Migration of an Operating System to an Object-Oriented Paradigm

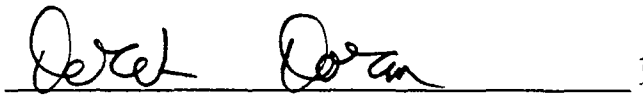
Derek Doran B.Sc.

Submitted for the award of Master of Science

Supervisor:
Dr. John Waldron
School of Computer Applications
DCU

Month of Submission : January 1996

I hereby certify that the material which I now submit for assessment on the program of study leading to the award of Master of Science is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work

Signed  Date 20-1-96

Acknowledgements

This thesis is dedicated to my parents, John and Aileen Doran, whose endless support and faith in my abilities have made it impossible for me to foul up, however hard I have tried

I am extremely grateful to my supervisor and friend, Dr John Waldron He has guided this thesis from a convoluted idea to its current state At every stage of its development he was generous with his time, his comments, and his criticisms John helped keep the thesis (and me) on the straight and narrow and it would have been a much inferior piece of work without his involvement

Finally a big thank-you to my friend and co-conspirator Sinead Masterson. Her understanding of my popcorn addiction, help with the Crossaire, and love of movies were an integral part of this thesis The experience will inspire me to great things some day, I'm sure

Derek Doran

Table of Contents

Table of Figures

***Derek Doran** (94970777)*
Computer Applications
Dublin City University

Dr John Waldron
M Sc Thesis Abstract
March 15th 1996

Object-oriented MMURTL: The migration of a microkernel operating system to an object-oriented paradigm

Operating System design has moved from monolithic systems such as UNIX, where all system services are implemented in a single kernel, to microkernel designs where the majority of system services are conducted in user space A recent trend in operating system design has been to use architectural models based upon the object-oriented paradigms This approach promotes the modelling of system resources and resource management as an organized collection of objects in such a way that the mechanisms, policies, algorithms, and data representations of the operating system are suitably encapsulated by the objects

Much of the research in this area to date has concentrated on the uses and benefits of object-oriented operating systems in the distributed systems arena Similarly, almost all of these systems have been designed from the ground-up

I believe that the progression towards object-oriented operating systems is likely to involve current operating systems incorporating and assimilating object-oriented features into their existing designs in a gradual manner, rather than an overnight switch to a new technology

In this light, the purpose of my thesis is to take an existing operating system and to propose a design which would migrate the original operating systems' facilities and features to an object-oriented paradigm. This thesis also evaluates the advantages and disadvantages of such a design over the existing one Finally, future enhancements and directions are proposed based on the new operating system design

Chapter 1

Introduction

1.1 Overview

This chapter briefly describes the progression of operating system research from monolithic to object-oriented systems This is followed by a description of the aims and intentions of this thesis An introduction to object-oriented concepts follows This is given as a precursor to a discussion of the object-oriented approach to operating system design and implementation Finally, a brief of summary of each of the remaining chapters of this thesis is presented

1.2 Operating System Research Trends

Operating system design has moved from monolithic systems such as UNIX [Ritchie 75] where all system services are implemented in a single kernel, to microkernel designs such as Mach [Rashid 86], where the majority of system services are conducted in user space

Microkernel architectures are designed to isolate the most essential functions of an operating system to a small core of code that runs in privileged mode The remainder of the system is supported as a set of applications that run in user space, isolated from the kernel by a set of well defined interfaces Such systems have been found to be easier to maintain, more extensible and more scalable than monolithic designs One of the major problems with such microkernel systems, however, is the high overhead caused by cross domain interprocess communication (IPC) calls This raises further questions as to which services should be situated inside the kernel and which should remain outside This is a design decision which is dependent on individual system implementations [Campbell 95]

Despite advances in these technologies; current operating systems still suffer from problems which have either been inherited from previous generations, or else which have appeared as a result of the rapid advancement of hardware technologies [Mitchell 93] These include

- the cost of maintaining and evolving the system, including both the kernel and non-kernel code
- the lack of system support for software reuse
- the difficulty of building distributed, multi-threaded applications and services
- the difficulty of supporting time-critical media (eg audio and video), especially in a networked environment

A recent trend in operating system design has been to use architectural models based upon object-oriented paradigms This approach promotes the modelling of system resources and resource management as an organized collection of objects in such a way that the mechanisms, policies, algorithms, and data representations of the operating system are suitably encapsulated by the objects [Campbell 91]

The movement to object-oriented operating systems is driven by two main motivations Firstly, there is a desire to create systems whose design enables easier and faster development and modification of system components Secondly, there is a movement towards operating systems where the distinction between system components and third-party applications is reduced in such a manner that additional components can become linked to the operating system with ease [Hamilton 93]

1.3 Research Aims

I beheve that the progression towards object-oriented operating systems will be more a migratory movement than a revolutionary one That is to say, instead of switching directly to new operating systems with radically different designs and technologies, existing operating systems will instead begin to incorporate and assimilate object-oriented features into their existing designs in a gradual manner This is based upon the belief that unless users are presented with a bridge between old and new technologies, the movement to operating systems with radically new designs will be a slow and problematic one A possible example of this in action has been the disappointing markets performance of the NextStep operating system, a system which has received much critical acclaim for both its design and implementation

Most research in the area of object-oriented operating systems to date, however, has centred on creating brand new operating systems which have been designed from scratch with relatively little emphasis placed upon existing systems Object-oriented operating systems designed in this manner tend not to have solved the problems found in previous generations of operating systems, but instead have placed them to one side and started again This is perfectly acceptable when designing a state-of-the-art system, however a gap remains in attempting to discover a suitable migration from existing technologies to object-oriented operating systems

In this light, the purpose of this thesis is to present a study of the migration of an existing non-object-oriented operating system and proposing a design which replicates its original facilities and features using an object-oriented paradigm.

The emphasis of this thesis is placed on the identification of the problems which were encountered, the architectures which were investigated and the solutions which were provided during the course of the migration to the object-oriented paradigm. In addition a comparison is made between the original system and its new, object-oriented, incarnation

In order to do this, this thesis concentrates on providing an object-oriented architectural design which as much possible duplicates the capabilities, though not the implementation, of the original operating system. Although it would have been possible to improve MMURTL's capabilities as a result of the introduction of the object-oriented paradigm, this would not have allowed a fair comparison between the original system and the newly designed one

The migration of an existing operating system to the object-oriented paradigm is an important step, however it is only the first of many This thesis also discusses the state of the new Object-Oriented MMURTL (OO-MMURTL) as it stands In addition a proposal as to future enhancements and directions which can be undertaken based upon, and as a direct result of, the new operating system design is presented The most obvious of these being a distributed implementation of MMURTL which embraces objects at the core of its design

1.4 Object-oriented concepts

This section presents an introduction to the concepts of object-orientation. Features relating specifically to object-oriented operating systems are dealt with later in this chapter. There are many different interpretations of the object-oriented concepts. This is probably highlighted by the terms *Object* and *Class* whose definitions may vary from one system to another. The definitions presented below are based upon those given by [Weiner 90], and are used throughout this thesis.

1.4.1 Objects and Classes

An *object* is an encapsulated entity including code and data. Each object is fully described by a combination of its data, referred to as *attributes*, and its functions, referred to as *methods*. Objects can be dynamically created and deleted. An *instance* of an object is the realisation of is interface.

A *class* defines a collection of objects which exhibit identical behaviour. Objects of the same class share common code, but each instance of the class, i.e. each object, retains its own set of data. A class allows a taxonomy of objects to be built based upon an abstract or conceptual entity, whereas an object defines the specific state of each particular entity.

1.4.2 Encapsulation and Information Hiding

By storing both a set of code and data together in an object, the designer is stating that the information held in the attributes and the actions performed by its methods which may be performed upon them are conceptually related, and it makes sense to store them together. This is known as *Encapsulation*. The essence of encapsulation is to enable a programmer to view a set of related items as a conceptual unit.

An object may have *public* or *private* methods. Its private methods may only be invoked from within the object, while its public methods describe the methods which may be called from inside or outside of the class. Hence the public methods and public attributes are often referred to as the interface of an object. The principle whereby certain elements of a class remain private is known as *Information Hiding*.

Information hiding allows a class to publicly declare what actions it is capable of performing, without divulging how these actions are performed This provides two important benefits Firstly, information hiding allows both the designer and programmer to take a more abstract view of objects Secondly, the internal representation of an object may be developed in the future without affecting the public interface, and therefore application programs which invoke the object

1.4.3 Abstract Classes

An abstract class is a template for other classes Similar to ordinary classes, it is described by a set of attributes and methods, however at least one of its methods is not fully described It is instead defined without being implemented Such methods are known as *virtual methods*

Since they are not fully described, abstract classes may not be instantiated Instead, other classes will inherit from an abstract class, implementing their own versions of its virtual methods Such classes are known as *concrete classes* They can be instantiated In this fashion, abstract classes can be used as templates to describe partial behaviour of concrete classes

1.4.4 Class Hierarchies

A set of classes can be hierarchically organised in order to further describe their reuse and evolution A *class hierarchy* is an architectural design for describing an object-oriented system. Class hierarchies provide infrastructural and architectural guidance while designing the interrelationships between component objects of a system.

Class hierarchy diagrams depict the relationships between objects and the interface inheritance between them. By using class hierarchies, system objects can be classified into categories The root leaf of a hierarchy is often represented by an abstract class By presenting information about object classification in this way, a detailed yet high level description of the system may be achieved

In further levels of the hierarchy, concrete implementations of classes will replace abstract classes, thus specialising and further describing the behaviour of the system objects being documented

A system described by a class hierarchy is, at its highest level, a single framework which guides the design of subframeworks The framework for a system provides generalised components and constraints to which specialised subframeworks must conform.

The root of the class hierarchy is shown at the top of the diagram. Specialisations of classes are represented in a top-to-bottom manner In Figure 1 1, SubClassOfA is a subclass of ClassA.

