An I/O and Stream Inter-Process Communications Library for a Password Capability System

By
Carlo Kopp
B.E.(hons), PEng

Department of Computer Science
Monash University

Thesis submitted for examination
for the Degree of
Master of Science

March 1996

Table of Contents

Summary

This thesis presents an I/O library and stream Inter-Process Communications (IPC) mechanism for the Walnut password capability system, developed in the Computer Science Department at Monash University. This library provides a C language programming interface, mostly conforming to the ANSI *stdio* model, to functions which provide stream I/O services to the virtual memory system, processes and I/O devices.

Chapter one introduces the architecture of the Walnut kernel. The Walnut virtual memory system, capabilities and objects, money, access rights, process and subprocess structures, message mechanism and system calls are examined.

Chapter two is a survey of design strategies used in the provision of I/O library and stream IPC services. This chapter examines the problems inherent in providing such services, and reviews the System V Unix model, the 4BSD Unix model, the Mach model and the Alloc Stream model. This is done to provide an understanding of alternative solutions to the problem of providing a common programming interface to I/O services and stream IPC.

Chapter three discusses the design of the file access and stream IPC mechanisms. The Walnut I/O model is reviewed, design requirements and considerations are examined, and the design rationale for the stream IPC mechanism, file object access mechanism, stream communications programming interface and process environment initialisation is explained. The data structures used in the provision of these services are examined in detail.

Chapter four reviews the implementation of the library design. Functions specific to the Walnut are examined, and the ANSI *stdio* stream operation, character I/O, direct I/O, formatted I/O, file positioning and error handling functions are examined in detail.

Chapter five discusses the library design and implementation from the perspectives of programming interface, throughput performance, portability, robustness and security. Specific advantages which accrue from the design strategy used are examined, performance testing results reviewed, and a range of further improvements to the library are proposed.

Chapter six summarises the major issues examined in this thesis.

Acknowledgements

In 1990 I set out to do a Masters degree in Computer Science, seeing a bright future in this fascinating discipline. After a promising start to my project, external circumstances conspired to impede my progress, and I was unable to return to my research until early 1994. This thesis is largely the product of effort expended in the last two years.

Throughout this protracted and often difficult period my supervisor, Professor Chris Wallace, provided me with the guidance, motivation and critical insight which are so vital to the successful completion of such projects. Without his patience and ability as a scientist and as a teacher, it is unlikely that I would have succeeded in resurrecting my project. My sincerest thanks.

I am also indebted to Maurice Castro, for his advice on the Walnut kernel and constructive critique of the library design, to Glen Pringle, who developed numerous components of the Walnut system and was always ready to assist, and to Dr Ronald Pose, for his insightful advice and critique.

Declaration

I declare that the thesis contains no material which has been accepted for the award of any degree or diploma in any university and that, to the best of my knowledge, the thesis contains no material previously published or written by another person other than where due reference has been made in the text.

Signed:
Carlo Kopp
Department of Computer Science,
Monash University,
Melbourne, Australia, 3168
March 1996

Chapter 1 Introduction

1.1 History and Objectives of the Walnut Kernel Project

In 1985 Anderson, Pose and Wallace at Monash University completed a tightly coupled multiprocessor system, and implemented an operating system to provide virtual memory support for this system. The purpose of this project was to demonstrate a tightly coupled multiprocessor [POSE89] which employed a password capability based virtual memory system [ANDERSON87]. The work built upon earlier Monash research in capability based addressing schemes [ABRAMSON82], [GEHRINGER82].

In 1990, the development group, then comprising Castro, Kopp and Wallace, commenced a port of this operating system to the Intel 386 architecture [INTEL84], [CRAWFORD87]. The intention was to exploit the combined segmented and paged memory management hardware embedded in the Intel chip, to produce a simplified derivative of the multiprocessor password capability operating system, capable of running on reliable low cost hardware. This port was also seen as an opportunity to improve the design, building on the experience gained with the multiprocessor project.

Analysis of the source code of the multiprocessor operating system indicated a large number of dependencies upon the NS32032 architecture [NS85] and custom designed memory management hardware used in the multiprocessor design. It was subsequently decided that a completely new implementation was preferable to a port. The new implementation, written by Castro and Wallace, was subsequently designated the Walnut.

The Walnut implementation is at this time hosted on a Personal Computer, using an Intel 486 processor and a set of generic I/O boards. As such this implementation only demonstrates single processor operation.

1.2 The Walnut Architecture

The Walnut architecture is directly derived from the architecture of password capability multiprocessor kernel. It shares many fundamental design features with its predecessor. These features are the virtual memory system, the capability mechanism, the money mechanism, substantial parts of the process structure and a capability access rights scheme. Unlike the multiprocessor system, the Walnut system also has a name serving function. The name serving function is implemented as a library, and is not embedded in the Walnut kernel.

This discussion will focus on those aspects of the Walnut architecture which are relevant from the perspective of implementing an I/O library. A more detailed treatment of the kernel is contained in [CASTRO95].

1.2.1 Virtual Memory

The virtual memory scheme used in Walnut kernel is based upon the multiprocessor password capability model [APW86], [POSE89]. In this model, the virtual address space is divided into a number of volumes, each of which contains a number of objects. Each volume has a permanent and unique 32-bit identifier, termed a volume number.

The volume number is permanently associated, in both existing implementations, with a specific fixed or removable storage device. The volume number is assigned when a device is configured for operation, and is fixed for the life of the volume. A volume number can only ever be reused if the volume originally assigned the volume number is destroyed.

The existing implementation uses an arbitrary number. A production implementation is intended to use a number unique to the physical device used, such as the trailing digits of a physical device production serial number.

While the volume is an essentially permanent construct, the objects contained within the volume are not. Objects may be created or destroyed, and can only ever exist within the address space of a given volume. Objects cannot be split across multiple volumes.

Objects are identified with a 32-bit serial number. The combination of the 32-bit volume identifier and the 32-bit serial number uniquely identifies each object. A serial number can only be assigned to one object on any given volume, and is a fixed value throughout the life of this object. Two different volumes may each possess an object with a common serial number, because the volume identifiers are unique. Within the address space of all uniquely numbered volumes in existence, each object is thus uniquely identified. Serial numbers may be reused.

The choice of a 32-bit volume identifier and a 32-bit serial number was a compromise between providing a suitably sized object address space, and compactness of storage. To date this size has proven to be acceptable.

1.2.2 Capabilities and Objects

The Walnut kernel, as well as the earlier multiprocessor and its associated operating system, employ a password capability based scheme for accessing objects [APW86]. Each capability is defined by a 128-bit value. This 128-bit value is composed of a 64-bit object identifier field, and a 64-bit password field. In this scheme, any object may be accessed through an arbitrary number of capabilities, differing only in their passwords and constrained by the available space on the volume.

VOLUME	SERIAL	PASSWORD 1	PASSWORD 2
32 BITS	32 BITS	32 BITS	32 BITS

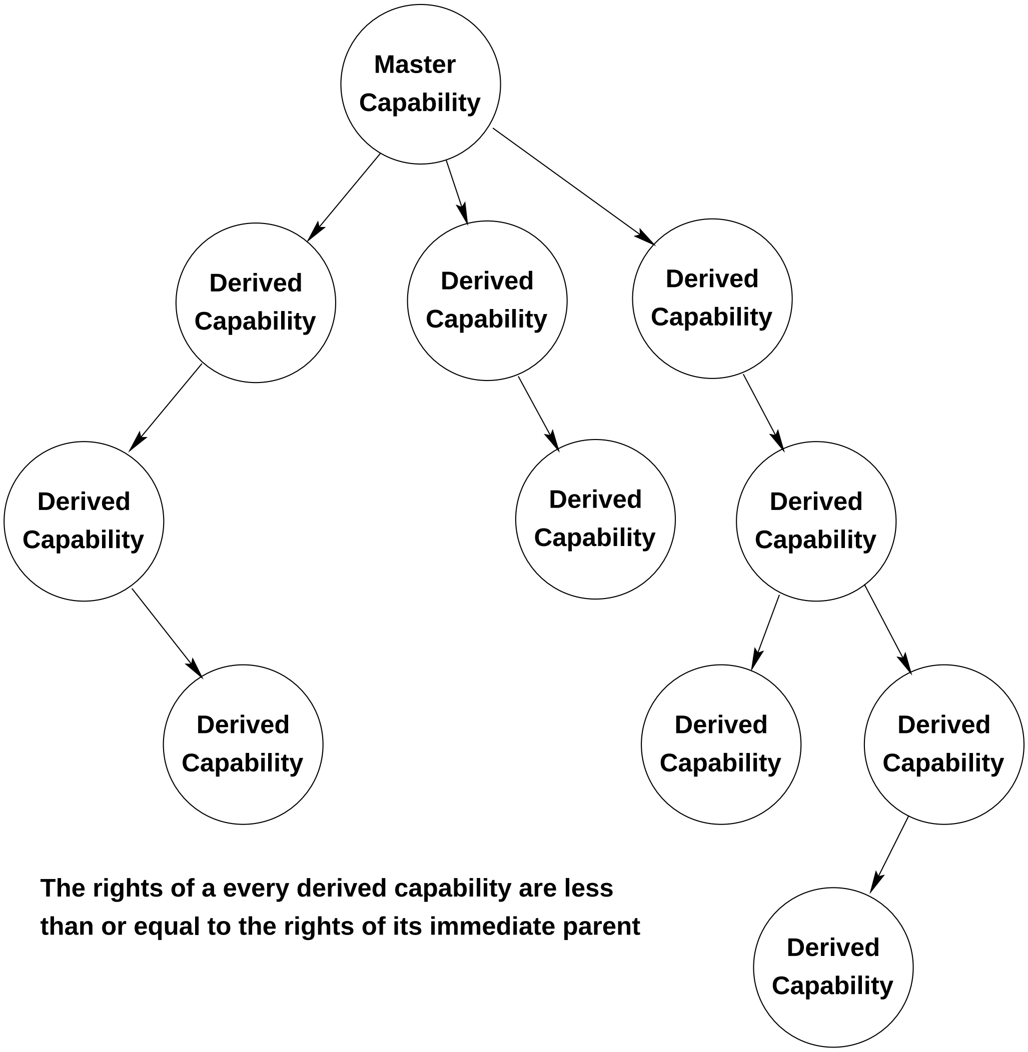
Figure 1.1 A 128 Bit Password Capability

Capabilities to access any given object may have differing or identical access rights. Each capability is however a unique value, regardless of whether two or more capabilities provide identical access rights.

Capabilities associated with any given object will share identical volume identifiers and identical serial numbers, but will possess unique 64-bit password fields. The password is intended to be a number which is unrelated to the access rights of the capability, therefore no mapping computable by a user exists between the password field and the access rights mask of the capability, in either direction. The system holds the mapping from password to rights in a protected "capability table" associated with each object and stored on the same volume as the object.

A capability is said to be created at that time, when it provides access to an object. It is destroyed at that time, when it can no longer provide access to an object. The validity of any given number as a capability is determined by its ability to grant access to an object on a volume. Capabilities are associated with volumes, and are regarded as part of an object. Should a volume be moved between systems, all capabilities which exist on the volume retain their validity in relation to that volume.

The use of the 64-bit password cannot provide an absolute guarantee of security, but does provide a very low probability of the security scheme being defeated. Even should many capabilities to an object exist, the probability of the correct password being guessed is of the order of ten to the minus fifteen.



The rights of a every derived capability are less than or equal to the rights of its immediate parent

Figure 1.2 Master Capability and Derived Capabilities

Were the password derived from the volume and serial numbers, the system could be more easily defeated, in comparison with the random guessing of what are essentially sparse valid password values. For this reason the passwords are intended to be generated randomly. The present Walnut implementation uses an simple pseudo-random number generator for this purpose. The final implementation will use the physically random number generator which was developed for the original multiprocessor design [WALLACE90].

The password scheme is further enhanced by the money mechanism [WP90]. As each attempt to guess the password will incur a cost penalty to the guessing process, systematic attack will be prohibitively expensive to the attacking party.

A Master Capability is produced whenever an object is created. The creator of the object will specify its rights, and no other party can alter these. To derive a capability, we require the value of the parent capability and a desired set of rights. The new derived capability will have rights which are a subset of the rights held by the parent capability. Additional rights not held by the parent capability cannot be added to a derived capability.

A model which can be applied to describe the interdependencies between the master capability and its derived capabilities is that of an inverted tree. The root of the tree is the master capability, and every other node in the tree is a derived capability. The master capability is thus an ancestor of all derived capabilities in the tree. When a new capability is derived, it is attached to the tree as a child of the capability from which it was derived. When a capability is destroyed, all of its descendants are also destroyed. Should the object be destroyed, then the whole tree of derived capabilities is also destroyed. Any data held within the object is also destroyed.

There is no concept of ownership in the virtual memory scheme used. A capability is implicitly tied to the object it is derived from, but may be used and where appropriate manipulated by any user who knows it.

1.2.3 Money

The money mechanism forms the basis for a cash economy scheme built into the Walnut kernel. While it was created to support resource allocation, its function in the earlier multiprocessor kernel was expanded to provide a far more general scheme for providing usage rights to facilities in the system [WP90]. This function was subsequently continued in the later Walnut kernel.

Money in the Walnut kernel is a quantity which may be transfered between objects, and may be viewed as a transferable right to use system services. Its most important attribute is that it is consumed when the service is provided. Objects in the virtual memory system are charged rent, and if they run out of money, the system destroys them. Processes are charged money each and every time they invoke a system call. Should they have insufficient money for the call invoked, the system will refuse to service the request.

A user must have a valid capability and sufficient money to access or manipulate a capability within the virtual memory system. Possession of a capability and money does not however subsume the function of the access rights within the capability. Even should sufficient money be available, should access rights prohibit an operation, the operation cannot be provided by the system.

All objects must contain money. Money is conserved during transfers between objects, in that transferring money from one object to another will see the source object’s money decreased by the transfered amount, and the receiving object’s money increased by the same amount. Manipulating the money held in an object requires a capability with appropriate access rights. The system may charge rent for objects held on a volume.

1.2.4 Rights

The Walnut kernel provides a range of system and user rights to a capability. Rights restrict what operations may be carried out upon a capability. A derived capability may have a set of rights which is identical to or a subset of the set of rights held by the parent capability. Some of the rights are specific to process objects.

The rights in the Walnut kernel are divided into system rights and user rights. System rights determine what operations may be carried out on a capability by any entity on the system. User rights determine what operations may be carried out by user processes and are not interpreted by the kernel.

System rights may be grouped into categories, according to how they provide access or enable operations by the capability.

System rights which allow the creation and destruction of capabilities are:

- **SRDERIVE** - the right to derive further capabilities

- **SRSUICIDE** - the right of a capability to destroy itself

System rights which allow manipulation of money are:

- **SRDEPOSIT** - the right to deposit money into a capability

- **SRWITHDRAW** - the right to withdraw money from a capability

System rights which provide a view of the capability are:

- **SRREAD** - the right to read the capability

- **SRWRITE** - the right to write to the capability

- **SRMULTILOAD** - the right to load the capability in the address space of a process

- **SRUSER** - the right of user processes to use the capability

System rights related to processes are:

- **SREXECUTE** - the right to execute the process

- **SRPEEK** - the right to query the state of the process

- **SRSEND** - the right to send messages to the process

1.2.5 Processes and Subprocesses

The Walnut process structure introduced a number of additional functional features not used in the multiprocessor operating system [ANDERSON87], [CASTRO95]. These are a subprocess mechanism, and a more complex set of process states.

1.2.5.1 Processes

A process extant in the Walnut environment is defined as a sequence of executed instructions and system calls, which are performed above the level of the virtual memory interface. Functions required to support the virtual memory interface, such as swapping, storage management and scheduling, are performed within the Walnut kernel and are not implemented as processes.

A process in the Walnut environment is represented by a process object. The process object contains all of the state information required for the execution of the process. A process must contain at least the following items:

- **Parameter Page**
- **Sub-process Table**
- **Message Slots (i.e. mailbox)**
- **Table of Loaded Capabilities**
- **Money**
- **Lock Words**
- **Address Map**
- **User Defined Code and Data**

A user process may only reach certain parts of the process object. These are the Parameter Block, the read-only Address Map and the User Defined Code/Data, if the latter exists. All other components of the process object may only be accessed by the kernel.

The Parameter Page is used by the process to communicate with the kernel. It comprises a Parameter Block, and a Message Area. The Parameter Block is the means via which parameters associated with system calls are passed between the process and the kernel. The Message Area is the means via which messages are received and sent to other processes.

The process object also contains a startup code area and a private data table. The startup code area will contain, by convention, a small sequence of instructions used during the startup of a process. The private data pointer table is indexed by the capability index of the executing code, and is used to locate private data used by the executing code.

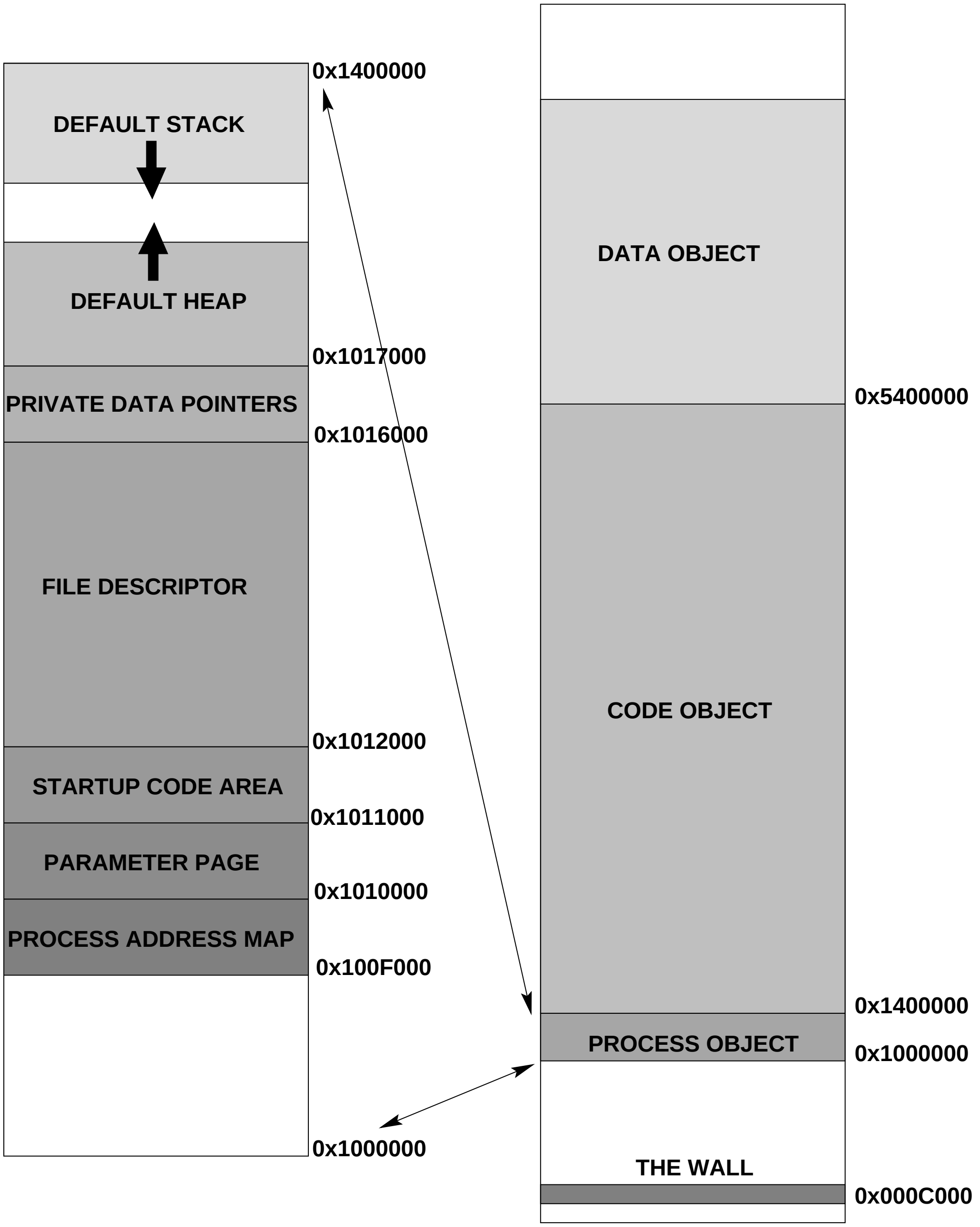


Fig 1.3 Walnut Process Virtual Address Map

The intent of this design is to allow multiple instances of a process to be created by sharing code objects and copying multiple instances of data objects. Each process will therefore possess private process objects, data objects and share a code object. By convention, initialised data is located at the beginning of the data object to simplify copying.

In addition to the process, code and data objects mapped into the process address space, each process also has a read only page designated the Wall mapped into its address space. The Wall contains public information about the system, such as real time counters and capabilities to public utilities.

1.2.5.2 Subprocesses

The process in the Walnut environment supports a subprocess mechanism. Subprocesses are threads of execution which share a common address space. No protection mechanism exists to prevent subprocesses from accessing structures belonging to their peers, therefore a programmer must ensure that a subprocess does not impair the operation of another subprocess by overwriting state information.

At process creation time, a fixed number of subprocess slots is allocated in a subprocess table. This table stores state information for each subprocess. When a subprocess is created, its scheduling priority, starting address and stack pointer address are specified.

Messages directed to a process must specify which subprocess they are intended for. Subprocess zero has a special purpose. It is used for controlling the state of the process. Operations performed by subprocesses other than subprocess zero are executed by code within the process address space. Operation performed by subprocess zero are executed by the Walnut kernel.

The subprocess zero mechanism allows a process to control the state of itself, or another process, if it possesses a valid capability to send a message to the subprocess zero of the recipient. Messages to and operations upon subprocess zero are the highest priority functions of a process.

Subprocess zero operations support the following functions:

- **freeze** - a frozen process is removed from the scheduler queue

- **thaw** - receiving a thaw message, a frozen process is placed into the scheduler queue and it becomes runnable

- **wakeup** - wakeup messages set the wakeup time of a process, which commences execution at that time

• **cooee -** the cooee message is a status enquiry through a message, rather than a system call against the capability of the process

• **protected freeze** - a process is frozen with a magic number used to prevent thawing by parties not holding this number

• **protected thaw** - a complementary call to protected freeze, protected thaw allows thawing of a process by a party holding a valid magic number

For a process to perform useful work it must execute a sequence of instructions, and typically access some data as operands. Executable code is mapped into the process address space as a code object. Data is mapped into the address space as a data object. In addition to code and data objects, the process object also provides optional space for a default stack and a default heap. These facilities will therefore support most conventional compilers. Up to 250 objects may appear at any time in the address space of a process.

Subprocesses may be created and destroyed by appropriate system calls. The scheduling mechanism will execute subprocesses which are runnable. The subprocess with the highest nominal priority is scheduled first, subprocesses of equal nominal priority are scheduled in the order of their position in the process subprocess table. Prior to scheduling subprocesses, the Walnut kernel will examine the process mailboxes for messages sent to all subprocesses. A subprocess which has received mail becomes runnable, and will be scheduled accordingly.

1.2.6 Messages

The message mechanism employed in the Walnut kernel is more powerful than that used in the multiprocessor kernel. Enhancements were provided in several areas. The first is in the support of the subprocess mechanism, the second is the provision of external send and receive calls, and the third the provision of an additional mailbox state management call.

In the Walnut kernel, messages are addressed to specific subprocesses. This is achieved by means of a system call parameter specifying the subprocess number, or by using a derived capability which allows access only to a specific subprocess.

The contents of a message to be sent are held in the Message Area [Section 1.2.5, Fig. 1.3], which is restricted in size to sixteen words. Should a larger message need to be sent, this can be accomplished indirectly by sending a capability to a larger buffer.

The Walnut kernel will filter messages directed to a process on the basis of two message parameters. These are a message prefix and a subprocess number. If these parameters are acceptable, and an empty mailbox is available, then the message can be sent. If not, an error is returned to the sending process. The use of the message prefix string allows a subprocess to retrieve messages in a specified order, by reading only those with a matching prefix string.

A mailbox may be open or closed. The Walnut kernel provides system calls to open a mailbox, close a mailbox and to receive a message and then close a mailbox.

1.2.7 System Calls

The Walnut kernel provides at this time thirtyone system calls. The procedure for executing a system call is very simple. It requires that the operation code for the call be placed into the Parameter Block *reserved* field, the parameters for the call written into the Parameter Block, and the execution of *system_call*. When the Walnut kernel returns from the call, the Parameter Block error field is inspected to verify whether the call has been successful or not. If the error code is zero, the call was successful, and the Parameter Block will contain the return values from the call. System call functions are summarised as follows:

- **K_MAKEOBJ** - creates an object. Parameters which may be specified are the volume and size, system and user rights, object type, object limits and initial money.

- **K_MAKECAP** - derives a capability from a specified capability. Parameters which may be specified are the rights and the size of the view, the latter constraining what part of the parent capability may be accessed.

- **K_DEL** - deletes the specified capability and all of its derivatives.

- **K_DELDER** - deletes all derivatives of the specified capability

- **K_RESIZE** - resizes the specified object.

- **K_SHRINK** - shrinks an object to the size of its current contents

- **K_WAIT** - puts a subprocess to sleep until the specified wakeup time has been reached, or until a message arrives.

- **K_LOADCAP** - loads a view of a capability into the process address space. Specified parameters allow loading of large and small windows sizes, specified sizes at arbitrary or specified offsets in the address space.

- **K_UNLOADCAP** - unloads a capability from a process address space

- **K_CAPID** - returns information about the specified capability. The capability may be specified by an index into the table of loaded capabilities, offset in address space or capability.

- **K_MAKEPROC** - creates a process by creating a process object and initialising its state information, and then loading the created process object into the address space of the calling process.

- **K_SEND** - sends a message to a process loaded within the address space of the calling process. The message includes a non-negative amount of money.

- **K_RECV** - recovers a message from a subprocess message queue. The only parameter is the size of the match string used to filter messages.

- **K_EXTSEND** - sends a message to a specified process. The message includes a non-negative amount of money.

- **K_EXTREAD** - reads a specified number of bytes at a specified offset within a specified capability. This call is not intended for use in future versions of the kernel, as it is inefficient.

• **K_EXTWRITE** - writes a specified number of bytes from the process message area at a specified offset within a specified capability. This call is not intended for use in future versions of the kernel, as it is inefficient.

• **K_BANK** - transfers a specified amount of money from the calling process to the specified capability, or vice-versa.

• **K_RESTRICT** - reduces the rights of a specified capability, using masks for both urights and srights.

• **K_CAPSTAT** - returns status information about the specified capability and its associated object.

• **K_RENAME** - changes the password values of the specified capability and invalidates all derivatives of the specified capability.

• **K_MAKESUBP** - creates a new subprocess within the calling process. Specified parameters are the number, wakeup time and priority of the created subprocess.

• **K_DELSUBP** - destroys the specified subprocess.

• **K_LOADREG** - loads subprocess context information from the process message area into the subpn table entry of the specified subprocess. Required for subprocess restart.

• **K_SAVEREG** - copies subprocess context information from the subpn table entry of the specified subprocess into the process message area. Required for subprocess restart.

• **K_SETTRAP** - directs traps generated by a specified subprocess to a specified subprocess.

• **K_RECV_CLOSE** - recovers a message from a subprocess message queue and closes the subprocess mailbox.

- **K_ACCEPT_MAIL** - opens a mailbox for a specified subprocess, and sets a match string for filtering incoming messages.

- **K_CLOSE_BOX** - closes specified mailboxes

- **K_COPYOBJ** - creates a copy of a specified view of an object. Cannot be used to duplicate processes.

- **K_PEEK_PROC** - returns the state and wakeup time of the specified process.

- **K_SET_HEIR** - specifies the heir of the calling process. The heir process will receive the remaining money and a death message upon the destruction of the calling process.

1.3 The Nameserver Model

The Nameserver library [PRINGLE95] provides facilities which allow an ASCII format string to be bound to a capability. Bindings between strings and capabilities are held within an object termed a database.

The nameserver library provides a set based mechanism for grouping bindings. This mechanism is recursive, and allows sets to contain other sets. Bindings may be associated with a set or a capability. A database may contain sets and bindings to capabilities. A set may contain other sets. These are then termed included sets. When the nameserver searches a set for a binding, it will also search all included sets.

Nameserver sets may be arbitrarily organised either as sets connected in a graph, or as a directory hierarchy. The latter paradigm (which is a subset of the former) allows users to organise their bindings in a fashion analogous to established operating systems such as Unix.

The nameserver library provides a number of useful functions which may be exploited in the design of a stdio library. These functions are :

- **namec()** - searches a specified database for a specified binding. It returns a structure which describes the binding and its associated capability.

