Research into the Development of the RHODOS Multi-threaded Microkernel*

G. Wickham, M. Hobbs, A. Goscinski

{gjw, mick, ang}@deakin.edu.au

School of Computing and Mathematics
Deakin University, Geelong
Victoria 3217, Australia.

Abstract

An important factor contributing to an operating systems' performance is the design and efficiency of the kernel code. This is especially true for microkernel based operating systems where the microkernel and kernel provides support for system servers. RHODOS is a microkernel based distributed operating system designed as a testbed for the study of the numerous research topics available in this area. The current version of RHODOS uses a single threaded microkernel, which fulfilled our requirements of providing a stable platform. A significant drawback of a single threaded architecture is the difficulty with which paging can be implemented. This report details the current state of the RHODOS single threaded microkernel and provides justification for the implementation of a multi threaded microkernel.

*This work was partly supported by Australian Research Council under Grants A48831034 and A49232429, and the Deakin University Research Grant 0504010151.

INDEX

1 Introduction

The desire to improve the performance, efficiency and utilisation of current computer systems has resulted in a shift in research effort, with a greater concentration being made within the area of distributed systems. A specific field of distributed systems that is proving to be a dynamic area is with that of distributed operating systems. One such system is RHODOS, a high performance distributed operating system designed to act as a test bed for studying the many components which go to form a distributed operating system [Gerrity et al. 91]. A current trend in modern operating system design is to build the system as a set of cooperating kernel server processes supported by a lightweight kernel, commonly known as a microkernel. RHODOS has been designed with the microkernel architecture forming an abstract machine on which kernel and system servers provide the bulk of the services and facilities expected from any operating system [Toomey et al. 92], [De Paoli et al. 94]. Other systems based on the microkernel approach include MACH [Accetta et al. 86], QNX [Hilderbrand 92] and Chorus [Rozier et al. 88].

The current implementation of the RHODOS microkernel has a simple restriction that process scheduling cannot be performed whilst executing any system calls within the microkernel. This mode of operation is termed ‘single-threaded’. Only when in user mode can a process be scheduled out and a new one scheduled in.

To improve the performance characteristics and overall flexibility of the RHODOS microkernel it has become important that the present single-threaded microkernel be modified to accommodate multi-threading of both synchronous and asynchronous events. The primary objectives obtained from this change will be:

- The ability to perform demand paging, and hence implement a richer virtual memory system; and
- The implementation of pre-emptive scheduling. This means that a process can always be rescheduled whenever a context switch clock tick (interrupt) occurs. This is in stark contrast to an implementation of a single-threaded microkernel where pre-emption cannot occur if a system call or hardware interrupt handler is executing.

A multi-threaded microkernel is one that supports multiple threads of execution within microkernel code itself. Explained more simply, it permits a system call to be at any time during the call, suspended and to remain suspended while another process is given execution time.

The transformation of RHODOS from a single-threaded to a multi-threaded microkernel will occur in two parts. The first being the implementation of separate system call stacks for each pro-

cess, and the second being a protection mechanism for data structures used within the microkernel.

In this report, firstly the Nucleus — RHODOS Microkernel is characterised (Section 2); with emphasis placed on the limitation of the single-threaded architecture. The design requirements of a multi-threaded architecture are presented in Section 3. In particular, the stacks which form the basis of the multi-threaded microkernel are discussed. The manipulation of the internal data structures of the Nucleus are presented. The conclusion shows the design and implementation steps to be performed in order to build a multi-threaded microkernel.

2 The Nucleus

The Nucleus, which is the current implementation of the RHODOS microkernel, only supports pre-emptive scheduling when the currently executing process is in user mode. With the single-threaded paradigm it is impossible to perform a context switch if the context switching routine is invoked before a system call is completed. However a mechanism has been implemented where the process can invoke the scheduling algorithm directly at the very end of a system call if a context switch has occurred. This method only works because the system call can run to completion and hence leaving the Nucleus in a restartable state.

2.1 Changing Mode

A hardware technique often employed to support protection and to ensure integrity of operating system code is through the use of privilege modes. A simple implementation of privilege modes would include two levels (as is used with the RHODOS implementation platform [Motorola 85]), generally known as *User* and *Supervisor* modes. Where the *User* mode is the general working mode used by normal user and server processes, whilst the *Supervisor* mode is the higher privileged mode that allows access to privileged memory locations and instructions.

