"A Single Shot Protocol", Systems Research Group Note, University of Cambridge Computer Laboratory, April 1979

[31] N.J. Ody,
"The Machine Interface to the Terminal Concentrator - Virtual Terminal Protocol", Systems Research Group Note, University of Cambridge Computer Laboratory, November 1980

[32] M.S. Powell,
"Experience of Transporting and Using the SOLO Operating System", Software - Practice and Experience, July 1979, pp. 561 - 570.

[33] Ian E. Powers,
"NESTAR MODEL A - A low cost network for microcomputers",
Local Networks and Distributed Office Systems, Online Publications Ltd., 1981, pp. 65-72.

[34] D.D. Redell, Y.K. Dalal, T.R. Horsley, H.C. Lauer, W.C. Lynch, P.R. McJones, H.G. Murray, S.C. Purcell,
"PILOT: An Operating System for a Personal Computer",
Communications of the A.C.M., Vol. 23, No. 2 (Feb 1980), pp. 81-92.

[35] M. Richards,
"BCPL: A Tool for Compiler Writing and System Programming", AFIPS S.J.C.C. Proceedings 35, pp. 557-566, 1969.

[36] M. Richards, A.R. Aylward, P. Bond, R.D. Evans, and B.J. Knight,
"TRIPOS - A Portable Operating System For Minicomputers", Software - Practice and Experience, June 1979

[37] M. Richards and C. Whitby-Strevens,
"BCPL: The Language and its Compiler", Cambridge University Press, 1979

[38] M. Richards and J.K.M. Moody,
"A Coroutine Mechanism for BCPL", Software - Practice and Experience, October 1980

[39] D.M. Ritchie and K. Thompson,
"The UNIX Time-sharing System", Communications of the A.C.M., Vol 17, No. 7 (1974), p. 365.

[40] D.M. Ritchie, K. Thompson, S.R. Bourne, and S.C. Johnson,
"The UNIX Time-sharing System",
"UNIX Implementation",
"A Retrospective",
"The UNIX Shell",
"Portability of C Programs and the UNIX System",
Bell System Technical Journal, July-August 1978.

[41] W.D. Shepherd,
"Ancilla - A Server for the Cambridge Model Distributed System", Software - Practice and Experience, November 1981, pp. 1185-1196.

[42] C.R. Snow,
"An Exercise in the Transportation of an Operating System",
Software - Practice and Experience, Vol 8, no. 1 (1978), pp. 41-50.

[43] J.E. Stoy and C. Strachey,
"OS6 - An Experimental Operating System for a Small Computer",
Part 1: "General Principles and Structure", Computer Journal, 15, No. 2, (1972), pp. 117 -124.
Part 2: "Input/output and Filing System", Computer Journal, August 1972, pp. 195 - 203.

[44] H.E. Sturgis, J. Mitchell, and J. Israel,
"Issues in the Design and Use of a Distributed File System", ACM Operating Systems Review 14(3), July 1980.

[45] Dominic Sweetman,
"A Distributed System Built with a Cambridge Ring",
Local Networks and Distributed Office Systems, Online Publications Ltd., 1981, pp. 451-464.

[46] D.C. Swinehart, G. McDaniel, D.R. Boggs,
"WFS: a Simple Shared File System for a Distributed Environment",
ACM Operating Systems Review 13(5), (Nov 1979).

[47] C.P. Thacker et al.,
"Alto - A Personal Computer", Computer Structures: Readings and Examples, Sieworek, Bell and Newell, eds., McGraw-Hill, 1980

[48] R.D.H. Walker,
"Basic Ring Transport Protocol", Systems Research Group Note, University of Cambridge Computer Laboratory, October 1978

[49] M.V. Wilkes and D.J. Wheeler,
"The Cambridge Digital Communication Ring", Proc. Local Area Communications Network Symposium, Mitre Corp. and Bureau of Standards, Boston, May 1979

# APPENDIX 1

## SUMMARY OF BLIB ROUTINES

The list presented below includes all the routines in the BCPL-written library BLIB, with a brief indication of what each one does.