- **setCapName()** - binds a string to a specified capability in a specified database

- **deleteBinding** - deletes a specified binding from a database.

All nameserver functions require that a nameserver database capability be provided. Any operations carried out on bindings and sets are only done in relation to the database specified by that capability. However, the database may include bindings to sets in other databases.

Chapter 2 Survey and Critique

2.1 Introduction

The design of stream I/O services, stream IPC services and the implementation of libraries to provide these services have a long and interesting history. A range of important technical issues must be addressed in both design and implementation. These will be examined in the context of a number of historically important designs.

For the purpose of this discussion a stream is defined as a connection which reliably transfers bytes of data in sequential order. Data is written and read from a stream in a first-in first-out (FIFO) fashion. A stream is unidirectional, unless specified otherwise. Buffering of data is provided by the stream mechanism, and is not visible to the programmer.

2.2 An Overview of I/O Libraries and Stream Inter Process Communications

2.2.1 I/O Libraries

The function performed by an I/O library is that of providing a programmer with an abstracted interface to a computer system’s physical I/O devices. The interface seen by the programmer should hide as many of the idiosyncrasies of the hardware as is possible. This is necessary to minimise the effort required in moving programs between different types of computer system. Should I/O library programming interfaces for a given language differ across systems, moving a program between systems will incur a significant overhead in time and effort required to both change the program, and if necessary to debug the changes.

The evolution of portable I/O libraries in FORTRAN provides a good example [PLAUGER92]. Early implementations, such as FORTRAN II, required explicit specification of an output or input device, such as a tape drive or card reader. Later implementations, such as FORTRAN IV, provided Logical Unit Numbers (LUN) rather than explicit physical device identifiers. A programmer could therefore write programs which produced I/O operations against logical devices, rather than physical devices. I/O libraries linked in at run time provided the mapping between logical and physical I/O devices.

The development of Unix and the C Language during the 1970s [RITCHIE93] produced a important set of improvements to I/O programming interfaces. The most notable improvement was the adoption of an I/O programming model in which I/O devices were abstracted as files. Program I/O to storage devices and display and entry devices employed the same type of programming interface. The specific routines which handle device dependent I/O were embedded in the Unix kernel, and thus hidden from the programmer.

To support such a paradigm, Unix required a standard format for I/O transfer to and from an I/O library. This format is a transparent binary stream. Text format data (eg ASCII) is treated as lines which are separated by newline characters. The mapping between a text stream and the format required by the I/O device is typically performed by the Unix operating system.

The C language interface to all I/O devices utilised a *FILE* structure. Each process used a table of such structures, and the index to the table was designated a file descriptor. A programmer accessing I/O devices of any type in Unix uses a common set of system calls for *open*, *close*, *read*, *write* and other C specific functions, all of which operate on the *FILE* descriptor. The *stdio* library [RITCHIE93], developed to provide a portable I/O package, further extended this model to use a pointer to a *FILE* structure.

The Unix *stdio* programming paradigm treats all I/O interfaces as stream communication channels (the Unix *ioctl* interface is intended for device status operations). Implementations of Unix will provide type specific support for various types of buffering, all implementations however share the attribute of largely concealing the buffering mechanism from the programmer.

The migration of the C language from Unix to other operating systems led to the adoption of the ANSI X3.159-1989 C language standard [ANSI89], produced by the ANSI X3J11 committee. The X3.159-1989 standard was designed to provide a portable I/O interface for systems which did not provide the Unix I/O model. The *FILE* pointer paradigm was retained in the ANSI standard, but provisions were included to allow implementors to support I/O models specific to other operating systems. The transparent binary stream I/O model was not retained, and the ANSI model distinguishes between binary streams and text streams. This measure was required to support systems in which text streams are treated differently from binary streams. The low level I/O functions such as *open*, *close*, *read, write* and *lseek* were not included in the ANSI standard, because some of their functionality was considered by the ANSI committee to be too closely tied to Unix. These functions have been included in the IEEE 1003.1-1990 POSIX standard [IEEE90].

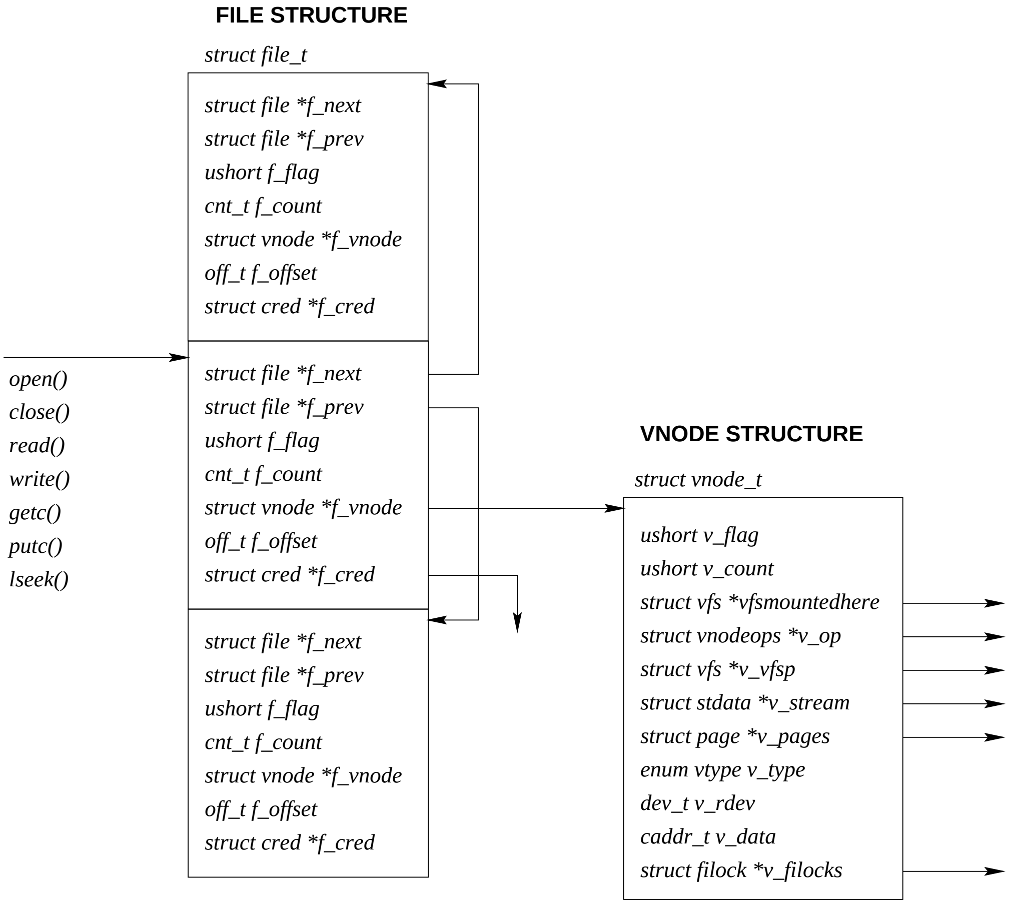


Figure 2.1 System V Release 4 Unix I/O Programming Model

2.2.2 Stream Inter Process Communications

Stream IPC facilities allow two processes to exchange information over a stream channel. This style of IPC is well suited to tasks such as piping the output from one program to the input of another. Programs such as text filters may be used in cascades, each filter in the cascade connected by a stream.

If a stream interface is used for I/O as well as IPC, the input to a cascade of processes may be an input device or file, and the output of the cascade may be directed to a file or an output device such as a terminal or printer.

Two areas are of principal interest in the implementation of a stream IPC mechanism. The programming interface is of interest because it will determine the amount of effort required to use the facility, and impose constraints upon how the facility may be used. The transport mechanism is of interest because it constrains the functionality, achievable throughput performance and the robustness of the stream facility.

2.3 The AT&T Unix Model

The evolution of AT&T Unix since the 1970s has seen the progressive refinement of the operating system's stream support for I/O devices and IPC. The current version of this product, System V Release 4 (SVR4), is widely licenced and used as the basis for Silicon Graphics Irix 5 and 6, Sun Microsystems Solaris 2, Hewlett Packard HP/UX 10, Novelle UnixWare and Santa Cruz Operation (SCO) Unix. All of these derivatives share the basic stream model of the original AT&T Unix System Laboratories SVR4. System V Unix was released in 1983, and the current System V Release 4 was released in 1988.

System V Unix employs the layered STREAMS interface [GOODHEART94] for supporting stream mode IPC and provision of an interface to stream oriented character devices, such as dumb terminals. In the STREAMS model , a single programming interface termed a stream head is used. A queue of linear buffers containing data and control messages is written to and read from the stream head. To accommodate filters and protocol modules, the STREAMS model employs a stack of stream modules. Each module will operate on a buffer, and then pass that buffer to the next stream module. In this fashion, the channel between a stream head and device driver may employ multiple stream modules to manipulate the contents of the data stream in the desired manner. An example would be the cooking of terminal I/O traffic.

SVR4 employs a virtual filesystem model. In this model the *FILE* (*file_t*) structure points to a virtual inode or *vnode* structure, which contains parameters specific to the stream interface used. Each process maintains a table of *file_t* structures, which are managed as a linked list of entries, using the *f_next* and *f_prev* pointers. The *f_flag* parameter is a bitmask of flags which describe the modes with which the interface was opened.

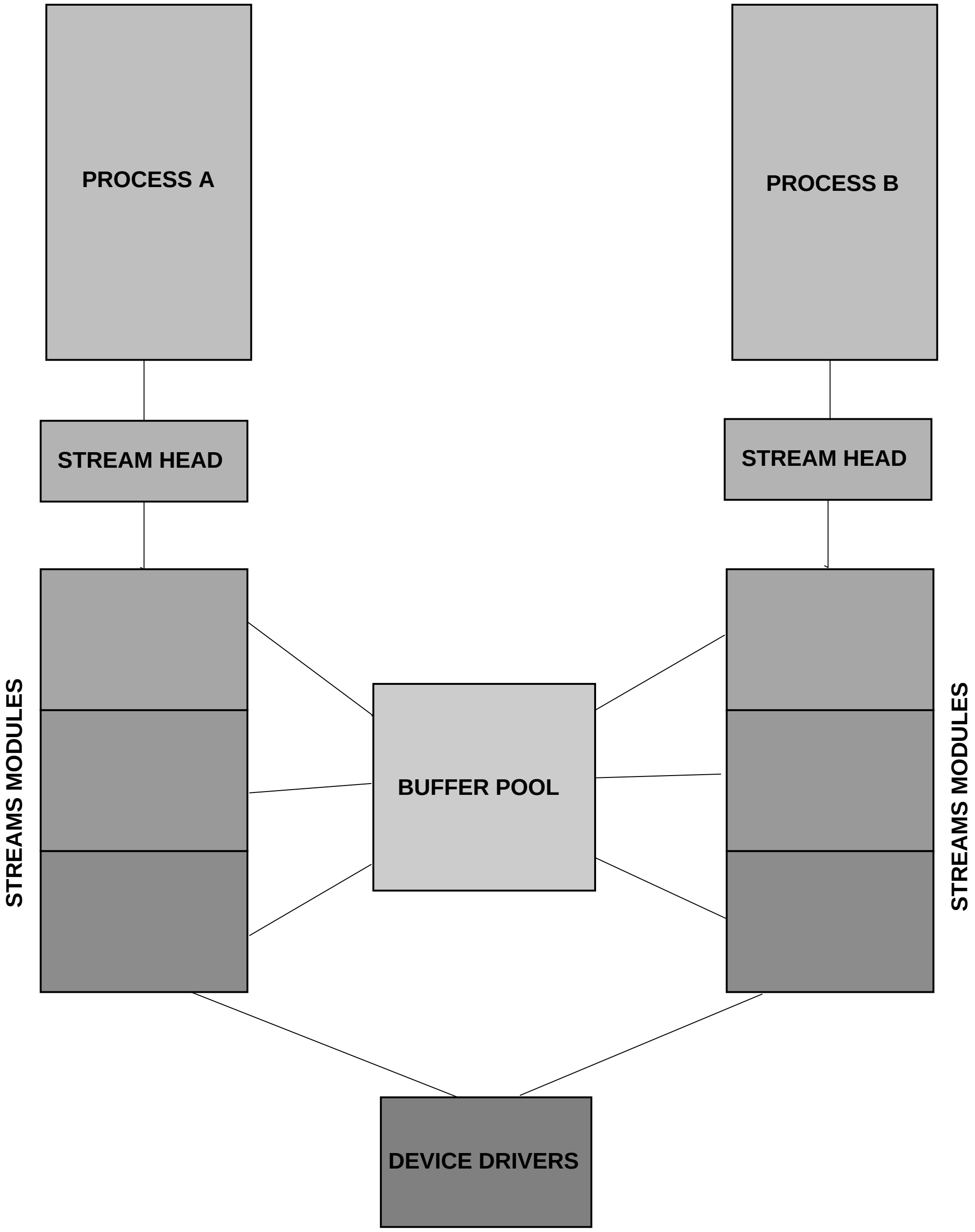


Fig.2.2 STREAMS Transport - Conceptual Model

The *f_count* parameter is a reference count which indicates the number of *FILE* pointers pointing to the *file_t* structure. This scheme is employed to prevent closure and deallocation of the file table entry by any other than the last *close* operation against the entry. The *f_offset* parameter is the character count offset into the file, and is adjusted by operations which *read*, *write* or *seek* on a file. The *f_cred* parameter contains the security credentials of the process which has opened the file.

The *vnode* structure is designed to accommodate a wide range of interfaces, such as physical storage devices with a range of installed filesystem types, symbolic links, block and character mode devices. STREAMS devices may also be accessed.

The *v_flag* parameter contains a bitmask which indicates the type of file opened, including pipes which are implemented as STREAMS devices. The *v_count* parameter is a reference count which provides a similar function to the *f_count* parameter. This parameter ensures that only the last process to close a file will deallocate the *vnode*. The *vfs_mountedhere* parameter is used only with directories which are mount points to a filesystem. The *v_op* points to a structure which contains an array of pointers to filesystem type specific operations. When the *vnode* is created upon opening a file or a pipe, the pointers are initialised to point at functions which are specific to the type of device to be used.

The *v_vfsp* pointer points to a structure which describes the filesystem type used. If the I/O is to a stream, then the *v_stream* pointer points to the STREAMS device used. The *v_pages* pointer points to a page list for the *vnode*. The *v_type* parameter specifies the type of device, for instance a regular file or pipe. The *v_rdev* parameter stores the major and minor device number for special files. The *v_data* filed points to the filesystem specific structure associated with the file, such as an *inode* used with a regular file. The *v_filocks* pointer points to a list of filock structures, used to implement file and record locking.

The *v_ops* pointer mechanism accommodates 37 filesystem specific operations, many of which are supported only for Unix file systems. These operations include reading, writing, opening, closing, fetching and setting of attributes, flushing (Unix sync) of buffers, and the locking and unlocking of files.

The intention of the designers of the SVR4 *file_t*/*vnode* model was to significantly simplify the implementation of I/O libraries installed on the operating system. A Unix I/O system call operating on a file or a stream will be pointed, via the *vnode*/*v_ops* pointer mechanism, to the specific function required. The addition of further filesystem types to the operating system can be readily accommodated by addition of instances of *vnodeops*, *vfs*, *vtype* and *v_data*.

The SVR4 Unix model provides both mandatory and advisory file locking mechanisms. Both mechanisms will allow the locking of records within the file, or the locking of the whole file. Process ID numbers are used to identify the processes which have placed the locks.

The buffering strategy used is device dependent. In all instances, a pool of linear buffers is used.

The STREAMS mechanism is used to implement pipes, named pipes (which appear in the filesystem) and stream IPC across networks. Where the SVR4 implementation provides a BSD socket IPC compatibility library, this is typically implemented in STREAMS.

2.4 The BSD Unix Model

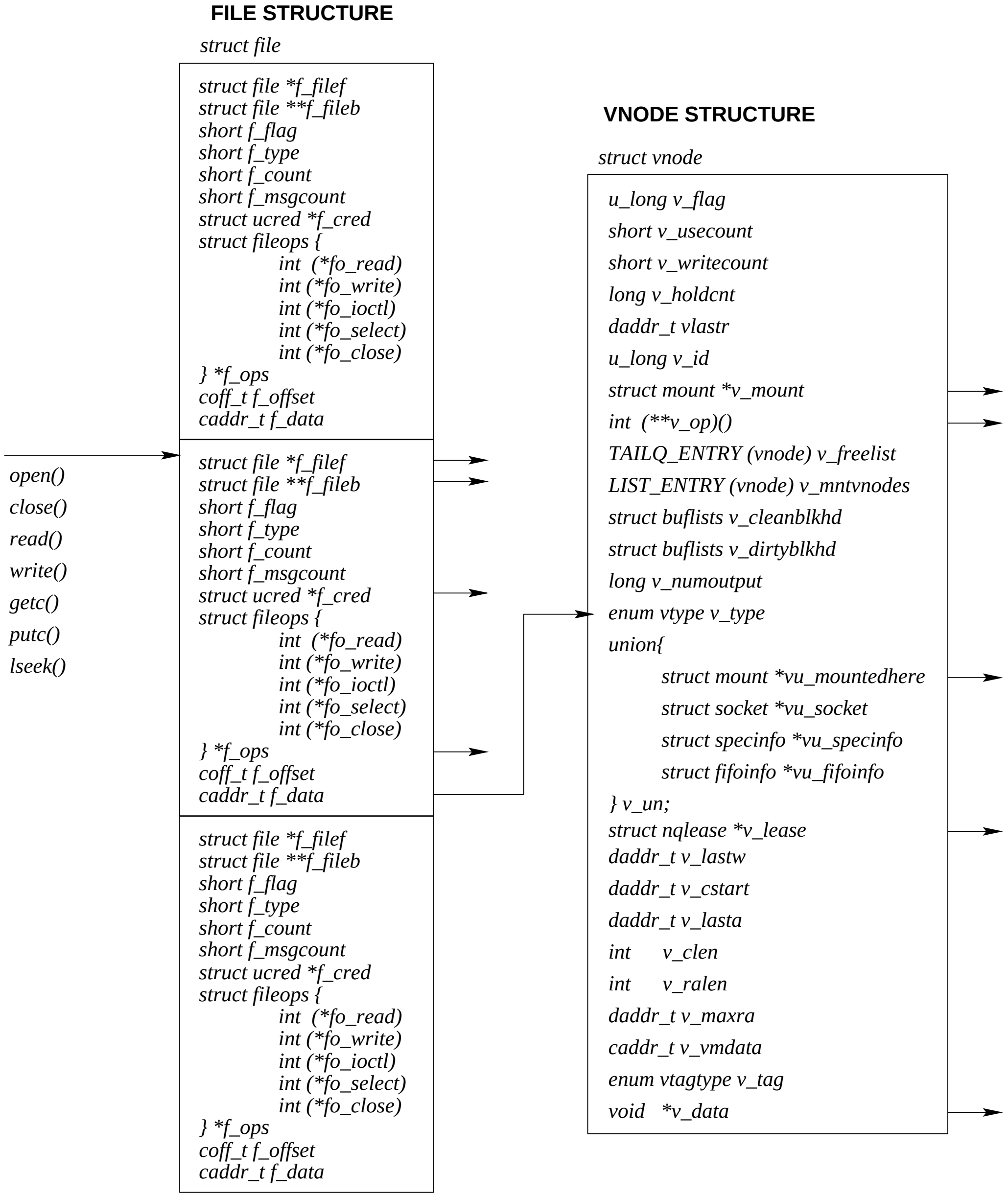
The most important derivative of AT&T Unix is BSD Unix, developed by the University of California at Berkeley. The first Berkeley Unix, designated 1BSD, was derived in 1977 from AT&T Unix Sixth Edition. This release was followed by 2BSD in 1978, and 3BSD in 1979. The most important release of Berkeley Unix was 4BSD, derived in 1979 from AT&T Unix Seventh Edition.

The 4BSD operating system became the basis for Sun Microsystems SunOS, which dominated the Unix workstation market during the 1980s. At the time of writing, versions of 4.2 and 4.3BSD are still widely used. The subsequent 4.4BSD, completed in 1993, is the basis for the commercial BSD/OS and the public domain FreeBSD 2.0. It is therefore reasonable to expect that BSD Unix will be used for some time yet, and thus its design is of more than historical interest.

The most recent release of 4BSD, 4.4BSD, employs a file table and vnode model analogous to that in SVR4 Unix. The implementation is however quite different.

The 4.4BSD system manages file structures by maintaining a list of pointers to individual *file* structures. The *file* structures exist on the process heap. Whenever a file is opened, address space is allocated on the heap and the *file* structure is initialised. On closing the file, the space is freed, and the list amended. The *f_filef* pointer points to the list entry for the file, and the *f_fileb* pointer points to the head of the list. A global variable is maintained for the number of open files [BSD44].

The *f_flag* field is a bitmask which contains the flags with which the file was opened. The *f_type* field describes the file type. The *f_count* field is the reference count which is identical in function to that in SVR4. The *f_msgcount* field is used in managing the state of socket connections. The *f_ucred* pointer points to a structure containing process credentials, and is analogous to that in SVR4. The *f_data* field is the address of the vnode structure used.



```
struct file *f_filef
struct file **f_fileb
short f_flag
short f_type
short f_count
short f_msgcount
struct ucred *f_cred
struct fileops {
            int  (*fo_read)
            int (*fo_write)
            int (*fo_ioctl)
            int (*fo_select)
            int (*fo_close)
} *f_ops
coff_t f_offset
caddr_t f_data
```

```
struct file *f_filef
struct file **f_fileb
short f_flag
short f_type
short f_count
short f_msgcount
struct ucred *f_cred
struct fileops {
            int  (*fo_read)
            int (*fo_write)
            int (*fo_ioctl)
            int (*fo_select)
            int (*fo_close)
} *f_ops
coff_t f_offset
caddr_t f_data
```

```
struct file *f_filef
struct file **f_fileb
short f_flag
short f_type
short f_count
short f_msgcount
struct ucred *f_cred
struct fileops {
            int  (*fo_read)
            int (*fo_write)
            int (*fo_ioctl)
            int (*fo_select)
            int (*fo_close)
} *f_ops
coff_t f_offset
caddr_t f_data
```

```
u_long v_flag
short v_usecount
short v_writecount
long v_holdcnt
daddr_t vlastr
u_long v_id
struct mount *v_mount
int  (**v_op)()
TAILQ_ENTRY (vnode) v_freelist
LIST_ENTRY (vnode) v_mntvnodes
struct buflists v_cleanblkhd
struct buflists v_dirtyblkhd
long v_numoutput
enum vtype v_type
union{
        struct mount *vu_mountedhere
        struct socket *vu_socket
        struct specinfo *vu_specinfo
        struct fifoinfo *vu_fifoinfo
} v_un;
struct nqlease *v_lease
daddr_t v_lastw
daddr_t v_cstart
daddr_t v_lasta
int     v_clen
int     v_ralen
daddr_t v_maxra
caddr_t v_vmdata
enum vtagtype v_tag
void   *v_data
```

Figure 2.3 4.4BSD Unix I/O Programming Model

The *f_offset* field is the file position pointer. Placement of the position pointer in the *file* structure allows multiple processes to operate upon the file pointer while not interfering with one another.

The most significant difference in the 4BSD file structure against the SVR4 design, is the inclusion of the *f_ops* field, which points to an array of pointers to file operation functions.

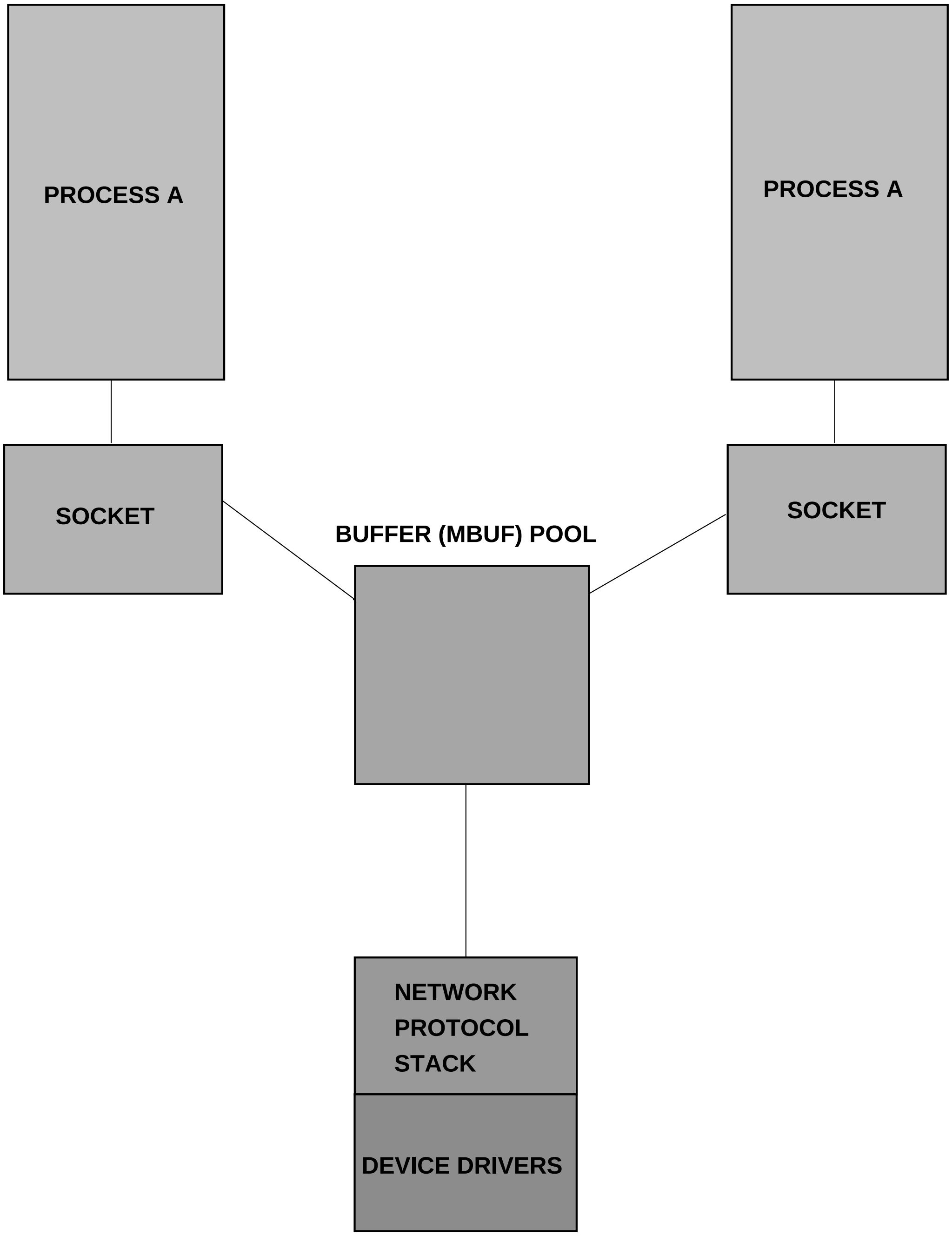


Fig 2.4 BSD Socket Transport - Conceptual Model

The intention of the 4BSD designers [LMKQ89] was to provide an object-oriented *file* structure. The *f_ops* structure is initialised at open time with type specific instances of read, write, ioctl, select and close functions. This design feature significantly simplifies the implementation of library functions which operate on the *file* structure [BSD44], [MCKUSICK94].

The vnode structure employed in 4.4BSD is significantly more complex than its equivalent in SVR4. This is largely due to the design requirement in 4.4BSD, to support additional filesystem types. As 4.4BSD was used initially as a research platform, changes were incorporated to accommodate the Log Structured Filesystem and the NQNFS protocol.

The fields which are common in purpose to the SVR4 design are the *v_flag* field, the *v_usecount* field, the *v_type* field, the *v_mount* structure, the *v_ops* pointer, and the *v_data* pointer. The *v_un* field is a pointer which is initialised to type specific pointers to state variable structures. The types included in the *v_un* field are regular files, sockets, special files and fifos.

Stream IPC in 4.4BSD Unix is implemented using the BSD socket mechanism [LFJLMP]. BSD releases prior to 4.2BSD implemented pipes in the filesystem, all subsequent releases implement pipes using the socket mechanism.

The socket mechanism was designed to provide transparent stream and datagram oriented IPC between processes running both locally and remotely over a network. The intent of the designers was to provide a programming interface which was independent of the underlying communications channel, and common to both local and remote operations.

The programming interface used was intentionally different from the interface for filesystem objects. The designers chose not to overload the established *open* system call with additional functionality to support the socket scheme. This was intended to improve the portability of programs [LMKQ89]. Opening a socket connection requires the creation of the socket with a *socket* system call, binding a socket address to the socket with a *bind* call and initiating the connection with a *connect* call. The socket call returns an index into the process file table, termed a file descriptor in Unix. Once the connection is open, the programmer may use both socket specific calls or the established Unix *read* and *write* system calls. The 4.4BSD *pipe* system call implementation will open a read socket and a write socket to provide a bidirectional channel [BSD44].