In RHODOS (as with most operating systems) there are three methods by which a process can change mode from that of a user process to that of the microkernel. These include:

- *System Calls* — A process may request services or resources from the microkernel via a ‘trap’ into the Nucleus;
- *Exceptions* — If a running process causes an error (i.e., divide by zero, page fault, invalid address, etc.) the Nucleus forwards a message detailing this event onto the respective kernel server process; and
- *Hardware Interrupts* — While a user process is running, hardware devices may cause asyn-

chronous interrupts indicating that they require attention. The processor is placed in supervisor mode and an interrupt frame is placed on the interrupt stack. This frame contains the details of the interrupt, which the interrupt handler uses to correctly process the event.

The three events that change the processor from user mode to supervisor mode are shown in Figure 1. System calls and exceptions are synchronous events in the respect that they are generated directly from the actions of a process running in user mode, whilst the hardware interrupt is an asynchronous event in that its behavior is stochastic.

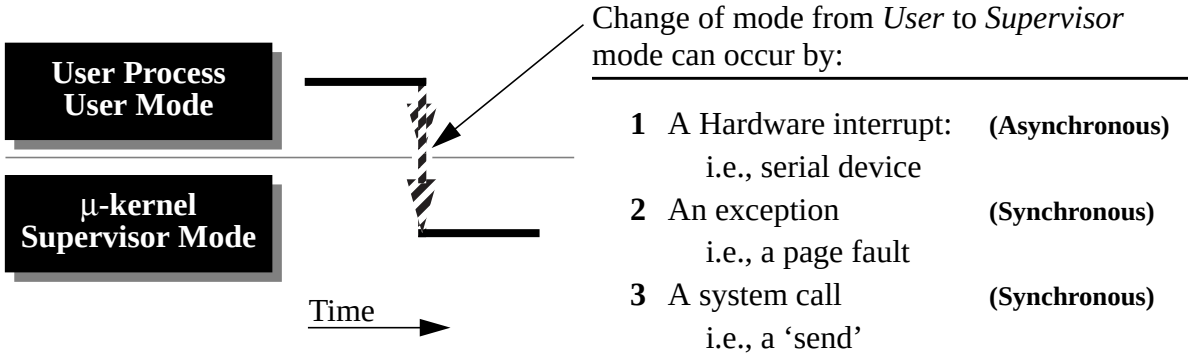

Figure 1: User to Supervisor mode events.

2.2 Limitation of the Single-threaded Microkernel

The RHODOS microkernel (or Nucleus) presently allows only one system call to be active at any instant in time and prohibits a context switch from occurring during the execution of a system call. This means that when an event such as a page fault is produced and in turn this interrupts a system call, the microkernel must handle the event fully itself and then return to complete the offending system call before any context switch can occur.

In line with the microkernel paradigm, the kernel servers and system servers should handle most events in a hardware independent manner, with the microkernel providing the interface between the servers and the hardware. This presents a problem in that a server may need to be scheduled in to resolve an event, which is impossible if the original process is suspended during a system call. Hence, system calls must run their full cycle, or be able to be placed into a state that can be restarted at a later time. Both of these solutions involve an unjustifiable overhead in preserving the state of the original process.

An example of a solution that is currently implemented is that of the page management scheme 'Copy-on-Write' (COW). When a page protected by COW is written to, the microkernel

will: acquire a page from the free page list; duplicate the offending page; and then restart the process. Currently, this is perfectly acceptable and works well, however serious problems may occur if there is a shortage of free pages. If there is no free page available then this will cause a serious error within the microkernel, and possibly an ungraceful halt of the operating system.

Demand paging, as would be used to implement virtual memory, is a veritable nightmare to implement with a single-threaded microkernel. In theory a page fault can occur at any time, whether in a system call or user space. Suspension of a process in user space presents no difficulties. However, with respect to a previously mentioned point, if a page fault occurs within a system call then the complexity of the handling is significantly increased.

To be able to handle page faults from within a system call it is necessary to construct them in a re-entrant fashion. This means that if a system call cannot complete a task then it must checkpoint the location where it was up to and then invoke the context switching algorithm. When the process can be re-scheduled the system call must be re-invoked from its beginning, and check to see if any checkpoint was set. If so, the process needs to jump to that location and then continue execution. However, building a greater awareness and intelligence into the RHODOS system call is an overhead that cannot be warranted within the microkernel paradigm.