### I/O Stream functions

| | |
|---|---|
| INITIO | Initialize input/output |
| RDCH | Read one character from current input stream |
| UNRDCH | Step back one character in input stream |
| WRCH | Write one character to current output stream |
| READWORDS | Read vector of words from input stream |
| WRITEWORDS | Write vector of words to output stream |
| FINDINPUT | Open an input stream |
| FINDOUTPUT | Open an output stream |
| FINDUPDATE | Open a stream for input and output |
| SELECTINPUT | Select a new current input stream |
| SELECTOUTPUT | Select a new current output stream |
| ENDREAD | Close the current input stream |
| ENDWRITE | Close the current output stream |
| ENDSTREAM | Close a specified stream |
| INPUT | Return the current input stream |
| OUTPUT | Return the current output stream |

### File Operations

| | |
|---|---|
| DELETEOBJ | Delete a file or directory |
| RENAMEOBJ | Rename a file or directory |
| LOCATEOBJ | Return a lock on a file or directory |
| FREEOBJ | Release a file or directory lock |
| COPYDIR | Make a copy of a directory lock |
| CREATEDIR | Create a directory |

**Formatted I/O**

| | |
|---|---|
| READN | Read a number from the input stream |
| NEWLINE | Start a new output line |
| WRITED | Write a decimal number in fixed width |
| WRITEN | Write a decimal number in necessary width |
| WRITEHEX | Write a hexadecimal number |
| WRITEOCT | Write an octal number |
| WRITES | Write a string |
| WRITEF | Write values according to a format string |

**Packet operations**

| | |
|---|---|
| SENDPKT | Send a packet containing the given values and await its return |
| RETURNPKT | Return a packet with the given results |
| PKTWAIT | = TASKWAIT by default: exists so it can be redefined for special purposes (see chapter 3) |
| DELAY | Wait for given period (sends packet to timer) |

**String and Character Operations**

| | |
|---|---|
| PACKSTRING | Compress word vector of characters into string |
| UNPACKSTRING | Expand string into word vector of characters |
| CAPITALCH | Convert given letter to upper case |
| COMPCH | Compare two characters (equating letter cases) |
| COMPSTRING | Compare two strings (equating letter cases) |
| SPLITNAME | Split a string at a given character |

**Command Argument Decoding**

| | |
|---|---|
| RDARGS | Read and decode arguments from current input stream, given a keyword format string |
| RDITEM | Read one item (i.e. word) from the input stream |
| FINDARG | Locate a particular keyword in a string containing a list of keywords |

**Program Loading**

| | |
|---|---|
| LOADSEG | Load a program from a named file |
| UNLOADSEG | Delete a loaded program |
| CALLSEG | Call a program from file as a subroutine, deleting it when it returns control |

**Miscellaneous Routines**

| | |
|---|---|
| DATSTRING | Give the current date, time, and day as 3 strings |
| DATSTAMP | Give the current date and time in binary |
| ENDTASK | Delete the current task, and unload a specified segment of program code |
| DEVICETASK | Find the task number of a device handler from the device name (see chapter 3) |
| FAULT | Write out the message corresponding to the given fault code |

# APPENDIX 2

## RING LIBRARIES, SERVICES, AND COMMANDS

This appendix describes the TRIPOS Ring software components not included in chapter 5. These are the BCPL libraries of Ring routines, the services available for other machines to call, and the Ring-related commands.

## A2.1 Libraries

Some routines of general use to Ring programs are made available in libraries. The libraries are in source form, so that they can be included in programs by the BCPL "GET" directive.*

There are two such libraries, called SSPLIB and BSPLIB.

### SSPLIB

SSPLIB contains routines for making SSP calls, looking up names in the Nameserver, and obtaining names from station numbers. The SSP routine can call a service either by its name, or from a supplied Ring address previously looked up.