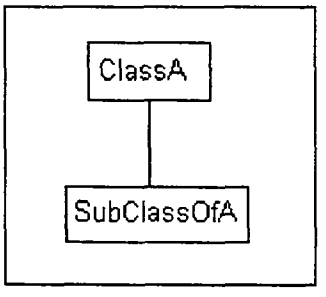


Figure 1 1 A simple class hierarchy

Several classes can inherit from the same class Object-Oriented MMURTL does not require, and therefore does not support multiple inheritance, hence each class may inherit from only one parent class

Abstract classes are shown in bold type This is demonstrated in Figure 1 2 which shows the class hierarchy for the Process subsection of the Chorus operating system, which is described in further detail in *Chapter 3 Object-Oriented Operating Systems*

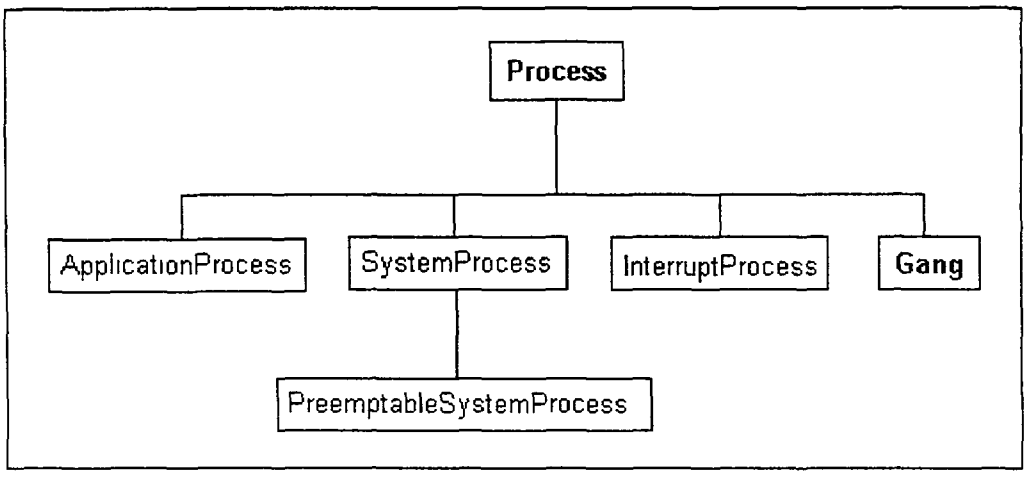


Figure 1 2 A class hierarchy featuring abstract classes

In Figure 1 2, *Process* is the highest level class in the hierarchy It is an abstract class It serves as a template for all further process objects in the operating system. It defines the default behaviour of a process in terms of its methods along with its default attributes

The virtual methods of the *Process* class must be defined in its subclasses In this way, each of the subclasses will become a specialised version of the default *Process* object, retaining the behaviour of the original class, while at the same time introducing new behavioural nuances specific to the subclass

In Figure 1 2, the behaviour of a *Process* is further defined and specialised in the subclasses *ApplicationProcess*, *SystemProcess*, and *InterruptProcess* In turn, *PreemptableSystemProcess* is a specialisation of *SystemProcess* This demonstrates the ability to provide a more specific subclass implementation at each level of the class hierarchy

A *Gang* is a specialisation of *Process* however it too is an abstract class Therefore it is impossible to create an instance of a *Gang* object Instead, system developers or application programmers must create a new subclass of the *Gang* class, providing implementations of its abstract methods, so that the new subclass is a concrete class and may therefore be instantiated

1.5 What is an Object Oriented Operating System?

In its purest form, an object-oriented operating system is one in which all system components, mechanisms and resources are modelled as objects. Object-oriented techniques such as abstraction, encapsulation and inheritance are used to perform the modelling.

The system developers design a set of objects which will serve as generic system components. These are then customized through object-oriented interfaces and specialization to suit the needs of each individual component of the system. In this way, inheritance encourages the reduction of the implementation of different system services to a small number of classes that can be specialized and combined to achieve a desired result [Campbell 93a]. This approach supplements the microkernel, client/server operating system organizations used in systems like Mach [Rashid 86], V [Cheriton 88], and Amoeba [Tannenbaum 90] to introduce more flexibility for application support.

The object-oriented approach to operating system design and implementation also serves to provide a distinct boundary between the various system components, thus easing their development, maintenance and extension.

There are two non-exclusive approaches to incorporating objects in the architecture of an operating system [Krakowiak 93]. Each of these is described below, with example implementations of each system given.

1.5.1 Object-Based Operating Systems

These operating systems provide support for objects. This usually takes the form of additional structures above the basic layers of the operating system which will support objects. The level at which object support is provided varies from one implementation to another.

Guedes [Guedes 92] presents an implementation of the Mach 3.0 multi-server system which supports objects. The operating system is decomposed between a microkernel, a set of system servers running in user-mode, and an emulation library executing in the address space of applications.

In this implementation of ObjectMach [Julin 89], all of the interfaces provided by the system servers are object-oriented and all of the services provided by them are defined in terms of system objects, such as files, directories, devices etc In addition, both the servers and the emulation library are written in an object-oriented language

1.5.2 Object-Oriented Operating Systems

In this architecture, the entire operating system is implemented as a set of objects Object-Oriented Operating Systems encourage customisation of not only the interfaces, but also the operating system itself through the use of object-oriented techniques such as inheritance

Choices is a fully object-oriented operating system which also provides support for the object-based architecture It is described in detail in *Chapter 3 Object-Oriented Operating Systems*

1.6 Thesis Overview

This is a brief summary of each of the chapters in this thesis, with the exception of this introduction chapter

Chapter 2 The MMURTL Operating System This chapter describes the basic architecture of the original MMURTL operating system. This is presented as a precursor to later chapters which, while describing the new implementation of OO-MMURTL, allude to MMURTL's original design as a point of reference

Chapter 3 Object-Oriented Operating Systems This chapter introduces practical examples of the design of existing object-oriented operating systems This will demonstrate the wide variety of possible implementations when objects are introduced to the design of an operating system. The operating systems presented in this chapter range from those developed by professional developers to those developed by academic institutions, and from object-based operating systems to completely object-oriented systems

Chapter 4 Object-Oriented MMURTL This chapter introduces the basic design and implementation of OO-MMURTL The explicit implementation of each component of the operating system is described in detail in subsequent chapters Operating system concepts and entities which are common to most or all of OO-MMURTL's system components are introduced here This includes the use of objects and the design of object storage containers

Chapter 5 The Process Management Model This chapter describes in detail the design and implementation of the OO-MMURTL process management model, including the class hierarchy and component classes which support this area of the operating system.

Chapter 6 The Messaging Model As in its original implementation, messaging is at the core of much of OO-MMURTL's system behaviour This chapter describes in detail the design and implementation of the OO-MMURTL messaging model, including the class hierarchy and component classes which support this area of the operating system.

Chapter 7 The Memory Model This chapter initially describes the advanced memory concepts involved in MMURTL's original memory model. Following this, the design and implementation of the OO-MMURTL memory model, including the class hierarchy and component classes which support this area of the operating system, are presented

Chapter 8 *Additional Classes* This chapter introduces additional classes which are integral to OO-MMURTL and support the classes presented in previous classes, while not belonging to the subsystem in question The design and implementation of the various class hierarchies and classes which support OO-MMURTL's ready queue, interrupts and timer are described in detail here

Chapter 9 *Design Testing* This chapter briefly describes my experiences during the design and testing of OO-MMURTL

Chapter 10 Conclusions This chapter describes the state of OO-MMURTL as presented in this thesis In addition, possible future enhancements to the new design are offered, along with suggestions as to the new directions which can be undertaken based upon the new operating system design

1.7 Summary

This chapter discussed recent research trends in the area of object-oriented operating systems Problems with the area of migration to object-oriented operating systems were highlighted Following this, the intent of this thesis, i e to redesign an existing operating system architecture using an object-oriented paradigm and document the resulting observations, was set out A brief explanation of object-oriented concepts and terms was given, before the general area of object-oriented operating systems was introduced Finally, a brief overview of each of the remaining chapters in this thesis was given

Chapter 2

The MMURTL Operating System

2.1 Overview

This chapter introduces the origins, intentions, architecture and design of the MMURTL operating system [Burgess 95] In addition, an introduction is made to the three components which form the core of MMURTL its task model, messaging model, and memory model Each of these descriptions will provide a basic overview of the component in question, while further details of each are presented in later chapters, which deal with OO-MMURTL's redesign of the three primary system components

2.2 MMURTL

MMURTL has been under development by Richard Burgess since 1991 It was designed to run on most 32-bit ISA PC compatibles However, despite its incompatibilities with other operating systems, MMURTL remains an extremely powerful operating system.

MMURTL is a 32-bit message based, multitasking, real-time operating system designed around the Intel 80386 and 80486 processors on the PC Industry Standard Architecture (ISA) platforms The name is an acronym for Message based MUltitasking, Real-Time, kerneL

MMURTL has been designed from the ground up, resulting in its incompatibility with other popular operating systems Its file system however, is FAT compatible, allowing MMURTL to reside on a DOS formatted hard drive in the same partition as DOS itself.

Although DOS is required to run before MMURTL is invoked, DOS is not powerful enough to accommodate MMURTL's mechanisms As such, MMURTL merely makes use of DOS as a boot program. Once the MMURTL loader is executed, the operating system executive gains control of the hardware and all traces of DOS are removed from memory

2.3 Reasons for using MMURTL

Burgess recommends the use of MMURTL as a learning and reference tool for programmers working in 32 bit environments In the field of education, MMURTL can be used to teach introductory multitasking theory, paged memory operation and management, hardware and software task management, and real-time message based operating systems

Although MMURTL is essentially a little known operating system, there were several motivations for the use of MMURTL as the basis of this thesis

- *Code Availability* - A major factor which dictated the choice of operating system was the essential need to have access to the complete source code of the system. The importance of this requirement dictated by the need and desire to achieve a complete understanding of the inner mechanisms and workings of the operating system being studied, so that an accurate model of its behaviour, both internally and externally, could be made

- *Documentation* - This requirement is related to the previous point In order to completely understand the system's mechanisms and policies, an operating system was sought which provided a set of documentation which was sufficiently detailed and accurately documented

- *Programming* - The operating system was required to provide a well-defined application programming interface which would allow both system developers and application programmers to harness its power and capabilities while at the same time ensuring ease of use In addition, the possibility of ill behaved programs causing damage to the rest of the operating system must be minimised

- *Platform* - The vast majority of work in the area of object-oriented operating systems research takes place on workstation platforms In order to present a diverse view of an object-oriented operating system, a system which was based on an IBM PC compatible format was sought With this in mind, a further specification with regards to power had to be made Due to the advanced nature and slower processing of object-oriented operating systems, the Intel based operating system had to have been designed explicitly to gain advantage of the advanced capabilities of later generation chips such as the 80386, 80486 and Pentium.