2.3 Overheads on System Calls

The microkernel paradigm implies an image of a very lean kernel that is used extensively for communication between servers and user processes. As the microkernel itself is completely unaware of any services offered to users at the application layer the microkernel does not need to offer many system calls. All communication between user processes and the system and kernel servers occurs via the use of remote procedure calls (RPCs). As every system call at the application layer requires communication with a server and this is achieved via RPCs, it is obvious that the communication primitives provided by the Nucleus will be used heavily. To ensure that the operating system operates as efficiently as possible, it is important that the microkernel is very lean to accomplish this.

However, as the RHODOS Nucleus is a single-threaded microkernel it requires system calls be written to stringent guidelines which impair their simplicity and efficiency. The basic guidelines followed in writing system calls in the current version of the Nucleus include:

- Before a system call attempts access to any page within the users virtual address space, it must determine that the page is currently mapped in.
- If at any stage the system call cannot continue because of a missing resource (page) then it

must: inform the appropriate manager for request of the resource; checkpoint the current position within the system call; and then set the system call restart flag and terminate the system call (this will re-start the system call the next time this process is scheduled).

The above two points presents a severe performance impairment of the single-threaded microkernel. As most system calls access user space for either reading or writing, each structure that is accessed must be verified by interrogating the internal memory maps.

A simpler solution would allow page faults to be generated by the memory management unit (as with any other exception or event), with notification issued to the appropriate servers by the microkernel. However, this simpler solution is impossible with the current version of the Nucleus.

2.4 User and System Stacks

An executing process requires an area in which to store temporary and transient data. This area is called a ‘stack’ and is directly supported in hardware by most, if not all, processor architectures. Each process is allocated its own private stack which is termed a ‘user stack’. While on processors with hardware support for more than one level of privilege there is usually support for multiple stacks.

RHODOS is currently being implemented on a Sun 3/50 which uses a Motorola 68020 [Motorola 85] as a central processing unit. This processor supports three separate stacks, which are the: user stack, interrupt stack and master stack. Whenever the processor is in the lower privilege mode it uses the ‘user stack’. When the hardware privilege flag is enabled the processor uses the ‘interrupt stack’ or the ‘master stack’ depending on the setting of an additional flag in the status register.

Currently within RHODOS, as shown in Figure 2, there is a separate user stack for each process, one interrupt stack and one master stack. The master stack is not purposely used by RHODOS but must be defined, whilst the interrupt stack is used for all system calls, exceptions and interrupts.

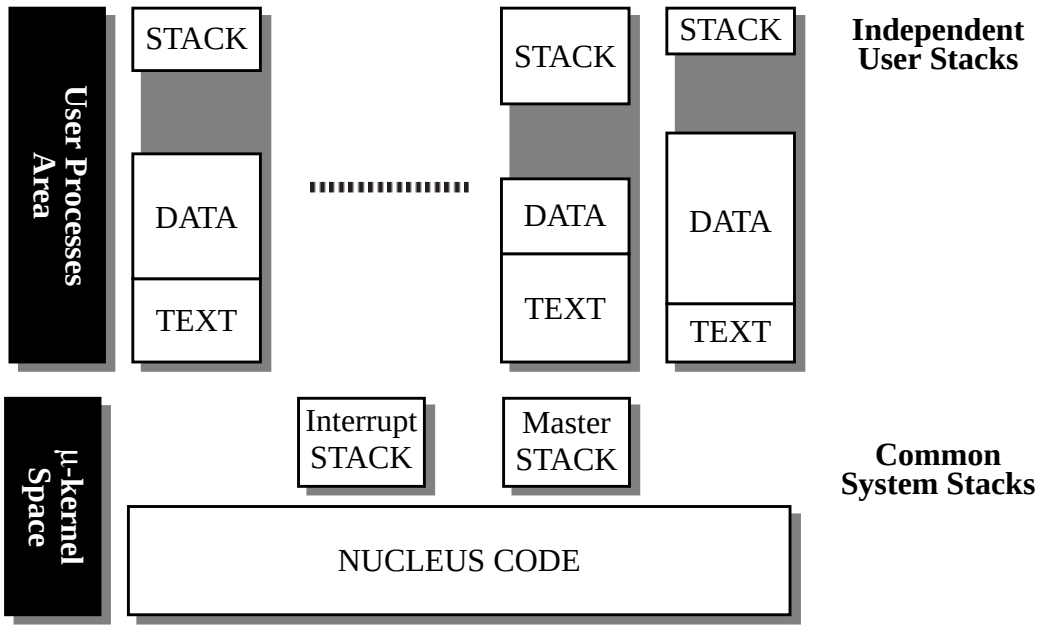

Figure 2: Current RHODOS Stack Implementation.

