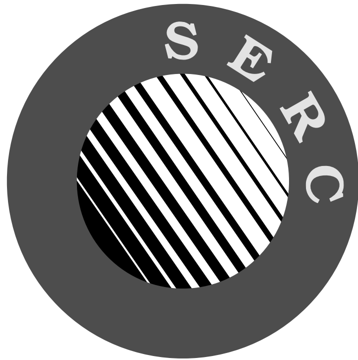

SERC-0011
August, 1995
Issue 1

The Walnut Kernel: Program Implementation Under The Walnut Kernel

Maurice Castro[1]
Glen Pringle[2]
Chris Wallace[3]

[1] Software Engineering Research Center, CITRI, 723 Swanston St, Melbourne 3053
[2] Department of Computer Science, Monash University , Clayton 3168
[3] Department of Computer Science, Monash University , Clayton 3168

Copies of this document may be obtained by contacting:

Director
SERC
723 Swanston St
Carlton, Victoria, 3053

Prepared by the Software Engineering Research Center.

Copyright ©1995 SERC.
All Rights Reserved.

Abstract

The research community has had a long term interest in the use of capabilities in operating systems. Many advantages have been claimed by designers of systems based on the capability paradigm. At present only a few capability-based systems are in use in non-research environments. Relatively little work has been published describing the implementation of applications in a capability based environment.

This paper describes the Walnut Kernel - a capability-based operating system developed in the Department of Computer Science, Monash University - and a number of application programs implemented under it. The development of these applications resulted in changes to the design of the kernel. These influences and lessons learned from the design and implementation of these applications are discussed.

This technical report has been released by both the Software Engineering Research Center, Collaborative Information Technology Research Institute, 723 Swanston St, Melbourne, Australia 3053 as technical report SERC-0011 and the Department of Computer Science, Monash University, Clayton, Australia 3168 as technical report 95/230.

1 Overview

The Walnut Kernel is a capability based operating system under development in the Department of Computer Science at Monash University. This paper describes a number of applications that have been written to operate under the Walnut Kernel. These applications have allowed programmers to explore the possibilities offered by a capability-based operating system and provided feedback to the operating system designers. This feedback in turn has resulted in changes to the design of the kernel.

Four programs are described:

- Initproc - the initialization process
- Glui - a screen multiplexor
- Shell - a user shell
- Wyrm - an arcade style game

Initproc is responsible for deriving capabilities used by processes which manage access to devices. Shell and Glui form a user level interface which allow access to the functions of the kernel and objects within the system. The game Wyrm is an example of a highly interactive application which demands fast response times from the system and is IO bound.

Section 2 briefly describes the kernel and capability based systems. Program structures and data structures used in the applications are described in section 3. Sections 5 to 8 discuss the application programs.

2 Walnut Kernel

The Walnut Kernel is based on the concept of *password capabilities* which were originally developed for use in the operating system for the Password-Capability System [1]. This section briefly introduces capabilities and the environment provided by the Walnut Kernel to application programmers.

2.1 Capabilities

Capabilities are essentially a naming scheme used to provide a uniform mechanism for controlling access to and the protection of resources [8]. The naming scheme can be extended beyond identifying objects by viewing a capability as a form of address [9]. This allows all system resources to be modelled as memory objects. Capabilities perform two roles: naming - a capability is used to uniquely identify an object or part of an object, and access control - associated with a capability are a set of rights that the possessor of the capability is allowed to exercise over the object associated with the name. Possession of a capability is synonymous with access to the rights associated with the capability. In combination with persistent memory objects this provides a simple and uniform programming model.

The capability model promotes the sharing of memory objects. Possession of a capability enables the holder to use the functionality conveyed by the capability. This allows multiple processes to access the section of an object covered by a capability.

The persistent nature of capability-based systems provides the further advantages of a reduced code size and simplicity. Typically 30% of a program is devoted to accessing data in secondary storage. Capability-based systems eliminate the need to explicitly access secondary storage resulting in savings in size. Complexity

is reduced by using a single access mechanism for both temporary and persistent storage [3].

A number of capability based operating systems have been developed. These include: Monads, KeyKOS, and Opal.

The development of Monads [11] was motivated by a desire to enforce the use of modular programming practices. It employed capabilities as a naming and access control mechanism. Specialized hardware was built to support the use of capabilities and enforce the information hiding principles implemented in the Monads programming model.