The socket transport mechanism is layered over the network protocol drivers. Traffic to remote processes must be handled by the drivers. Traffic local to the host bypasses the network protocol stacks. Management of the pool of *mbuf* linear buffers is performed by the networking software. This approach was employed to simplify the design by constraining the function of the socket to that of providing the programming interface alone.

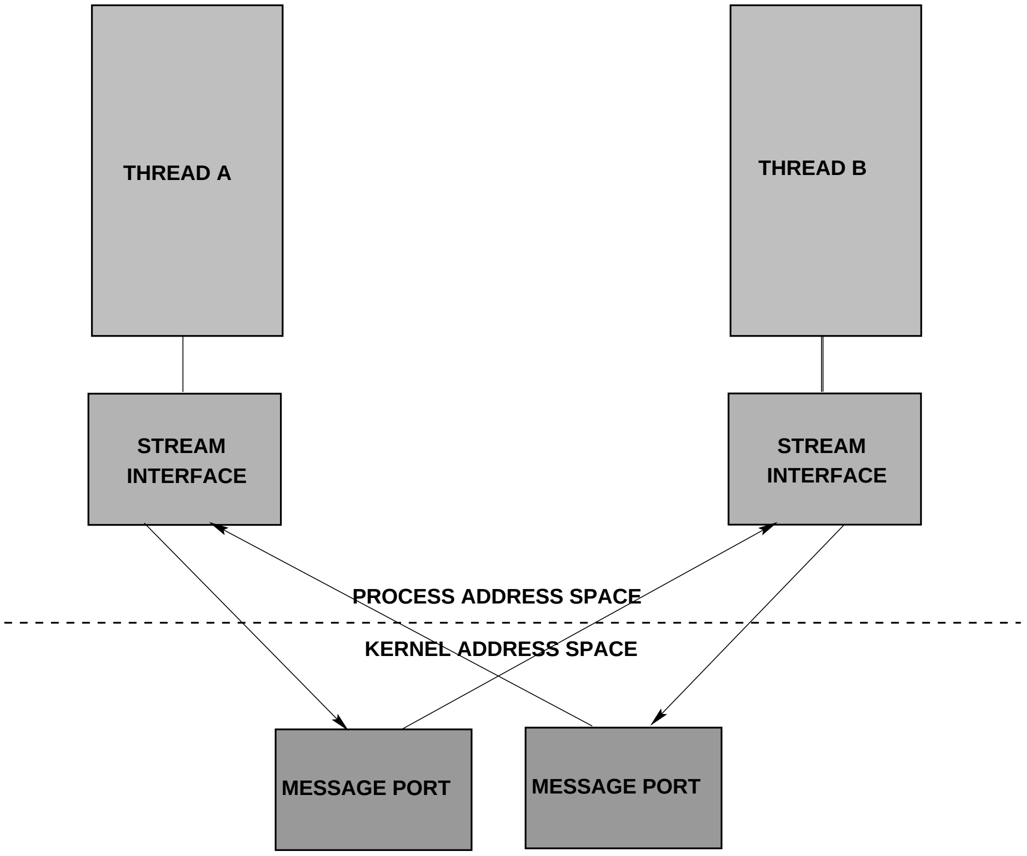


Fig 2.5 Mach 3.0 Stream Communications - Conceptual Model

2.5 The Mach Model

The Mach operating system was developed by Carnegie Mellon University as a research platform for the purpose of investigating issues in microkernel design. The Open Software Foundation OSF/1 operating system utilises the Mach 2.5 kernel and is the Unix system supplied by Digital Equipment Corporation with their Alpha architecture systems. The Mach 3.0 kernel was released in 1989, and provided some improvements to the IPC facilities used in the earlier Mach 2.5 kernel [DRAVES91]. Mach is of interest in the context of stream oriented IPC because it implements a stream mechanism which is layered over a message passing mechanism.

The Mach 3.0 kernel IPC facility is based upon a message passing model, in which access to message communication ports is provided by capabilities. The interface to the message passing mechanism is designed to support Remote Procedure Calls, object-oriented client server operations, and stream communications between processes.

The central element in the Mach message mechanism is the port, which is implemented as a message queue in the address space of the kernel [OSF93]. The kernel enforces security of the port by allowing messages to be passed only if the sending party has the capability for the connection. Messages received by a thread within a process are copied into the address space of the process. The size of the message passed through a port is arbitrary, and may be as large as the address space of the process.

In the Mach 3.0 stream implementation, messages from any given thread are delivered in order, which satisfies one of the basic requirements for stream communication. Flow control of stream traffic is implemented by limiting permissible queue lengths. Threads which require different amounts of buffering can alter the queue length associated with a port. Operations which block due to a full or empty queue can be set to restart when the queue is able to accept or provide further messages. If non-blocking behaviour is required, the kernel can notify the communicating thread with a message indicating that the queue is ready again.

The OSF/1 operating system provides access to the Mach messaging mechanism, but implements stream IPC with BSD style sockets. A separate socket implementation is used.

2.6 The Alloc Stream Model

The Alloc Stream Facility (ASF) was developed in the early 1990s by researchers at the University of Toronto [KSU94] as a means of improving the performance of Unix I/O libraries. The ASF model is of interest as it demonstrates how appropriate use of memory mapping techniques and buffering strategy can be employed to improve throughput performance at the library interface to the operating system.

The starting point in any discussion of the ASF facility is the behaviour of established Unix I/O libraries. A typical Unix C language *stdio* library implementation will map I/O library calls such as *fopen, fclose, fread, fwrite* and *fseek* into the corresponding Unix systems calls, *open, close, read, write* and *lseek*. This mapping will typically involve the substitution and addition where necessary of arguments, and in read or write operations, the buffering of data within the library.

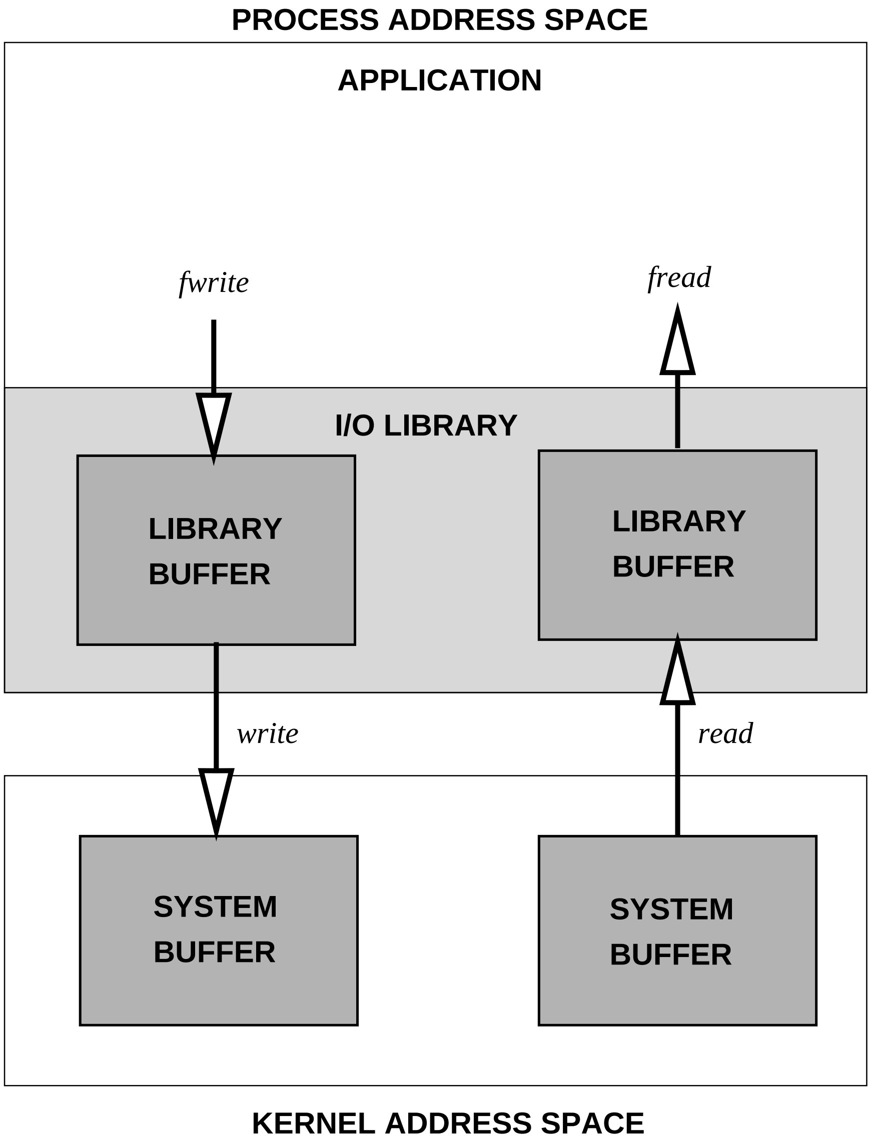


Fig 2.6 Generic Unix stdio Library Implementation

Because the *stdio* library is interfaced to the Unix kernel through system calls, all transfers of data between the library and kernel incur an overhead in copying data, and an overhead in executing the system call. Both of these overheads can be significant in a modern Unix system, which typically employs a RISC architecture processor.

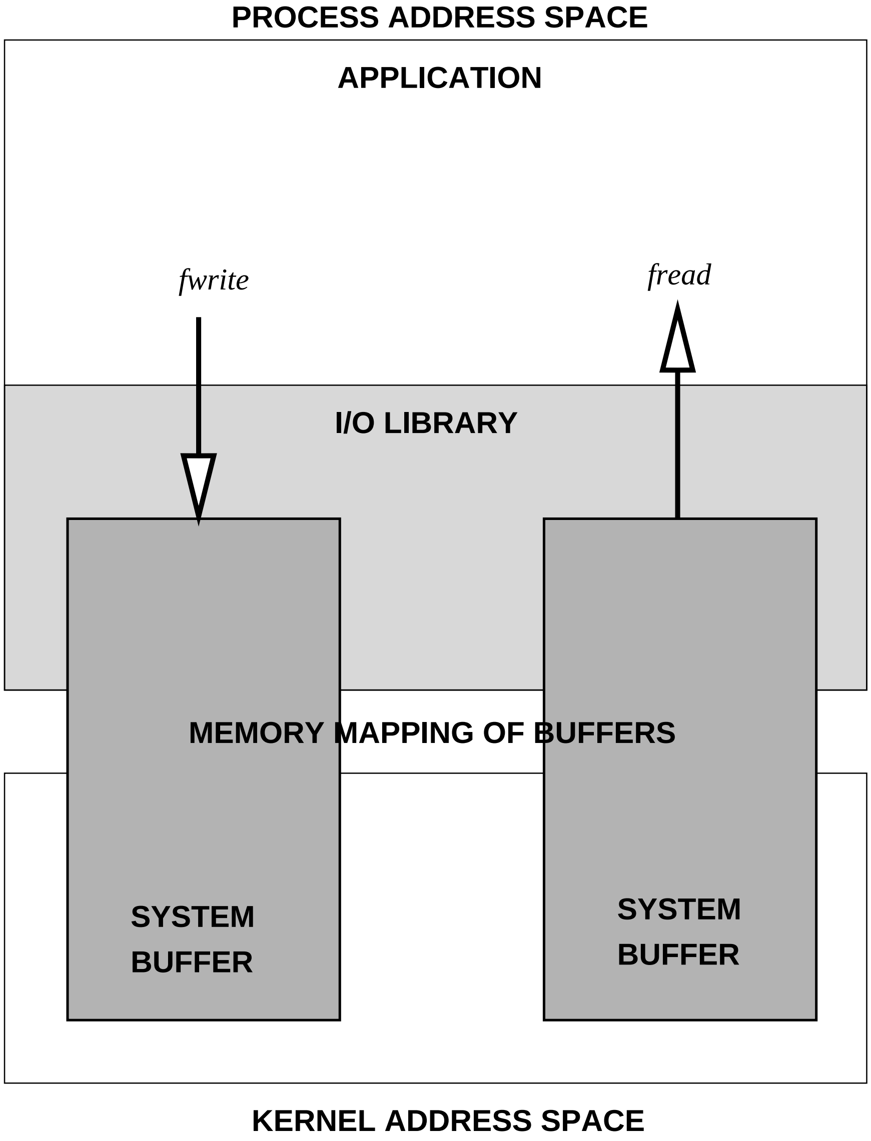


Fig 2.7 Unix stdio Library Implementation Using ASF Model

System calls require a context switch which will result in the saving of the process state. This can be an expensive operation where a large set of registers must be saved. This problem is exacerbated where the application executes frequent read and write operations containing small amounts of data, as the overhead is incurred for each operation.

The performance loss due repeated copy operations can be significant. In the conventional Unix model, each operation will result in two copy operations, as the data is first copied from an application into a buffer in the library, and then copied from the library to a kernel buffer.

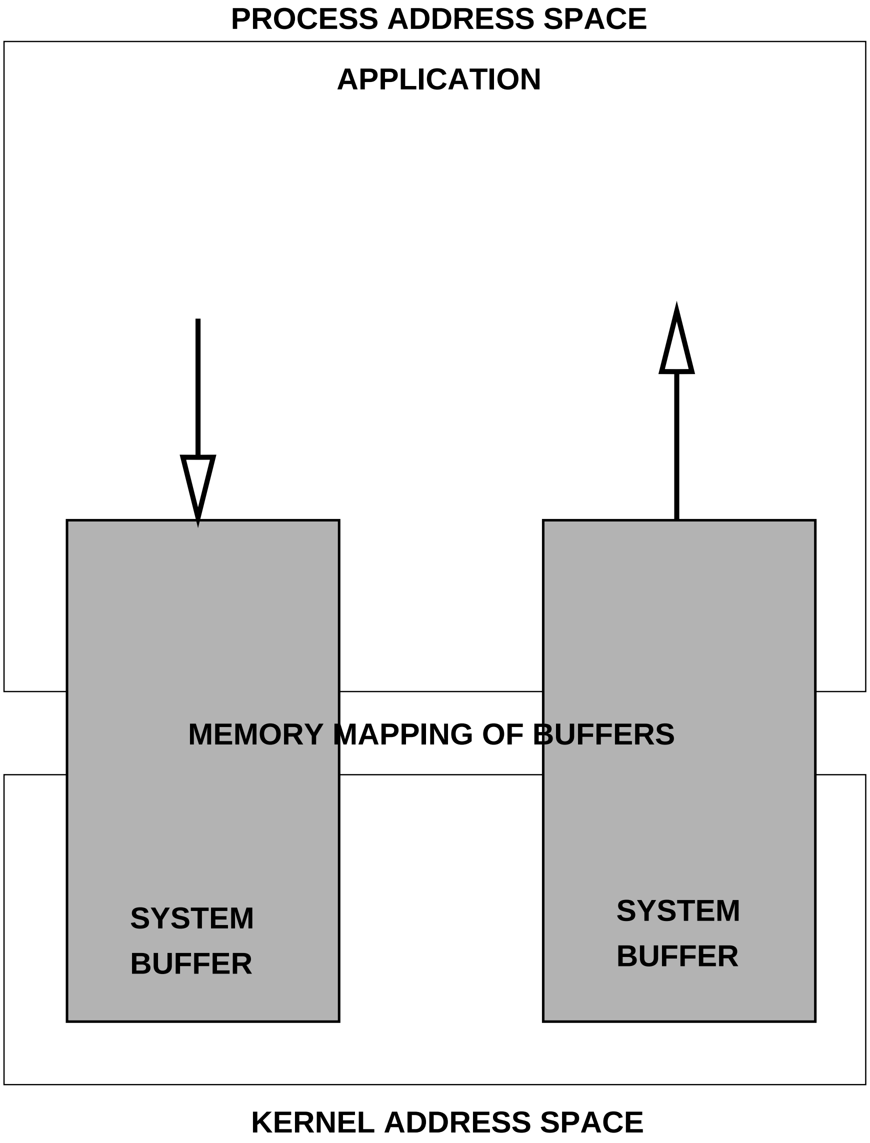


Fig 2.7 Application I/O Using the ASF ASI Interface

The ASF model is built on the idea of using the platform’s virtual memory system to map the kernel buffers into the address space of the user process. The library or application then need only copy the data once, to or from the memory mapped kernel buffer.

The ASF design provides a user level interface, termed the Alloc Stream Interface (ASI). This interface may be used directly by an application programmer, or by a library implementor. Buffer allocation and mapping by the ASI is performed by an *salloc* call, buffer deallocation and unmapping by the *sfree* call. These calls are analogous to malloc and free operations.

The ASF may be exploited in two ways. The first is the reimplementation of existing I/O libraries. The reimplemented library interfaces to the kernel buffers using the ASI interface. The overhead incurred is that of the single copy between the application and the buffer in the library. This strategy allows existing application program code to be retained without modification.

The second method is for the application programmer to rewrite the application, replacing the previous *stdio* library calls with memcpy calls directly to and from the mapped kernel buffers. While this method incurs a software development time overhead, it removes the execution time overhead of the library call.

The authors of the ASF design report useful improvements to the performance of standard Unix applications linked to a reimplemented *stdio* library, and significantly improved performance to Unix applications reimplemented to use the ASI interface.

2.7 Other Operating Systems

The Multics system [ORGANICK] is not discussed in detail as it does not employ a stream IPC mechanism, and historically precedes the emergence of Unix and the C language. The I/O interface employed in Multics provided a mechanism for binding device names to physical devices.

The CHORUS system [ROZIER91] is also not discussed in detail, as stream oriented communications are not provided in its kernel. In the CHORUS system IPC is implemented via a message passing mechanism which is similar to that in Mach.

Chapter 3 The Stream IPC and File Access Model

3.1 Introduction

The design of the stream IPC and file access mechanism is central to the function of the Walnut I/O library. This mechanism is used to provide a programmer with a transparent byte oriented communications channel between a pair of processes, or with a stream interface to a file object.

A stream IPC channel may be used for piping the output of one program into another, as well as to provide a clean interface to a process which provides a common service, such as support for a hardware device.

The Walnut architecture provides a user process with two types of programming interface to an I/O device. Disk storage devices are accessed directly through the virtual memory mechanism. Devices not accessed through the virtual memory mechanism, such as floppy disk drives, tape drives or stream oriented devices such as dumb terminals or printers, use a different access model.

Two mechanisms may be used for accessing such a device. One mechanism is that of mapping a page containing device control and status registers into the address space of a process. A process which knows the capability to this page can then sense and manipulate the state of the device by reading and writing device registers respectively. This mechanism is applicable to interrupt and non-interrupt driven devices.

Interrupt driven devices must employ a further mechanism. This mechanism is an interrupt service routine which is linked with, but otherwise largely independent of the kernel. The interrupt service routine will directly access device registers. Communication between an interrupt service routine and process is accomplished by sharing a page at a fixed physical address in the address space of the interrupt service. A process which knows the capability to this page can then interact with the interrupt service routine.

This model therefore allows a process to first initialise a device, and then read and write data through a shared buffer and interrupt service. While any process knowing capabilities to a device can access that device, in practice the only process to do so will be a device manager or server process.

This is analogous to the Unix paradigm in that a driver is split into an "upper part" and a "lower part", the latter being the interrupt services [LMKQ89]. Unlike Unix, where the "upper part" of the driver is embedded in the kernel, the "upper part" of a Walnut device driver typically runs as part of a device manager process.

A user process may access the device manager process through a shared page, a message, or a stream connection. Access may occur only if it has the capability to access the shared page, send a message to or to make a stream connection to the device manager process.

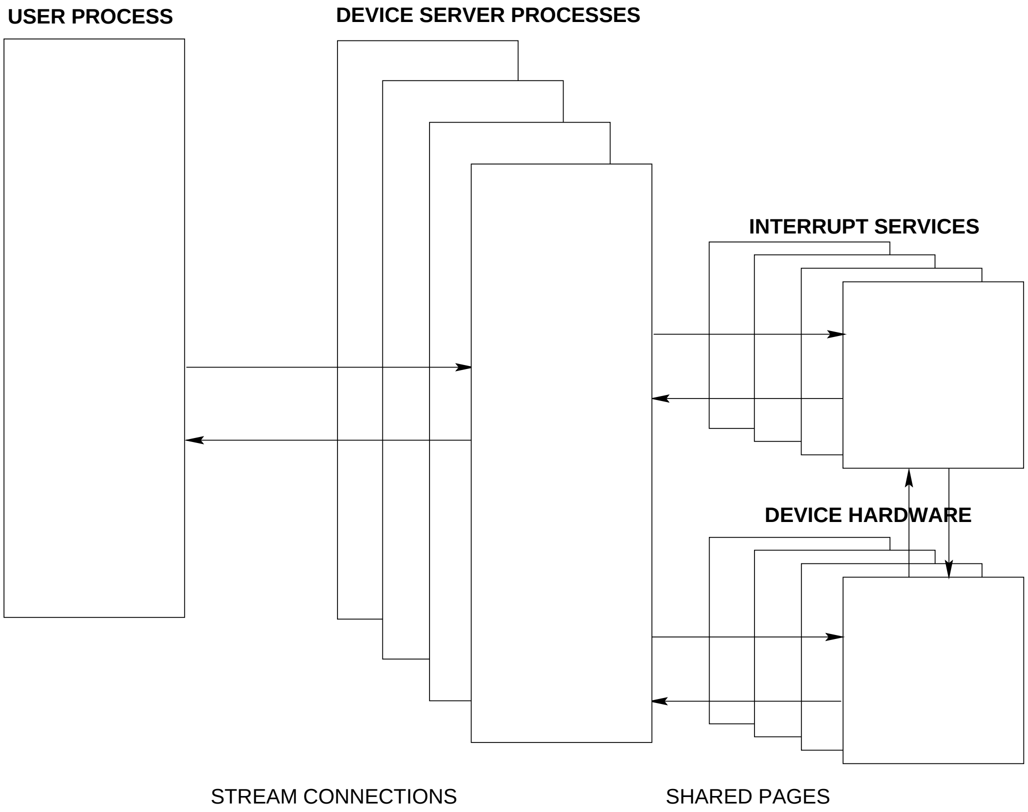


Figure 3.1 Walnut Library I/O Interface Model

The device manager process is typically the only entity within the system which has direct access to the stream device. This design strategy provides a uniform interface to all devices, and a measure of security in accessing the devices as a valid capability to connect to the device manager process must be known.

The implementation of device driver functions in processes rather than inside the kernel provides a number of advantages.

- the design of the kernel is simplified

- robustness is improved, as a problem in the lower part of the device driver or hardware will not impair the operation of the kernel. A problem in the upper part of the device driver can impair only the operation of the device, or those processes using the device through the device manager process.

- device drivers may be added, removed or altered without changing the design of the kernel

- user processes may be provided with a uniform programming interface to device manager processes.

- in multiprocessing systems, dedicated I/O processors share a common type of user programming interface with simpler directly accessed devices

The Walnut *GLui* console manager is implemented using this model. *GLui* is a virtual terminal manager for a PC frame buffer console and keyboard, and emulates a number of dumb terminals. The Walnut floppy drive manager also employs the device manager model, although the existing implementation uses a shared page rather than a stream connection to transfer data to and from a user process.

Because the stream IPC mechanism becomes the means of supporting much of the I/O as well as piping between processes, throughput performance and robustness are important issues.

To produce a simple and robust design for the stream IPC and file access mechanism, careful consideration had to be given to the data structures used, to allow these same data structures to be used for stream IPC and library I/O operations against file objects.

The need to provide a substantial degree of compliance with the ANSI C standard dictated that some established conventions be retained, and these are detailed more closely in 3.2.1

Whilst some consideration was given to including the POSIX 1003.1 standard into the design constraints, this was subsequently rejected as too many aspects of the POSIX model were implicitly tied to Unix and thus irrelevant in the provision of *stdio* functions. Should a port of Unix to the Walnut be intended, analogous to that in Mach or Chorus, then the development of a POSIX library would be justified [POSE93]. The same argument may be applied to a BSD socket programming interface.

3.2 Walnut I/O Library Design

3.2.1 Requirements

A number of basic design constraints were set, to provide both compliance with the language environment used, as well as to provide objective criteria for assessing the usefulness of various design alternatives. An important aspect of this process was to determine which attributes of the established ANSI programming interface model were relevant for the Walnut environment, and which were not.

The process of defining requirements was iterative. Starting from a series of basic requirements, these were repeatedly refined by comparing constraints implicit in both the ANSI standard and the Walnut kernel. This strategy was adopted to ensure that the final set of design requirements provided the best possible fit.

Importantly, as development of the software proceeded, further constraints and technical issues became evident, and these in turn were added into the final functional requirement set for the library. Some care was taken to ensure that the design was clearly separated from the implementation. This was to ensure that the library can be ported with a minimum of effort, when the Walnut kernel is ported to different machine architectures.

The initial set of design requirements were defined as:

- the mechanism employed would exploit the Walnut kernel’s ability to map an object into the address space of a process.

- the stream transport mechanism would be implemented with a circular buffer scheme.

• a file structure would be used to preserve the ANSI C programming interface structure and syntax.

• within a file object, the offset pointer can be moved to an arbitrary byte address within the capability.

• operations must be available to open and close file and stream objects.

• the syntax for open, close, seek, read and write calls should be compliant with the ANSI C programming interface.

• support would be provided for the Walnut nameserver library, to allow the use of name and path based object addressing syntax as is used in the ANSI standard.

• support would be provided for both blocking and non-blocking behaviour when reading and writing both file and stream objects.

• a set of additional function calls would be added, where appropriate, to provide functionality unique to the Walnut. These would where possible retain the style of programming interface used in the ANSI C standard and the POSIX 1003.1 standard.

• ANSI compliance would be mandatory, where such compliance did not incur a significant development time overhead.

The requirement to preserve the ANSI and POSIX 1003.1 style of programming interface has some important implications. Both of these models are derived from the Unix paradigm, in which the *FILE* structures associated with open files are managed as a table [Sections 2.3, 2.4]. Indeed, the argument used to identify an open file in the POSIX environment is an integer index into this table. As a consequence of this, the Walnut *FILE* structure would also have to be managed as a table to ensure that the same behaviour is provided.

An important and implicit objective, as is evident from the initial requirements, was to provide a programmer familiar with the ANSI C and POSIX 1003.1 environments with a comfortable programming environment.

This would minimise the effort required to port existing software from other platforms to the Walnut, as well as enabling a programmer to become proficient in the use of the library as quickly as possible.

3.2.2 Design Considerations

Whilst a number of design constraints were defined, these did not encompass all considerations which would be relevant to the implementation. As such, design considerations are discussed separately. Many of the design considerations were apparent from the outset. Some became apparent during the implementation of the library.

• **locking vs lock free buffer management** - the *stdio* library is intended for use with single processor as well as future multiprocessor implementations of the Walnut kernel [CP94]. Robustness in a multiprocessor environment is important, and the use of locking mechanisms will increase complexity should robustness be required.

• **the use of common vs unique structure types for file objects and stream objects** - because file objects and stream objects have different functional requirements, a library implementation may use either a common data structure type for the object, or unique data structure types. Whilst the use of different data structure types would simplify the data structures, it would increase the complexity of a number of the library functions.

• **distinguishing file objects from stream objects** - two strategies were considered for this purpose. The first strategy is to define separate object types in the kernel, using the parameter block type field. The second strategy is to use a magic number at the beginning of the object.

• **programming interface for opening and closing streams** - two approaches were considered, these being the use of unique library calls for file and stream open operations, or the overloading of a single function call for both purposes.

• **mapping ANSI file behaviour on to an object** - a memory mapped object has a number of properties which differ from those of an opened file object, as defined in the ANSI C standard. While memory mapping is a more powerful approach than treating the object as a sequentially accessed file, compliance with the standard dictated usage of the latter. Important issues were how much of the object is to be mapped in, and how the state of the object and the position of the file pointer are managed.

- **process environment** - the Walnut kernel provides a process with an address space which contains a number of defined variables. The use of the library however required the addition of a number of additional variables and structures, to support its function. The foremost of these was a table of *FILE* structs. In addition, it was found that other functionality such as error handling was required.

- **file structure strategies** - the design of the *FILE* structure required reconciliation of a number of ANSI standard related constraints, as well as the functional requirements of the stream and file object handling.

- **nameserver issues** - the nameserver library provided a means of binding names to capabilities. The stream IPC mechanism would need to exploit where appropriate, nameserver functions, to provide a good fit with the ANSI programming interface model.

- **performance issues** - the transport used in the stream IPC mechanism was to be designed to minimise the computational overhead of transmitting data through the channel, thus maximising throughput performance. While the use of a circular buffer scheme in a shared object was a design requirement, the justification of this requirement lies in a performance advantage. Furthermore, other library operations were to use the minimal number of system calls required.