MMURTL is a 32-bit message based, multitasking, real-time operating system, designed explicitly for use on Intel 80386 and 80486 platforms It makes use of processor specific structures to provide more advanced facilities through the operating system. An example of this is the use of the 386/486 memory paging mechanisms to dispense with the segmented memory model as used by DOS and to replace it by a simpler yet more powerful flat 32-bit virtual memory model.

The entire MMURTL source code is available at the Dr Dobbs Journal FTP[1] site on the Internet, and also on a CD-ROM available inside Richard Burgess' recently published book entitled *"Developing Your Own 32-bit Operating System"* [Burgess 95], which uses MMURTL as the basis for its case study of a practical, modern, 32-bit operating system.

In addition, Burgess provides clear and concise documentation of MMURTL along with plentiful use of comments throughout his source code Finally, MMURTL's system calls are well designed and clearly defined providing ease of use in the C programming language

The only specification which MMURTL failed to satisfy was that of providing a sufficiently safe programming and operating environment in which to program both applications and the system. Although a debugger forms an integral part of MMURTL, little language or run-time support is provided by the operating system. MMURTL provides no inbuilt mechanisms to reduce the possibility of errant programs bringing down the entire operating system.

[1]ftp dobbs com in directory /pub/source/MMURTL

Although this is a serious omission in a modern operating system, it was acceptable in the circumstances, given that C++ as a language, and object-orientation as a design technique, both provide mechanisms (such as C++ exception handling), which can enhance the possibility of implementing a safe operating environment System protection could be introduced as an integral feature of a future implementation of OO-MMURTL without having to redesign or reengineer the new system.

In summary, almost all of the initial operating system specifications required for this thesis were met with by MMURTL, and in some cases they were surpassed

2.4 Design goals of the MMURTL project

Since its original inception, MMURTL has extended from a simple operating environment undertaken as an exercise in system programming and design, to a powerful though raw operating system. Burgess defined the following as the original design goals of the MMURTL project

- *True Multitasking* - Given the capabilities of systems such as UNIX and Mach to provide true multitasking, implementing a non-multitasking environment would have served merely to mimic DOS' failings on the same platform. For this reason, the ability of a single program to create several threads of execution that could communicate and synchronise with each other while carrying out individual tasks was seen as an essential feature

- *Real-Time operation* - The ability to react to outside events in real-time would be an additional feature which would extend the possible uses of the MMURTL operating environment This goal provided the impetus that MMURTL be a message-based system.

- *Client/Server design* - The ability to share services with multiple client applications on the same machine or across the network provides MMURTL with a valuable model which could lend itself easily to future expansion, in particular as a distributed operating system. MMURTL's message-based design provides the mechanisms to accommodate this

- *Flat 32-bit Virtual Memory Model* - MMURTL uses the memory paging capabilities of 386/486 processors to provide an easy 32 bit flat address space for all applications running on the system.

2.5 The Task Model

MMURTL has a real-time prioritised tasking model The currently executing task in MMURTL is always the one with the highest priority which is not in a wait state

In order to conduct the prioritising of ready tasks, each task is assigned a priority value when it is created MMURTL has 32 priority levels with 0 being the highest and 31 the lowest General purpose applications (editors, compilers, word processors etc) should all run at a priority of 25 This leaves 0 through 24 for more important things, and only 26 through 31 for less important things (spoolers etc)

In MMURTL, a task may be in one of three states Firstly, when a task is initially created, it is presumably ready to run Secondly the task with the highest priority in the system is executing If there is more than one task with the highest priority, then execution is shared between them in a time-sliced manner Finally, a task can be in a wait state, waiting for a communication from another task, from a hardware notification, or from an external source It will not be able to continue executing until the event they are waiting upon occurs

Tasks that are ready to run are placed in the system prioritised ready queue This is actually implemented as an array of thirty-two queues, each holding a set of tasks which are ready to run, of the same priority Thus the zeroth queue in the array holds the ready tasks with a priority of zero, the first queue in the array holds the ready tasks with a priority of one, and so on The task which is at the top of the ready queue, is the one at the top of the first non-empty queue in the array of thirty-two queues

If more than one task is on the highest priority non-empty queue, then time-slicing occurs, otherwise the task with the highest priority executes until it relinquishes the processor because it has entered a wait state or has run to completion This is described in detail below

Tasks that are waiting upon a message or another task, remain at an exchange (see *Section 2.6 Messaging subsystem*). When they receive a notification at the exchange they will move to the ready queue, or if they have a priority higher than the currently running task they will become the running task and the previous running task will be returned to the ready queue.

MMURTL does perform some time-slicing but this is only between tasks with equal priorities. If the priority of the currently running task is 25, the Task Manager will slice time between that task and any others in the ready queue which have an equal priority. Note that tasks waiting at an exchange with the same priority will not be time sliced since upon becoming the running task they would merely be capable of performing a busy wait.

MMURTL switches between tasks in response to the following events:

1. An outside event caused a message to be sent to an awaiting task which has an equal or higher priority than the currently running task.

2. The currently running task can't continue because it needs more information from the outside world. In this case it sends a *request* and goes into a wait state and control is passed to the task with the next highest priority.

3. The operating system has detected that there is a process with a priority on the ready queue with the same or higher priority as the task that is currently running, and a predetermined amount of time has lapsed.

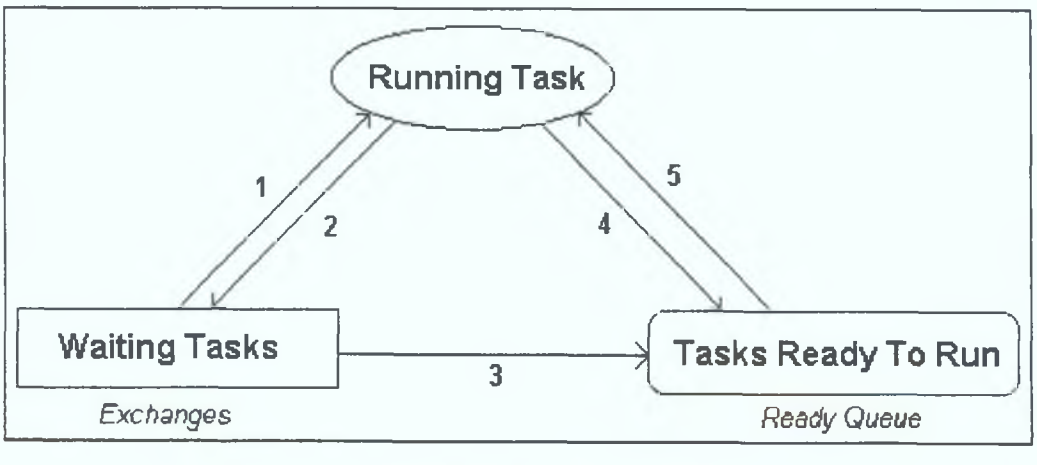


Figure 2.1. MMURTL's Task Switching Model

Figure 2 1 shows the movement of tasks as performed by the system Task Manager This shows that a Running Task which needs a communication from another task, hardware, or an external source, will move into a wait state at an Exchange (2) When the notification is received by the waiting task, it will be placed on the Ready Queue (3) unless it is has a higher priority than the currently executing task, in which case it becomes the Running Task (1) If the Running Task has completed, or if the Running Task's time slice has expired (4) and there are other tasks with an equal or higher priority, then the next highest priority task in the Ready Queue becomes the Running Task (5)

2.6 The Messaging Subsystem

MMURTL is capable of both synchronous and asynchronous messaging Tasks exchange information with each other by sending messages There are two basic forms of messaging in MMURTL The first type is a *Request* for services which should receive a *Respond* message in response The second type is a *non-specific message* which doesn't expect a response The Request/Respond mechanism is a key component of MMURTL's client/server model.

Central to the messaging subsystem is the concept of an *Exchange* An exchange is the place at which a message is left for a task, and so it is the place to which tasks look when seeking or awaiting messages In order to send a message in MMURTL, a task must first have an exchange to which it can address the message Exchanges are allocated by the *AllocExch* function

Sending a message in its simplest form requires a *SendMsg* call which sends a given message to a specified exchange At this point one of two things may happen

Firstly, a task may be expecting the message and therefore it would wait at the exchange for it to arrive This is done by calling the *WaitMsg* function Once a message arrives at the exchange at which a task is waiting, the task may continue running An exchange is one of only two places a task may wait The other is the ready queue

Secondly, the message will arrive at the exchange at which there is no waiting task In this case the message will wait at the exchange until a task looks there using the *CheckMsg* function to detect if a message is waiting

2.6.1 Request and Respond

The Request and Respond messaging primitives are designed so that a system service which provides shared processing for all applications on the system can be installed The processing is carried out by the system service in response to a Request The results and perhaps data are returned via the Respond primitive to the requester

Message based services provide shared processing functions that are not time critical. The service is provided to all applications Example uses of system services include file systems, keyboard input, printing services, and Email services

Each service is given a name upon installation The name must be unique on the machine This name is registered with the operating system Name Registry The registration takes place only the first time the service is installed At the same time the service name is assigned an exchange number In this way, the service can be referred to in future by its name, such as 'KEYBOARD', as opposed to its exchange number The exchange may change each time the machine is booted but the service name remains the same It is not unusual for up to thirty system services to be installed at a given time

A *Service Code* is associated with each service Each of the possible 65,533 service codes may represent a function provided by that system service The Service Code is transparent to the operating system as it is uniform across all system services

2.6.1.1 Request

A Request for a system service is more complex than a simple *SendMsg* and as such it requires more information Each request must include a Service Name which identifies the service being requested and a Service Code which specifies the exact service function being requested Two pointers to memory in the address space of the requester are sent which may contain data required by the service This data could point to a text string, a array or a structure of data, depending on the service

Each request must also be accompanied by the number of an exchange which has been previously allocated by the requesting task This exchange will be used by the service to respond to the request

2.6.1.2 Respond

The Respond primitive is less complicated than the Request It merely requires as parameters the handle to the request block which is being responded to and a status or error code being returned by it

2.6.2 Example of the Request-Respond Mechanism

The following steps illustrate a possible communication from an application to a service and the resulting response using the Request/Respond mechanism.