KeyKOS [4] is significant because it demonstrates the potential value of capability-based operating systems in commercial applications. Systems using KeyKOS were used as the basis of British Telecom's Tymnet service. This service required accurate accounting, the ability to support mutually antagonistic users and 24-hour uninterrupted operation. Capabilities provided strong access control[4]. The persistent nature of memory objects and regular checkpointing, meant that systems could be restarted rapidly and start functioning from the point of failure.

Opal [6] is a single address space operating system employing password-capabilities as an access control mechanism. It provides a 64-bit address space which is sparsely populated by memory objects. Objects are identified by capabilities which are collected into protection domains. Processes operating in a protection domain can name an object by its address and exercise the access rights for the object provided by the capability stored in the protection domain.

2.2 Types of Implementations

Currently there are three major mechanisms for implementing capabilities within an operating system:

Tagged Architectures employ a hardware supported tag bit which distinguishes the memory containing capabilities from other user program memory. Only a limited set of operations are permitted on capabilities, and the hardware is used to prevent forgery.

Capability Lists are collections of capabilities managed by the operating system. The operating system moderates all access to the contents of capability lists, hence preventing forgery. This mechanism does not require specialized hardware support.

Password-Capabilities are statistically secure. The capability consists of an identifier for an object and a randomly selected value. Only a small number of the total range of values are valid capabilities. This ensures that forgers would have to spend a prohibitively large amount of time searching the name space for a valid capability.

Capability-lists have been the most popular form of implementation as they do not require specialized hardware support. Monads and KeyKOS both employed capability-lists. Password-capabilities are a more recent development and have been employed in: the Password-Capability System, Opal, and Mungi[10].

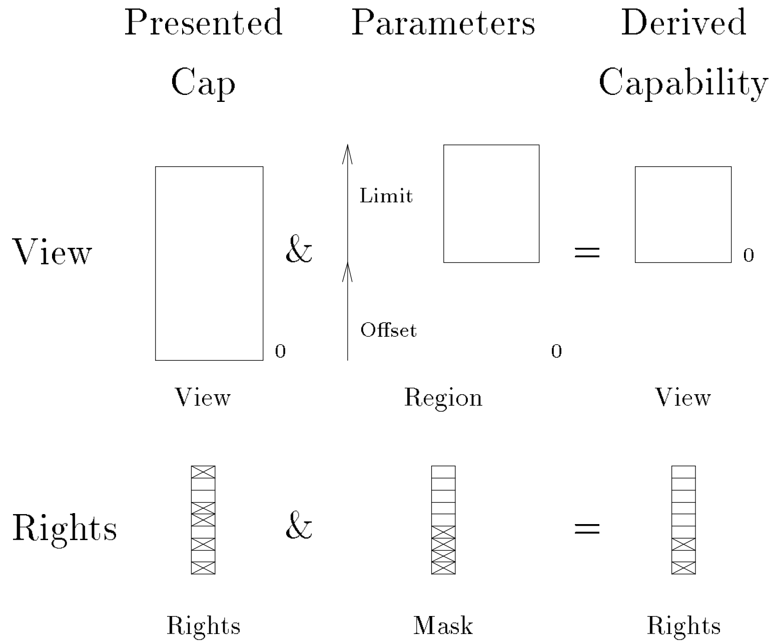

Figure 1: Derivation

2.3 The Password-Capability Model

A password capability consists of two parts. The identifier portion of a capability uniquely identifies an object. The password component identifies the set of rights and the view associated with the capability.

The rights associable with a capability are system dependent. Typically the rights include: user accessible, read, write, execute, derive, and suicide. The first four of the rights are related to access to memory, the other rights are used to control how the capability is used. Capabilities without derive right cannot be used to make new capabilities. A capability without suicide right cannot be deleted directly, it can only be deleted through the use of a capability able to destroy the parent capability.

A view is an attribute of a capability. It is the region of an object that can be addressed by a process having the capability. The region is contiguous and defined by an offset from the base of the object and an extent.

The master capability for an object is returned to the process which creates the object. The master capability possesses all the rights which may be exercised over the object. All other capabilities applying to the object are derived from the master capability. Derivation (see figure 1) is the fundamental operation of a password-capability based system. It is performed by taking a capability's rights and performing a logical-and of those rights with a rights-mask to produce a set of rights not more powerful than the parent capability. In addition, the view presented by the parent capability and an offset and extent are combined to produce a capability which covers an equal or smaller section of the object.