- **security issues** - the stream IPC mechanism should not degrade the security of access inherent in the Walnut kernel. The two processes communicating through the stream IPC channel should not be given access to each other’s address space, other than to buffer related structures shared for purposes of data transmission.

3.2.3 Design Rationale

The requirement to fit the ANSI programming interface to the Walnut paradigm suggested a simple two tier model. A file structure analogous to that in ANSI and Unix implementations would provide the entry point into the mechanism. A stream structure would contain pointers into the view of the stream buffer object or into the view of the file object.

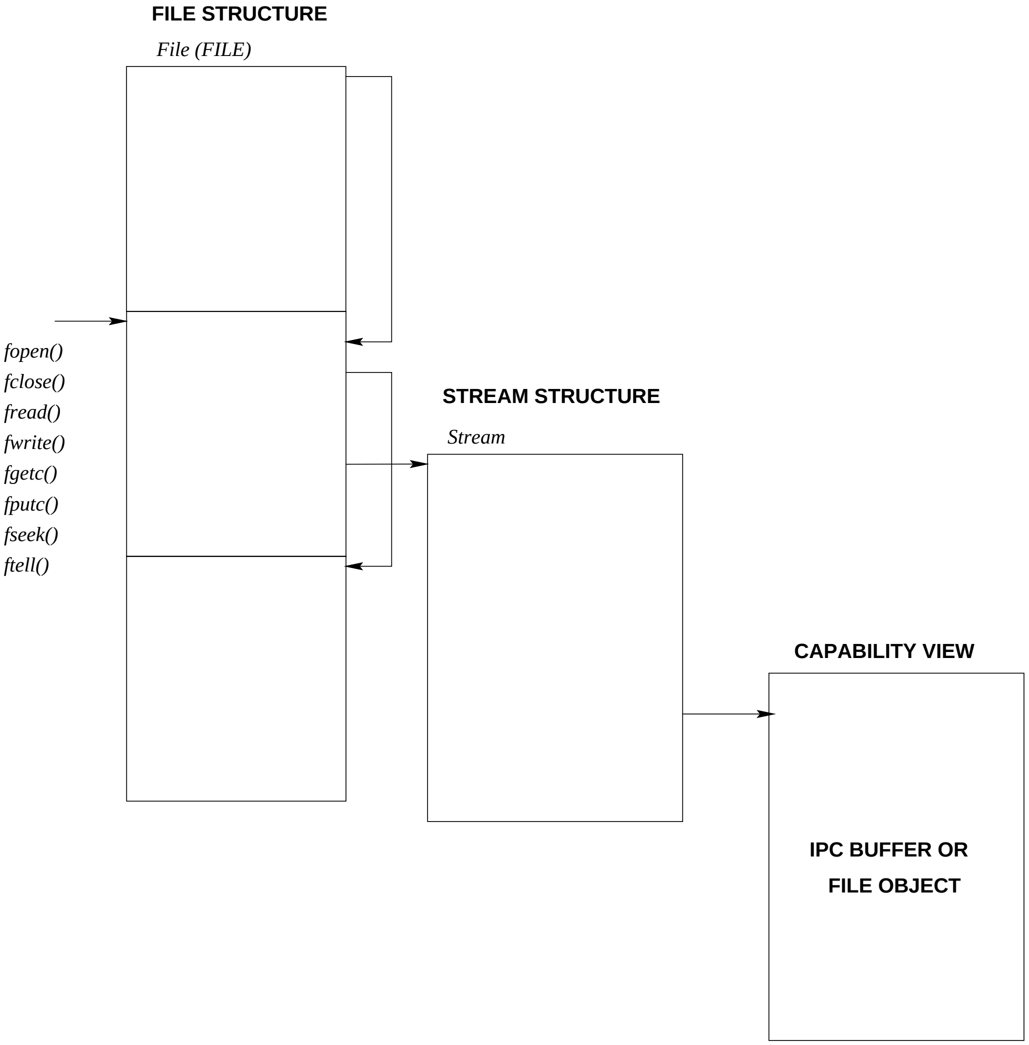


Figure 3.2 Walnut Library I/O Programming Model

3.2.3.1 Buffer and Stream Structure Design

Several options were considered for the design of the Stream structure and buffer format. Initially, an arrangement with three separate objects was proposed. The first object contained the circular data buffer. The second object contained buffer parameters, the in-index (write index) and out of band (OOB) message passing parameters, which were to be a capability and state flags. The third object contained the out-index (read index) and out of band (OOB) message passing parameters, which were also to be a capability and state flags.

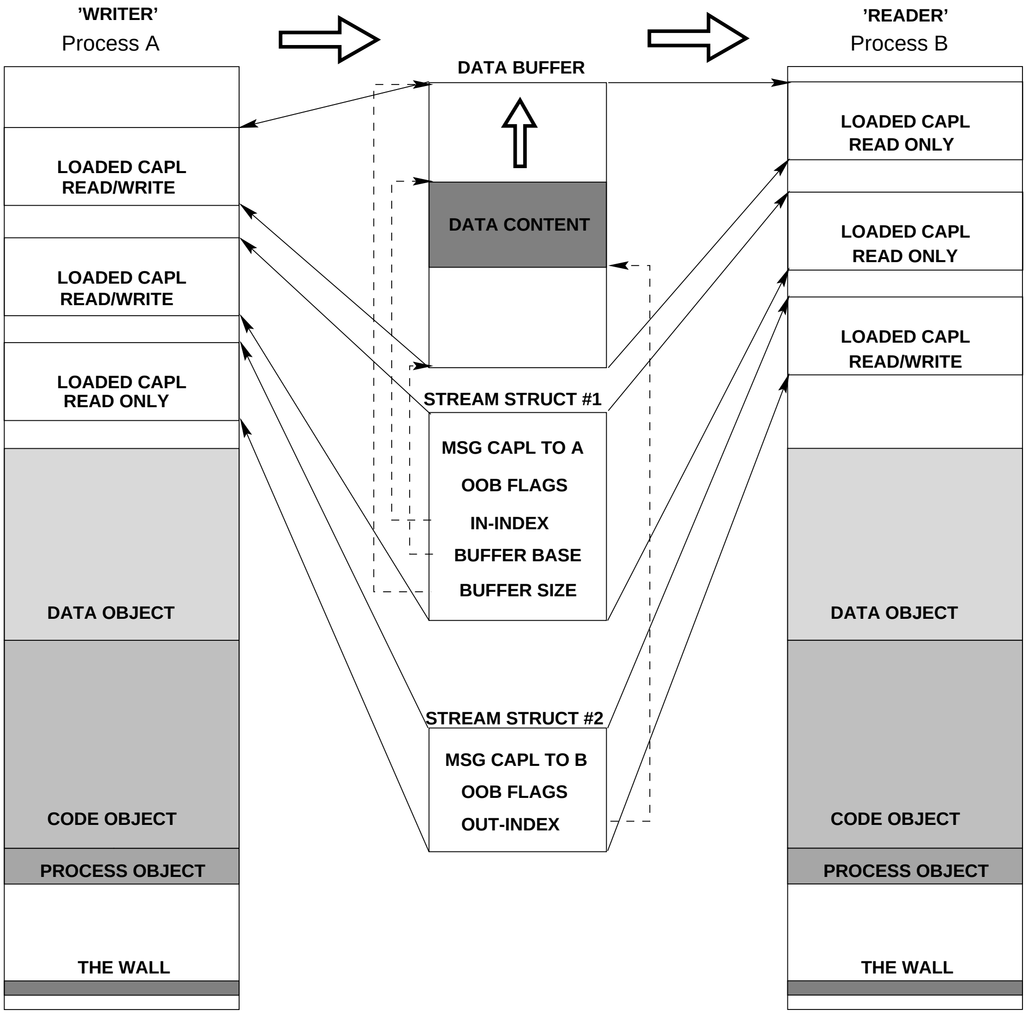


Figure 3.3 Walnut Stream Design - Initial Proposal

To provide protection from ill behaved subprocesses which could corrupt the contents of the objects, the reading process would map in the buffer and second object in read only mode. The writing process would map in the third object in read only mode.

This arrangement evolved. The first major change was the decision to collapse the three objects into one, as the overheads of managing three objects for a single stream connection could not be justified unless the protection issue was to be a significant problem.

The capabilities and the OOB message flag fields would provide for not only conventional blocking and non-blocking modes of operation, but also for a "wakeup" mode of operation. In such a mode of operation, a process which would otherwise block due to a buffer full or empty condition, would sleep until "woken" by a message from its peer in the connection.

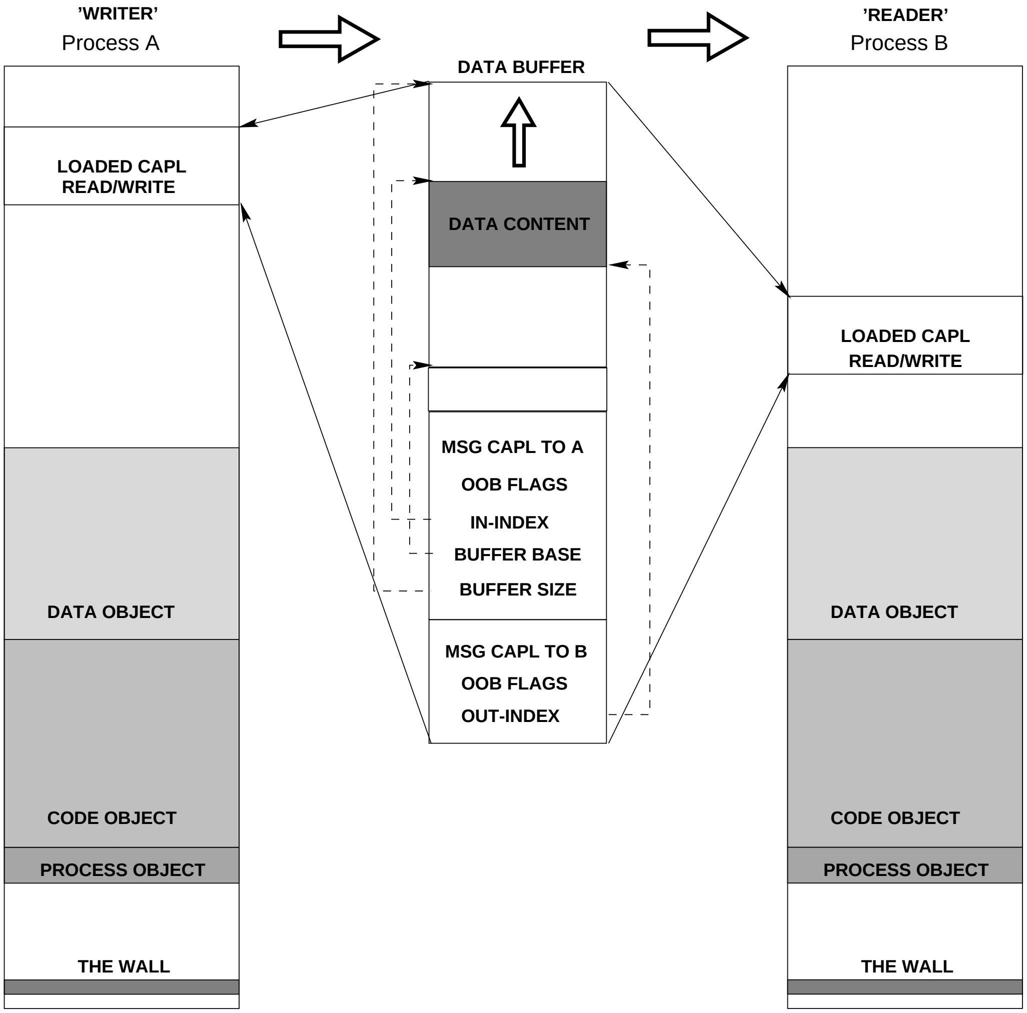


Figure 3.4 Walnut Stream Design - Single Capability Model

The circular stream data buffer was designed for simplicity and to provide the best achievable throughput performance. A scheme was devised, whereby the reading and writing processes need only access their respective index values into the buffer, and a tripwire value.

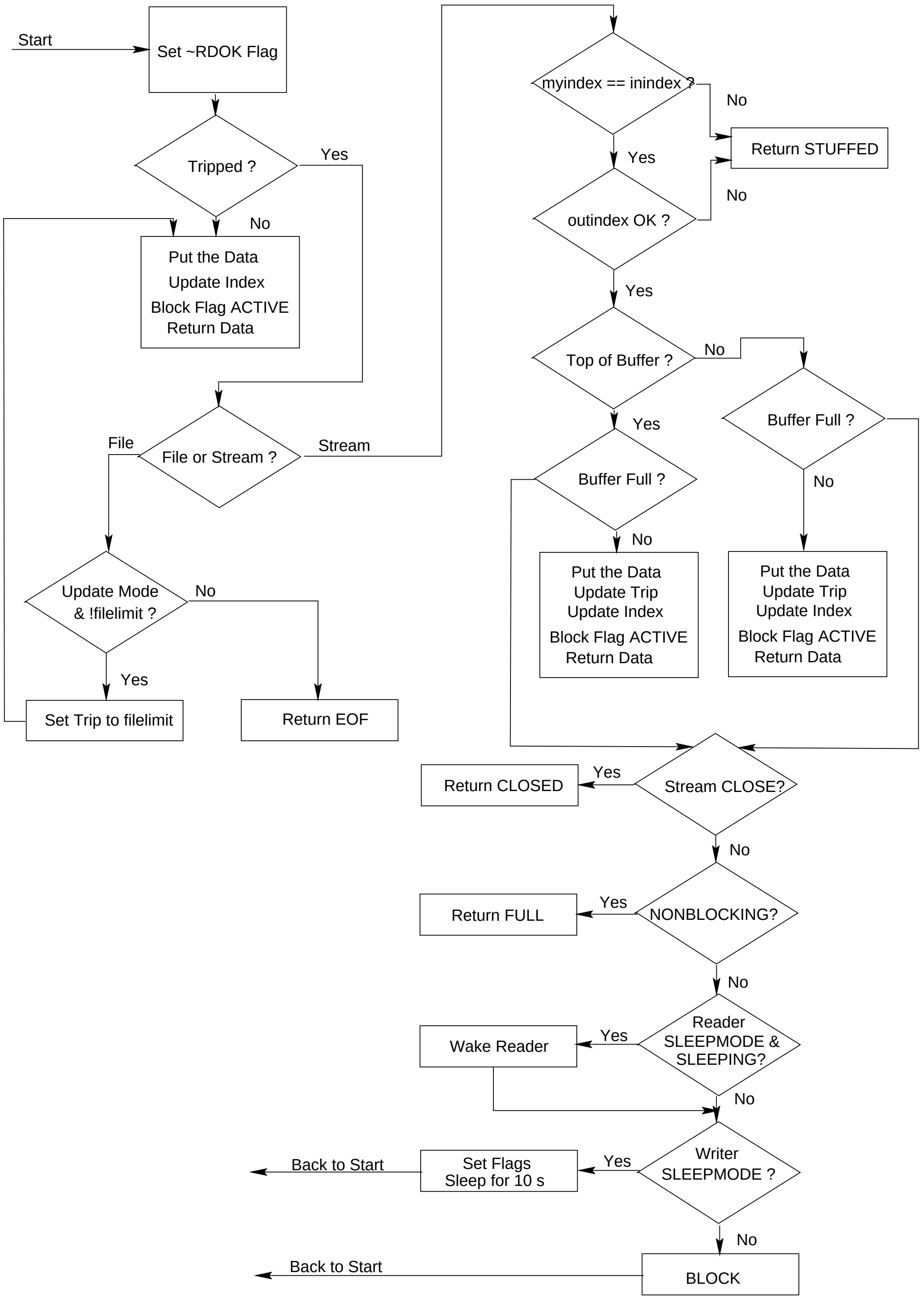


Figure 3.5 Walnut Stream Design - Simplified fputc Algorithm

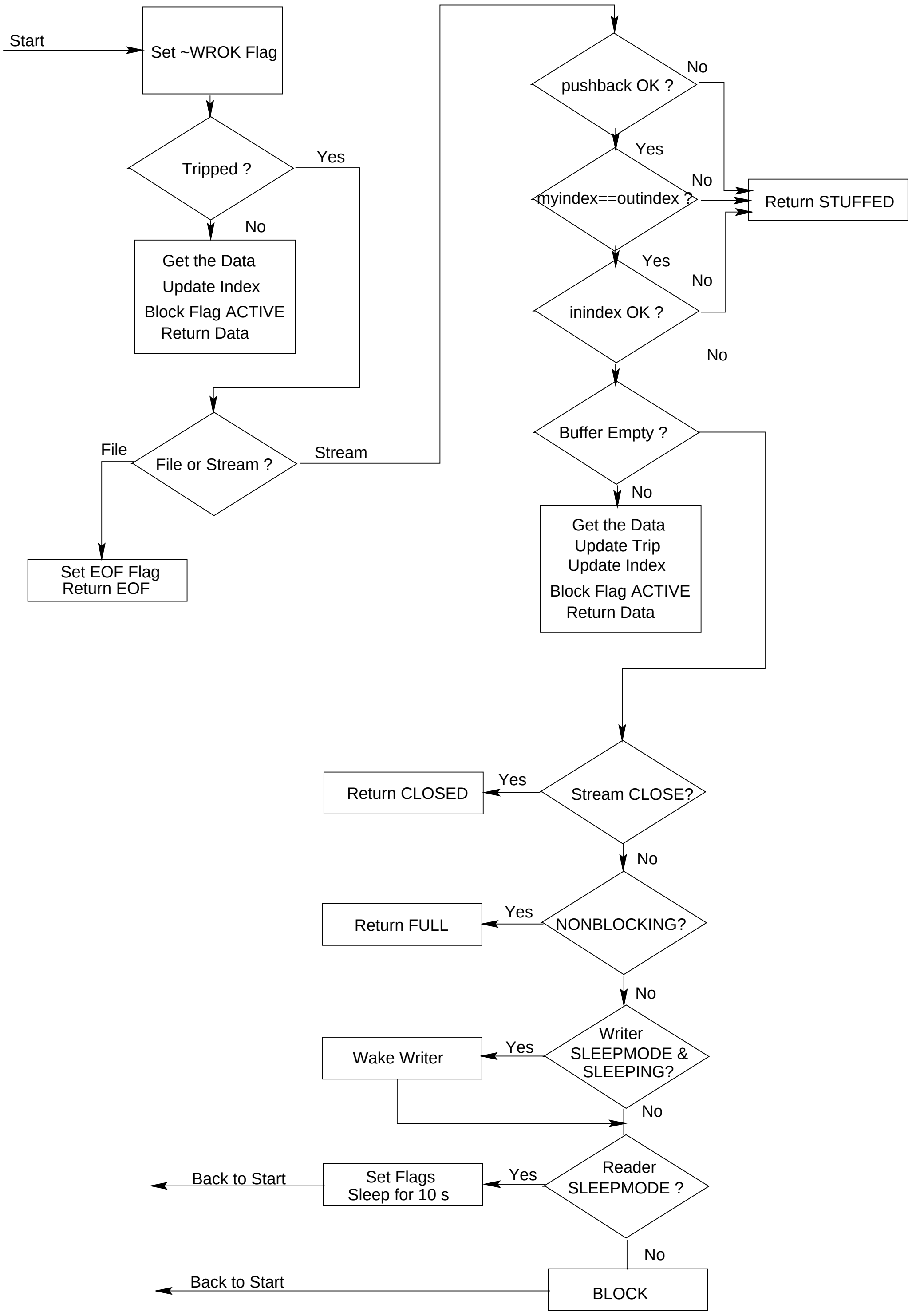


Figure 3.6 Walnut Stream Design - Simplified fgetc Algorithm

The tripwire value is used to detect special conditions in the buffer, such as full and empty states, or either index reaching the top to the buffer (designated the wrap condition). Because the trip condition is unique to each of the processes operating on the stream buffer, it was placed into the file structure.

To achieve high performance, the algorithm used in both *fputc* and *fgetc* was designed to minimise the number of operations performed in a regular access to the buffer (i.e. non-trip condition).

This was accomplished by having *fputc* and *fgetc* first set the status flag and test the trip condition, and then proceed to write or read the character, or if tripped, interpret the state of the buffer and proceed further (Figure 3.5).

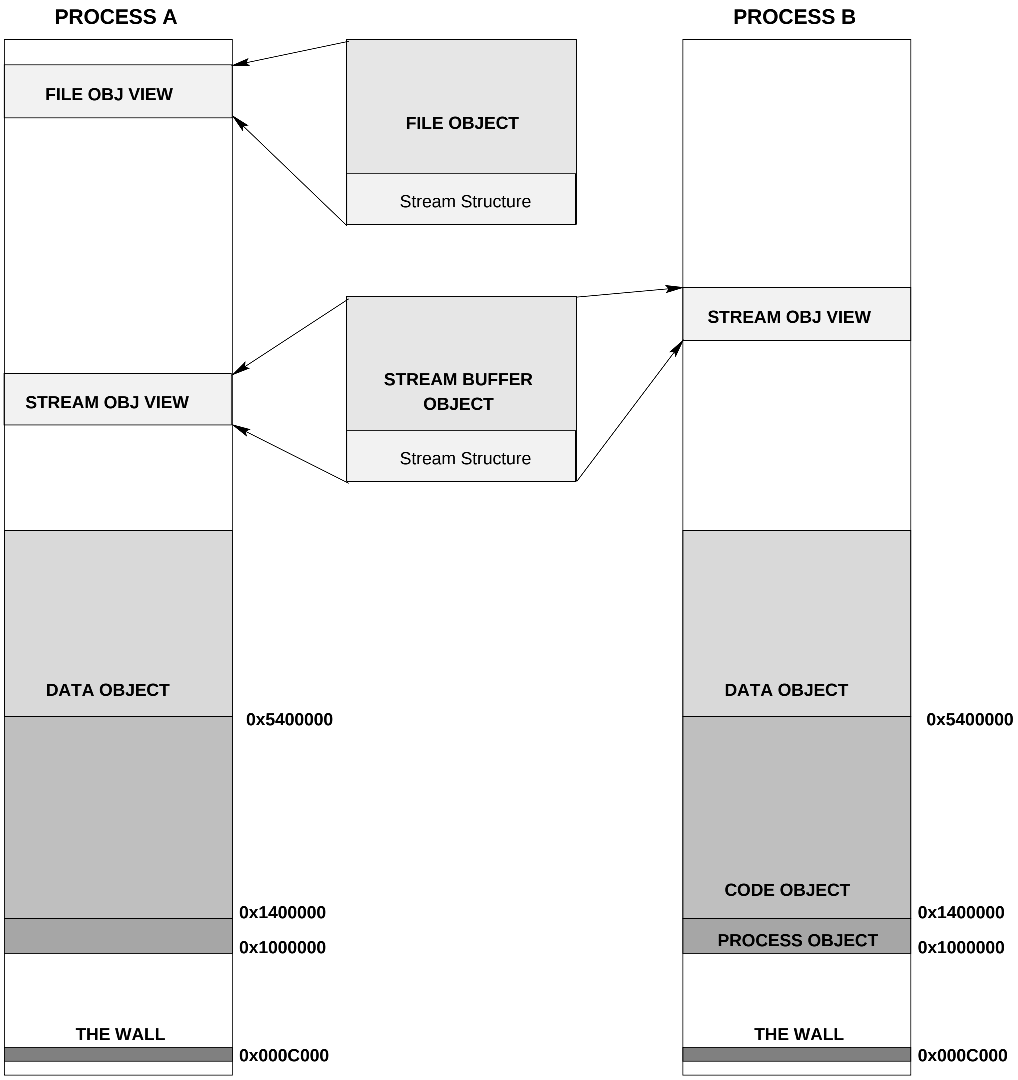


Figure 3.7 Walnut Library I/O Implementation Model

To implement the *ungetc* function, facilities for pushback of characters had to be provided. The mechanism adopted uses a compile time parameter (**NPUSHBACK**) which is used to offset the trip point for the condition where the write index is "catching up" with the read index. If the trip point in the buffer were not offset by the limit to the number of characters pushed back on to the stream, these characters could be overwritten by the writing process. The number of characters pushed back is held in the *Stream* structure.

The **NPUSHBACK** value could also be held in a field within the *Stream* structure, to be initialised at the time the structure is created. This would allow stream connections to have arbitrary values of **NPUSHBACK**, providing that these values are reasonable in relation to the buffer size. It is intended that a future version of the library include this facility.

The file structure was initially designed to operate with an IPC stream, and subsequently modified to operate with a file object. To improve performance, the file structure contains pointers to all of the fields in the *Stream* structure which are required for operations on the buffer. These are initialised when the file structure is initialised. A copy of the read or write index is included to allow integrity checking of the *Stream* structure on every operation. A field of flags was included to enable ANSI functionality to be implemented.

The unified *Stream* structure for both file objects and stream buffer objects was chosen to simplify the design. Since both stream buffer objects and file objects would be manipulated only by *stdio* library functions, there was no apparent advantage to the use of file objects comprising only data. The implementation adopted therefore uses the same *Stream* struct and data area format for both stream buffer objects and file objects.

Stream buffer objects and file objects are identified by the use of a magic number, rather than using the type code of the capability. Whilst no advantage was found to either technique for open operations, the use of the magic number technique was cheaper for operations on objects already mapped into the process address space. This is because the overhead of a **K_CAPSTAT** system call was not incurred.

The mapping of ANSI standard open modes for files on to Walnut access rights required some thought. For the creation of file objects, read and write modes would map directly, and the update mode would provide both read and write modes. If the modes specified in an open call would not match the access rights to an existing capability, the operation would fail. Because there is no distinction in the Walnut between binary and text streams, the optional

ANSI mode flag for binary streams, "b", is not used.

Because the Walnut stream is unidirectional, the update mode is not supported for operations on stream buffers. Because a stream buffer object must be destroyed on the closing of the connection, and must be loaded by at least two processes, the default access rights mask for a *Stream* buffer object is **SRSUICIDE | SRREAD | SRWRITE | SRUSER | SRMULTILOAD**.

The file structures are managed as a linked list. While the alternative method of allocating heap storage for every file structure to be used was considered, a file table managed as a linked list was adopted as this simplified the debugging of the design. Allocation of structures on the heap is more efficient in usage of address space.

The file table is located at a fixed address in the process object, and is typically initialised after process creation. Adoption of this mechanism allows the library to provide both the ANSI and POSIX programming interface conventions. The latter was not implemented.

3.2.3.2 File Object Opening and Object Creation

The availability of the nameserver function library allowed the use of a name based syntax which preserved the ANSI programming interface. Using the interface requires that a binding to a valid capability exists. If the capability does not exist, the open operation must fail. If a binding does not exist, the default ANSI behaviour is to assume that the intent is to create a file.

Because the ANSI *fopen* call does not include parameters for default file opening, two alternatives were available for setting parameters for the file object to be created. One alternative is to define a default value in the library with a global variable, the other is to exploit the ANSI standard [ANSI89] and add an additional set of open mode flags unique to the Walnut. Expediency dictated the adoption of the former approach using the *setvol* macro to set the volume number, although a good case can be made for the latter. The form of the latter could be a set of *fopen* parameters such as *"wa,vol=0x8888,limit=1000,send=0,money=10000"*, where the volume parameter specifies the volume to create the object on, the limit parameter sets the value of the argument for **K_MAKEOBJ**, the send flag prevents the capability from being distributed to other processes, and the money parameter sets the money to be put into the object. It is intended that this be implemented in a future version of the library.

The default behaviour for *fopen* is for an object to be created with **INITIALCASH** money, and a large limit size, on the volume number set with *setvol*. Alternatively, a user may create the object with a **K_MAKEOBJ** call, bind it to a name with a nameserver library call and then call *fopen* to open the file object.

3.2.3.3 Stream Open Programming Interface

The programming interface for opening stream connections was implemented by overloading the programming interface for file objects with additional qualifiers. This approach was considered preferable to designing a separate interface, as the interface is simpler and thus more easily debugged. At this phase in the project, the use of name bindings to processes was considered a possibility, but not planned for implementation.

During the development of the library, stream open operations were executed using the *opencap* call, which operates directly on a capability to send a message to a process. This proved to be cumbersome during debugging. The advantages of a scheme where name bindings were attached to processes became very apparent. Not only was the design of client processes simplified, but operations from the Walnut shell command line were simplified. An errant process identifiable by name is much easier to delete. An *rm* utility designed for destroying file objects could be used to destroy a no longer required process.

The fopen call was therefore modified, and additional utilities implemented which allow a process to "name" itself once it begins execution, and to "unname" itself when it destroys itself. The bindings are held in the process' default database.

3.2.3.4 Process Environment Initialisation

The basic process environment provided by the Walnut kernel does not provide facilities for the *stdio* library. A process which is intended to call functions from the library requires initialisation of a number of library variables and structures. This is performed by the *initenv* function which must be run before any other library function.