The reason why RHODOS is currently termed a single-threaded microkernel is that with only one stack being used for the hardware privilege level, that is the interrupt stack, it is impossible to perform a context switch if this stack is not empty. This is due to the fact that it is impossible to guarantee the order in which the suspended processes will be restarted. The order is determined by the scheduling policies and external events that are affecting the system at that instance.

3 Design Requirements of a Multi-threaded Microkernel

The intrinsic benefit of a multi-threaded microkernel is the ability to suspend a process when it has not completed a system call and to perform a context switch, thus permitting another process to execute. To achieve this goal it is necessary for each process to have its own private privilege stack, which is called a ‘master stack’ on the M68020 [Motorola 85]. This concept is shown in Figure 3.

The separate system stack for each process will be used only for the processing of system calls (software traps) invoked from a user process. If a hardware exception or an interrupt occurs, it will be processed in the environment of the interrupt stack thus separating interrupts and hardware exceptions from software traps. This separation will permit a context switch to occur when-

ever the context switch algorithm is invoked the interrupt (driven by a periodic timer).

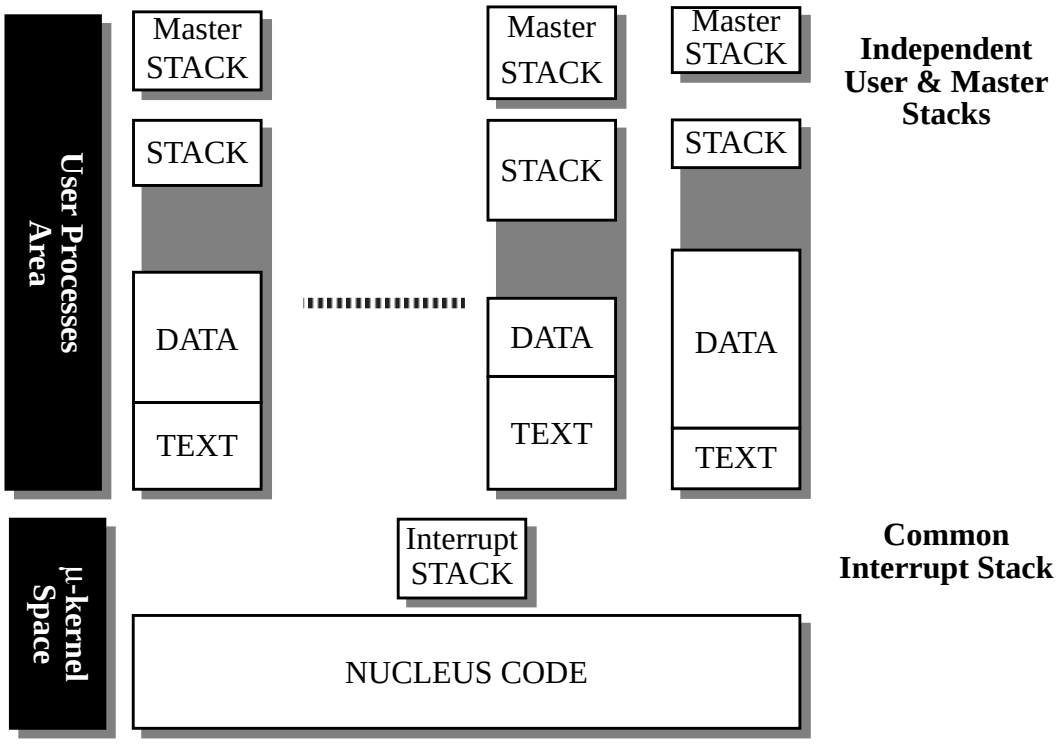

Figure 3: Proposed RHODOS Stacks Usage.

Likewise, any hardware exceptions generated by the memory management unit will be processed on the interrupt stack, thus isolating the offending process from the underlying mechanisms of kernel server notification and scheduling.

3.1 Basic Problems

With a multi-threaded microkernel system calls can operate in a non-contiguous (or atomic) way. It is possible for a system call to be suspended during execution and to be resumed after many other processes have had the chance to execute. For example, a process may be part way through a system call when the scheduler is invoked. Another process may then issue a ‘kill’ to the first process that results in the original process being terminated before the system call can complete.