[4] Illicit access to an object was only possible if the object could be named, and a capability for that name forged. The compromise of an object under a capability-based system does not result in the compromise of other objects. To compromise other objects it is necessary to repeat the process required to breach the security of the first object.

Figure 2: Structure of a Walnut Kernel Capability

The capabilities of an object are notionally arranged in a tree structure. To maintain this structure the deletion of a parent capability results in the destruction of its children, and hence all its descendents.

The password-capability model allows any process that knows the name of a capability to load the capability and make use of the rights and view associated with the capability.

Processes are distinguished from other objects by the existence of process-state information contained in the process object. The state information describes the contents of the address space seen by the process, the process's mailboxes and the stored processor state. Access to this information is typically moderated by the kernel. This prevents user code from circumventing hardware protection mechanisms (for example the user/supervisor bit). Process objects are the only active elements within the system apart from the kernel itself.

2.4 Walnut Kernel Implementation

The Walnut Kernel is based on the password-capability model. However, the model has been modified in a number of key areas as a result of our experiences in using the password capability model and restrictions imposed by avoiding features requiring hardware support not available on a wide range of processors. This section briefly describes the flavor of the Walnut Kernel, a complete description can be found in [5].

Under the Walnut Kernel a capability is 128-bit value that consists of 4 fields (see figure 2):

Volume field identifies the device on which an object resides.

Serial Number field identifies the object within its volume.

Password 1 & 2 fields identify the capability.

The volume and serial portion of the capability uniquely identify an object. The passwords are used to identify the set of rights and the view associated with the capability.

Typically the passwords of a capability are allocated randomly to ensure a sparse distribution and statistical security. The Walnut Kernel introduced the ability to specify the passwords of a derived capability. This serves two purposes:

- It allows capabilities to be *hard coded* into programs and operate correctly. This service is an alternative to advertising a capability.
- It allows the regeneration of capabilities after deletion. This feature is used by the initialization process to regenerate capabilities for the system object after a reboot.

The majority of commercially available processor designs only provide support for demand paged virtual memory, and they have no provision for supporting segmentation. The Walnut Kernel uses the paging hardware to control access to objects. This results in a page size protection granularity.

Views of objects can be mapped into the address space of a running process. The contents of the loaded views may be manipulated directly through the use of any operations, provided by the processor, which affect memory.

Memory is easily shared by Walnut Kernel processes as it only requires processes to load capabilities with overlapping views for part of an object to be shared. A further interprocess communication mechanism is provided: messages. Messages are short strings that are sent by a process to another process. On arrival, a message is stored in a mailbox and the process is woken up if it was sleeping. Messages can be used to provide synchronization between multiple processes.

Mapped into each process is a user readable page known as the **Wall**. This page is used to distribute system wide information. Typically this page contains capabilities used to communicate with manager processes and the system time. The system time is updated when the kernel transfers control to a user process.

A process contains a number of threads of control known as subprocesses. All processes support at least one user subprocess. Messages are addressed to a specific subprocess of a process. Associated with each subprocess is a priority which determines which subprocess to execute when more than one is runnable.

Processes can control the execution state of other processes by the use of **freeze** and **thaw** functions. When a process is frozen it is made unrunnable until it is thawed. The process will continue to accept messages while its mailboxes are not full.

A version of freeze and thaw which uses magic numbers was introduced in the Walnut Kernel; it is known as protected freeze and thaw. These operations require that a magic number be presented to the kernel each time a freeze or thaw operation is performed. The process is only runnable when all the freeze operations are undone by thaw operations, with magic numbers matching the magic numbers of the freeze operations.

Access to hardware devices is provided through shared memory. The system object is a special object that covers the memory in which the kernel code and data are stored, the buffers used to transfer information to and from devices, and the memory locations where any memory mapped devices exist. An initialization process derives capabilities from the master capability for the system object. The derived capabilities are distributed to manager processes for devices.

Block oriented random access devices, typically, do not have an interface accessible by the user. These devices are treated as volumes. Allowing low level access to the devices would compromise the security of data stored on the media. In the case of floppy disks, low level access is prohibited when a volume is mounted and otherwise allowed. This allows tools which read MS-DOS formatted disks to be implemented in user code. This provides a level of security equivalent to the protection provided by the media. In the case of removable media, security is equivalent to the possession of the physical media[5].