The *initenv* function will perform the following operations:

• save the argument list if the process was created by a shell.

• set default values for stream object size and blocking mode.

• save process specific information held initially in the parameter block (i.e. process capability, maximum number of mailboxes, subprocesses and loadable capabilities).

• initialise the debugging monitor screen.

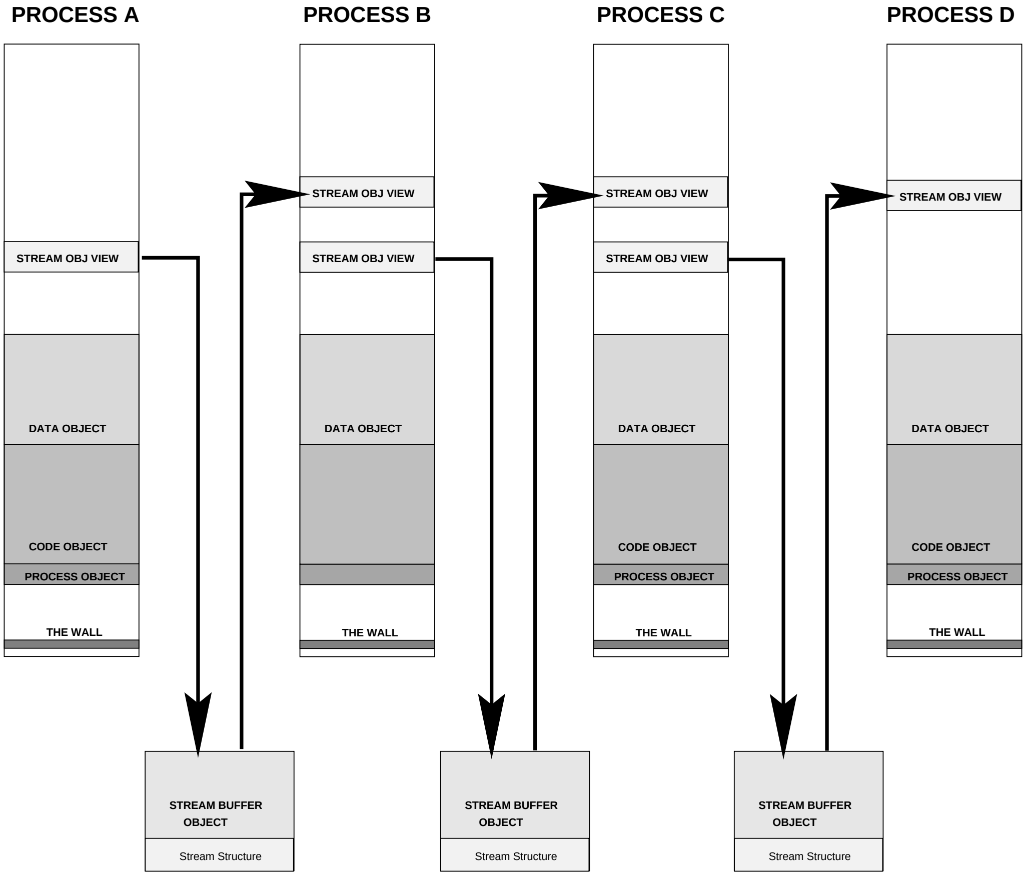


Figure 3.8 Walnut Library Pipe Implementation Model

• make and load the object used by the stream opening protocol for the purpose of passing arguments.

- enable the receipt of messages.

- create a table of mailbox states for all subprocesses.

- initialise the default name server database.

- initialise the file table.

- open file table entries for the *stdin* and *stdout* streams.

- initialise global variables used for debugging and error reporting.

The open *stdin* and *stdout* file table entries are an ANSI standard requirement. The library assumes that valid capabilities to suitable stream objects have already been copied into the file table entries for *stdin* and *stdout*. These will have been created by a shell or other program which has created the process. The ANSI *stderr* file table entry is aliased to *stdout* at compile time, as the ANSI model of directing *stdout* to buffered output and *stderr* to unbuffered output has no meaning in the Walnut context.

The ANSI style of file table initialisation and POSIX 1003.1 style of argument passing allows a shell to provide command line piping between programs, similar to that in Unix.

3.3 Stream/File Object Structures

The final configuration of the Stream structure provides the required common interface to file objects and stream IPC connections. The structure definition is as follows:

```
/* the fundamental types are defined as:
 *
 *          Uw          unsigned 32 bit
 *          Sw          signed 32 bit
 *          Uh          unsigned 16 bit
 *          Sh          signed 16 bit
 *          Uq          unsigned 8 bit
 *          Sq          signed 8 bit
 *
 */
```

```
typedef unsigned long Uw;
typedef signed long Sw;
typedef unsigned short Uh;
typedef signed short Sh;
typedef unsigned char Uq;
typedef signed char Sq;

/*
*       Standard structure at front of File and Stream objects
*/

typedef struct Streamst {
        Sw magic;       /* magic number determines object type */
        Sw type;        /* Basically, stream or file  */
        Sw strmsz;      /* Size in chars of data area  or fileobj size */
        Sw base;        /* Offset of data area from start of Stream struct */
        Sw inindex;     /* Index rel to base where NEXT char is to be put */
        Sw filelimit;   /* file object size limit */
        Uq lock;        /* Semafore */
        Uq readersblock;/* Flags indicating if reader is blocking */
        Uq writersblock;/* Flags indicating if writer is blocking */
        Uq dum3;
        /*
        * ------  The rest needed only for inter-process stream ------
        */
        Sw outindex;    /* Index whence next char will be read  */
        Capl reader;    /* Message capl to reader       */
        Sw rfdti;       /* Reader's file descriptor table index  */
        Sw rmode;       /* Read mode flag */
        Capl writer;    /* Message capl to writer  */
        Sw wfdti;       /* Writer's file descriptor table index  */
        Sw wmode;       /* Write mode flag  */
        Sw pushback;    /* pushback counter, inc'ed by ungetc, dec'd by getc */
        }       Stream;
```

The magic number, *magic*, determines the type of capability the Stream is associated with. Two valid types are defined at this time. The **FILEMAGIC** type is associated with ordinary files, which may be text (ASCII) or binary. Code and data objects when accessed by the library are treated as ordinary files. The **STREAMMAGIC** type is associated only with stream buffer objects.

The *type* field may be a **FILECHAR** or a **FILESTREAM**, and was used during the early development of the stream IPC implementation. To retain compatibility with other items of code produced prior to the introduction of the *magic* field, this field has been retained.

The *strmsz* field is overloaded with two usages. In a file object it contains the exact size of the file, which is less than or equal to the size of the object. In a stream buffer object, it contains the size of the stream buffer within the object.

The *base* field contains the size of the offset from the beginning of the object to the beginning of the file data or stream buffer.

The *inindex* field is overloaded with two usages. In a file object opened in write mode, it contains the file index value. In a stream buffer object, it contains the value of the write index.

The *filelimit* field is used only in file objects, and contains the absolute limit to the file size. The *lock* field is not currently used, but is intended as a semaphore in a multiprocessing environment.

The *readersblock* and *writersblock* fields are used to indicate the blocking state of the file object or stream buffer object, in the read and write directions respectively. These fields are used to indicate whether the operation is **ACTIVE**, **BLOCKED**, **SLEEPING** or **WOKEN**, in the respective modes of blocking.

The *outindex* field is overloaded with two usages. In a file object opened in read mode, it contains the file index value. In a stream buffer object, it contains the value of the read index.

The *reader* and *writer* fields are used in stream buffers only, and contain the capabilities to message the reading process and writing process respectively. When a stream connection is running in wakeup mode, these capabilities are used by a process to send a wakeup message to the other process using the connection.

The *rfdti* and *wfdti* fields are indices into the file tables of the reading and writing processes, respectively. These have been retained for compatibility with earlier developed code and may be removed in a future version of the library.

The *rmode* and *wmode* fields are used to indicate the intended blocking mode of the file object or stream buffer object, in the read and write directions respectively. It is also overloaded with the closing state flag, used during the closing of stream connections.

The *pushback* field contains the count of characters pushed back on to the file object or stream buffer object by the ungetc operation.

3.4 File Structure

The final configuration of the *File* (*FILE*) structure provides the required interface to a file table entry. Like the *Stream* structure, it overloads some fields with usages for stream communications and file objects. The structure definition is as follows:

```
/*
*       File (FILE) Pointer - File descriptor table entry
*/

typedef struct Filest   {
        Capl filecap;
        Sw windowsize;  /* Size in chars of loaded window onto buffer/file */
        Stream *obj;    /* Ptr to start of buffer or file object  */
        Sw strmsz;      /* Size of info area in chars   */
        Uq *strm;       /* Logical ptr to start of information characters */
        /*
        *       in- and out- indexes are indexes rel to strm
        *       inptr, outptr point to these values
        */
        Sw *inptr;      /* ptr to in-index  */
        Sw *outptr;     /* ptr to out-index  */
        Sw *pushback;   /* ptr to pushback counter */
        Sw tripwire;    /*
                         * A critical index value at which special action is
                         * needed
                         */
        Sw myindex;     /* Next index value to use      */
        Uq mode;
        Uq type;        /*  FILECHAR = file object,  FILESTREAM = stream */
        Uq dir;         /*  1 = read,   2 = write,   3 = read/write  */
        Sw next;        /* index to next free File */
        Sw flags;       /* eof, err flags */
        Sw fileindex;   /* file object posn pointer */
        }       File;
```

The *filecap* field is the capability to the file object or stream object to be opened, or open. It is used during the opening operation.

The *windowsize* field contains the size of the view of a file object mapped in. The existing version of the library does not utilise it, it is intended to be used in situations where the view mapped into the process address space is much smaller than the size of the object.

The *obj* field is a pointer to the beginning of the mapping of the file object or stream buffer object, where the *Stream* structure resides.

The *strmsz* field is overloaded with two usages. In a file object it is the size of the file, in a stream buffer object the size of the data buffer. Both values are in bytes (8-bit characters).

The *strm* field is a pointer which is overloaded with two usages. In a file object it points to the beginning of the data within the object (i.e. the first byte), in a stream buffer object it points to the beginning of the data buffer.

The *inptr* and *outptr* fields point to the inindex and outindex fields of the Stream structure respectively. The *tripwire* field is used to manage the state of the indices into the data buffer of a stream buffer object, or the index into a file object. The *myindex* field is a copy of the last value of the index written by the process accessing the file structure.

The *type* field is used to identify whether the file structure is associated with a file object or a stream buffer object. Valid values are **FILECHAR** and **FILESTREAM**.

The *dir* field identifies the direction in which the stream may be operated upon. It may assume the values of **READ**, **WRITE** and **UPDATE**.

The *next* field is used in managing the file table linked list, and points to the next free entry in the table.

The *flags* field is a 32-bit bitmask used in managing the state of the stream. A number of flags are defined:

○ **_EOF** - end of file condition encountered
○ **_ERR** - error condition encountered
○ **_TIMEOUT** - timeout error condition encountered
○ **_APPEND** - file open in append mode
○ **_UPDATE** - file open in update mode
○ **_RDOK** - read operations are permissible
○ **_WROK** - write operations are permissible
○ **_CAP** - capability has no name binding
○ **_TMP** - object is a tmpfile
○ **_FULL** - full condition encountered

The *fileindex* field is used only with file objects and stores the value of the index (pointer) into the file. If greater than the *strmsz* of a file object, its value is copied to *strmsz* on a flush or close operation.

Chapter 4 Library Implementation

4.1 Introduction

The implementation of the *stdio* library reflects the constraints and rationale defined in Chapter 3. The library can be functionally divided into ANSI C library functions and Walnut specific functions. Additional debugging tools are detailed in the Appendices to this document.

4.2 Walnut Library Functions

A C programmer can directly manipulate file objects and IPC streams using the Walnut library functions. These library functions are also used by the ANSI stdio functions for lower level manipulation of capabilities, and provision of services such as the opening of stream connections.

4.2.1 The opencap Function

The *opencap* function is similar in operation to the Unix and POSIX 1003.1 open functions, providing the low level programming interface for file and stream object operations. Because the Walnut kernel operates directly on capabilities, the programming interface for *opencap* provides arguments necessary for this purpose.

The function call for *opencap* and its arguments are defined as follows:

*FILE *opencap(Capl *capability, Uw flags, Uw vol, Uw money, Uw limit);*

- **Capl *capability**, is a pointer to a file object or target process object capability. The process opening a file object or IPC stream must have a valid capability and access rights to the file object or process to be opened, if it exists.

- **Uw flags**, is a 32 bit bitmask of flags which are to be used when opening a file object or a stream IPC connection. These flags will define modes and defaults should a file object need to be created.

○ **READ** - flag to open the capability for reading

○ **WRITE** - flag to open the capability for writing

○ **APPEND** - flag to open the capability for writing, and advance the pointer to EOF

○ **UPDATE** - flag to open the capability for update (i.e. read and write), valid only for file objects.

○ **CREATE** - flag to specify the creation of a file object, if the object to be opened does not exist.

○ **BLOCK** - flag to set stream I/O mode to blocking.

○ **NOBLOCK** - flag to set stream I/O mode to non-blocking.

○ **WAKEUP** - flag to set stream I/O mode to sleep on empty or full.

○ **SMALLWINDOW** - flag to specify an object created with 4 kbyte size, overriding the limit argument

○ **LARGEWINDOW** - flag to specify an object created with 4 Mbyte size, overriding the limit argument

○ **FILECAP -** flag to specify that the capability to be opened is to a file object.

• **Uw vol**, this argument specifies the volume for file object creation.

• **Uw money**, this argument specifies the amount of money to be placed into a file object when created.

• **Uw limit**, specifies the limit value to be used during object creation.

The *opencap* function will first check the value of the capability argument to determine whether it is a valid capability value, or a null capability. If it is a null capability and the **CREATE** and **FILECAP** flags are not set, then *opencap* will return with an error.

Once the capability value has been checked, the *opencap* function will then decode the flags argument to set values for access modes and blocking modes.

The *opencap* function will then execute a **K_CAPSTAT** system call to verify the existence of the capability. If the capability does not exist, and creation has not been specified, *opencap* returns with an error. If the capability exists, *opencap* saves the parameter block containing the details of the capability.

Once the details of the capability have been found, *opencap* tests the type field to determine whether the capability is to a file object or a process object. The flag bits for window sizes are tested to set the limit value to be used in creating a file object or a stream object.

If a file object is to be created, *opencap* executes a **K_MAKEOBJ** system call to create the file object, and saves its details. The object is then loaded into the address space of the process, and initialised with a valid Stream struct. During initialisation the magic number and type fields are set, the limit is set and the blocking modes are set. The initialised object is then unloaded.

If *opencap* is operating on a file object, the objects rights are then compared with the flags argument. A mismatch in open flags and rights will cause *opencap* to return with an error.

If *opencap* is opening a stream connection, it will open the process mailbox to enable the receipt of messages. Once this is done, *opencap* will execute a **K_PEEKPROC** call via the *pingtarget* function, to determine the state of the target process. If the target process doesn’t exist, does not provide the right to peek, is frozen, in probate or dead, then *opencap* returns an error.

If the target process is alive and thus assumed to be capable of producing a stream connection, *opencap* will employ the *sendrequest* protocol module to send an open stream request message. If a valid response message is received from the target process, *opencap* will decode the message. If the message indicates that the request has been serviced, the stream object returned is checked. The *opencap* function will execute a **K_CAPSTAT** call to verify the existence of the object.

At this point *opencap* will possess a capability to a file or a stream object which is known to exist. It may now proceed to complete the opening of the object.

A file table entry is now reserved for the object, and the file table linked list amended to reflect this. The file table entry is then initialised with the capability to the object, and the *fileinit* function called to complete the open. If *fileinit* does not return an error, *opencap* returns the file pointer, and the open is completed.

4.2.2 The fileinit Function

The *fileinit* (formerly *open2*) function is used in the final phase of opening a stream. Its purpose is to properly initialise the File and Stream structures associated with the open stream, and load the object into the address space of the calling process. The function call and its arguments are:

*Sw open2(Uw fdindex, Uw *offset, Uw *cindex, Uw block, Uw direction);*

- **fdindex** - index into the file table

- **offset** - specifies capability offset and returns offset value

- **cindex** - specifies capability cindex and returns cindex value

- **block** - specifies blocking mode

- **direction** - specifies read or write direction for stream or file objects

The *fileinit* function will first test the file table index for a valid value. If the value is valid, it then tests the capability held in the file table entry for validity. A **K_CAPSTAT** system call is then used to test for the existence of the capability. If the capability exists, it is then loaded into the process address space. The existence test is debugging code which was retained.

Once the stream or file object capability is loaded, *fileinit* will test the magic number and if valid, initialise the pointers and pushback counters held in the File structure. If the capability is a stream object, the index values and size are tested for validity.

The *fileinit* function will then initialise the common file structure fields for direction, stream base, stream or file size, type, flags and for a stream, tripwire. Stream or file object specific parameters are then initialised. These are the capability of the calling process, the file table index and the blocking mode. In file objects, the flags for file state are initialised.

4.2.3 The makestreamobj Function

The *makestreamobj* function is used to create a valid stream object. It is typically employed by a server process which is responding to a request for a stream connection.

*Sw makestreamobj(Capl *obj, Uw *size, Uw money);*

- **obj** - a pointer to the capability of the created stream object

- **size** - specifies the size of the stream buffer object to be created

- **money** - specifies the money value to be placed into the stream buffer object

The algorithm for setting the object size will take the size argument, add the size of the Stream structure and set the object size to be the minimum number of pages required to contain the structure and specified buffer size. An object is then made, loaded and initialised. The magic number, type, base, size and default blocking modes are set. The object is then unloaded and the function returns.

4.2.4 The copen Function

The *copen* function is analogous to the ANSI *fopen* function, but operates on a pointer to a capability rather than a pointer to a string. It was included in the library to allow a programmer to open capabilities which do not have name bindings.

*FILE *copen(Capl *cap, const Uq *mode);*

The mode argument is identical to that in *fopen*, and the *copen* function differs from *fopen* only in that it does not need to operate on the name-server binding to determine the value of the capability.

4.2.5 The removec Function

*Sw removec(Capl *capability);*

The *removec* function is analogous to the ANSI *remove* function, but operates on a pointer to a capability rather than a pointer to a string. The capability is destroyed by this function.

4.2.6 The cmap and cunmap Functions

The *cmap* and *cunmap* functions are analogous to the Unix and POSIX *mmap* and *munmap* system calls. Unlike the Unix calls, *cmap* and *cunmap* operate on a specified capability rather than a file pointer. The function is included to simplify porting of applications and is not used in the library implementation. The functions will load or unload a capability into or from the address space of the calling process, respectively.

*void *cmap(void * addr, Uw len, Uw prot, Uw flags, Capl *cap, Uw offset);*

- **addr** - address at which the capability is to be loaded

- **len** - specifies the length of the view to be loaded

- **prot** - unused in Walnut

- **flags** - unused in Walnut

- **cap** - pointer to capability to be loaded

- **offset** - offset to base of view to be loaded

*void cunmap(void *addr, Uw len);*

- **addr** - address of the mapping to be unloaded

- **len** - unused in Walnut

The *cmap* and *cunmap* function implementation is not compliant with the POSIX function. A compliant *cmap* implementation would need to create a derived capability with parameters defined by the flags argument, and load this capability.

4.2.7 The setmyname and clrmyname Functions

*Sw setmyname(char * myname);*
*Sw clrmyname(char * myname);*

These functions are used to create or delete a name binding to the calling process. The sole argument is a pointer to a string. A typical use is to simplify the design of client processes, which may locate a server by using a name rather than a capability value. The default nameserver database is used.

4.2.8 The kerror Function

The *kerror* function is used to report kernel errors returned by a system call which has failed. The function is analogous to the ANSI *perror* function, in that it will take a pointer to a string as an argument and write the string and its error message to the *stderr* file. Where *kerror* differs from the ANSI *perror* is in its ability to provide verbose reporting of kernel errors. A global library variable, *debug*, is employed to set the level of error reporting.

*void kerror(const char *s);*

A debug level of 2 provides terse reporting, which writes the kernel error code to *stdout*. A debug level of 3 provides verbose reporting, which decodes the kernel error code and writes a description of the error to *stdout*. Error reporting is detailed in [Appendix A].

4.2.9 The Client Server Protocol Functions

The Walnut *stdio* library implements a simple client server protocol. This protocol was designed to support process to process requests for stream opening, as well as to provide the basic functionality for the later implementation of a remote procedure call mechanism. The latter is not implemented, as it fell outside the scope of the library design and implementation.

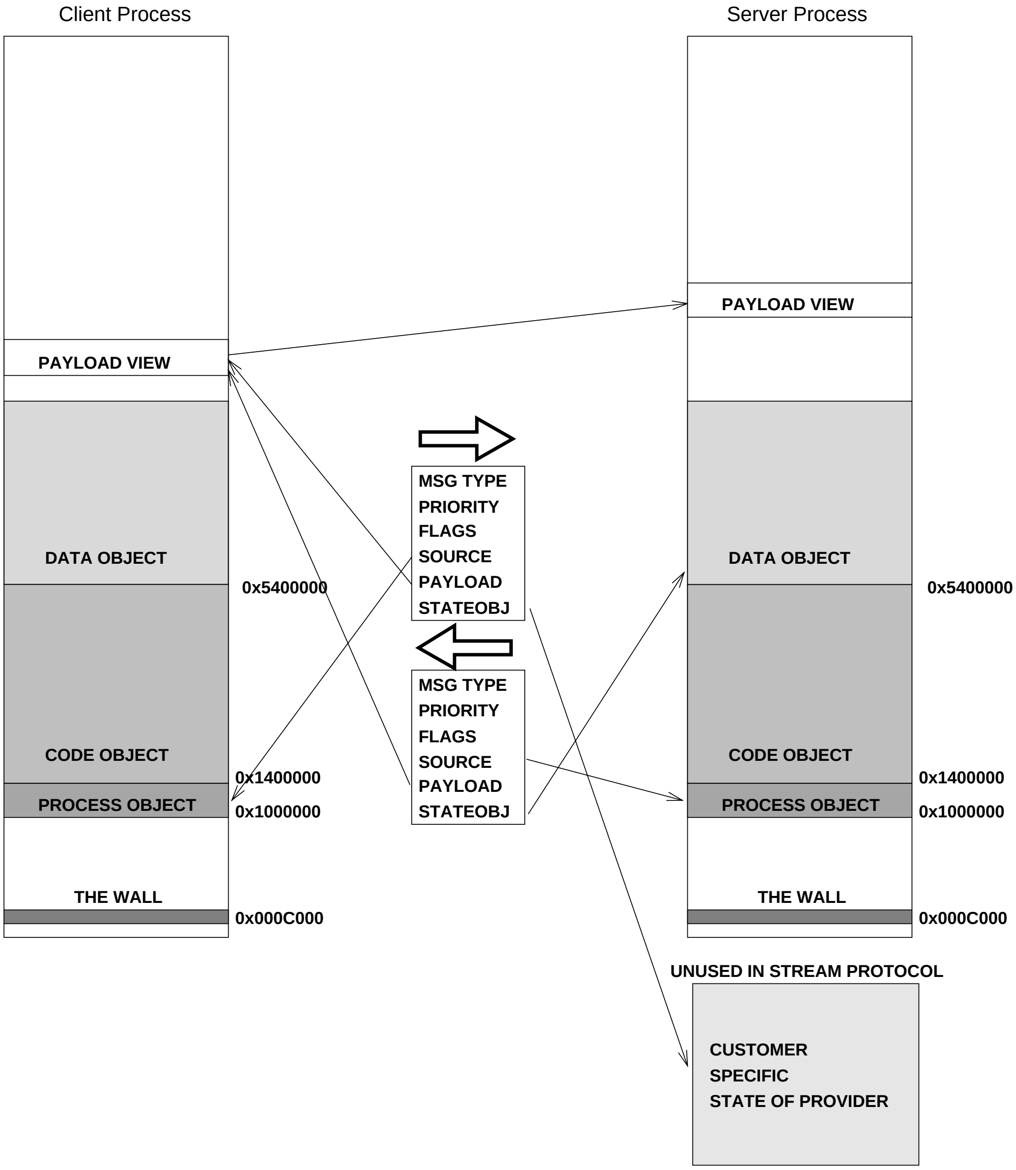


Fig.4.1 Client Server (RPC) Protocol Structures

4.2.9.1 Protocol Definition

The protocol is a simple request and response scheme. The client will send a request message to the server. The server will validate the request, and if appropriate, will supply a response. In the instance of stream opening, the response will contain the capability to a stream buffer object, which is created upon validation of the request.

Because the permitted message size in the Walnut kernel is small, a protocol which must pass arguments back and forth would require careful design of the message format to fit all the necessary fields into the available space. Should the protocol need to support Remote Procedure Calling (RPC), then the space in the message is simply too small for general use. The protocol design created therefore passes a capability to a argument buffer between the client and server. By using a capability to an argument buffer, the basic design becomes readily extensible. As a result, the effort required to provide an RPC service is minimised by exploiting the existing protocol modules used for stream opening.

The protocol message structure has the following format:

```
typedef struct {
    Uw     type;          /* protocol message type */
    Uw     money;         /* money sent or returned */
    Uw     priority;      /* priority of service */
    Uw     flags;         /* flags field */
    Capl   src;           /* originator of message */
    Capl   payload;       /* arguments passed/returned */
    Capl   persistent;    /* persistent values passed/returned */
    } SERVICE_REQUEST;
```

The message format is symmetrical, in that both requests and responses use the same structure. The values in the fields determine whether the message is a request for a service or a response. Valid message types are:

- **REQUEST_SVC** - request for RPC service
- **RESPOND_SVC** - response to RPC service request
- **REJECT_SVC** - rejected RPC service request
- **REQUEST_STRM** - request for stream IPC connection
- **RESPOND_STRM** - response to a request for stream IPC connection
- **REJECT_STRM** - rejection of a request for stream IPC connection

A request message is sent by a client to a server, a

REQUEST_STRM message asks for a stream connection with defined parameters. A response message is sent by a server to a client, acknowledging the provision of the requested service. A reject message is sent by a server to a client, to notify of the rejection of the request.

The client server protocol exploits the money mechanism. A server expects payment for the provision of a service. The *money* field is a redundant copy of the value of the money sent with the message, and was employed for debugging purposes. Should insufficient money be provided, the request is rejected.

The *priority* field is a facility intended to simplify the relative scheduling of service requests from multiple clients to a single server. A queue of received requests may be scheduled in the order of requested priority.

The *flags* field allows the transmission of status information. Valid flags are:

- **CPERSISTENT** - return persistent state information
- **CUSEPRIORITY** - use the priority field
- **PLOWMONEY** - request rejected due insufficient money
- **PNOSUCHSVCE** - request rejected as service not available
- **PBADMODES** - request rejected due incorrect modes
- **PBADPRIORITY** - request rejected due invalid priority value

The client process provides the server process with the capability to to send it a message in the *src* field. The *payload* field is the capability to a buffer of arguments. It is used for stream requests and is intended for use with RPC requests. The *persistent* field is intended for use with RPC requests. State information pertaining to a request from a specific client can be returned to the client, without read or write access. A subsequent RPC service request can then restore the state information before execution. This facility is not currently used.

When requesting a stream connection, the payload capability will contain the parameters of the requested stream. These are loaded in the following format:

```
typedef struct {
    Uw      size;          /* stream object size - customer/provider */
    Uw      blockmode;     /* stream blocking mode - customer/provider */
    Capl    streamobj;     /* capability to stream object - provider */
```

} STREAM_PARAMS;

The *size* field is the requested size, the *blockmode* field the requested blocking mode for the requesting party and the *streamobj* field is the returned capability to the stream object.

4.2.9.2 Programming Interface and Implementation

Three functions are implemented for the purpose of protocol handling and decoding. These are *sendrequest* and *getrequest*, to which the *sendresponse* and *getresponse* functions are aliased at compile time, and *decoderequest*.

```
Sw sendrequest(Uw type, Uw money, Uw priority, Uw flags, Capl *target,
                                        Capl *payload, Capl *persistent, Uw subpn);
Sw getrequest(Uw *type, Uw *money, Uw *priority, Uw *flags, Capl *source,
                                        Capl *payload, Capl *persistent);
Sw decoderequest(Uw type, Uw money, Uw priority, Uw flags, Capl *source,
                                        Capl *payload, Capl *persistent);
```

The *sendrequest* function copies the argument list into the message area, in a format defined by the **SERVICE_REQUEST** structure, and then calls **K_EXTSEND** to send the message to the intended recipient. The *getrequest* function is called by the receiving process to retrieve the message. This function will take the *type* field from the argument list and copy it into the message area, to ensure that only messages of the specified type are received. The *getrequest* function will then attempt to retrieve a message. If no message has been received, the function makes a **K_WAIT** call and sleeps until a message has arrived, upon which it repeat the previous step.