1 A task calls Request and asks a specific service to perform a certain function
2 The request arrives at the exchange in the form of an 8 byte message
3 The requesting task calls WaitMsg on the exchange specified in the Request call and sits at the exchange waiting for a reply
4 The request is noticed by the system service and serviced
5 If the service is performed successfully, the resulting data is placed into the memory address area of the requesting task The Respond primitive is called which notifies the task waiting at the exchange that the request has been carried out

2.7 Memory Management

MMURTL uses 386/486 hardware based paging for memory allocation and management As a result MMURTL dispenses with segmented programming (which is normally associated with DOS) In MMURTL there is only one memory model, which has two segments One is for code, the other for data and stack Each segment can be as large as physical memory

MMURTL doesn't provide memory management in the sense that compilers and language systems provide a Heap or an area that is managed and cleaned up for the caller Instead, MMURTL is a paged memory system. Each page is four Kilobytes of contiguous memory

Pages are allocated in response to a request by an application or system service and are returned to the pool of free memory pages when they are deallocated MMURTL manages all the memory in the processor's address space as pages

MMURTL uses almost no segmentation The operating system and every application share only four segments

- The OS Code Segment
- The OS Data Segment
- The User Code Segment
- The User Data Segment

MMURTL makes use of its own code segment to provide additional protection for system programming and for protection within the OS pages of memory

This memory management scheme allows MMURTL to use 32 bit data pointers exclusively This simplifies system and application programming and speeds up code execution.

2.7.1 Paging

MMURTL's use of paging simplifies its memory management mechanism by using tables to manage both physical and linear memory addresses These tables are used by hardware to translate physical memory addresses to linear memory addresses

The tables that hold these translations are called Page Tables (PTs) Each entry in a Page Table is referred to as a Page Table Entry (PTE) Each Page Table Entry represents one four kilobyte page There are 1,024 Page Table Entries, each using four bytes, in each Page Table Therefore one four kilobyte Page Table can represent four megabytes of memory.

In order to locate the appropriate Page Table, the memory management system makes use of Page Directories (PDs) Each task is assigned a Page Directory In turn each entry in a Page Directory Entry (PDE) Each Page Directory Entry holds the physical address of a Page Table Once again each entry requires four bytes and represents a single Page Table which represents four megabytes of memory Since each Page Directory has 1,024 entries, each task can access 4 gigabytes of linear address space

2.8 Summary

This chapter introduced the MMURTL operating system. The criteria for finding an appropriate system on which to base this thesis were laid out initially, followed by a brief summary of the aims of the MMURTL operating system. Next, each of the three primary components of MMURTL were described - the task model, the messaging subsystem, and the memory model. Each of these systems will be described in further details in later chapters which concentrate on the redesign of each of these components in OO-MMURTL

Chapter 3

Object-Oriented Operating Systems

3.1 Overview

This chapter provides a brief examination of several object-oriented and object-based operating systems This serves to demonstrate the wide variety of possible implementations when objects are introduced in the design of an operating system. The operating systems presented in this chapter range from those developed by professional developers, such as Sun Microsystems' *Spring* [Mitchell 93] system, to those developed by academic institutions such as the University of Grenoble's *Guide* [Balter 91] system. They also range from object-based operating systems such as Carnegie Mellon University's *Mach* [Julin 89] to completely object-oriented systems such as *Choices* [Campbell 93b], which was developed by the University of Illinois

3.2 Spring

Spring is a highly modular, distributed, object-oriented operating system, which is currently under development by Sun Microsystems Inc A commercial release has yet to appear

The design of Spring concentrates on the notion of an object-oriented operating system with a strong and explicit architecture, particularly with reference to the interfaces between its software components By stating strength of interface as an important design factor, Spring's system designers imply that a well defined definition of *what* each software component does is given, while at the same time very little about *how* the component is implemented is given

This summary of the Spring system presents its use of an Interface Definition Language, a discussion of Objects in Spring, a brief examination of Spring's structure, and finally a look at the Spring Nucleus

3.2.1 Interface Definition Language

In order to achieve this strength of interface, the Spring designers developed an Interface Definition Language (IDL) which is based upon the IDL adopted by the Object Management Group as a standard for defining distributed, object-oriented software components

Typically, an IDL compiler is used to produce three pieces of source code in a chosen target language, for example C++ or Smalltalk The first piece of source code which is produced is, in the case of C++ for example, a header file with the definitions of the methods, constants, types and classes which were defined in the IDL Secondly, client side stub code is produced This stub code is dynamically linked into a client's program to access an object which is implemented in a different address space Finally, the server side stub code is dynamically linked into an object manager in order to translate incoming requests for object invocation into the run-time environment of the object's implementation

Several compilers are provided to support the IDL, for example, Spring could provide an IDL-to-C and an IDL-to-C++ compiler allowing a client written in C and a server written in C++ to use the same IDL The use of this IDL helps Spring to achieve strong interfaces between software components while remaining a flexible, extensible and open design, thanks to its language independence and capacity for dynamic linking

3.2.2 Spring Objects

Almost all of Spring is implemented as a suite of object managers, for example the file system which provides file objects etc In addition, object managers are themselves objects As a consequence it is as easy to add new system functionality as it is to write a new application in Spring, and all such functionality is inherently part of a distributed system.

The combination of object-orientation and strong interfaces endows Spring with an open, distributed, extensible and secure computing environment, in addition to uniformity of access to objects and location transparency

The users of a Spring object invoke operations upon it based upon the operations defined in its interface As a result of its distributed nature, how and where the operation is actually performed is transparent to the application performing the object invocation

There are two forms of objects in Spring, server-based objects and serverless objects Server-based objects are implemented in servers that reside in different address spaces from their clients Support is provided for these objects is given through the generation of stubs by the IDL These stubs take the arguments for the invocations, dispenses them for transmission to the server, and retrieves any results before returning them to the client application This form of transmission makes use of the Spring *subcontract* mechanism which allows control over object runtime operations such as how object invocation is implemented and how object references are transmitted between address spaces

Serverless objects always exist in the address space of the client When a serverless object is passed between address spaces, its state is copied to the new address space Both server-based and serverless object invocation are shown in Figure 3 1

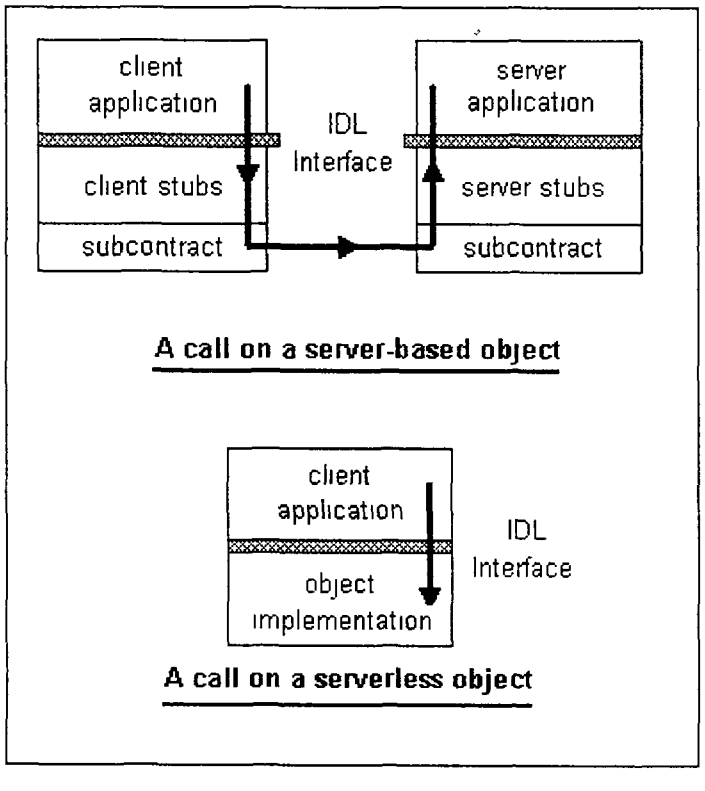


Figure 3 1 Object Invocation in Spring

3.2.3 The Spring Nucleus

The nucleus is Spring's microkernel It supports three basic abstractions *domains*, *threads*, and *doors* A domain is analogous to a UNIX process or a Mach task It provides an address space for an application to run in and acts as a container for various kinds of application resources such as threads and doors Threads execute in domains Each thread executes a single portion of a domain, usually while other threads are executing concurrently within the same application Doors support object-oriented calls between domains A door represents an entry point to a domain, represented by both a client computer and a unique value nominated by the domain

3.3 Chorus

Choices is a parallel object-oriented operating system designed as a collection of interconnected frameworks It was developed at the Department of Computer Science at the University of Illinois Chorus is an original operating system, designed from its conception as an object-oriented operating system implemented in C++ It supports distributed and shared memory multiprocessor applications and virtual memory

3.3.1 Frameworks

The design of Chorus is based upon the design of and interrelationships between frameworks and subframeworks A framework is an architectural design for object-oriented operating systems and are used to describe the components of such a system and the way they interact The interactions in Chorus are defined in terms of classes, instances, constraints, inheritance, polymorphism and rules of composition

The Chorus operating system is implemented as a framework, which guides the design of the subframeworks of its subsystems Each subframework refines the general operating system framework according to the requirements of the particular subsystem which it represents