A major change was made to the capability model with the introduction of the SRMULTILOAD right. Capabilities without this right can only be used by processes with a serial number which equals the password 2 of the capability. This allows processes to transfer access rights for an object to a specific process, and prevent any other process from acquiring those rights. One application of this mechanism is to use it in combination with a protected freeze and thaw operation to grant controlled access to physical devices (see section 6).

[5] Encryption can be employed to make the data less easily accessible when the removable media is not mounted

2.5 Devices in the Current Implementation

Circular buffers are used to transfer information to and from serial ports, parallel ports and the keyboard. There are two interfaces to the screen. A VT100 emulator which uses a circular buffer to transfer information and the memory mapped screen memory. The VT100 emulator manipulates the position of the cursor on the screen when the memory mapped screen is used.

All the circular buffers used to interface with devices employ the same format and conventions. This simplifies design and improves reliability by encouraging software reuse.

3 Structures

This section describes a number of common structures found in programs operating under the Walnut Kernel. It addresses both organization of programs and data structures.

3.1 Program Structures

Walnut Kernel programs are similar to programs which are implemented under GUIs in that both types of programs respond to external events. GUIs provide two constructs for handling events:

Message Loops are a loop which contains a call to a function that accepts an event from a queue of events, and then calls a function to handle the event.

Callback Functions are registered with the user interface and are invoked with a set of parameters when an event occurs. Callback functions are used to handle asynchronous events.

The Walnut Kernel supports constructs which perform similar tasks, but are implemented differently.

```
while true
begin
    wait(-1)
    receive(msg)
    server_function(msg)
end
```

Figure 3: Pseudocode for a Message Loop

The Walnut Kernel typically handles messages by using a **message loop** (see figure 3). This simple construct places the process (or subprocess) into a sleep state until a message arrives, receives a message, handles the message, and returns the process to a sleeping state. As a process cannot sleep when there are messages waiting for it, the message loop can handle multiple messages without the need to test for the presence of a message before going to sleep.

Asynchronous events which would be handled with a call back function under a GUI are implemented through the use of subprocesses. A subprocess is a thread of control within a process to which a message can be specifically addressed. When a message arrives for a subprocess, the subprocess is made executable. Typically a

message loop is used to receive and handle the message before putting the subprocess back to sleep.

3.2 Data Structures

Persistence, sharing and relocation shape the types of data structures in common use under the Walnut Kernel.

File oriented operations are typically performed on a stream of data, converting the contents of an input stream to an output stream. Persistent data structures do not require conversion to and from a secondary storage format, eliminating the stream orientation imposed by the file mechanism. In addition, programmers are able to perform random access operations on input and output data structures without the overheads that would be present on a stream oriented system. The absence of these constraints provides an new degree of freedom in the design of data structures.

A hash table is an example of a data structure that benefits from a persistent implementation. On a persistent system the hash table is stored in a directly usable form. This can be contrasted with a file oriented system which has the choice of extracting the data from the table and storing it in a linear form, or storing the table as a block of memory dumped to disk. The former requires either a complex transform on the data to recover it in the correct order for storage, or an additional data structure that keeps track of the order in which data should be stored. The latter approach requires the table to be read at the beginning of the program and written at the end of the program, introducing a significant IO overhead.

The easy sharing of data requires programmers to be aware of synchronization, access control, and locking issues. Currently systems programmers work in an environment where sharing considerations are important. Application programmers need to become aware of the issues and techniques for managing shared data. Provision must be made in shared data structures for collective access to the data structure. This may include choosing data structures that allow simultaneous access (circular buffers) or employ locking.

Programmers have a choice of loading an object at a fixed address or allowing the loading of an object at an arbitrary address. If an object is always located at a fixed address, pointers may be used within the object to refer to other parts of the data structure. This arrangement has the advantage of speeding references within an object. However, it causes a loss of flexibility and may restrict the sharing of objects. This is because programs will only be conveniently able to load one object at a time that occupies a set of points in the address space. Relocatable objects use index values to refer to parts of the object. This requires an addition operation before a dereference operation can be performed resulting in a potential loss of performance.

Circular buffers (see figure 4) are used to transfer stream oriented information between processes. Two implementations are used:

- A minimal implementation is used by character mode devices such as serial ports and the keyboard to communicate with their manager processes.
- An optimized version is used for interprocess communication.