If a message of the proper type is received, *getrequest* will then load the contents of the message area into the corresponding variables pointed to by the argument list. The *decoderequest* function is then used to validate the contents of the message. This function will return an operation code which the server will act upon.

4.2.10 The accept Function

The *accept* function is analogous to the 4BSD *accept* function used with the socket interface. The Walnut *accept* function will wait for the arrival of a valid request for a stream IPC connection, and then return a pointer to the file structure associated with the open connection.

*File * accept(Uw dirn, Uw maxsize, Uw myblockmode);*

The *dirn* argument specifies the direction of the stream as seen by the calling process. A request for a stream with the same direction specified will fail. The *maxsize* argument imposes a limit on the size of the stream object to be provided. The *myblockmode* argument specifies the blocking mode required by the calling process.

The *accept* function will upon receipt and decoding of a valid stream connection request, load the payload capability and extract the parameters of the stream connection. It will then create the stream object using *makestreamobj*, return a valid response to the requesting process, and then load the stream object, initialise the file table entry with *fileinit*, and return a file pointer. The server process may then use the stream connection.

4.3 ANSI stdio Library Functions

4.3.1 Stream Operation Functions

The ANSI stream operation functions manipulate the state of a file or stream connection.

4.3.1.1 The fopen Function

*FILE *fopen(char *filename, const Uq *mode);*

The *fopen* function is substantially compliant with the ANSI model. The most significant difference in the programming interface is that the function will open both file capabilities and stream IPC connections to named processes. The overloading of the function call with both stream IPC and file operations was advantageous. The use of the Walnut nameserver library to bind names to processes allows retention of the same syntax as used with file access, thus avoiding the need for either a separate programming interface, or a non compliant version of the ANSI interface.

The *mode* argument fully conforms to the ANSI standard, supporting read, write, append and update modes in file object access. Stream access is constrained to write or read access, as the append and update (read and write) modes are not defined for stream connections in the Walnut environment.

The Walnut *fopen* function will first decode the open mode to generate flag values for *opencap*. It will then call the *namec* function to find the capability which is bound to the *filename* argument. If no such capability exists, it will assume that the intent is to create and open a file object, and appropriate default values will be set.

If the name is bound to a capability, *fopen* will extract the parameters contained in the binding to determine whether the capability is to a file or a process. If the binding is to another capability type, *fopen* returns with an error. Defaults specific to a file or a process are set if the binding is to one of these two types. The *opencap* function is then called to open the file object or stream connection.

If the binding did not exist, and a file object has been created, *fopen* will then bind the string pointed to by the filename argument to the capability for the created object.

4.3.1.2 The fclose Function

*Sw fclose(FILE *stream);*

The *fclose* function conforms to the ANSI standard. This function will first call *fflush* to update the file pointer, or flush a stream connection. If a stream connection is being closed, *fclose* will set the mode flags for closure to notify the other process that the connection is being closed.

The capability is then unloaded, and if it was a tmpfile it is destroyed. If the capability is to a stream object, and *fclose* has been called as a result of a closure condition detected in the flags, *fclose* will destroy the object. The file table entry is then released and *fclose* returns.

4.3.1.3 The fflush Function

*Sw fflush(FILE *stream);*

The *fflush* function conforms to the ANSI standard. For file objects, *fflush* will update the file size to the value of the current write pointer, and update the trip value, and set the **_RDOK** flag for files in update mode. The latter is an ANSI requirement which specifies that when in update mode, a write operation must be followed by a flushing operation before a write can be done.

For stream objects where the other process is in wakeup mode, *fflush* will send a wakeup message to expedite the flushing of the buffer. The *fflush* function will then poll the buffer until it is emptied. During each poll, *fflush* relinquishes its time slice. Once the buffer is empty, or file operations complete, *fflush* returns.

4.3.1.4 The rename Function

*Sw rename(const char *oldname, const char *newname);*

The *rename* function is not implemented. An implementation would first locate the binding associated with the *oldname* argument, test it for validity, check the status of the capability, and if valid, then create a new binding to *newname* and delete the binding for *oldname*. This would provide behaviour which conforms to the ANSI standard.

4.3.1.5 The remove Function

*Sw remove(const char *filename);*

The *remove* function is an extension of the ANSI programming interface. It provides the ANSI operation on file objects, but can also remove a process if sufficient rights are provided to the process.

The *remove* function will first locate the binding associated with the *name* argument, and if it is associated with a file, code, data or process object, it will extract the capability. The binding is then deleted and the capability destroyed.

4.3.1.6 The tmpfile Function

*FILE *tmpfile(void);*

The *tmpfile* function conforms to the ANSI standard. It will set up default values for file creation, call *opencap* and then set the **_TMP** flag in the file table entry so that the object is destroyed on closing. A name binding is not produced.

4.3.1.7 The tmpnam Function

*char *tmpnam(char *s);*

The *tmpnam* function is not implemented. An implementation would generate a unique name string for the nameserver database in use and save the binding in the database.

4.3.1.8 The freopen Function

*FILE *freopen(char *filename, const char *mode, FILE *stream);*

The *freopen* function nominally conforms to the ANSI standard, but is untested. This function will close the file capability associated with the stream argument, open the capability bound to the filename argument in the specified mode, and then return a pointer to the file table entry.

4.3.1.9 The setvbuf and setbuf Functions

*Sw setvbuf(char *file, char *buf, Sw mode, Uw size);*
*Sw setbuf(char *file, char *buf);*

The Walnut *stdio* library does not support additional buffering in the library. The *setvbuf* and *setbuf* functions are provided for ease of porting, and do not perform any function.

4.3.2 Character I/O Functions

4.3.2.1 The fgetc and fputc Functions

*Sw fgetc(FILE *stream);*
*Sw fputc(Sw c, FILE *stream);*
*Sw getc(FILE *stream);*
*Sw putc(Sw c, FILE *stream);*
Sw getchar();
Sw putchar(Sw c);

The *fgetc* and *fputc* functions are supersets of ANSI functions. Both functions operate directly on the *File* structure and *Stream* structure, using the scheme described in Chapter 3. ANSI character I/O functions will return EOF if the end of file is encountered, or an error condition is encountered. The Walnut library functions will provide the same behaviour for end-of-file conditions, but will return specific negative error codes for other conditions. Unique error codes are provided to identify empty, full, closed or invalid buffer conditions.

The *getc*, *putc* and *getchar*, *putchar* functions are implemented as aliases. This was done to minimise coding effort, as no additional functionality is provided by these functions. The *getc* and *putc* functions when implemented as macros offer a performance advantage as the function call overhead is not incurred.

4.3.2.2 The ungetc Function

*Sw ungetc(Sw c, FILE *stream);*

The *ungetc* function is a superset of the ANSI model. This function will allow the programmer to push back up to **NPUSHBACK** characters. The **NPUSHBACK** parameter is set at library compile time and is thus fixed. If the programmer attempts to push back more than **NPUSHBACK** characters, an EOF is returned.

4.3.2.3 The fgets and fputs Functions

*char *fgets(char *buf, Sw BUFSIZ, FILE *stream);*
*Sw fputs(char *buf, FILE *stream);*

The *fgets* and *fputs* functions both conform to the ANSI standard.

Both functions will repeatedly call *fgetc* or *fputc*, respectively, and will test the returned values for error conditions.

4.3.3 Direct I/O Functions

4.3.3.1 The fread and fwrite Functions

*size_t fread(void *array, size_t elementsize, size_t count, FILE *stream);*
*size_t fwrite(void *array, size_t elementsize, size_t count, FILE *stream);*

The *fread* and *fwrite* functions conform to the ANSI standard programming interface. When operating on file objects, these functions will use *memcpy* operations to efficiently copy data between the *Stream* buffer and the buffer specified in the *array* argument. When operating on an IPC stream, these functions employ the *fgetc* and *fputc* functions respectively to emulate the behaviour defined in the ANSI standard.

4.3.4 Formatted I/O Functions

4.3.4.1 The fprintf and printf Functions

*Sw fprintf(FILE *stream, const char* fmt, ...);*
*Sw printf(FILE *stream, const char* fmt, ...);*

The *fprintf* function does not fully conform to the ANSI standard. The function was ported from the Walnut kernel *printf* function. It will accept arguments only in the **%d, %x, %c** and **%s** formats, and cannot handle floating point arguments. The *printf* function calls *fprintf* with the stream argument set to *stdout*. The ANSI *vfprintf* and *vprintf* functions are not implemented.

4.3.4.2 The fscanf and scanf Functions

*Sw fscanf(FILE *stream, char* fmt, ...);*
*Sw scanf(FILE *stream, char* fmt, ...);*

The *fscanf* function does not fully conform to the ANSI standard, and will provide functionality typical of Unix hosted implementations which predate the ANSI standard. The function was ported from the 4.3BSD *stdio* library *fscanf* function. The *scanf* function calls *fscanf* with the stream argument set to *stdin*. The ANSI *vfscanf* and *vscanf* functions are not implemented.

4.3.5 File Positioning Functions

4.3.5.1 The fseek and ftell Functions

*Sw fseek(FILE *stream, Sw offset, Sw lastoffset);*
*Sw ftell(FILE *stream);*
*Sw rewind(FILE *stream);*

The *fseek* and *ftell* functions conform to the ANSI standard where the view of file object is equal to the size of the object. Where the view is smaller than the object, the EOF indication provided by these functions indicates that the end of the view has been reached. Operations are defined only for file objects. The *rewind* function calls the *fseek* function with an *offset* argument of zero and a *lastoffset* argument of **SEEK_SET**, as per the ANSI standard.

4.3.5.2 The fgetpos and fsetpos Functions

*Sw fgetpos(FILE *stream, Uw *pos);*
*Sw fsetpos(FILE *stream, Uw *pos);*

The *fsetpos* and *fgetpos* functions conform to the ANSI standard. These functions call *fseek* and *ftell* respectively.

4.3.6 Error Handling Functions

*Sw feof(FILE *stream);*
*Sw ferror(FILE *stream);*
*Sw clearerr(FILE *stream);*
*void perror(const char *s);*

The error handling functions conform to the ANSI standard. The *feof* function will test the **_EOF** flag in the file table entry. The *ferror* function will test the **_ERR** flag in the file table entry. The *clearerr* function will clear the **_EOF** and **_ERR** flags in the file table entry. The *perror* function will print the string pointed to by its argument, followed by an error message. Defined error messages are :

- **bad file table index**
- **null capability**
- **bad magic number**
- **bad stream direction**
- **object load failed**
- **bad buffer object**
- **bad capability**
- **make object failed**
- **rights masks inappropriate**
- **bad buffer index**
- **file table full**
- **message send timed out**
- **inconsistent arguments to open**
- **name server error**
- **capl doesn’t exist**
- **no such process**
- **no right to peek**
- **process is frozen**
- **process in probate**
- **process is dead**
- **insufficient money in request**
- **non-existent service**
- **requested illegal modes**
- **request has invalid priority**

Chapter 5 Discussion

5.1 Introduction

This chapter discusses how well the Walnut *stdio* library meets its design objectives, in terms of programming interface, throughput performance, portability, robustness and security. A number of improvements to the design are proposed.

5.2 Programming Interface

The Walnut *stdio* library provides a programming interface which substantially conforms to the ANSI C standard. Whilst a number of less frequently used ANSI functions have not been implemented, their absence has to date not caused any difficulty.

The principal difference between the Walnut library and the ANSI standard lies in the handling of file objects and the extension of the *fopen* and *fclose* interfaces to support stream connections [Section 3.2]. Neither of these differences compromise the initial requirement of providing ANSI compliance where it did not incur significant overheads in development time.

The adoption of a unified programming interface for file objects and streams has provided a simple and elegant solution to the problem of stream opening. The integration of the name server function with stream opening creates interesting possibilities in shell design, as server processes may be easily accessed from the command line. An example would be:

testfile | filter1 | filter2 | magtape

In this command line, *testfile* is a file object, *filter1* and *filter2* programs which manipulate a data stream, and *magtape* a device manager for a tape drive. A single command line from the shell has allowed the user to manipulate data in a file and write it to tape.

The *stdio* library programming interface meets its design objectives, and provides a tool for further research on the Walnut system.

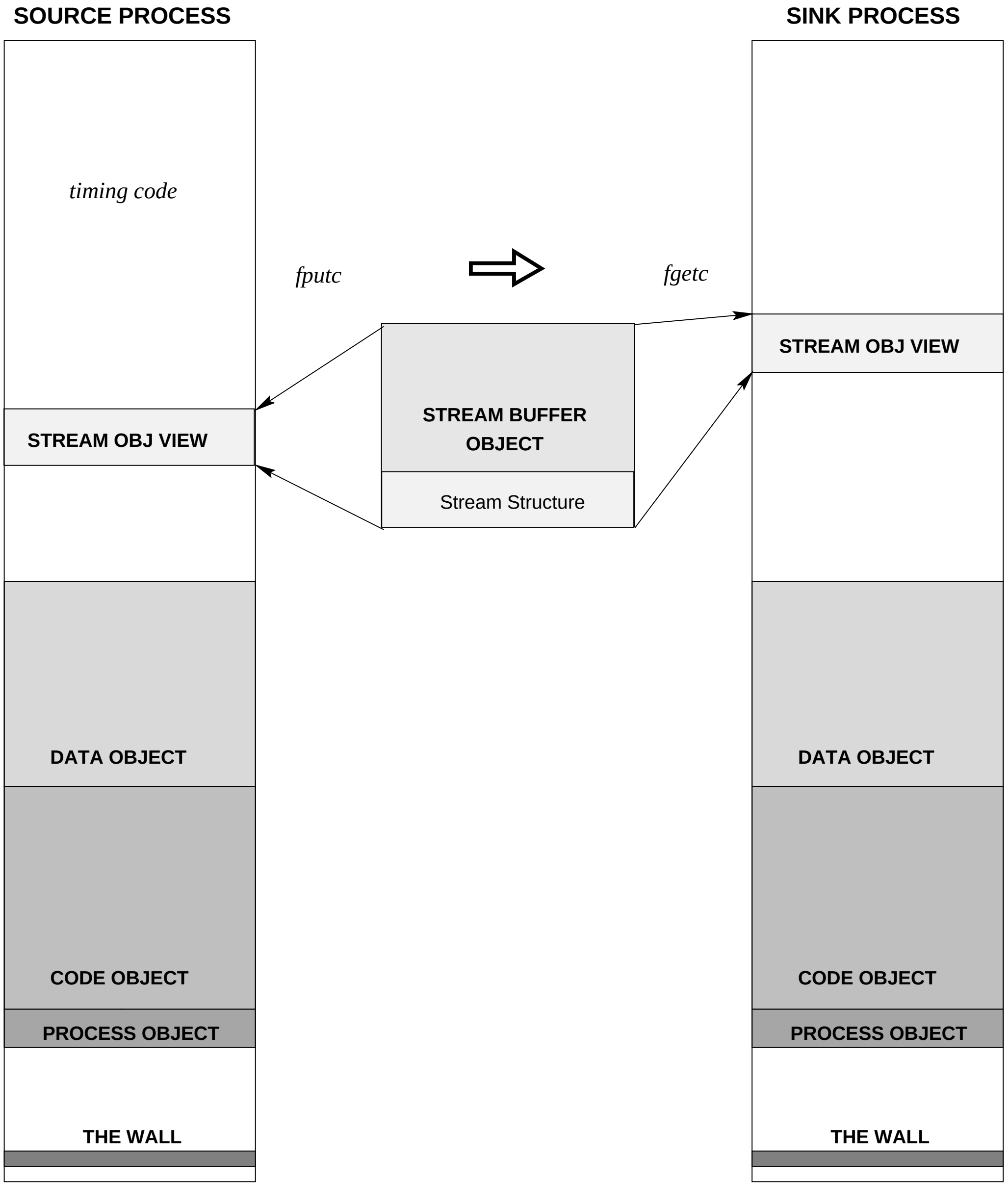
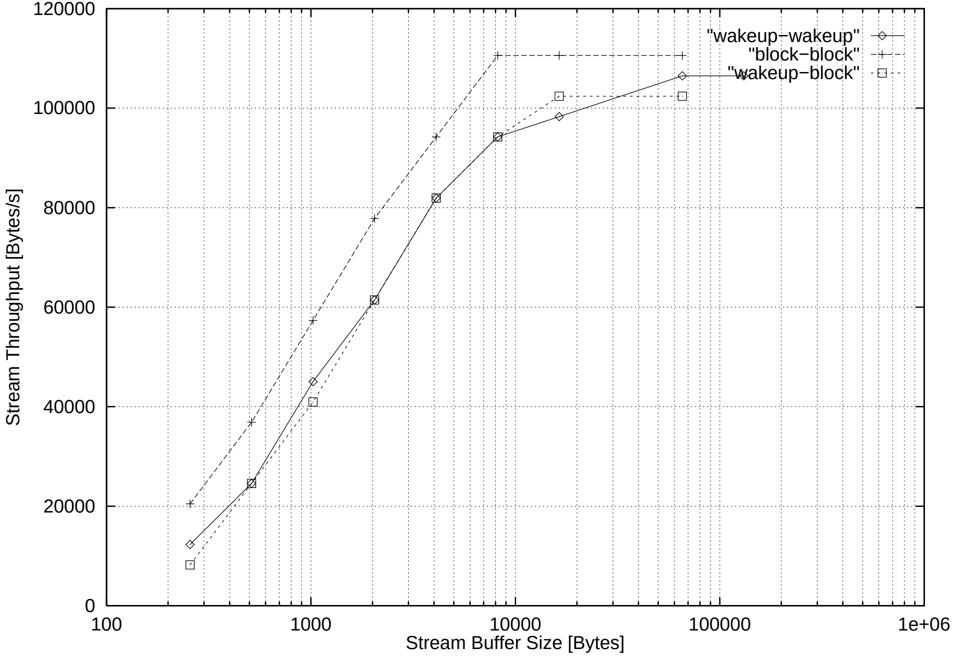


Figure 5.1 Stream Throughput Testing Model

5.3 Throughput Performance

The throughput performance of a stream IPC scheme is an important measure of quality in design and implementation. In a system such as the Walnut, where the stream IPC mechanism is employed for purposes of piping between processes, and I/O operations to stream oriented devices, poor stream IPC performance will significantly affect the performance of the whole operating system. Poor response times will force the use of faster hardware to achieve a desired user response time. Faster hardware can impose significant cost penalties upon the owner of the system. Throughput performance was therefore an important consideration in the design and implementation of the stream IPC mechanism [Section 3.2.3.1].

Fig 5.2 Stream Throughput Performance (Process to Process)



To determine how successfully the design and implementation addressed this requirement, it was necessary to measure the achieved performance. To accomplish this, test programs were produced. The model for the testing, depicted in Fig 5.1, is that of a client process writing test data to a server process. The client process uses the timer registers in the Wall to measure the time from the beginning to the end of the transmission of a buffer of test data. The server process reads the test data from the stream and discards it. Each test run copied test data buffer sizes of 64k, 128k, 256k, 512k, 1024k and 2048k. This was done to ensure that measured performance was consistent for various durations of transfer. The test platform was a 40 MHz Intel 486 Personal Computer.

Each test run was conducted for a different stream buffer size and blocking mode. Stream buffer sizes of 256, 512, 1024, 2048, 4096, 8192, 16384, 65536, 128k and 256k bytes were used as test points. Tests were conducted on processes which were paired in blocking mode and wakeup mode, as well as a test with a writing process in wakeup mode and a reading process in blocking mode. The test programs used the *fwrite* and *fread* functions, which currently employ the *fputc* and *fgetc* functions respectively.

The performance measurements indicate that operation of a stream with both ends in blocking mode always performs better than a stream with one or both ends in wakeup mode. In blocking mode, both processes will always be runnable. Each and every time a blocked process is scheduled to run, it will test the condition of the stream buffer to see whether it can continue. In wakeup mode, a process which is unable to continue will sleep until woken by its peer. This will save the CPU time which would have otherwise been required to poll the state of the buffer, but incurs a time delay between the sending of the wakeup message and the commencement of reading or writing by the process' peer. The use of wakeup mode therefore offers an economy in machine cycles executed at a cost of about 5 to 15 kbytes/s in performance.

The dependency of throughput performance upon stream buffer size exhibits, as we would expect, an improvement in throughput performance with increasing buffer size, for small buffer sizes. At buffer sizes beyond 8192 bytes there is no measurable improvement in performance. The best performance achieved with the existing implementations of *fwrite*, *fread*, *fgetc* and *fputc* is about 112 kbytes/s. The default stream buffer size was set to 4096 - *sizeof(Stream)*, which is a convenient value because it results in a page sized stream object. The throughput performance is about 85% of the maximum measured.

For purposes of comparison, the performance measurement test code was ported to 4.3BSD, using the Walnut *fwrite* (*fputc* based) algorithm. The port included compensation for Walnut debugging code delays. This code was then tested on a hardware platform identical to the Walnut testbed. Repeated measurements yielded performance between 172 and 176 kbytes/s, which is about 50% faster than the Walnut system. The test processes consumed about 80% of total system CPU time during the test.

While this test suggests that the Walnut design does not perform as well as the 4.3BSD socket based pipe, it makes no allowance for the performance of the operating system and the effect of the scheduling policy. The 4.3BSD system is a mature production system which uses a complex priority based scheduler, whereas the Walnut is a prototype using a simple round robin scheduler. A more accurate comparison could be achieved by porting the whole Walnut stream mechanism to Unix, and testing the complete mechanism. This would remove the effects of context switching performance and scheduling policy, and is an area for future study.

The throughput can be improved upon by a number of changes to the implementation of the library. The performance of *fwrite* and *fread* can be improved by replacing the prototype implementation with a true block mode implementation [Section 4.3.3.1], using *memcpy* operations to and from the stream buffer. This would remove the time overhead incurred by the function call required for each and every character transferred, as well as the time overhead incurred by updating buffer index values for each and every character transferred.

The performance of the *fgetc* and *fputc* implementations could be improved by two changes. The first change would be to implement *getc* and *putc* as C language macros, which is the convention in Unix and ANSI C. This would remove the time overhead incurred by the function call required for each and every character transferred. The second change would be to recode the algorithms into assembly code macros, which would further improve performance. The use of assembly code macros would however be at the expense of portability.

5.4 Portability

The long term intention of the Walnut project is to port the operating system to a second generation multiprocessor system. This multiprocessor is intended to use a RISC instruction set processor and system specific memory management, bussing and I/O interfaces. For this reason, portability of the library design is important.

The existing Walnut *stdio* library design and implementation contain no features or facilities which are specific to the Intel architecture used in the Walnut testbed system. All macros are written in C and no Intel assembly code is used.

The library is compiled on a Unix system (FreeBSD 1.1.5) using a GNU (gcc 2.4.5) compiler, and a standard Unix make. The library test suite [Section 5.5] allows compilation for running under Unix and well as the Walnut testbed. Running the test suite under Unix allows initial detection of possible compiler dependencies during porting, while also simplifying the task of debugging where required.

The use of generic Unix development tools, the absence of architecture dependencies in the design and implementation of the library, and the use of the test suite has resulted in a library which will be simple to port.

5.4.1 Application Development and Porting

The proof of portability lies in successfully porting programs to the intended system. In the instance of the Walnut *stdio* library, the obvious target are the standard Unix utilities. However, analysis of 4.4BSD source code suggests that only more recent utilities exploit the *stdio* library, historically older utilities are written around the Unix / POSIX 1003.1 system call interface.

For the purposes of demonstration, two Unix-like utilities were created, and two 4.4BSD utilities were ported.

A Unix-like *rm* utility was written. Whilst this utility does not support the Unix command line options, it does have the ability to remove both processes and file objects, should suitable access rights be known [Section 3.2.3.3]. This utility was found to be very useful during the latter phases of library testing, as it subsumes the functions of the Unix *rm* and *kill (-9)* utilities.

A Unix-like *cat* utility was also written. Again, this utility does not support the Unix command line options, but is written to accept its input stream from either a file or a pipe. Use in the latter mode will require a shell capable of setting up pipes between command line argument specified processes.

The first 4.4BSD utility to be ported was *head*, chosen for ease of conversion. This tool was soon followed by *uniq*, selected also for ease of porting. An attempt to port *hexdump* was abandoned due time constraints, as this tool is substantially more complex than the preceding two. Source code for *rm* and *head* is included in Appendix C, to provide examples of application development and porting, respectively.

The porting procedure requires that include files be replaced, ANSI C prototypes produced, function argument lists converted to ANSI format, Unix specific types be converted to their Walnut equivalents, and finally, *setmyname* and *clrmyname* calls added in at the beginning and end of the program respectively.

The inclusion of *setmyname* and *clrmyname* calls is a convention which should be retained. Any program or utility run from the shell will create a name binding in the format "myname-running", when it commences execution, and delete this binding once it has completed. A program which fails in a loop can be quickly identified and killed off by its user from a command line, using the *rm* utility.

5.5 Robustness

Robustness is a measure of how well a design and implementation handle error conditions and variations in input data. In a stream IPC design and implementation, robustness will be determined by the ability to provide error free transmission of data regardless of the frequency and size of reads and writes, and by the ability to handle error conditions arising from corrupted data structures and ill-behaved peer processes in a connection.

Integrity of data transmission through pipes and I/O devices is essential. Data must be transferred free of errors and in order. The stringency of this requirement was addressed by comprehensive testing, in several phases. The test algorithm, devised by Wallace, writes and reads bursts of data with pseudorandom sizes into and from the stream connection. This strategy ensures that the stream buffer logic is forced to handle repeated full and empty conditions for a wide range of buffer index positions. This test method successfully detected a number of defects in the earlier implementations of the algorithm.

The first phase of testing was carried out on a Unix system, with a single process writing and reading to and from a *Stream* structure within its address space. Once a sufficient number of error free operations was accumulated (in excess of 24 hours of operation), the design was moved to the Walnut testbed system.

The second phase of testing was carried out on the Walnut testbed, using a client server pair of processes. The client process executed the test algorithm, while the server provided a simple transparent loopback.

The third phase of testing was an extension of the second phase. The *ungetc* algorithm was testing by inserting an *ungetc* and *fgetc* operation into the existing test code. This testing phase was then extended to accumulate two weeks of uninterrupted error free operation.

Robustness in the library implementation was achieved by several means. The algorithms used in library functions were designed with very frequent integrity checks on the capabilities operated upon, and the *Stream* structures contained therein.

A suite of validation tests was written to test most library functions. This suite, designated *validate*, may be compiled and run under Unix, or compiled as a program which may be run on the Walnut testbed. The validate test suite was incrementally extended as additional library functions were produced. Each function was tested under Unix and the Walnut testbed to ensure that it was error free. In addition, a convention adopted during development was that any change to a function or addition of a function would not be considered complete until the validation suite was shown to work error free for all functions in both environments. This proved to be a wise strategy, as changes to a number of the low level library functions late in the project did indeed introduce bugs which would not have been found without the comprehensive test method adopted. The validate suite contains the following twentysix tests, which are described in [Appendix D]:

0. perror integrity test
1. opencap create object test
2. fprintf/putc write object test
3. write exclusion test
4. ferror test
5. clearerr test
6. file close test
7. opencap read mode test
8. file read test
9. feof test
10. fseek/ftell test
11. fsetpos/fgetpos test
12. file append test
13. file update test
14. removec/kerror test
15. copen create test
16. copen write test
17. copen update test
18. tmpfile test
19. stream fflush test

20. fopen create test
21. fopen read test
22. fopen update test
23. remove test
24. ungetc pushback test
25. fscanf test
26. makestreamobj test

The adoption of rigourous testing techniques has therefore yielded a robust library design and implementation.

5.5.1 Persistence

Any discussion of robustness in the Walnut context must include the subject of persistence. One of the properties of the Walnut kernel is that all objects are persistent. If a Walnut system is shut down, upon a restart all processes active at the time of shutdown will resume execution. Because any object in the system's memory is an image of the object on disk, all processes may be cleanly restarted. Providing that all changes to the object are flushed to disk during a shutdown or crash, a restart should be transparent to a user.

A useful side effect of this property is that the Walnut stream IPC mechanism is also persistent, and may be restarted. This is because all state information pertaining to a stream is held within the process object, the process data object, and the stream object itself. If these objects are successfully flushed to disk during a shutdown or crash, all state information is saved. This characteristic is not shared by established operating systems such as Unix, where stream state information is perishable and lost during a shutdown or a crash.

The persistence property was successfully demonstrated during the testing and debugging of the Walnut *stdio* library. A client server process pair running a stream test restarted after a shutdown.

5.6 Security

The Walnut kernel virtual memory system provides a secure environment for a process. The address space of the process is protected from access to third parties not holding a capability to access the process.