The frameworks for the system provide generalised components and constraints to which further specialised subframeworks must conform. Each subframework introduces additional components and constraints to the components of its framework Subframeworks may be recursively refined in order to provide further specialisations

In this manner, frameworks assist the initial operating system and Chorus' extensibility by providing an architectural design that has common components and interactions throughout the entire network of subsystems A framework augments the traditional layered design of operating systems [Campbell 92] A layer represents an abstract machine that hides machine dependencies and provides new services, while a framework introduces classes of components that encapsulate machine dependencies and define new services

3.3.2 Classes

The Choices framework consists of three abstract classes A *MemoryObject* is used to store data A *Process* represents a thread of control which executes a sequential algorithm. A *Domain* is an environment that binds the names processed by the threads of control to storage locations The cardinality of their interrelationships are as follows

- Each process must have one and only one domain - Each thread may only operate in a single execution environment
- Several processes may share the same domain - Several threads may perform concurrently in the same execution environment
- Each domain may be have several memory objects - An execution environment may consist of several blocks of memory, in which code and data are stored
- Each memory object may be associated with several domains - Each block of memory may be shared between co-operating execution environments

Each of the three abstract classes described above must be specialised through subclasses before they can be invoked The constraints, such as the cardinality relationships, between classes remain true between subclasses They can, however, be further defined and specialised

3.3.3 Process Management

As an example of a Chorus framework in action, a brief introduction to the Chorus Process Management framework is given here Almost all of Chorus' frameworks consist of related or communicating components The process management framework has five such components, each of which is described below

- *Process* - This is a path of execution through a group of C++ objects There are three specialisations of a Process A *SystemProcess* executes in the kernel and is non-preemptable An *ApplicationProcess* runs in user and kernel space, while an *InterruptProcess* is used to handle the occurrence of an interrupt Note that context switching is light-weight between Processes in the same domain and is heavy-weight between Processes in different domains

- *ProcessorContext* - This is responsible for saving and restoring the machine dependent state of a Process Every Process has exactly one ProcessorContext and each ProcessorContext belongs to a single Processor

- *Processor* - This encapsulates the processor dependent details of the central processing unit This includes the hardware CPU identification numbers and the state of the hardware mechanism.

- *Gang* - This is a group of Processes that should be gang scheduled (run simultaneously) on the processors of a multiprocessor machine The Gang allows the collection of Processes to be manipulated as a single unit

- *ProcessContainer* - This component is responsible for implementing scheduling in Choices Subclasses of ProcessContainer inherit the scheduling interface and are responsible for implementing different scheduling policies as required by the operating system. For example, in order to implement a multilevel feedback queue scheduling policy, the ProcessContainer insertion and removal methods must be specialised in a subclass of ProcessContainer Processes are executed by inserting them into instances of ProcessContainer, the Processor then removes the Process from its ready queue before dispatching it An example class hierarchy of the ProcessContainer class is shown in Figure 3 2

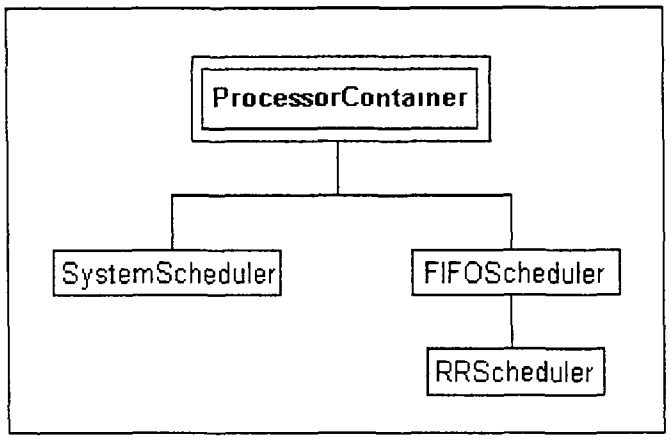


Figure 3 2 An Example Chorus Class Framework

3.3.4 MicroChoices

The most recent version of the Choices operating system is called MicroChoices The most important development in this version is the splitting of the kernel into two distinct portions, resulting in the removal of machine dependent code from the majority of the operating system.

The low-level portion of the newly designed kernel is the *nanokernel* This provides hardware dependent support for the remainder of the operating system. It is built as a framework of classes that captures the essential properties of the low-level hardware, presenting a useful interface to the higher levels of the kernel in a machine-independent manner The nano-kernel is a single modular subsystem that provides the mechanisms for implementing higher-level abstractions, such as processes, timers, and virtual memory The framework of abstract classes which the nano-kernel is composed of are specialised for particular hardware implementations through the creation of subframeworks The remainder of the kernel, the microkernel, and the operating system as a whole can now benefit from enhanced portability The nanokernel design has now reduced the task of porting MicroChoices to a matter of simply providing new machine specific subframework of the nano-kernel

The MicroChoices model describes an idealised machine architecture at the lowest level supporting a machine independent micro-kernel interface to the remainder of the operating system. Guidelines for the intermediate levels of the system are provided through the use of object-oriented frameworks

3.4 Guide

The Guide system was developed at the University of Grenoble, France It is a fully object-oriented distributed operating system for the development and operation of distributed applications Every resource or abstract entity in the system is an object, and communication between processes takes place through shared objects

The GUIDE [Krakowiak 93] model views the system as a distributed shared universe organised as a set of objects Guide objects are passive, that is to say that processes or threads are defined independently from the objects they operate on In addition, GUIDE objects are capable of persistence, their life can extend beyond that of the process or thread which created them.

The design of GUIDE is based on the tenet that an object support layer is provided by a lower level kernel, such as a microkernel, whose object support remains independent of the object model which it supports Thus a single object support microkernel is capable of supporting different object models

The development of GUIDE has occurred in two phases, each of which is described below

3.4.1 Guide-1

The first phase of Guide, Guide-1, was a single-language, UNIX-based system. Its primary aim was to investigate the use of objects as a unifying structuring mechanism in an operating system. A new programming language was composed in order to provide the required freedom of design required by the project The operating system was fine-tuned to the performance of this language, and vice versa

Guide-1's execution model was organised into *Tasks* A Guide Task is a virtual address space in which objects are mapped Concurrent activities run inside Tasks Both tasks and activities may be distributed

The object memory was internally organised as a two-level store, as shown in Figure 3 3, although this division was hidden from applications Both levels were transparently distributed

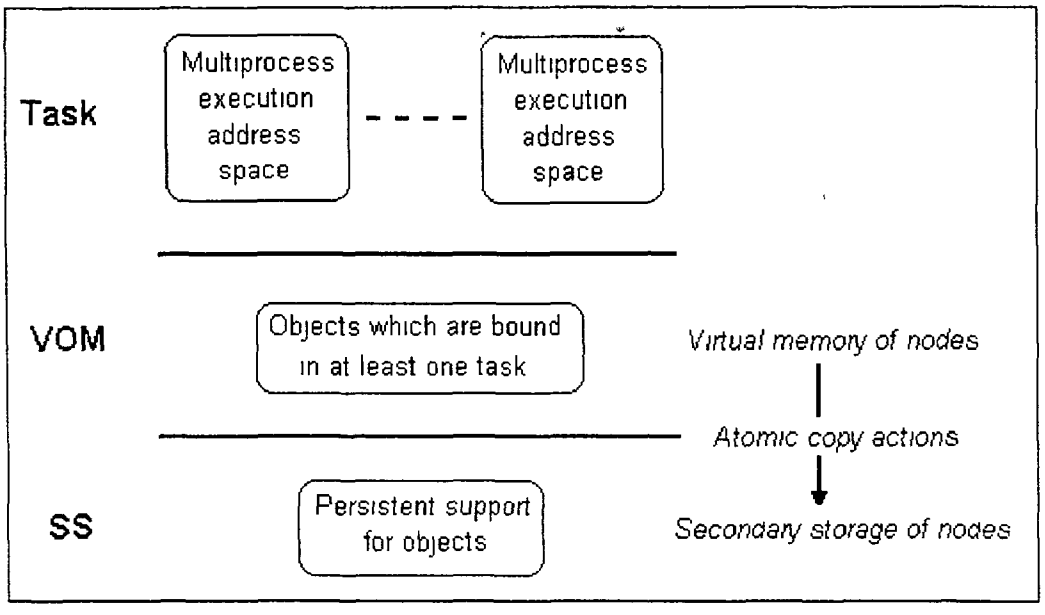


Figure 3 3 Guide's Internal Object Memory Organisation

The Virtual Object Memory provided support for executing methods on shared, synchronised objects, while the Secondary Storage provided permanent storage space for objects The Virtual Object Memory acted as a cache for the Secondary Storage Garbage collection was performed at the Secondary Storage level. Each Guide Task was associated with a distributed address space The Virtual Object Memory consisted of the address space of all active Tasks

In order to be used by the activities of a Task, an object must be present in the address space of that Task This was done by mapping the object into the Task's address space, and subsequently loading the object through the underlying paging mechanism.