Both circular buffer implementations do not require locking, but ensure that data is correctly transfered from the sender to the receiver. They operate by giving the sender read/write access to the write-pointer, and read-only access to the read pointer. The receiver has read/write access to the read-pointer, and read-only access to the write pointer. This eliminates contention over updating the pointers. The data structure operates safely even if information relating to the position of the

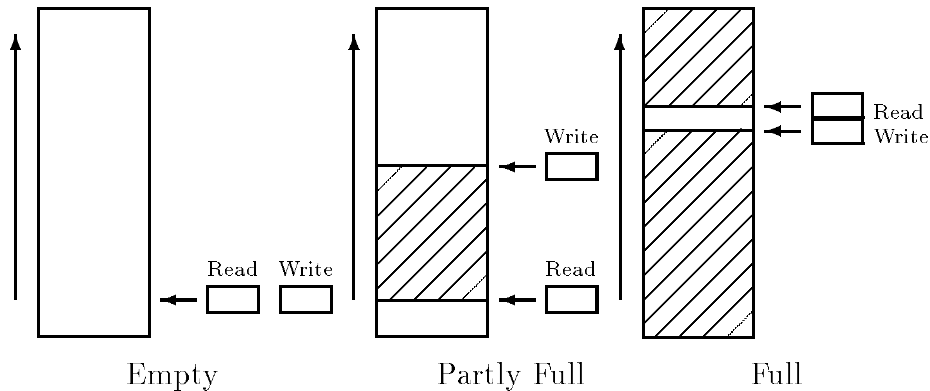

Figure 4: A Circular Buffer

other pointer is old. The data structure has a minor inefficiency in that there is always a single wasted slot when the data structure is full.

In the more efficient implementation, both the sender and receiver have a private pointer known as the **tripwire**. The tripwire is set to point to either the value of the other pointer or the top of the buffer. Before sending or receiving an element from the circular buffer, the value of the pointer is compared against the tripwire to determine if there is the risk of overfilling the buffer or crossing the end of the buffer. This mechanism saves the cost of a comparison on most accesses to the buffer by converting the separate tests for overfilling and wrapping, from top to bottom of the buffer, into a single test. If the comparison indicates that either of the boundary conditions has been reached, further tests are carried out to determine which of the two conditions caused the problem, and the tripwire is set to a new position.

4 Legacy Code

A library has been constructed that emulates many of the functions found in the C *stdio* library. This library has two roles:

- It allows the reuse of a large quantity of existing C code, reducing development effort.
- It provides an environment that is familiar to a large range of programmers allowing them to use existing skills while learning about the features the Walnut Kernel environment offers them.

Under the emulation library files and streams are implemented using the same circular buffer code. Files gain no advantage from being implemented using circular buffers; however, there is no performance penalty either. By choosing to implement the two mechanisms in the same way, code volume is reduced and code maintenance is simplified.

5 Initproc

When a Walnut Kernel is booted, it generates an object known as the system object. This object contains all the memory pages occupied by kernel code, kernel data, and device driver interfaces and buffers. The initialization process derives capabilities from the system object used by the processes which manage devices. The

restrict operation is then applied to the master capability. This operation removes rights associated with a capability without affecting the rights of the children of the capability. This eliminates a potential security hole associated with the existence of a capability allowing unfettered access to the kernel and device interfaces. After deriving the set of less powerful capabilities, Initproc notifies the scheduler that it is safe to schedule other processes, and sends messages to all manager processes containing the capabilities they require to access the devices they manage. Initproc completes its operation by entering a message loop and waiting for a message indicating that the system is to be reconfigured.

Initproc illustrates a number of features of programming under the Walnut Kernel; however, the process is unique among Walnut Kernel processes in that it is restarted from a fixed address each time the kernel is booted. Apart from always starting Initproc from a fixed address, the kernel provides no special functions to support this code. Thus all of Initproc's code operates at the user level, requiring no special kernel support or privileges. All other processes resume their operations from the point at which they were stopped when the system was shutdown. Furthermore, as the system object is stored in volatile storage, the system object does not retain information about the capabilities applying to it over a reboot. The initialization process is responsible for remaking the capabilities used by the manager processes before allowing other processes to be scheduled.

The kernel scheduler monitors a word in the Wall. When the word becomes non-zero, the kernel scheduler allows the scheduling of any runnable process. Initproc derives a capability for the Wall from the system object. This capability is sent to the Wall manager and used by Initproc to notify the scheduler.