The stream IPC mechanism employed in the Walnut does not compromise the inherent security afforded by the kernel. The only area of the process address space which may be accessed by another process is the stream buffer, which is only accessible to a peer process using the same connection. Because the stream buffer object is unloaded and destroyed upon the closing of the connection, a third party cannot use it to gain access to the address space of either process.

5.7 Future Directions

There exist a number of areas in which the Walnut *stdio* library design and implementation can be improved and extended.

The *fgetc*, *fputc*, *fwrite* and *fread* implementations can be made to run more efficiently [Section 5.4]. The utility of the *ungetc* function can be improved by making the **NPUSHBACK** parameter dynamically configurable during stream opening [Section 4.3.2.2]. The formatted character I/O functions *fprintf* and *fscanf* can be extended to provide full conformity with the ANSI standard, and the ANSI *v*- and *vf*- versions of these functions implemented. A number of infrequently used ANSI functions which are partially implemented or untested can be completed.

The *fopen* programming interface can be extended to provide flexible parameters for file object or stream opening [Section 3.2.3.2]. The Client Server protocol modules used in stream opening could benefit from more flexible decision logic for decoding requests.

While the existing Client Server protocol has provisions to enable its use for RPC operations, an RPC programming interface is neither defined nor implemented. Given the availability of source code for the ONC RPC protocol [SMI], it would be an obvious candidate for porting and integration with the Walnut design.

A project derived from the first multiprocessor design was a port of the 4.3BSD system call interface to the password capability kernel. Should a similar port be intended for the Walnut, it could benefit significantly from the existing Walnut *stdio* library. A library conforming to the POSIX 1003.1 or SVID interface definitions could be implemented quite efficiently by using existing low level functions in the *stdio* library, and deriving many other functions from their ANSI equivalents. A Unix or Unix like interface library would not be complete without a BSD socket library. A socket library could be easily implemented using components of the existing *stdio* library.

A case can be made for porting the Walnut stream mechanism to Unix, exploiting the *mmap* interface, to provide fast and simple user level pipes. This could also provide a good comparison of achievable throughput performance between the two mechanisms [Section 5.3].

The presence of low level functions to support RPC operations provides the foundation for building a more complex object oriented client-server programming interface, as is characterised by OMG CORBA [OMG91]. The implementation of such an interface is however a substantial undertaking within itself.

The existing Client Server protocol could also be exploited to provide for a unified command and status programming interface to device manager processes. This interface could emulate the POSIX 1003.1 *ioctl* call, which is used by a process to alter the configuration of a device or a stream. The existing programming interface between a user process and a device manager process uses either a device specific programming interface, or the *fopen/opencap* interface which has no provisions for altering device configuration once the stream is open.

Finally, the presence of a working name based stream opening protocol will allow improvements to the existing implementation of the *GLui* console manager, the Walnut *Shell* and the floppy disk manager. All of these programs use very little if any part of the Walnut *stdio* library and could benefit from a rewrite to take advantage of the library’s features.

Chapter 6 Conclusion

The Walnut kernel provides a virtual memory system in which objects are accessed through capabilities. The kernel does not provide a programming interface for I/O devices and stream IPC.

An I/O library and stream IPC mechanism for the Walnut system has been designed and implemented. This library provides a programming interface through which users can access the virtual memory system, I/O devices and other processes through a stream connection.

The programming interface mostly conforms to the ANSI C language standard. File objects may be manipulated in the same fashion as files in operating systems such as Unix. Stream connections to other processes may be opened using the ANSI *fopen* syntax. Processes are identified by names, thus providing a simple and elegant programming interface, common to files and streams.

The stream IPC mechanism uses a circular buffer, which resides in a stream object. The stream object is mapped into the address space of both processes in the connection, and is destroyed upon the closing of the connection. The design allows a user to push back multiple characters on to the stream.

A common set of library routines and data structures is used for both file objects and stream connections. Differences between files and streams are accommodated by overloading members in data structures, and by the use of alternate logic in the algorithms used.

The library is implemented in two layers. The lower layer comprises functions which operate directly upon capabilities, and provides a number of simple utility and debugging functions. The upper layer provides the ANSI standard programming interface, and utilises, where appropriate, lower level functions.

The protocol for opening stream connections between processes has been designed to accommodate stream connections and RPC requests.

The performance of the stream IPC mechanism has been measured, and the library implementation tested with a suite of validation programs. A number of Unix-like utilities were written, and a number of 4.4BSD utilities ported using the library.

We have proposed that the library be extended to provide a programming interface for POSIX functions and BSD sockets, and that the name based stream opening syntax be further exploited in the design of shells and client server applications.

The Walnut *stdio* library design has provided the means of demonstrating a number of programming interface and functional features which are not commonly used in established operating systems.

The shared use of names for accessing files and processes is the foremost such feature. At the programming interface level, this significantly simplifies the programming interface to device manager processes handling I/O, to server processes providing user specific or system wide services, and to any future RPC or object management mechanism. Moreover, at the command line interface level, this mechanism allows a user to monitor and if necessary manipulate the activity of running processes by simply listing the bindings in the current working set or directory. Attempts to provide similar functionality in Unix are demonstrably cumbersome.

Because a server process providing either a user specific or system wide service can be identified and accessed from a command line, this feature can be further exploited in the design of new shells, built around a client-server model. Such shells can provide a rich set of features without the performance penalties associated with running individual commands as separate processes. This is an area for future research.

The name based access model for file and stream opening can be extended further. An area worth further study is that of using the *fopen* interface to create processes. The existing model does not exploit the Walnut type identifier in a capability, and treats code objects identically to file objects. An *fopen* on a code object could therefore be made to result in the creation of a process running this code object. This is yet another area for future research.

The Walnut IPC mechanism exploits the persistence properties inherent in the Walnut virtual memory system. Streams retain all state information in objects which are persistent. As a result of this, a stream connection may be cleanly stopped and restarted through a Walnut shutdown and boot. This is a characteristic which is not available in conventional systems.

The properties inherent in the Walnut IPC mechanism also enable the redirection of running streams. Redirection of streams would be particularly useful for command line operations by a user, who can then interrupt the operation of a program to redirect its input or output to another source or sink. A shell design which can exploit this behaviour is another area for future research.

The provision of an ANSI standard programming interface to a password capability virtual memory system provides a programmer with a well known interface to an unconventional kernel. The library design maps the behaviour of the Walnut kernel into ANSI calls without compromising the properties of the Walnut virtual memory system. Programs may thus be developed for the Walnut with a minimum of effort expended, as the complexities of the Walnut system call interface are effectively hidden from the programmer. This will make the Walnut a more attractive target for researchers, as it combines ease of use with a powerful virtual memory architecture.

The Walnut *stdio* library and its embedded IPC mechanism provide the means for further productive research into the area of password capability systems.

References

ABRAMSON82

Abramson D.A., Computer Hardware to Support Capability Based Addressing in a Large Virtual Memory, PhD Thesis, Department of Computer Science, Monash University, 1982

ANSI89

ANSI XJ311 Committee, Rationale for American National Standard for Information System - Programming Language - C, ANSI, 1989.

ANDERSON87

Anderson, M., A Password Capability System, PhD Thesis, Department of Computer Science, Monash University, January 1987

APW85

Anderson M., Pose R.D., Wallace C.S., A Password Capability System, Technical Report No.52, Department of Computer Science, Monash University, March 1985

APW86

Anderson M., Pose R.D., Wallace C.S., A Password-Capability System, The Computer Journal, Vol. 29, No. 1, 1986.

BSD44

4.4BSD Lite Source Code Tree, University of California, Berkeley, published as FreeBSD 2.0.5 Release, Walnut Creek CDROM, Walnut Creek, 1995.

CASTRO95

Castro M., The Walnut Kernel: User Level Programmer’s Guide, Technical Report No.95/222, May 1995

CATHRO88

Cathro, D., An I/O Subsystem for a Multiprocessor, MSc Thesis, Department of Computer Science, Monash University, January 1988

CP94

Castro M., Pose R.D., The Monash Secure RISC Multiprocessor: Multiple Processors without a Global Clock, Australian Computer Science Communications, Vol. 16, No. 1, 1994, pp. 453-459.

CRAWFORD87

Crawford J.H., Gelsinger P.P., Programming the 80386, Sybex, 1987

DRAVES91

Draves R., A Revised IPC Interface, Working Paper, Mach Project Group, CMU Dept of Computer Science, 1991.

GEHRINGER82

Gehringer E.F., MONADS: A Computer Architecture to Support Software Engineering, MONADS Report No.12, Department of Computer Science, Monash University, January 1982

GOODHEART94

Goodheart B. and Cox, J., The Magic Garden Explained - The Internals of Unix System V Release 4, Prentice-Hall, 1994.

IEEE90

IEEE, Information Technology Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language], IEEE Standard 1003.1-1990.

INTEL84

iAPX386 High Performance 32-Bit Microprocessor Product Preview, Intel Corporation, Santa Clara, 1984

KSU94

Krieger O., Stumm M., Unrau R., The Alloc Stream Facility - A Redesign of Application Level I/O, IEEE Computer, March 1994. Also Technical Report by same authors, Computer Systems Research Institute, University of Toronto, 1994.

LMKQ89

Leffler S.J., McKusick M.K., Karels M.J., Quarterman J.S., The Design and Implementation of the 4.3BSD UNIX Operating System, Addison-Wesley, 1989.

LFJLMP

Leffler S.J., et al, An Advanced 4.3BSD Interprocess Communication Tutorial, Computer Systems Research Group, Dept of EE and CS, University of California, Berkeley.

MCKUSICK94

McKusick M.K., UNIX Kernel Internals: Data Structures, Algorithms, and Performance Tuning, Course Notes, Australian UNIX Users Group, Summer, 1994.

NS85

National Semiconductor Corporation, Series 32000 Databook, National Semiconductor Corporation, Santa Clara, June 1985

OMG91

Object Management Group, The Common Object Request Broker: Architecture and Specification, OMG Document Number 91.12.1, Revision 1.1, Draft 10 December, 1991.

ORGANICK

Organick, E.I., The Multics System: An Examination of its Structure, The MIT Press, Massachusetts, 1972.

OSF93

Open Software Foundation, Design of the OSF/1 Operating System, P T R Prentice-Hall, New Jersey, 1993.

PLAUGER92

Plauger P.J., The Standard C Library, Prentice Hall, New Jersey, 1992.

POSE89

Pose R.D., Capability Based Tightly Coupled

Multiprocessor Hardware to Support a Persistent Global Virtual Memory, Proceedings of the 22nd Hawaian International Conference on System Sciences, Vol. 2, pp. 1-10., 1989.

POSE93

Pose R.D., Porting Unix to the Password-Capability System, submitted to the First International Workshop on Architectural and Operating Support for Persistence, 1993.

PRINGLE95

Pringle G., Walnut User System Documentation, Draft Technical Report , Department of Computer Science, Monash University, December 6, 1995

RITCHIE93

Ritchie D.M., The Development of the C Language, Proceedings of the Second History of Programming Languages conference, ACM, Cambridge, Mass., 1993.

ROZIER91

Rozier M., et al, Overview of the CHORUS Distributed Operating Systems, Technical Report CS-TR-90-25, Chorus systemes, 1991.

SMI

Sun Microsystems, Inc, RPC Programming, Network File System: Version 2 Protocol Specification

WALLACE90

Wallace C.S., Physically Random Generator, Computer Systems Science and Engineering 5, 2, 82-8, 1990.

WP90

Wallace C.S., Pose R.D., Charging in a Secure Environment, Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information, Bremen, FRG. 1990., pp. 24-1..24-11.

Appendix A Release Notes

The Walnut *stdio* Library

RELEASE NOTES V1.1

Carlo Kopp
carlo@cs.monash.edu.au
20th March, 1996

1. Introduction

The Walnut kernel *stdio* library provides standard C language *stdio* library functions and an environment for user processes running on the Walnut microkernel. The library provides most ANSI standard *stdio* functions, although some functions are not fully featured. The ANSI specific v- and vf- versions of the formatted print and scan functions are not implemented, as are the linear buffer management functions.

The design philosophy of the library was to provide ANSI compliance, where such compliance did not incur substantial additional development effort over a K&R C library.

Central features of the *stdio* library are the provision of preloaded and initialised *stdin* and *stdout* I/O streams for user processes, the provision of a stream communication mechanism for Inter Process Communications, unified stream and file object structures, a family of ANSI-C and POSIX like functions for capability operations, and support for the Walnut kernel nameserver library. The library includes embedded debug message reporting, accessible by an application programmer.

2. Process Environment

The Walnut kernel provides a simple process environment which does not provide explicit support for *stdio* functions, or stream communications. These functions are provided by the *stdio* initialisation module, **initenv()**. The **initenv()** call sets up the environment for the *stdio* library within the process address space.

The process environment provides the following facilities:

- file pointer (descriptor) table

- global environment variables used by the library at runtime

- a debug environment variable

- support for screen debug messaging

- nameserver library initialisation

- initialised *stdin* and *stdout* file pointers

The **initenv()** module must always be linked in with the *stdio* library, as without it the library will be unable to function. The module is called with a conventional C-language argument list:

initenv(int argc, char **argv, char **envp);

These arguments provide the parent process with the means of passing specific arguments down to the child process. This allows programs such as shells to execute commands out of process, by creating child processes which execute programs with specified argument lists.

2.1 The *FILE* Table and the Initialisation of *stdin, stdout* Descriptors

The file table is an array of **FILE (File)** structs. Each struct contains the facilities which allow a *stdio* stream I/O function to read and write a shared stream object or a file object. Each opened file object or stream object requires a **FILE** struct and is accessed by passing a pointer to the **FILE** struct as the stream argument of the function call.

When the process is created the file table is also created at a fixed address in the process address space, and the stream object capabilities for *stdin* and *stdout* are preloaded into the appropriate fields. The stream objects for *stdin* and *stdout* are initialised by the parent process.

The **initenv()** module completes the initialisation of the file table, by performing the following tasks:

- setting up the linked list pointers used during **fopen()** and **fclose()** to maintain the file table free list

- removing **fdt[0]**, *stdin*, and **fdt[1]**, *stdout*, from the free list

• executing a low level **fileinit** call on *stdin* and *stdout* to complete the stream opening process, thus enabling their use for I/O

2.2 Global Environment Variables

The *stdio* library employs a number of global environment variables. These are private to the process and cannot be read by other processes unless the process makes them readable to another party. These variables are initialised by the **initenv()** module, and may be accessed by programs executing in the *stdio* process environment. The variables are described as follows:

• **Uw errno**, ANSI global error flag used by library calls

• **Uw iomode**, specifier for default I/O blocking mode

• **Uw max_ld_cap**, maximum number of loaded capabilities

• **Uw max_mail_box**, maximum number of mailboxes

• **Uw max_subp**, maximum number of subprocesses

• **Uw max_auto_ld_cap**, maximum number of auto loaded capabilities

• **nameserverDB mydb**, name server database object to be loaded

• **Capl nullcap = { 0, 0, 0, 0 }**, constant null capability

• **Capl mycap**, process capability id

• **Param mypb**, process parameter block

• **char *screen**, pointer to **debug** buffer

• **Uw debug**, debug level, used to enable verbose error reporting

• **int argc**, the number of command line arguments

• **char **argv**, array of pointers to command line argument strings

• **char **envp**, array of pointers to environment variables for process

These variables should not need to be used in the course of application programming, but may be useful for system programming tasks, and for custom extensions to the *stdio* library.

2.3 Nameserver Library Initialisation

The default nameserver database capability is loaded into the process address space at creation time, by the parent process. The **initenv()** module will initialise the nameserver library, with a **initDB(&mydb, envp)** call, using the default database and the process' preloaded environment variables.

Should the user require the use of a different nameserver database, the **initDB()** must be repeated with the appropriate arguments.

3. ANSI C *stdio* Functions

The Walnut kernel *stdio* library implements a substantial subset of the ANSI/ISO X3.159-1989 (X3J11) C language *stdio* library. As the original C standard I/O library was written around the Unix operating system, many aspects of the ANSI library reflect its origins, such as the stream operations functions. In the Walnut kernel implementation, most of the behaviour specified in the ANSI standard is replicated, although some differences do exist.

As the ANSI standard functions are comprehensively documented both in hard copy and the BSD manual pages, this document will concentrate on known differences between the Walnut kernel library and ANSI standard.

The most substantial differences at a functional level arise from the fundamental differences between the Walnut kernel's memory mapped object scheme, and the traditional Unix I/O mechanism. The conventional Unix filesystem object scheme provides access to objects via stream operations which are embedded in the monolithic kernel, and the kernel buffers accessed blocks in memory as required. In the Walnut kernel stream I/O functions are provided within the library, and objects are accessed by loading them into a process address space.

In the Walnut process environment the *stdio* library implementation thus subsumes a number of functions which users may be accustomed to finding in traditional monolithic kernels.

3.1 Stream Operation Functions

FILE *fopen(char *filename, const Uq *mode);
Sw fclose(FILE *stream);
Sw fflush(FILE *stream);
Sw rename(const char *oldname, const char *newname);
Sw remove(const char *filename);
FILE *tmpfile(void);
char *tmpnam(char *s);
FILE *freopen(char *filename, const char *mode, FILE *stream);
Sw setvbuf(char *file, char *buf, Sw mode, Uw size);
Sw setbuf(char *file, char *buf);

The **fopen()** call is a compliant implementation of the ANSI standard function call, and will open either a file object or an I/O stream in read, write or with file objects, also update mode, subject to flag usage. The update mode is not defined for I/O stream objects, and thus can only be used for file objects.

The current implementation of the library supports only *unbuffered binary* streams, as defined by the ANSI standard. This provides transparent transport for both binary and text streams, with no intervening buffering between the object and the calling function (a future implementation may support line buffering of text I/O streams on receive).

The **fopen()** call recognises the following flag types:

• **r** - read only flag, the stream can only ever be read

• **w** - write only flag, the stream can only ever be written

• **w** - append flag, write mode with the file pointer positioned to the end of file position

• **+** - update flag, the file object can be read or written. A file positioning function or **fflush()** call must be interposed between consecutive read and write, or write and read operations

When handling I/O streams, the **fopen()** call is given a name which maps into the capability to message a process. If the process does not respond to or rejects the request to open the stream, the call will return a null pointer and set the global error code. The use of **perror()** is recommended, when using **fopen()**. The value of the default volume for object creation is set using the *setvol* macro. The **fclose()** call is ANSI compliant.

The **fflush()** function differs in its behaviour from the ANSI standard, which assumes the use of linear buffers. The **fflush()** function exhibits the following behaviour:

• file objects - in write, append or update mode the file size value of the object is updated to the current value of the file pointer

• stream objects - in write mode, the **fflush()** call will wait until the buffer is emptied by the reading process, and then return.

The remaining file management functions are ANSI compliant, the buffer management functions are dummy functions included for compatibility.

3.2 Character I/O Functions

Sw fgetc(FILE *stream);
Sw fputc(Sw c, FILE *stream);
Sw ungetc(Sw c, FILE *stream);
char *fgets(char *buf, Sw BUFSIZ, FILE *stream);
Sw fputs(char *buf, FILE *stream);

The character mode I/O functions are designed to provide ANSI compliant behaviour for stream and file object types. At this time, the **getc()** and **putc()** calls, traditionally implemented as macros, are implemented as aliases to the **fgetc()** and **fputc()** function calls, and thus will not exhibit a performance advantage as with conventional implementations.

The **ungetc()** function is substantially enhanced against the minimal requirements of the ANSI standard, and provides guaranteed pushback of up to **NPUSHBACK** characters, where **NPUSHBACK** is a compile time parameter defined in the *filedefs.h* include file.

3.3 Direct I/O Functions

size_t fread(void *array, size_t elementsize, size_t count, FILE *stream);

size_t fwrite(void *array, size_t elementsize, size_t count, FILE *stream);

The direct I/O functions emulate the behaviour of the ANSI standard, although the current implementation uses character mode I/O for stream access, and thus will not exhibit an advantage in transfer rate performance over the character I/O functions.

3.4 Formatted I/O Functions

Sw fprintf(FILE *stream, const char* fmt, ...);
Sw fscanf(FILE *stream, char* fmt, ...);

The formatted I/O functions provide a partial implementation of the ANSI standard functions. The **fprintf()** function at this time does not support floating point arguments, and thus accepts only **%d, %x, %c** and **%s** type arguments. The **fscanf()** function was ported from BSD 4.3 source, and thus implements commercial standard K&R functionality.

3.5 FILE Positioning Functions

Sw fseek(FILE *stream, Sw offset, Sw lastoffset);
Sw ftell(FILE *stream);
Sw fgetpos(FILE *stream, Uw *pos);
Sw fsetpos(FILE *stream, Uw *pos);

The file positioning functions have defined behaviour only for file objects, as with most *stdio* library implementations. Due to the memory mapping of objects into the process address space, these functions will not return a meaningful EOF indication if the mapped window into process memory is smaller than the object size. Where the mapped size is larger than or equal to the object size proper, these functions will exhibit nominal ANSI behaviour.

3.6 Error Handling Functions

Sw feof(FILE *stream);
Sw ferror(FILE *stream);
Sw clearerr(FILE *stream);
void perror(const char *s);

The error handling functions all exhibit nominal ANSI standard behaviour. Use of the **perror()** function requires the inclusion of the *werrno.h* file.

4. Walnut Capability Functions

The Walnut *stdio* library contains a number of extensions to the ANSI *stdio* library suite. These extensions provide for ANSI like and POSIX (IEEE 1003.1) like functions which operate directly on capabilities, and thus do not require use of the nameserver library. Where an application program directly manipulates capabilities, these functions should be used in preference to ANSI functions, as the capability function library extensions are more efficient.

4.1 The opencap() Function

FILE *opencap(Capl *capability, Uw flags, Uw vol, Uw money, Uw limit);

The **opencap**() function is directly analogous to the **POSIX.1** and **Unix open**() system call, but takes a pointer to a capability instead of the file name argument. The argument list is defined as follows:

- **Capl *capability**, pointer to file object or target process object capability

- **Uw flags**, flags for opening, or default object creation

The following flags are defined for **opencap**():

○ **READ** - open the capability for reading

○ **WRITE** - open the capability for writing

○ **APPEND** - open the capability for writing, and advance the pointer to EOF

○ **UPDATE** - open the capability for update, i.e. read and write (file only)

○ **CREATE** - if capability doesn’t exist, create on open

○ **BLOCK** - stream I/O mode is blocking

○ **NOBLOCK** - stream I/O mode is non-blocking

○ **WAKEUP** - stream I/O mode is wakeup

○ **SMALLWINDOW** - object created with 4 kbyte size, overrides limit argument

○ **LARGEWINDOW** - object created with 4 Mbyte size, overrides limit argument

- **Uw vol**, specified volume for object creation

- **Uw money**, specified money for object creation

- **Uw limit**, specified limit for object creation

The **opencap**() call will first check its arguments, then execute a **CAPSTAT** system call to confirm the state of the capability. If the capability doesn’t exist and the **CREATE** flag isn’t set, **opencap**() returns a null pointer, else it creates a file object with parameters defined by the argument list, and initialises the object header. If the capability does exist, and is a file object, then **opencap**() tests its access rights against the flags in its argument list, should these be inconsistent, **opencap**() returns a null pointer.

If the capability is a process object, **opencap**() will attempt to open an IPC stream to the nominated process, with parameters determined by the argument list. If the stream cannot be opened, **opencap**() returns a null pointer. The limit parameter is overloaded to define the requested stream size. Within the range of object sizes supported, any stream buffer size may be requested.

The **opencap**() function will then fetch a descriptor from the file table, returning a null pointer if the table is full or in a erroneous state. The descriptor number is then passed to the low level **fileinit**() function, which initialises the file descriptor (pointer) and loads in the object.

The **opencap**() call returns a valid **FILE** pointer if successful, or a null pointer if in error. The error condition may be analysed using **perror**().

4.2 The copen() Function

FILE *copen(Capl *cap, const Uq *mode);

The **copen()** function is analogous to the ANSI standard **fopen()** function, but takes a pointer to a capability as a first argument, rather than a pointer to a filename string. The mode flags used are identical to those used by fopen(). The **copen()** function returns a **FILE** pointer if successful, or a null pointer if unsuccessful.

4.3 The removec() Function

Sw removec(Capl *capability);

The **removec()** function is analogous to the ANSI standard **remove()** function, but takes a pointer to a capability as its argument. The **removec()** function returns 0 if successful, or -1 if unsuccessful.

4.4 The cmap and cunmap Functions

The *cmap* and *cunmap* functions are analogous to the Unix and POSIX *mmap* and *munmap* system calls. Unlike the Unix calls, *cmap* and *cunmap* operate on a specified capability rather than a file pointer. The function is included to simplify porting of applications and is not used in the library implementation.

void *cmap(void * addr, Uw len, Uw prot, Uw flags, Capl *cap, Uw offset);

- **addr** - address at which the capability is to be loaded

- **len** - specifies the length of the view to be loaded

- **prot** - unused in Walnut

- **flags** - unused in Walnut

- **cap** - pointer to capability to be loaded
- **offset** - offset to base of view to be loaded

void cunmap(void *addr, Uw len);

- **addr** - address of the mapping to be unloaded
- **len** - unused in Walnut

The *cmap* and *cunmap* function implementation is not compliant with the POSIX function.

4.5 The setmyname and clrmyname Functions

Sw setmyname(char * myname);
Sw clrmyname(char * myname);

These functions are used to bind and unbind names to and from the calling process. The sole argument is a pointer to a string. A typical use is to simplify the design of client processes, which may locate a server by using a name rather than a capability value.

4.6 The setvol macro

setvol(Uw volume);

The *setvol* macro is used with the ANSI *fopen* function. It sets the global value of the default volume for file or stream creation. If *setvol* is not invoked before an *fopen* is called, *fopen* will fail.

5.Error Reporting Facilities

The Walnut kernel *stdio* library provides comprehensive error reporting facilities for the debugging of applications. In the existing implementation, there are three levels of debug error reporting.

The first of these is enabled by setting the global variable **debug = 1**, and reports library error messages using the ANSI style **perror(const char *function)**.

The second level of debug reporting provides abbreviated Walnut kernel kernel error code reporting, and is enabled by setting the global variable **debug = 2**.

Full Walnut kernel kernel error reporting is then enabled by using the third debug reporting level, which is enabled by setting **debug = 3**.

Default operation is at **debug = 0** which is effectively a library reporting silent mode, where the user has the option of independently using the ANSI **perror()** reporting function.

An example of using the error reporting follows:

```
/*
 * enable full error reporting for the fopen() function, then disable it
 */
debug = 3;
fd = fopen("blogs","r");
fwrite( buf, 4, 4096, fd );
fclose( fd );
debug = 0;
```

The debug facilities should be used selectively, as the *stdio* library functions, in the course of operation, will often execute kernel calls which fail, such as **CAPSTATs** on non-existent file objects. If used indiscriminately, error reporting will clutter the user display with irrelevant messages.

5.1 The kerror() Function

void kerror(const char *s);

The **kerror()** function provides Walnut kernel kernel error reporting in terse and verbose modes. Its function is analogous to the ANSI **perror()** function, in that it accepts a single string argument to identify the location of the error, and then prints either a numerical error code, or a numerical error code and one line listing of the error message. The *kerror.h* file must be included.

NB: this library call uses the returned value of the parameter block error field **parameter->error** to identify the error state. This field cannot be zeroed until the **kerror()** call has returned.

5.2 The KERROR() Macro

KERROR(function,err)

The **KERROR** macro provides a packaged invocation of perror and kerror, and some trivial screen positioning to provide an easily readable error report. Usage of the **KERROR** macro is analogous to the concurrent usage of **perror**() and **kerror**(), with **function** used to identify the name of the calling function (i.e. location of the failure), and **err** set to the value of the returned error code. The *kerror.h* file must be included.

NB: the error code is not the returned value from a *stdio* library function, which is defined by the ANSI standard. **KERROR** should be used when coding directly with Walnut kernel system calls. The following internal error codes are defined in the Walnut kernel *stdio* library:

EBADFDINDEX - bad file table index
ENULLCAP - null capability
EBADMAGIC - bad magic number
EBADDIRN - bad stream direction
EBADLOAD - object load failed
EBADBUFOBJ - bad buffer object
EBADCAP - bad capability
EBADMAKE - make object failed
EBADRIGHTS - rights masks inappropriate
EBADINDEX - bad index
EFDTFULL - file table full
EMSGTMOUT - message send timed out
EBADARGS - open args inconsistent
ENSERROR - name server error
ENOEXIST - capl doesn’t exist
EKERNEL - kernel error

6. Utility and Debugging Functions

The *stdio* library provides a number of utility and debugging functions, which may be productively used for system programming tasks involving the library. These functions provide for formatted display of file descriptor structs, object headers and general screen output bypassing the *stdin* and *stdout* paths.