3.4.2 Guide-2

Whereas the original Guide-1 system was a UNIX-based system, Guide-2 was designed based upon a microkernel-based architecture Similarly whereas Guide-1 was a single-language system, Guide-2 aimed to provide generic support for object-oriented languages which conformed to certain criteria, namely that the language is class based, where the classes are organised in a hierarchy by the *is-a-subclass-of* relationship, and that objects are named by universal references

The Guide virtual machine provides three basic abstractions in its object model for building complex structures *Instance-objects*, *class-objects*, and *code-libraries* The relationship of these objects is shown in Figure 3 4

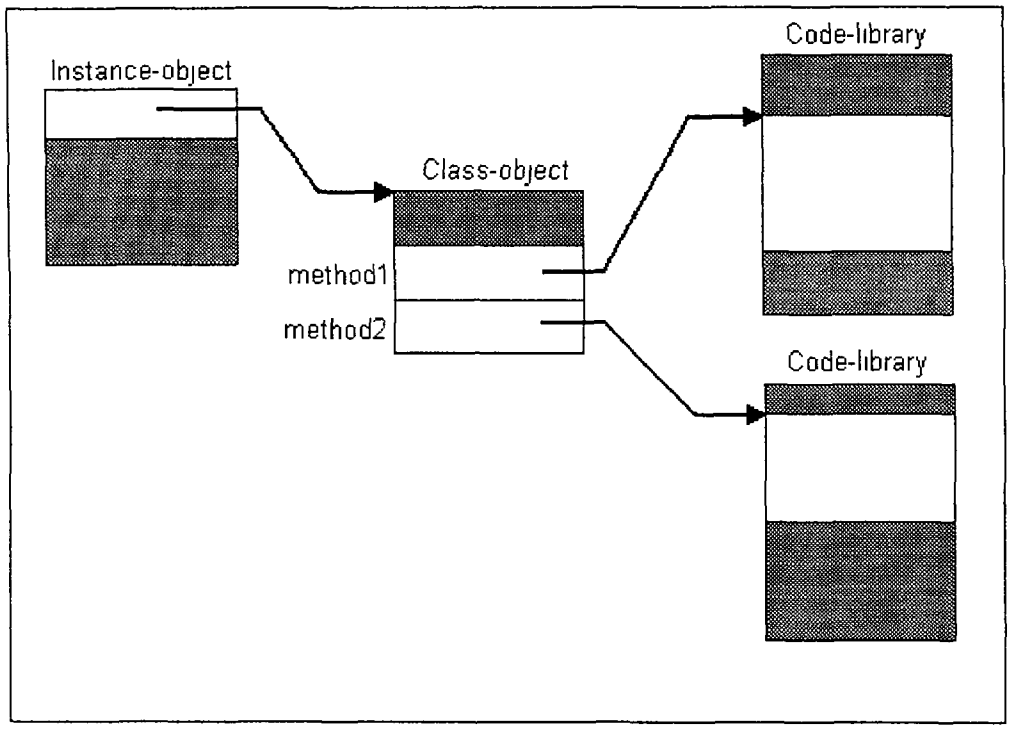


Figure 3 4 Object Relationships in Guide

3.5 Mach

Mach is one of the most popular microkernel systems today, being the basis for commercial operating systems such as NeXT, OS/2 and Windows NT Mach 3 0 was designed to provide core system functions required in order to support higher system levels It is intended to be a foundation on which operating systems can be built

The Mach microkernel performs a small set of operations in order to reduce the size of code running in both kernel space and user space The operations performed by the microkernel fall into five categories - Virtual Memory Management, tasks and threads management, interprocess communication, I/O support and hardware management, and both host and processor services These components define the abstract processing environments for application programs

The Mach 3 0 multi-server system makes use of two object-oriented techniques in its operating system construction Firstly, an object-oriented model is defined for all interactions between clients of the operating system and the system itself and secondly, object-oriented techniques are used to structure those interactions

The functionality of the operating system is provided by a three-layer architecture At its lowest level lies the Mach 3 0 microkernel which provides the basic Mach abstractions, such as virtual memory management, task management and device handling The next level is comprised of a set of mostly generic system servers These execute as tasks in user mode, implementing all of the high level functionality required by a complete operating system, such as file management and networking Finally, the third and highest level consists of a collection of emulation libraries executing in the address space of the user tasks, which provide access to the generic services supported by the services

The interactions between system components are defined using an object-oriented model, and the system itself is implemented using an object-oriented language All of the services provided by the various servers are defined in terms of operating system objects such as files, directories, devices, etc Although each server is an abstraction of an entire operating system component, as a result of its object-oriented nature only a subset of its operations may be invoked This gives the system service programmer more flexibility in the implementation of the server, which may be split, combined, or optimised, through the use of inheritance and polymorphism, without affecting the manner in which the clients interact with it

IBM is currently working with Taligent on a version of its Workplace OS, using Mach as its basis, which will provide object-oriented abstractions from the microkernel to the system service level, supporting object-oriented services such as IBM's System Object Model, Distributed System Object Model, and Taligent's frameworks

3.6 Summary

This chapter introduced four different implementations of object-oriented operating systems Spring, Choices, Guide and Mach The four were chosen based upon the diversity of their design and implementation, and the nature of the institutions which developed them.

Although there were several common conceptualisations between the four operating systems, the number and variety of differences between them, in terms of both their design and implementation was apparent This exemplifies the lack of consensus among designers and implementors of object-oriented operating systems In turn, perhaps this serves to demonstrate that research in the area of object-oriented operating systems has yet to approach maturity In a similar fashion, OO-MMURTL shares some commonalities with these operating systems, but also displays some diversities too

Chapter 4

Object-Oriented MMURTL

4.1 Overview

This chapter introduces Object-Oriented MMURTL (OO-MMURTL) In particular, new entities which have been added to the operating system are focused upon In subsequent chapters each component of the operating system is examined, such as the process management subsystem and the memory management subsystem, and their object-oriented designs will be described in detail The strength of each of their designs, however, is reliant upon the concepts introduced in this chapter

When object-orientation is used as the basis for the design and implementation of an operating system, it is inevitable that new entities will be introduced at various stages of its realisation Previous chapters have provided definitions of the concepts which underlie object-oriented technology However, the strength of an object-oriented operating system comes as a direct result of the strength of the basic components from which the remainder of the system is inherited If the initial building blocks lack a firm foundation, the underlying weaknesses will have a heavier effect on each subsequent sublevel of the system design The two basic entities which form the basis of OO-MMURTL are described below

4.2 Object-Oriented Entities in OO-MMURTL

The level of activity in the area of object-oriented operating systems, particularly in the distributed systems area, suggests that object-oriented technologies can provide powerful new tools at the hands of the system and application developer As with any other technology however, the importance and value of object-orientation lies not in its suggestion, but in maximising its effect on the performance and capabilities of the operating system itself Object-orientation is realised in OO-MMURTL by identifying and introducing two primary concepts - objects and object stores

4.2.1 Objects

Objects are the most basic entity in the OO-MMURTL operating system. The first step undertaken in the migration of MMURTL to an object-oriented design was the identification of the core components of the original system. Within these components, entities which would be better implemented as objects in the new operating system design are identified. Subsequently, a class hierarchy is designed in order to encapsulate the behaviour of each set of related objects which belong to a specific component of the operating system.

4.2.2 Object Stores

The second way in which object-orientation is realised in OO-MMURTL is through the introduction and use of object stores. As was previously mentioned, the introduction of objects is not sufficient in itself to reap the rewards of object-orientation. The behaviour of the objects has to be supported by a suitable infrastructure within the operating system. In OO-MMURTL this role is performed by the Object Managers, which are described in detail later in this chapter.

4.2.3 Documentation of Objects and Object Stores

Subsequent chapters discuss each of the main components of the OO-MMURTL operating system which have been redesigned using the object-oriented paradigm, namely:

- Process management
- The messaging subsystem
- Memory management
- The ready queue
- The interrupt mechanism
- The timer

Each of these chapters follows a set pattern

1 The behaviour of the relevant Object Manager is described
2 The class hierarchy which encapsulates the components behaviour is introduced
3 The individual classes are described
4 The implications of object-orientation on the component in question are discussed

Additional concepts and functionality which have been introduced to support a given component are introduced during the course of the relevant chapter

4.3 Storage Containers

Every operating system must store a wide range of information concerning the processes and resources which it supports This information ranges from the address and size of a data segment belonging to a specific task, to the contents of a message which is being transported between two tasks The individual system developer must decide how best to organise this data while remaining faithful to the goals and constraints of the system design

MMURTL makes use of tables to store its system information Fixed-sized structures are defined for each resource Each of these structures is padded so that the default MMURTL page size is a multiple of the structure size Figure 4 1 shows a table which represents examples of these structures, along with their size, and the amount of structures which could be allocated to an operating system memory page (1024 bytes)

Structure	*Size (bytes)*	*Per Page*
Job Control Block (JCB)	512	2
Task State Segment (TSS)	512	2
Exchange (EXCH)	16	32
Request Block (RQB)	64	8

Figure 4 1 MMURTL Table Structures

4.3.1 Advantages of Tables

The decision to use tables in the implementation of MMURTL was due to the following factors

- *Fast access* - This is perhaps the single most important benefit of tables Speed of access is paramount in the mind of a system designer at every stage of development Because the system data stores are used so frequently by most system operations, fractions of a second delay in retrieving or modifying the information could result in a needless reduction in productivity This is compounded in the case of tables that must be accessed in a mutually exclusive manner In these cases, other tasks could be blocked while the running task accesses the data, causing a further waste of CPU cycles

- *Simplicity of their implementation* - Tables are one of the simplest structures in which to store and maintain data They are easily accessed and maintained, substantially simpler than a structure such as a linked list, whose pointers must be fastidiously guarded at all times

- *Easily Managed* - MMURTL makes use of static tables That is, each table is allocated a predetermined amount of operating system memory pages at boot time From this point on, a simple list or bitmap is used to track free table entries from entries which are in use When a task is allocated a table-based resource, it is given a unique number which represents the resource in question For example, each JCB has a unique Job Number This number will serve not only to identify the relevant job but can also be used to index the table directly, since each identifier also doubles as a table entry number

4.3.2 Disadvantages of Tables

Despite the advantages mentioned above, in my opinion tables are an inherently limited form of information storage for certain entities in an operating system, in particular an object-oriented operating system, for the following reasons

- *Distributed management* - In order to facilitate speed of access, there are no specific functions whose responsibility it is to maintain the system tables. Instead, each function which accesses, and in particular modifies, a system table must repeat the same actions as every other task which performs the same function. This lack of accountability towards system tables is directly related to the next problem.

- *Inadequate protection* - MMURTL uses two of Intel's four possible protection levels - user and system. As a result, any system task can access any other part of the system without restriction. A task may easily, yet inadvertently, corrupt a system table therefore compromising system integrity with possibly disastrous effects on other tasks.