In addition to the easy sharing of data demonstrated by the above application, persistence is also exploited in Initproc. The derivative capabilities generated from the system object are stored in an array. When Initproc is restarted following a shutdown, it examines this array and generates derivative capabilities with the same name and rights as those found in the array before restarting the scheduler. This simplifies the design of the manager processes as the capabilities given to managers by Initproc appear to persist over the reboot. Holders of derivatives of capabilities distributed by Initproc will find that those capabilities no longer work.

6 Glui

Glui is the manager process for the screen and the keyboard. It provides several stream mode interfaces to the keyboard and screen. A series of keystrokes are used to switch between sessions. In addition, Glui supports a mechanism for giving direct access to the screen memory for a number of processes. Like Initproc, Glui functions using system calls available to all processes.

When the Walnut Kernel is booted, Initproc sends a message with a capability for the resources managed by each manager process. On receipt of the messages containing capabilities for the keyboard, the screen, the VT100 emulator built into the kernel, and the Wall, Glui creates 10 virtual screens and derives a capability which allows messages to be sent to Glui. This capability is then placed on the Wall.

The screen and keyboard IO architecture of the Walnut Kernel is illustrated in figure 5. To provide terminal multiplexing facilities, Glui intercepts all keyboard input scanning for control sequences. If no control sequences are found, the keyboard input is placed in the input buffer for the application currently being displayed on the screen. The output buffer of the current application is polled periodically. If new information is found in the buffer, it is passed to the VT100 emulator code built into Glui. This emulator writes its output directly to the memory mapped

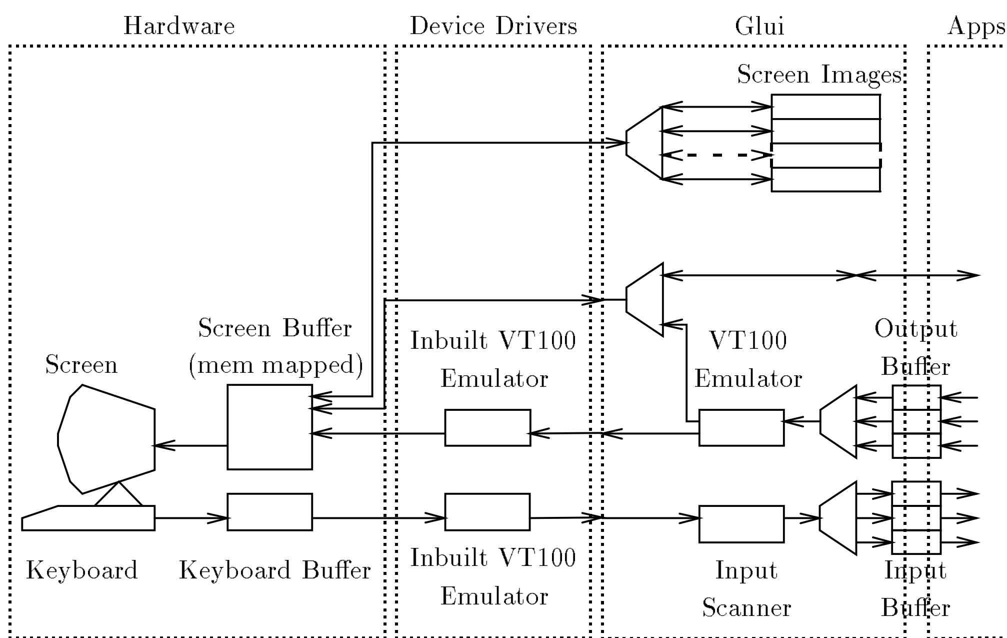

Figure 5: Keyboard and Screen IO

screen buffer. To move the cursor and sound the bell, Glui passes control codes to the VT100 emulator built into the device drivers.

To change the display to another application, Glui stops accepting input from the current client program. The current contents of the screen are copied to a buffer associated with the current application. This buffer is located within Glui and is not made accessible to other programs. The buffer corresponding to the new application is copied to the screen, and the output buffer of the new application is read to update the screen. Keyboard output is directed to the input buffer of the new application.

Glui supports two types of output services:

- a VT100 emulator
- the hardware screen buffer

When a process requires IO through the VT100 emulator and keyboard, it sends a message to Glui using the capability on the Wall. If there is a virtual screen available, Glui sends a message back which contains the capability for a keyboard buffer and a screen buffer. These buffers use the circular buffer protocol discussed in section 3.2. The process requesting the screen may send data containing VT100 screen control sequences via the output stream. Input is received via the input stream.