6.1 Debug I/O Display Functions

```
void outs(char *string);
void outh(Uw h);
void outi(Sw i);
void newline(void);
void outc(int c);
int  getx();
int  gety();
void gotoxy(int x, int y);
void clrscr();
```

The debug screen debugging facility allows processes to display messages on a character mode display, wholly bypassing the *stdio* stream communication channel. The debug screen uses a separate frame buffer to the console display. This can be of use should difficulties be encountered with library operation. Enabling the screen debug facility requires recompilation of the library with the **-DDEBUG** flag.

The **outs()** function takes a pointer to a string of chars, and prints these to the debug monitor. This function cannot display a newline character, unlike ANSI I/O functions.

The **outh()** function takes an unsigned integer and prints it to the debug monitor.

The **outi()** function takes an signed integer and prints it to the debug monitor.

The **newline()** implements a return-newline sequence on the debug monitor.

The **outc()** function takes a character and prints it to the debug monitor.

The **getx()** function returns the x location of the cursor on the debug monitor.

The **gety()** function returns the y location of the cursor on the debug monitor.

The **gotoxy()** function sets the position of the cursor on the debug monitor.

The **clrscr()** function clears the debug monitor and moves the cursor to the top left of the monitor.

6.2 Debug Display Formatted Stream Functions

```
void printfd(FILE *fd);
```

void printobj(Stream *obj);

The formatted stream display functions are debugging tools which dump the formatted contents of a **Stream** (stream / file object header) and **FILE** (File table entry) to the debug monitor. This can be of use when debugging precludes access to the *stdout* I/O stream. Both functions take pointers as arguments.

6.3 Formatted Structure Display Functions

void fprintpb(Param *pb);
void fprintcap(Capl *cap);
void fprintfd(FILE *stream);
void fprintobj(Stream *streamobj);
void fprintncd(struct namecdata *ncd);

The formatted structure display functions provide the formatted display of key *stdio* library structures, for debugging purposes. Output from these functions is directed to the *stdout* output stream. All functions take pointers as arguments.

The **fprintpb()** function displays the contents of the specified Walnut kernel kernel parameter block. The **fprintcap()** function displays the contents of the specified capability. The **fprintfd()** function displays the contents of the specified file descriptor. The **fprintobj()** function displays the contents of the specified stream or file object header. The **fprintncd()** function displays the contents of the specified name server database entry.

6.4 The fileinit() Function

Sw fileinit(Uw fdindex, Uw *offset, Uw *cindex, Uw block, Uw direction);

The **fileinit**() function provides low level operations used during the final phase of opening a stream or a file object. The argument list is defined as follows:

- **fdindex** - index into the file descriptor table

- **offset** - specifies capability offset and returns offset value

- **cindex** - specifies capability cindex and returns cindex value

- **block** - specifies blocking, nonblocking or wakeup I/O mode

- **direction** - specifies read or write direction for stream or file objects

The **fileinit**() call will first check the index provided and the volume number for a null volume, returning -1 if these are not usable. If they are usable, **fileinit**() will execute a **CAPSTAT** system call to verify the state of the capability, and then load the capability into the process address space at the specified offset. If the offset is zero, the argument value is overwritten with the returned offset.

Once the object is loaded, **fileinit**() will check the magic number for a **FILESTREAM** (I/O stream object) or **FILECHAR** (file object) value, the base size against the header size, the total object size against the mapped in window size and the object index values, returning -1 if in error. Once the integrity of the object is confirmed, **fileinit**() will proceed to initialise the remaining fields in the object header and the file descriptor (**FILE** pointer).

On successful completion, **fileinit**() returns 0.

7. Build Environment

Building executable targets which can be run on the Walnut kernel system requires a Walnut kernel build environment. This environment comprises the *libstdio.a* library, the *libnameserv.a* and a *Makefile* derived from the *Makefile.tmpl*

The *Makefile* must be suitably modified to include the modules intended to be linked with the library. The result of a successful compilation and link will be code and data object files, *myfile.cbn* and *myfile.dbn* respectively, which can be directly loaded and run on the Walnut kernel.

All files must include the *stdio_c.h*, *filedefs.h* and *stdfiles.h* include files, which contain essential function prototypes, macros, aliases and defines. Should error reporting be required, the *kerror.h* and *werrno.h* files must also be included.

Appendix B Source Code (*fputc, fgetc, ungetc*)

```
/*
 * wfputc.c - put character system library call
 * Monash multi Intel version
 * Author: Carlo Kopp
 * Created: 31 May, 1994
 * Modified: 1st June, 1994 - revised design
 * Modified: 4th July, 1994 use CSW file descriptor
 * Tested: 6th October, 1994 Carlo/Maurice
 * Modified: 19th October, 1994 Chris Wallace (revised trip, new algorithm)
 * Modified: 17th March, 1995 Carlo Kopp - pushback buffer added
 * Tested: 17th March, 1995 Carlo Kopp
 */

/* $Id: thesis.ms,v 1.1 1996/02/28 01:20:13 walnut Exp walnut $ */

#include <string.h>
#include <funtype.h>
#include <param.h>
#include <stdfiles.h>
#include <filedefs.h>
#include <cap.h>
#include <request.h>

#ifdef UNIX
#include <stdio.h>
#else
#include <stdio_c.h>
#endif

#ifdef UNIX
extern Param    localpar;
extern File     localfdt[];
extern Uw       localmesg[];
#define NPUSHBACK 8
#endif

/*
 * The circular buffer scheme used by the Multi stdio library makes the
 * following assumptions about the buffer indices:
 *
 * inindex (write) may assume values between 0 and (strmsz - 1)
 *
 * outindex (read) may assume values between 0 and (strmsz - 1)
 *
 * The buffer-empty state is inindex = outindex
 *
 * The buffer-full state is EITHER
 *
 * inindex = outindex - 1 - NPUSHBACK
 *
 * OR
 *
 * inindex = strmsz - 1 ,  outindex = NPUSHBACK
 *
 * OR
 *
 * inindex = strmsz - 1 - ( NPUSHBACK - outindex )
 *
 * NPUSHBACK is a compile time parameter which sets the size of the
 * buffer pushback zone. NPUSHBACK < strmsz - 1, and in practice is the
 * line size for line buffered character mode, as per ANSI C standard
 * Typical NPUSHBACK values are << strmsz, examples would be
 * strmsz = 4096 ; NPUSHBACK = 256 , strmsz = 64k ; NPUSHBACK = 2048
 *
 * NB NPUSHBACK is always adjusted by the value of stream->pushback to
 * prevent backward creep of the tripwire with ungetc() calls
 */

/*
 * Unix test environment requires NPUSHBACK < 39
 */
#if defined(TESTIO)
#define NPUSHBACK 16
#endif

Sw
wfputc(Sw data, File * stream)
{
	Param		*par;
	Sw		inindex, outindex = NPUSHBACK;
#ifndef	UNIX
	extern Uw	lineNumber;
#else
	Uw		lineNumber;
#endif

#ifdef BLOGS
#define LN lineNumber=__LINE__ + 0x30000
#else
#define LN
#endif

	extern Uw	debug, *the_wall;

	/*
	 * This is the common code section executed on every pass. The _RDOK
	 * flag is cleared to protect from a following read. The trip test
	 * determines whether a special case exists, in which event case
	 * specific code is executed. retry and writeok are specific entry
	 * points used by the special case handlers.
	 */
```

```
retry:
        stream->flags &= (~_RDOK);
        LN;
        if (stream->myindex >= stream->tripwire)
                goto tripped;
        LN;
writeok:
        (stream->strm)[stream->myindex] = (Uq) data;
        LN;                     /* put the character */
        stream->myindex++;
        LN;
        *(stream->inptr) = stream->myindex;
        LN;                     /* update inindex */
        stream->obj->writersblock = ACTIVE;     /* flag not full */
        LN;
        return (data);

        /*
         * Here we test for file or stream, if it's a file we have hit the
         * end and we return EOF, else continue
         */
tripped:
        if (stream->type == FILESTREAM)
                goto stream;
        LN;

        /*
         * we have tripped on a file object write ...
         *
         * NB: update mode requires that tripwire is reset to limit, and char is
         * put
         */
        if ((stream->flags & _UPDATE) &&
            (stream->tripwire < stream->obj->filelimit)) {
                stream->tripwire = stream->obj->filelimit;
                goto writeok;
        } else
                return (EOF);

        /*
         * We have a stream. Test for full and also for top-of-buffer.
         */
stream:
        /*
         * Lets be paranoid and test to see if anyone has stuffed up
         */
        inindex = stream->myindex;
        LN;
        if (inindex != *(stream->inptr))
                goto stuffed;
        LN;
        outindex = *(stream->outptr);
        LN;
        if ((outindex < 0) || (outindex >= stream->strmsz))
                goto stuffed;
        LN;
        /*
         * The stream object seems OK        for top of buffer
         */
        if (inindex == (stream->strmsz - 1)) {
                /*
                 * Have reached top of buffer. If outindex = NPUSHBACK,
                 * the buffer is full.
                 */
                if (outindex == NPUSHBACK)
                        goto full;
                /*
                 * Buffer is not full. Can place data and reset inindex
                 */
                (stream->strm)[inindex] = data;
                LN;
                /*
                 * Next trip condition must be catching up with outindex
                 */
                stream->tripwire = outindex - 1 - NPUSHBACK;
                LN;
                stream->myindex = 0;
                LN;
                *(stream->inptr) = 0;
                LN;
                stream->obj->writersblock = ACTIVE;    /* flag not full */
                LN;
                return (data);
        }
        /*
         * Not at top of buffer. Buffer may be full, so test
         */
        if (inindex == (outindex - 1 - NPUSHBACK))
                goto full;
        if (inindex == (stream->strmsz - 1 - NPUSHBACK + outindex))
                goto full;
        LN;
        (stream->strm)[inindex] = data;
        LN;
        /*
         * What trip will we hit next?  If inindex < outindex, will hit it
         * before (or same time as) hitting top of buffer.
         */
        if (inindex < (outindex - NPUSHBACK)) {
                stream->tripwire = outindex - 1 - NPUSHBACK;
                LN;
        } else if ((inindex >= outindex) && (outindex > NPUSHBACK)){
                stream->tripwire = stream->strmsz - 1;
                LN;
        } else if ((outindex <= inindex) && (outindex <= NPUSHBACK)){
                stream->tripwire = stream->strmsz - 1 - NPUSHBACK + outindex;
                LN;
        } else
```

```
                return (STUFFED);
        LN;
        stream->myindex = inindex + 1;
        LN;
        *(stream->inptr) = inindex + 1;
        LN;
        return (data);

full:
        /*
         * First we test for a stream closed by the other party, then we test
         * for nonblocking mode, from which we return, else we block ( ie
         * wait and retry until a char is written)
         */
        if (stream->obj->rmode == STREAMMODECLOSING)
                return (CLOSED);
        LN;
        if (stream->obj->wmode == STREAMMODENONBLOCKING)
                return (FULL);
        LN;
        par = (Param *) PARAMADDRESS;
        the_wall = (Uw *) 0xc000;
        LN;
#ifdef NOWAKEMODE
        stream->obj->writersblock = BLOCKED; LN;/* flag full */
        release(par); LN;                       /* give up slice */
#else
        /*
         * NB: writer wakes reader if reader's mode is STREAMMODESLEEPONEMPTY
         * regardless of state of writer
         */
        if ((stream->obj->rmode & STREAMMODESLEEPONEMPTY)
                && (stream->obj->readersblock & SLEEPING)
                && !(stream->obj->readersblock & WOKEN)) {
                wakereader(par,stream->obj); LN;
        }
        /*
         * NB: writer puts itself to sleep if its mode is STREAMMODESLEEPONFULL
         * regardless of state of reader
         */
        if (stream->obj->wmode & STREAMMODESLEEPONFULL){
                stream->obj->writersblock |= SLEEPING; LN;/* flag asleep */
                sleepten(par); LN;
                flushmsg(par); LN;
                stream->obj->writersblock = ACTIVE; LN;/* flag active again */
                goto retry;
        }
        /*
         * Here we block ... STREAMMODEBLOCKING default
         */
        stream->obj->writersblock = BLOCKED; LN;/* flag full */
        release(par); LN;      /* give up slice */
#endif
        goto retry;
        /*
         * Buffer object corrupted
         */
stuffed:
        return (STUFFED);

}                               /* end wfputc() */
```

```
/*
 * wfgetc.c - get character system library call
 * Monash multi Intel version
 * Author: Carlo Kopp
 * Created: 31 May, 1994
 * Modified: 1st June, 1994 - revised design
 * Modified: 4th July, 1994 use CSW file descriptor
 * Tested (OK): 6th October, 1994 Carlo/Maurice
 * Modified: 19th October, 1994 Chris Wallace (revised trip, new algorithm)
 */

/* $Id: thesis.ms,v 1.1 1996/02/28 01:20:13 walnut Exp walnut $ */

#include <string.h>
#include <funtype.h>
#include <param.h>
#include <cap.h>
#include <request.h>
#include <stdfiles.h>
#include <filedefs.h>

#ifdef UNIX
#include <stdio.h>
#else
#include <stdio_c.h>
#endif

#ifdef UNIX
extern Param    localpar;
extern File     localfdt[];
extern Uw       localmesg[];
#define NPUSHBACK 8
#endif

/*
 * The circular buffer scheme used by the Multi stdio library makes the
 * following assumptions about the buffer indices:
 *
 * inindex (write) may assume values between 0 and (strmsz - 1)
 *
 * outindex (read) may assume values between 0 and (strmsz-1)
 *
 * The buffer-empty state is inindex = outindex The buffer-full state is EITHER
 * inindex = outindex-1    OR inindex = strmsz-1,  outindex = 0
 */

Sw
wfgetc(File *stream)
{
        Param
                        *par;
        Sw
                        inindex,
                        outindex;
        Uq
                        data;
#ifndef     UNIX
        extern Uw       lineNumber;
#else
        Uw              lineNumber;
#endif

#ifdef BLOGS
#define LN lineNumber=__LINE__ + 0x20000
#else
#define LN
#endif

        extern Uw       debug, *the_wall;

        /*
         * This is the common code section executed on every pass. The _WROK
         * flag is cleared to protect from a following write. The trip test
         * determines whether a special case exists, in which event case
         * specific code is executed. retry and readok are specific entry
         * points used by the special case handlers.
         */

retry:
        stream->flags &= (~_WROK);
        LN;
        if (stream->myindex >= stream->tripwire)
                goto tripped;
        LN;
/*
 * readok:
 */
        data = (stream->strm)[stream->myindex];
        LN;             /* get the character */
        stream->myindex++;
        LN;
        if (*(stream->pushback) > 0) *(stream->pushback) -= 1;
        LN;
        *(stream->outptr) = stream->myindex;
        LN;             /* update outindex */
        stream->obj->readersblock = ACTIVE;
        LN;             /* flag not empty */
        return (data);

        /*
         * Here we test for file or stream, if it’s a file we have hit the
         * end and we return EOF, else continue
         */
tripped:
        if (stream->type == FILESTREAM)
                goto stream;
        LN;
        /*
         * read and update mode on file set the EOF flag and return EOF
         */
        stream->flags |= _EOF;
        LN;
        return (EOF);
        LN;

        /*
         * We have a stream. Test for empty and also for top-of-buffer.
         */
stream:
        /*
         * Lets be paranoid and test to see if anyone has stuffed up
         */
        if (*(stream->pushback) > NPUSHBACK)
                goto stuffed;
        LN;
        outindex = stream->myindex;
        LN;
        if (outindex != *(stream->outptr))
                goto stuffed;
        LN;
        inindex = *(stream->inptr);
        LN;
        if ((inindex < 0) || (inindex >= stream->strmsz))
                goto stuffed;
        LN;
        /*
         * The stream object seems OK
         */
        if (inindex == outindex)
                goto empty;
        LN;
        /*
         * Buffer is not empty. Can read a character
         */
        data = (stream->strm)[outindex];
        LN;
        stream->obj->readersblock = ACTIVE;   /* flag not empty */
        LN;                     /* get the character */
        outindex++;
        LN;
        if (outindex == stream->strmsz) {
                /*
                 * Have reached top of buffer.  Reset outindex to 0. Next
                 * trip condition must be catching up with inindex
                 */
                stream->tripwire = inindex;
                LN;
                stream->myindex = 0;
                LN;
                if (*(stream->pushback) > 0) *(stream->pushback) -= 1;
                LN;
                *(stream->outptr) = 0;
                LN;
                return (data);
        }
        /*
         * Not at top of buffer.  Outindex has been stepped on. What trip
         * condition can we hit next?  If outindex <= inindex, will hit it
         * before (or same time as) hitting top of buffer.
         */
        if (outindex <= inindex) {
                stream->tripwire = inindex;
                LN;
        } else {
                stream->tripwire = stream->strmsz - 1;
                LN;
        }
        stream->myindex = outindex;
        LN;
        if (*(stream->pushback) > 0) (*stream->pushback) -= 1;
        LN;
        *(stream->outptr) = outindex;
        LN;
        return (data);

        /*
         * First we test for a stream closed by the other party, then we test
         * for nonblocking mode, from which we return, else we block ( ie
         * wait and retry until a char is read)
         */
empty:
        if (stream->obj->wmode == STREAMMODECLOSING)
                return (CLOSED);
        LN;
        if (stream->obj->rmode == STREAMMODENONBLOCKING)
                return (EMPTY);
        LN;
        par = (Param *) PARAMADDRESS;
        the_wall = (Uw *) 0xc000;
        LN;
#ifdef NOWAKEMODE
        stream->obj->readersblock = BLOCKED;        /* flag empty */
        release(par); LN;       /* give up slice */
#else
        /*
         * NB: reader wakes writer if writer’s mode is STREAMMODESLEEPONFULL
         * regardless of state of reader
         */
        if ((stream->obj->wmode & STREAMMODESLEEPONFULL)
                && (stream->obj->writersblock & SLEEPING)
                && !(stream->obj->writersblock & WOKEN)) {
                wakewriter(par,stream->obj); LN;
        }
        /*
         * NB: reader puts itself to sleep if its mode is STREAMMODESLEEPONEMPTY
         * regardless of state of writer
         */
        if (stream->obj->rmode & STREAMMODESLEEPONEMPTY){
                stream->obj->readersblock |= SLEEPING; LN;/* flag asleep */
                sleepten(par); LN;
                flushmsg(par); LN;
                stream->obj->readersblock = ACTIVE; LN;/* flag active again */
                goto retry;
        }
        /*
         * Here we block ... STREAMMODEBLOCKING default
         */
        stream->obj->readersblock = BLOCKED;       /* flag empty */
        release(par); LN;      /* give up slice */
#endif
        goto retry;

        /*
         * Buffer object corrupted
         */
stuffed:
        return (STUFFED);

}                                /* end wfgetc() */
```

```
/*
 * ungetc.c - unget character system library call
 * Monash multi Intel version
 * Author: Carlo Kopp
 * Created: 17th March, 1995
 */

/* $Id: thesis.ms,v 1.1 1996/02/28 01:20:13 walnut Exp walnut $ */

#include <funtype.h>
#include <param.h>
#include <stdfiles.h>
#include <filedefs.h>

#ifdef UNIX
#include <stdio.h>
#else
#include <stdio_c.h>
#endif

#ifdef UNIX
extern Param    localparam;
extern File     localstreamt[];
extern Uw       localmesg[];
#define NPUSHBACK 8
#endif

/*
 * The circular buffer scheme used by the Multi stdio library makes the
 * following assumptions about the buffer indices:
 *
 * inindex (write) may assume values between 0 and (strmsz - 1)
 *
 * outindex (read) may assume values between 0 and (strmsz-1)
 *
 * The buffer-empty state is inindex = outindex The buffer-full state is EITHER
 * inindex = outindex-1    OR inindex = strmsz-1,  outindex = 0
 *
 * NB: the pushback buffer zone size is set by NPUSHBACK, pushback is the
 * pushback counter which is used by putc
 */

Sw wungetc(Sw c, File *stream)
{
        Sw              inindex, outindex;
#ifndef     UNIX
        extern Uw       lineNumber;
#else
        Uw              lineNumber;
#endif
#define LN lineNumber=__LINE__ + 0x20000

/*
 *     beartrap for bad data such as error codes
 */
        if ( c < 0 ) return ( c );
/*
 *      This is code executed on every pass - if myindex is 0, we wrap around
 */
        stream->flags &= (~_WROK);
        LN;
        if (stream->myindex <= 0)
                goto tripped;
        LN;
        *(stream->pushback) += 1;
        LN;
        if(*(stream->pushback) > NPUSHBACK)return (EOF);/* overran pushback */
        LN;
        stream->myindex--;
        LN;
        (stream->strm)[stream->myindex] = c;
        LN;                     /* push the character */
        *(stream->outptr) = stream->myindex;
        LN;                     /* update outindex */
        return (c);

        /*
         * Here we test for file or stream, if it’s a file we have hit the
         * bottom and we return EOF, else continue
         */
tripped:
        if (stream->type == FILESTREAM)
                goto stream;
        LN;

        /*
         * read and update mode on file set the EOF flag and return EOF
         */
        stream->flags |= _EOF;
        LN;
        return (EOF);
        LN;

        /*
         * We have a stream. We wrap around to the top
         */
stream:
        /*
         * Lets be paranoid and test to see if anyone has stuffed up
         */
        if (*(stream->pushback) < 0)
                goto stuffed;
        LN;
        outindex = stream->myindex;
        LN;
        if (outindex != *(stream->outptr))
                goto stuffed;
        LN;
        inindex = *(stream->inptr);
        LN;
        if ((inindex < 0) || (inindex >= stream->strmsz))
                goto stuffed;
        LN;
        /*
         * The stream object seems OK
         */
        /*
         * Here we wrap around
         */
        *(stream->pushback) += 1;
        LN;
        if(*(stream->pushback) > NPUSHBACK)return (EOF);/* overran pushback */
        /*
         * here is where it all happens
         */
        LN;
        (stream->strm)[stream->strmsz - 1] = c; /* XXX */
        LN;
        stream->myindex = stream->strmsz - 1;
        LN;
        *(stream->outptr) = stream->strmsz - 1;
        LN;
        return (c);

stuffed:
        return (STUFFED);

}                               /* end wungetc() */
```

Appendix C Source Code (*rm, head*)

```
/*
 * rm - Walnut user level rm program
 *
 * Authors: Carlo Kopp
 *
 * revision history:
 *
 * Date            Author          Change
 *
 * 26/10/95        Carlo Kopp      created from filetest.c tests
 *
 * $Id: $
 */

#include <funtype.h>
#include <werrno.h>
#include <param.h>
#include <stdfiles.h>
#include <screen.h>
#include <filedefs.h>
#include <namec.h>
#include <nameserver.h>
#include <procenv.h>
#include <request.h>

#include <stdio_c.h>

#define main init

/*
 * line number for gcc debug
 */

Uw              lineNumber = 0;
#define LN lineNumber=__LINE__ + 0x30000
/*
 * code starts here
 */

void
main(int argc, char **argv, char **envp)
{
        extern Uw
                        errno,
                        debug;
        int
                        i = 0;

        setmyname("rm-running");
        debug = 3;

        for (i = 1; i <= argc; i++) errno = wremove(argv[i]);
        clrmyname("rm-running");
        vx();

    }
```

```
/*
 * Copyright (c) 1980, 1987, 1992, 1993
 *     The Regents of the University of California.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *     This product includes software developed by the University of
 *     California, Berkeley and its contributors.
 * 4. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 */


#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#include <funtype.h>
#include <werrno.h>
#include <param.h>
#include <stdfiles.h>
#include <screen.h>
#include <filedefs.h>
#include <namec.h>
#include <nameserver.h>
#include <request.h>
#include <stdio_c.h>

/*
 * head - give the first few lines of a stream or of each of a set of files
 *
 * Bill Joy UCB August 24, 1977
 * Walnut port Carlo Kopp February, 1996
 *
 */

Uw              lineNumber = 0;
#define LN lineNumber=__LINE__ + 0x60000

#define main init

int eval;
extern Uw errno;
```

```
void
head(File *fp, register int cnt)
{
        register int ch;

        while (cnt--)
                while ((ch = getc(fp)) != EOF) {
                        if (putchar(ch) == EOF)
                                err(1, "stdout: %s", strerror(errno));
                        if (ch == ’0)
                                break;
                }
}

void
obsolete(char *argv[])
{
        char *ap;

        while (ap = *++argv) {
                /* Return if "--" or not "-[0-9]*". */
                if (ap[0] != ’-’ || ap[1] == ’-’ || !isdigit(ap[1]))
                        return;
                if ((ap = malloc(strlen(*argv) + 2)) == NULL)
                        err(1, "%s", strerror(errno));
                ap[0] = ’-’;
                ap[1] = ’n’;
                (void)strcpy(ap + 2, *argv + 1);
                *argv = ap;
        }
}

void
usage()
{
        (void)fprintf(stderr, "usage: head [-n lines] [file ...]0);
        vx();
}

#include <stdarg.h>

void
err(int fatal, const char *fmt, ...)
{
        va_list ap;
        va_start(ap, fmt);
        (void)fprintf(stderr, "head: ");
        (void)vfprintf(stderr, fmt, ap);
        va_end(ap);
        (void)fprintf(stderr, "0);
        if (fatal)
                vx();
        eval = 1;
}
```

```
int
main(int argc, char *argv[])
{
	register int ch;
	File *fp;
	int first, linecnt;
	char *ep;

	setmyname("head-running");
	obsolete(argv);
	linecnt = 10;
	while ((ch = getopt(argc, argv, "n:")) != EOF)
		switch(ch) {
		case ’n’:
			linecnt = strtol(optarg, &ep, 10);
			if (*ep || linecnt <= 0)
				err(1, "illegal line count -- %s", optarg);
			break;
		case ’?’:
		default:
			usage();
		}
	argc -= optind;
	argv += optind;

	if (*argv)
		for (first = 1; *argv; ++argv) {
			if ((fp = fopen(*argv, "r")) == NULL) {
				err(0, "%s: %s", *argv, strerror(errno));
				continue;
			}
			if (argc > 1) {
				(void)printf("%s==> %s <==0,
				    first ? "" : "0, *argv);
				first = 0;
			}
			head(fp, linecnt);
			(void)fclose(fp);
		}
	else
		head(stdin, linecnt);
	clrmyname("head-running");
	vx();
}
```

Appendix D Validation Test Suite

0. perror integrity test

The perror integrity test sets the *errno* value, and then invokes the *perror* function. This is done for all defined error codes.

1. opencap create object test

The opencap create object test creates a file object in write mode.

2. fprintf/putc write object test

The fprintf/putc write object test writes a string into the object created in the previous test.

3. write exclusion test

The write exclusion test verifies that a write mode file cannot be read. It tests the exclusion flags which should be set in the *FILE* structure.

4. ferror test

The ferror test verifies that the *feof* function correctly tests the flags.

5. clearerr test

The clearerr test verifies that the *clearerr* function has cleared the _ERR and _EOF flags in the *FILE* structure.

6. file close test

The file close test verifies that a file can be closed.

7. opencap read mode test

The read mode opencap test verifies that a file can be opened in read mode. It operates on the file object created by test 2.

8. file read test

The file read test verifies that a file can be read. It operates on the file object created by test 2.

9. feof test

The feof test verifies that the *feof* function tests the _EOF flag correctly.

10. fseek/ftell test

The fseek/ftell test contains four sub-tests, each of which verifies that an *fseek* operation has produced the required change to the file index position.

11. fsetpos/fgetpos test

The fsetpos/fgetpos test repeats test 10, using the *fsetpos* and *fgetpos* functions.

12. file append test

This test opens the previously created file object in append mode, writes to it, closes it, opens it in read mode and confirms that the append mode write has been successful.

13. file update test

This test opens a file in update mode, writes to it, rewinds the file index, and reads back the contents.

14. removec/kerror test

This test destroys the file object used in the preceding test, and then attempts to open the non-existent file object to confirm the operation of the kernel error reporting function.

15. copen create test
16. copen write test
17. copen update test

These three tests repeat the operations carried out previously in the *opencap* function tests, using the *copen* function.

18. tmpfile test

This test verifies that a tmpfile can be created.

19. stream fflush test

This test executes the *fflush* function.

20. fopen create test
21. fopen read test
22. fopen update test

These three tests repeat the operations carried out previously in the *copen* function tests, using the *fopen* function.

23. remove test

This test confirms that a previously created file object can be removed.

24. ungetc pushback test

This test creates a file object, writes a string to it, reads back the string, pushes **NPUSHBACK** characters back to file, and then verifies that the characters pushed back are identical to the characters initially read.

25. fscanf test

This test verifies that the *fscanf* function can correctly decode arguments in the **%d, %x, %c** and **%s** formats.

26. makestreamobj test

The makestreamobj test uses the *fileinit* function to verify that a stream object created by the *makestreamobj* function is error free.