- *Static allocation* - Although this problem does not apply to all operating system implementations of tables, it does apply to them as they exist in MMURTL. The static allocation undoubtedly facilitates easier management of tables, however, it also leads to two potential problems :

    1. At boot-time, memory pages are allocated to each table, for example sixty-four TSSs are immediately allocated, using thirty-two kilobytes. As the memory available to operating systems increase, this may be regarded as a minor problem. At the same time any waste of system resources must be carefully monitored, and so this possible waste cannot be disregarded either.

    2. Of more concern is the inability for additional table entries to be allocated following boot-up. This could lead to the anomaly whereby a task requests that a new resource be allocated to it, there are enough resources available to satisfy the request, but there are no free structures left to represent the resource, hence the request fails.

4.4 New Container Criteria

Taking the deficiencies of the current storage system under consideration, the following criteria were set in order to determine a design for an adequate storage medium for OO-MMURTL:

- *Centralised processing* - In keeping with the object-oriented paradigm, the new storage system had to have centralised processing. In other words, the functions which could access and modify the tables ought to be grouped together. Updates to tables could no longer be performed anywhere in the operating system simply because it was possible. Each storage container must be responsible for all accesses to the information it contained. To do this, a set of clearly defined access methods must be made available by each container in order to facilitate every type of access and update that may be required, where permissible by user or system tasks.

- *Protection* - Since access to the system data stores will be through a centralised processing area, an integral component of the new container should be that it offers protection to the information it contains. No task should be allowed to gain access to a data object, or perform an action on one, without full error checking having been performed first. Examples of the checks that should be asked of a request include:

  - Is the object identifier valid?
  - If it is valid, does the object it refers to exist?
  - If it exists, have all of the resources required by the operation been allocated?

  Only when the container is satisfied that the task requesting access to the data can perform the desired action without causing a system error, will the container allow the task to proceed.

- *Intelligence* - This means that each container has some specific knowledge of the nature of the data it holds. Tables are generic by nature. They provide an area of data to store multiple entries of a given size. The only information that a function requires to access the relevant record is the offset of the table, the index of the entry, and each entry size. Although this calculation is both simple and rapid, it does not provide optimum support for either the data or for the tasks which are attempting to access it. Take a container which stores exchanges as an example of an intelligent container. Such a store should be able to receive a communication from a task and detect whether it is a message or a request. Having verified the correctness of the communication, the container will then forward it to the exchange belonging to the intended destination's task if it's a message, or in the case of a request then the intended system service will be looked up, before the communication is forwarded to it.

- *Future expansion* - The new container should also be capable of growth and expansion Tables cannot mature in this fashion The container must be as flexible as the data it holds If at some stage in the future OO-MMURTL was to become a distributed operating system, it would be necessary for each object container to provide additional fundamental behaviour by each of the objects it represents This could include the need for object persistence and location transparency Although this functionality would be provided by new aspects of the operating system, the existing constructs, in particular the object containers, would have to be capable of supporting the new behaviour Given that the new container was obviously going to be an object-oriented construct, the criterion of expansion becomes the easiest to satisfy as a result of object-oriented features, in particular that of inheritance

Taking the sum of these needs as a specification, a new construct has been introduced to the OO-MMURTL operating system to fill the gap in its facilities This construct is described below

4.4.1 Object Managers

Every object-oriented operating system has some form of repository for objects Some choose to group objects in related categories, others store objects in a global (within a particular system node) store There are as many names for such stores as there are differing implementations - containers, collections, bags etc

The term *Object Manager* was chosen because it best describes the function performed by the new construct It does more than contain or collect objects, equally it doesn't just store its objects in bags Each of OO-MMURTL's Object Manager's performs the following tasks

- Stores objects
- Verifies the validity of all accesses to objects
- Maintains responsibility for allocating and freeing memory used by objects and the resources allocated to them
- Provides a complete set of public access methods to the objects

4.5 Object Manager Hierarchy

Having defined the facilities which had to be provided by an object manager, the next step was to consider possible implementation strategies As was previously stated, the approach taken was to provide a specialised object manager for each major component of the operating system which maintained a number of objects under its supervision There would be a Process Manager, an Exchange Manager, a Memory Manager and so on The reasoning behind this was to ensure each manager had a strict, well-defined realm of control

Each of these managers would be responsible for the key area it represents, and this varied widely between one manager and the next, for example process objects behave differently and have a different life cycle than exchange objects, and so on Having said that, the basic functionality to store and retrieve a memory or an exchange object remain closely related

As a result, it became apparent that each Object Manager would have two sides to its behaviour Firstly, all of the managers shared the need to store their objects in a similar fashion On the other hand, the support provided by each manager for access to the objects they stored varied

The solution to satisfying both aspects of the behaviour of each Object Manager is provided by the following two-step implementation strategy

Step One A root CManager class is defined whose purpose it is to provide storage and retrieval functionality This class would manage the pointers to the objects it stored, it would allocate and deallocate memory appropriately in order to facilitate the storage of the objects. and it would provide methods which could be used by inherited classes to manipulate the data it stored

Step Two Subclasses of CManager are created for each component of the operating system. These subclasses would inherit all of CManager's functionality in order to manage the objects they contained In addition each implementation of a CManager subclass must provide methods which deal explicitly with the types of object they store These methods would define the public interface of each particular implementation of an Object Manager

These two steps combine to provide a set of private methods hidden from the user which store and maintain the objects, and a set of public methods which provide the functionality to perform tasks specific to each component of the system.

The CManager class hierarchy in OO-MMURTL is flat in nature, as shown in Figure 4 2 It is possible, however, to implement further subclasses of a CManager subclass if necessary, although such a need does not currently arise in OO-MMURTL

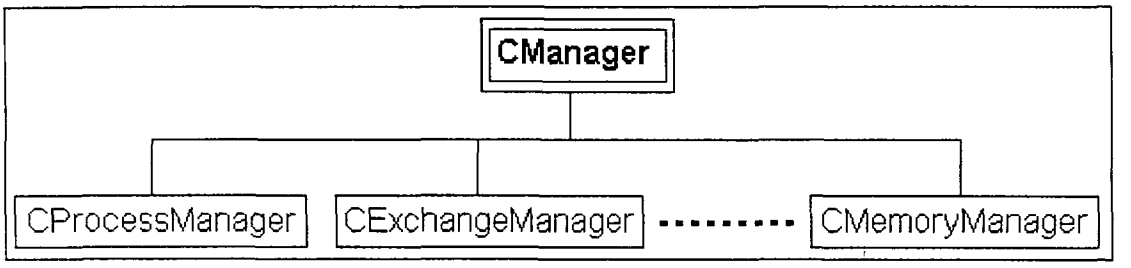


Figure 4 2 The CManager Framework

In order to exemplify their relationships, the following section describes an example transaction between a task, an Object Manager and an object represented by the manager

4.6 An Example Object Manager Transaction

The scenario which forms the basis for this example is that of a task sending a request to an exchange The steps which comprise this action in OO-MMURTL are shown in Figure 4 3 The bold bracketed numbers in the right column refer to the control flow diagram which is shown in Figure 4 4

Note that the Object Manager in question - the CExchangeManager - is described in detail in Chapter 6 along with the class hierarchy which represents the entire messaging system (including the CExchange class which also features in the example)

Step	*Description*
1	Task sends request message to CExchangeManager
2	CExchangeManager attempts to retrieve pointer to requested CExchange
3	CExchangeManager verifies existence of CExchange
4	CExchangeManager verifies validity of request
5	CExchangeManager creates a TRequest record
6	CExchangeManager sends the TRequest record to the relevant CExchange
7	CExchange attempts to remove a waiting task from its own task queue
8a	If no task was waiting, the request is queued (goto step 11)
8b	If a task was waiting, the request is given to it and the task is placed on the ready queue
9	The ready queue is re-evaluated
10	A switch is made to the highest priority task
11	Control returns to the CExchangeManager

Figure 4 3 Message from a task to an exchange (Steps)

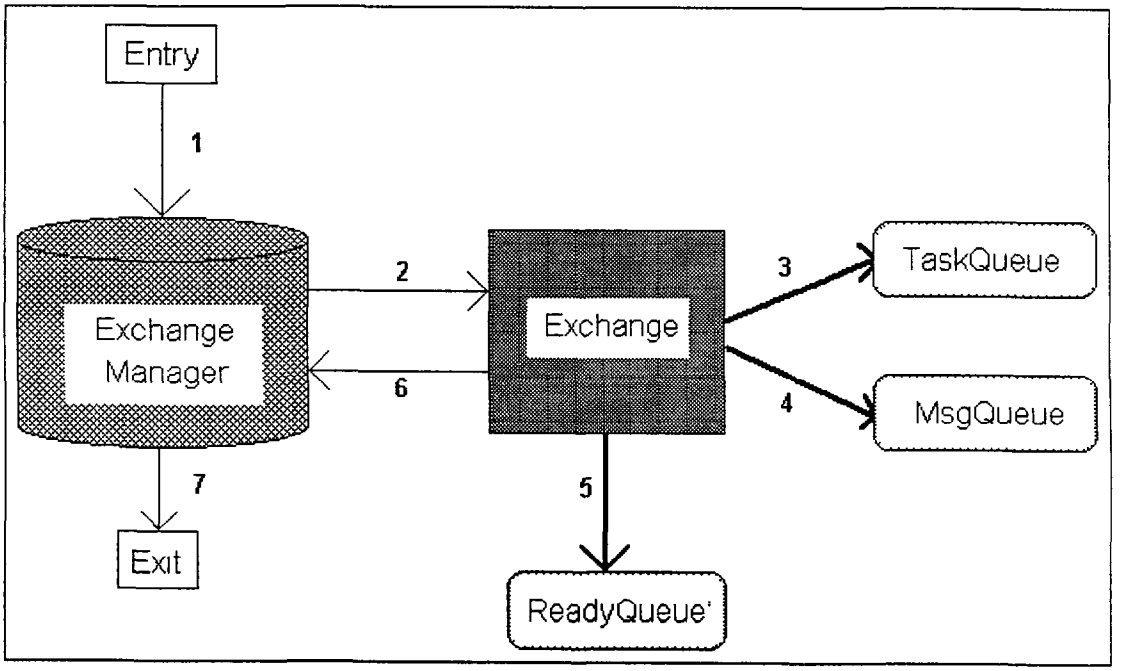


Figure 4 4 Message from a task to an exchange (Control Flow Diagram)

The control flow diagram depicted above shows the flow of control between the various entities in the messaging subsystem and the ready queue in response to a request for an exchange Control enters the system at the *Entry* node and leaves at the *Exit* node A thin arrowed line represents a transfer of control which does not immediately return, while a thick arrowed line shows a flow of control which does return

4.6.1 Advantages Demonstrated by the Example Object Manager Transaction

The following features of the previous example demonstrate the improvements which have been provided by the new Object Manager entity in object access transactions

- The message which is intended for the CExchange object, is initially sent to the CExchangeManager This allows the CExchangeManager to vet and possibly reject the request
- Further resources are only committed once the request has been validated
- The CExchange object is only made aware of valid, correctly formatted requests
- CExchangeManager retains control of the CExchange object at all times

4.7 Summary

This chapter introduced Object-Oriented MMURTL In particular, the use of two primary object-oriented entities which form the foundation of OO-MMURTL were described, namely objects and object stores The responsibilities of the new system's object stores were identified, subsequently a new operating system component called an Object Manager was described which would handle these responsibilities Finally, an example transaction was given which made use of an Object Manager, highlighting the advantages of the new mechanism.