When direct access to the hardware screen buffer is requested, the process must supply for itself a capability that allows the process to be frozen. If this capability is not provided, or does not allow Glui to send the freeze message, the request will be rejected. If a suitable valid capability is supplied, a capability without SRMULTILOAD right and with a password 2 equivalent to the requesting process's serial number is returned to the requesting process. When loaded by the the requesting process, this capability allows direct access to the screen buffer; however, this capability cannot be loaded by any other process.

Protected freeze and thaw are used on the processes granted direct access to the memory mapped screen buffer. This prevents other processes from thawing a process with a usable capability for writing to the screen. Glui is able to ensure that only one process writes to the screen at a time, preventing corruption of the screen's contents.

Both the SRMULTILOAD right and the protected versions of freeze and thaw were introduced to enable Glui to allow controlled direct access to the hardware screen buffer. Other solutions were considered, including locking processes [2] and schemes for the rapid revocation of capabilities.

Under the Walnut Kernel, a process is locked when it is created with a 63-bit lockword. This lockword is XORed with each 'alter' capability[6] before the capability is used by the kernel. The process can only use capabilities which have been XORed with the lockword, and then be passed to the process. This prevents a locked process from communicating with other processes without the assistance of a party who knows the lockword value. This mechanism was considered, but locking severely curtailed the ability of the client program to communicate.

Although a number of rapid revocation schemes were considered, the generalization of these schemes to a multiprocessor environment either resulted in a mechanism insufficiently responsive, or required an unacceptably high overhead to support a relatively infrequent operation.

7 Shell

Shell is a command interpreter. It provides mechanism for managing objects, organizing 'files' generated through the stdio emulation code and launching programs. Shell has detailed information relating to the structure of a process which follows the conventions adopted for the Walnut Kernel.

When Shell is first started it sends a message to Glui requesting a terminal emulator output buffer and a keyboard input buffer. On receipt of these capabilities, it presents the user with a prompt and awaits further instructions. Users can run processes in two modes:

- Yielding the screen to the new process. The input buffer and output buffer used by Shell are given to the new process for its use until the new process terminates.
- Creating a new screen for the new process. The shell requests a new set of buffers from Glui which are given to the new process.

The two modes differ in several respects. When the new process is to inherit the screen from Shell, the buffers are made available to the new process and the shell goes into a loop which polls the new process's status. Shell ignores the contents of the buffers and does not take any command input until it detects that the new process has ceased to function. Shell then resumes using the buffers and accepting commands. When the screen is not inherited from Shell, a set of buffers is requested from Glui and made available to the new process. Shell continues to interpret the input from the keyboard and remains active on the screen that it is currently connected to.

Two mechanisms were introduced into the Walnut Kernel to allow processes to determine the state of another process:

Cooee Messages are sent to a process and results in a **Cooee reply** message being sent to a capability specified in the cooee message. The Cooee reply

[6] A non-alter capability does not posses write rights and cannot be used to transfer information to another process. Alter capabilities can be used to transfer information.

message is automatically generated by the kernel and contains a field indicating whether the process is running, frozen, sleeping or dead.

Peek System Call returns a value which indicates whether the process is running, frozen, sleeping or dead.

The Cooee message was introduced first; however, polling processes to determine their state proved to be useful and popular, so the more efficient peek mechanism was provided. The peek mechanism has the advantage of a significantly lower overhead as it requires only a single system call and the message passing mechanism is avoided.

A process object conforming to the Walnut Kernel conventions contains:

Startup Code Area (optional) This area may contain a small amount of code used in starting a process.

File Descriptor Table (mandatory) This area contains the file descriptors for use by the process. Note: The first 3 elements of the File Descriptor Table are mandatory to allow for standard output, standard input and standard error. The entries in this table are used by the Unix emulation library.

Private Data Pointer Table (mandatory) This area contains pointers to private data. The table is indexed by the capability index of the executing code and is used to locate data used by the executing code.

Default Heap (optional) The default location for the creation of the heap.

Default Stack (optional) The default location for the creation of the stack.