--------------------

* The use of libraries supplied as object modules is awkward in BCPL, because linking is done via the global vector; the library would have to claim some globals not used by anything else, and a header file would still have to be provided to define the names and numbers used.

**BSPLIB**

BSPLIB contains routines for performing special operations on, and testing the state of, byte streams. These are functions which are peculiar to byte streams and hence are not provided by the normal stream interface. Examples are:

- forcing transmission of buffered data
- testing whether a byte stream RESET has occurred
- causing a RESET
- sending a 'close request' without closing the stream
- testing whether any input is pending (i.e. whether a call of RDCH would halt)

## A2.2 Services

The Ring services usually provided by machines running TRIPOS are the following:-

| | |
|---|---|
| WTO | 'Write To Operator'*: receive and display a one-line message |
| TAKEFILE | Receive a (character or binary) file |
| GIVEFILE | Transmit a file |
| READ | Create a stream to read a file |
| WRITE | Create a stream to write to a file |
| RATS | 'Remotely Activated Terminal Session': create a new CLI and virtual terminal handler to allow remote logging-on. |

Conventionally, all these services have names in the Nameserver constructed as <service name>-<machine name>. E.g. on the Nova, the services would be WTO-NOVA, TAKEFILE-NOVA, etc.

--------------------

* from the name of an IBM Assembler macro.

## A2.3 Commands

The TRIPOS Ring software includes many commands, to cover a variety of functions:-

- File transfer
- Sending operator messages
- Nameserver operations
- Status of other Ring stations
- Remote logging-on to other machines
- Starting, stopping and inspecting local Ring tasks

**File transfer**

```
GIVEFILE  "FROM/A,TO=MACHINE/A,AS/A,BINARY/S"
```

Copies a file from the local machine to another, either as characters or in binary.

```
TAKEFILE  "FILE/A,FROM=MACHINE/A,TO/A"
```

Fetches a file from another machine, giving it the specified name here.

**Operator Messages**

```
WTO   machine   message
```

sends the message to the named machine, where it should be displayed on the system console.

**Nameserver Operations**

```
LOOKUP "NAME/A"
```

This prints the Ring address and flags corresponding to NAME.

```
LISTNAMES "TO"
```

This lists the entire Nameserver table.

**Ring Status**

```
RPROD "MC/A,PORT,CLOSE,BOOT"
```

Sends a basic block containing one data packet to the specified machine, and reports on whether it was accepted. CLOSE causes the block to contain a BSP CLOSE command, so it can be used to encourage errant byte streams to go away. BOOT sets the data to the value which will cause a crashed PDP11 TRIPOS system to reboot.

```
RSTATUS
```

Sends a basic block (on port 1) to each Ring address in turn, and lists all those for which the block is not 'ignored'. This is an easy way of discovering which machines are 'alive'.

**Remote logon**

```
STAR "MACHINE/A"
```

Connects to the RATS service of the specified machine. Console input is passed on to that machine, and output is displayed. The connection ends either when the byte stream is closed by the far end (e.g. after a FINISH command), or can be forced by control-D break.

**Starting, stopping and inspecting Ring tasks**

The Ring handler is started by the command LOADRINGHAND. This command loads and creates the transmitter and receiver devices, and creates the Ring handler task, unless it was already running.

```
RHINFO "KILL/S"
```

Prints information about outstanding reception requests, reserved ports, and bad or unwanted blocks received.

The command RHINFO KILL is used to make the handler delete itself, abandoning any outstanding requests. Thus, any program which uses the handler should be killed first. In TRIPOS systems in the processor bank, the Ring handler is a resident part of the system, so does not need to be loaded, and does not respond to RHINFO KILL.

The Ring services task is loaded by the command LOADRINGSERV. Setting flag 4 (control-D break) or killing the Ring handler makes it go away.

The command BSINFO may be used to inspect the state of byte streams. It prints the task number of the BSP handler, and information on the states and sequence numbers of each open stream pair. The BSP handler may be killed by using the command BSINFO KILL. This causes the handler to go away when it next has no streams open.