An implementation of a multi-threaded microkernel needs to be able to guarantee that this type of situation (although pathological in nature) will not adversely affect the performance or stability of the operating system and therefore requires careful thought and design.

3.2 Manipulation of the Internal Microkernel Data Structures

The microkernel uses many data structures to store internal information and user data as it flows from process to process. All of these structures are implemented using linked list techniques which present special problems for the multi-threaded microkernel engineers.

To maintain microkernel integrity throughout its operation it is necessary to ensure that the following condition of implementation is always adhered to:

> *Any data structure that is maintained by the microkernel and manipulated by a system call must always be attached to a known linked list while outside of the atomic locking mechanisms.*

This statement stresses two main points. The first is that all linked list manipulations must be protected by a locking mechanism. It is possible to disable and enable processor interrupts on the Sun 3/50 through the use of a hardware register. Before linked lists are manipulated it is necessary to disable interrupts, perform the manipulation and then re-enable interrupts.

Whilst interrupts are disabled, they are not passed to the processor but are left in a pending state. The interrupts are forwarded as soon the hardware flag is re-enabled, thus ensuring that virtually no interrupts are left unprocessed.

The second point is that the system call must ensure that data structures donated by the microkernel must be attached to known list locations during the period in which interrupt locking is enabled. This feature will ensure that if the process is terminated at any time then the microkernel can reclaim data structures that are on loan to the user process.

4 Conclusion

The main disadvantage that has resulted from use of a single-threaded microkernel is that it is impossible to perform a context switch when a system call is active. This has resulted in the inability to implement a virtual memory page management technique whilst adhering to the microkernel design paradigm. A paradigm which requires only the bare minimum number of services and functions, thus providing an abstracted hardware foundation that supports an extremely flexible and modular system.

The major benefits of an upgrade to a multi-threaded microkernel is the ability of closer conformance for the RHODOS Nucleus to the microkernel paradigm, and also provide support for virtual memory. To perform the upgrade, the differences and requirements to transform RHODOS from a single-threaded to a multi-threaded microkernel architecture have been researched. This

resulted in two main areas requiring development and modification. The first, is the use of multiple stacks for each process so that system calls can be suspended part way through execution. The second is that of resource protection and reclamation within the system calls that are used by processes. These areas are detailed and include a general synopsis of a technique which could be used in relation to the current implementation platform of a Sun 3/50.

5 References

[Accetta et al. 86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young. "*Mach: A New Kernel Foundation For UNIX Development.*" Summer USENIX Conference, Atlanta, 1986.

[De Paoli et al. 94] D. De Paoli, A. Goscinski, M. Hobbs and G. Wickham. "*RHODOS — A Microkernel based Distributed Operating System: An Overview of the 1993 Version"*. Technical Report TR C94/, School of Computing and Mathematics, Deakin University, Geelong, (in preparation).

[Gerrity et al. 91] G. Gerrity, A. Goscinski, J. Indulska, W. Toomey and Z. Zhu. "*RHODOS — A Test Bed for the Studying Design Issues in Distributed Operating Systems*". Proceedings of the 2nd Singapore International Conference on Networks (SINCON'91). September 1991.

[Hilderbrand 92] D. Hildebrand. "*An Architectural Overview of QNX*". Proceedings of Workshop on Microkernel and other Kernel Architectures. April 1992.

[Motorola 85] "*MC68020 32—Bit Microprocessor User's Manual"*. Second Edition. Prentice Hall, Inc., Englewood Cliffs, N.J. 07632.

[Rozier et al. 88] M. Rozier, V. Abrossimov, F. Armand, M. Gien, M. Guillemont, F. Hermann and C. Kaiser "*Overview of the Chorus Distributed Operating System*". Montigny-le-Bretonneux (France), June 1988.

[Toomey et al. 92] W. Toomey, A. Goscinski and G. Gerrity. *"The Nucleus — Microkernel for the RHODOS Distributed Operating System"*. Proceedings of the 1992 IEEE Conference in Computers, Communications and Automation Towards the 21st Century — TENCON'92.

[Wickham et al. 94] G. Wickham, D. De Paoli, M. Hobbs. *"Implementation of the RHODOS Space Manager"*. Technical Report TR C94/, School of Computing and Mathematics, Deakin University, Geelong, (in preparation).