To start a process, the shell takes a code object and a data object for the program to run in the new process. The data object is duplicated. A new process is made by invoking the kernel. The capabilities for the code object and the duplicate data object are passed to the kernel as autoload capabilities[7], the stack pointer and program counter are set and the wakeup time for the new process is set to forever. After the new process is created, Shell, modifies the pages of the object loaded into the shell's address space. Shell writes into the file descriptor table the capabilities for the new process's standard input, output and error, and any other file descriptors that are required. The command line arguments and a capability for the object containing the process's environment strings are written into the heap space. A message is sent to the process to wake the process up.

The method used to create processes allows multiple copies of a program to be run simultaneously. The scheme is economical of both disk space and memory space as it shares a single image of the code. The data is duplicated to prevent multiple copies of a program interfering with each other.

8 Wyrm

Wyrm[8] is an arcade style game inspired by the games nibbles[7] and worm[12]. Apart from its frivolous value, Wyrm has been used to test the responsiveness of the interface and a number of IO mechanisms.

The current version of Wyrm makes use of the Unix emulation stream IO code to communicate via standard IO with Glui which draws the parts of the game on the

[7] Autoload capabilities are automatically loaded into the address space of a process when the process is created.

[8] A wyrm is a mythical creature of great power. The game was sarcastically named wyrm because of its lack of speed. After tracing a number of implementation problems in the stream IO code Wyrm now proudly lives up to its name.

screen. This version is highly responsive and shows that the two layers of software provided by the existing IO structure are sufficiently quick for highly interactive applications.

Early in the development of the Unix emulation libraries a misplaced *fflush* had caused us to believe that the system performance was inadequate. At that time, a version of Wyrm which made direct use of the screen was written in an attempt to determine where the bottleneck lay. This resulted in changes in the design of the kernel and Glui to correctly support the sharing of memory mapped buffers.

9 Conclusion

This paper has outlined four applications implemented under the Walnut Kernel. These initial applications have influenced the design of the kernel resulting in the introduction of several mechanisms:

- Protected freeze and thaw
- SRMULTILOAD system right
- Cooee messages and the Peek system call
- Capabilities with user specified passwords
- The publicly-readable Wall

The password-capability model has been shown to provide adequate performance for a variety of tasks. Although the SRMULTILOAD right fundamentally changes one of the bases of the capability model by introducing a capability useful only to a particular process, this change has left the remainder of the model unaffected.

Techniques familiar to programmers familiar with GUIs have been shown to be applicable to the Walnut Kernel. This similarity implies that an available pool of skilled programmers can be easily cross trained to work under capability based operating systems.

The IO and operating system have been demonstrated to be sufficiently responsive to be used for highly interactive tasks.

Acknowledgments

The authors would like to acknowledge the following contributions:

- Mr Carlo Kopp - author of the UNIX compatibility libraries.

The 'Secure RISC Architecture' project is supported by a grant from the Australian Research Council (A49030623). Maurice Castro is a recipient of an Australian Postgraduate Research Award.

References

[1] M. Anderson, R D. Pose, and C S. Wallace. A Password-Capability system. *The Computer Journal*, 29(1):1–8, 1 1986.

[2] M. Anderson and C S. Wallace. Security management in a password-capability system. Technical Report 56, Department of Computer Science, Monash University, 8 1985.

[3] M.P. Atkinson, R. Morrison, and G.D. Pratten. Designing a persistent information space architecture. In *10th IFIP World Congress, Dublin*, pages 115–120, 1986.

[4] Alan C. Bromberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonath on S. Schapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112. USENIX Association, 4 1992.

[5] Maurice Castro. The walnut kernel: User level programmer's guide. Technical Report 95/222, Department of Computer Science, Monash University, 5 1995.

[6] Jefferey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single address space operating system. Technical Report Technical Report 93-04-02, Department of Computer Science and Engineering, University of Washington, Seattle, USA, April 1993 (Revised January 1994).

[7] Microsoft Corporation. Qbasic nibbles, 1990. Source code in BASIC.

[8] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 3 1966.

[9] R S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, 7 1974.

[10] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. Mungi: A distributed single address-space operating system. In G. Gupta, editor, *Proceedings of the Seventeenth Australian Computer Science Conference*, pages 271–280, 1 1994.

[11] Leslie J. Keedy. The Monads view of software modules. In A J H. Sale and G. Hawthorne, editors, *Proceedings of the Ninth Australian Computer Science Conference*, 8 1982.

[12] Michael Toy. Worm, 1991. Part of the Berkeley Unix Distribution, Source code in C.