Chapter 5

The Process Management Model

5.1 Overview

This chapter describes OO-MMURTL's process management model A brief description of the improvements which result from the use of object-orientation in the design of the model is given Following this, definitions are provided for the new concepts which are introduced in the design of the new model. Next, the object manager which is responsible for this subsystem, the Process Manager is described At this point the hierarchy of component classes of the process management model are presented, before each of the classes is described in detail Finally, a summary is made of the advantages which have arisen as a result of the model's implementation

5.2 MMURTL's Tasking Model

MMURTL's tasking model is based upon the notion of jobs Each piece of executing code is treated as a job, regardless of whether it is a user program, a system program, or a system service Each job is represented by a Job Control Block (JCB) which contains information pertaining to it, such as its name, the address of its Page Directory, and the address and size of its code, data and stack segments

OO-MMURTL improves upon this in two ways

- A class model is provided which differentiates the various types of executing entities within the operating system. A basic pattern of behaviour is defined for each one By doing this, the new design enables the operating system to distinguish the intrinsic behavioural differences between each type of process
- Secondly, a specialised component has been introduced whose sole aim is to manage the behaviour of processes of all types, and to ensure that process requests are correctly stated and suitably behaved This component, which is called the *Process Manager* replaces the static table which previously held job information

5.3 Process Management Model Definitions

In light of the changes to the Process Model, it is necessary to define and clarify the concepts involved, before proceeding with a discussion of OO-MMURTL's new model

A *Process* is a control path through a group of C++ objects Each process is composed of one or more threads of execution Each thread is referred to as a *Task* Information concerning tasks is maintained in a Task State Segment (TSS) structure A TSS is a construct defined by the Intel processor The processor uses the TSS to manage its tasks

One of the advantages of using the TSS is its capacity for expansion The processor allows the operating system to append additional fields to the TSS One of the fields which is added to the TSS by OO-MMURTL is the address of the task's corresponding Process object In this way the operating system can use the TSS to directly access the Process for a given task, thus enabling actions (such as task switching, spawning new tasks etc) to be performed on it

Three basic entities can be identified in the OO-MMURTL Process Model

- A *Job* is an executing program whose life cycle consists of being loaded into the operating system, striving for system resources in order that it can complete its purpose, before exiting It aim is to run to completion with minimum delay A Job can be performed in privileged (OS) or non-privileged (User) mode

- A *System Service* is a Process which is usually loaded at boot time, but may also be dynamically loaded later, and which remains dormant for much of its existence Its purpose is to await requests from a Job or another System Service, requiring a specialised action to be performed When the action is performed, the service returns to a dormant state until another request is received Usually a System Service will be removed from memory when the operating system itself terminates The Keyboard module and File System are examples of modules which provide System Services

- The final operating system entity which can gain control of an execution thread is a *Device Driver* These are specialised pieces of code whose purpose is to control or emulate hardware on the system. They are low level entities which abstract the inner workings of hardware devices to simplify access to them by Jobs and System Services

5.4 The Process Manager

In MMURTL, Task Management is performed by two components

- The processor manages TSS structures
- The operating system manages Job Control Blocks

Each TSS points to a corresponding JCB which contains data required by the operating system about each job Job Control Blocks are structures of static length (512 bytes) which are stored in a table and referenced by an index into that table The operating system provided no single component to manage the JCB table Instead any job with system privileges would access the JCBs directly

The central component of the OO-MMURTL Process Management Model is the Process Manager, which replaces the JCB table The processor still manages TSS structures as before, with the exception that each TSS now points to a process object as opposed to a JCB structure

The Process Manager was introduced to overcome the following problems which were inherent in MMURTL's task model

- Because no single component was responsible for the management of the JCB table, any system task could access and amend, or even destroy it For this reason, the integrity of the table could be easily compromised

- Jobs were responsible for performing privileged operations upon themselves This led the system developer to perform tricks through Assembly language to overcome the paradoxes which resulted One example of when this occurred was when a program terminated itself and specified another program which was to run next What happens in this case is that the program to be terminated calls the *ExitJob()* function where its resources are freed Now, however there is no program to return to, it has been terminated Instead the *ExitJob()* function manipulates the calling stack so that the operating system returns control to the program specified to run next This is an unorthodox operation which can cause system crashes with ease if the slightest mistake is made

5.5 The Process Management Class Hierarchy

The purpose of the OO-MMURTL Process Management hierarchy, as with any other object-oriented framework, is to provide well-defined mechanisms that allow developers to extend and leverage the functionality provided, in order to increase system productivity and integration [Taligent '93]

The process management hierarchy has two separate components, which represent the different types of executing entities possible in OO-MMURTL The first and most common of these is the CProcess hierarchy

Figure 5 1 depicts the base class CProcess and its subclasses, CSystemService and CJob CJob's subclasses CUserJob and CSystemJob are also represented

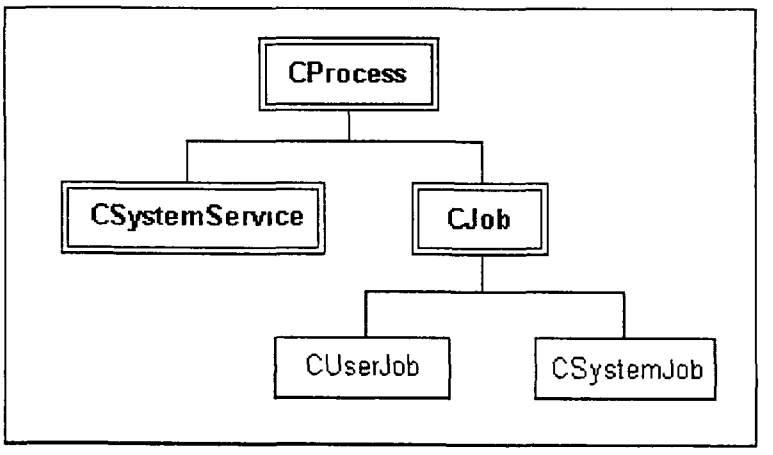


Figure 5 1 CProcess class hierarchy

CProcess, CSystemService and CJob are abstract classes They are insufficient in themselves to be invoked Their purpose it to define a template for the entities which they represent A programmer must derive a new class and write the abstract methods which are required in order to create a concrete class

CUserJob and CSystemJob are both concrete classes Classes of both of these types can be instantiated without need for a new inherited class This is because they have been sufficiently refined through the base class CProcess, its subclass CJob, and the relevant subclass CUserJob or CSystemJob

The second component of the process management hierarchy relates to device drivers They are not processes in their own right As such, their behaviour required them to be distinguished by a separate hierarchy This is shown in Figure 5 2

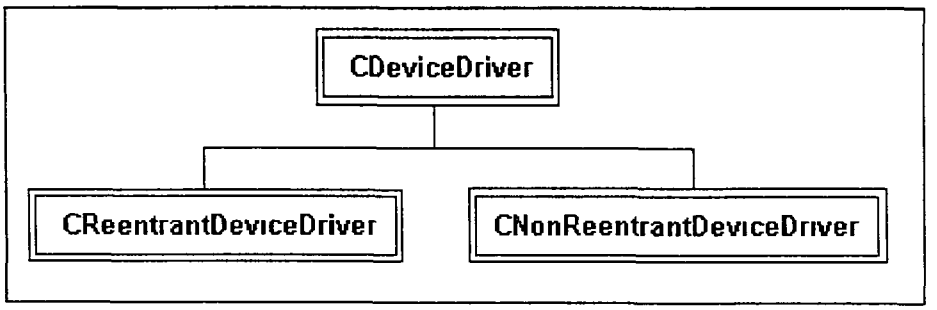


Figure 5 2 CDeviceDriver class framework

The reason the CDeviceDriver hierarchy is distinct from the remainder of the process management framework is because device drivers are not regarded by the operating system as independent executions CDeviceDrivers do not retain their own threads of control, instead control is passed from tasks belonging to CProcess objects A CDeviceDriver object is invoked and upon completion, control is returned to the calling CProcess

As depicted in Figure 5 2, CNonReentrantDeviceDriver, CReentrantDeviceDriver and CDeviceDriver are abstract classes The actual code which interacts with the hardware that each device driver controls must be inserted by the device driver programmer for each particular device

5.6 The CProcessManager class

In order to solve the inherent problems of the old JCB, the new Process Manager performs two basic functions

- It acts as a container to store and manage the process objects
- It acts as a gateway through which calls to processes are vetted before being allowed to be performed

In order to fully satisfy these requirements, the Process Manager is responsible for the following activities

- Creation of new Jobs, System Services and Device Drivers in response to requests from the operating system or existing Processes / Device Drivers
- Directing messages and method calls from the operating system or user programs to the appropriate Process / Device Driver
- Responding to messages and methods which have been dealt with by the appropriate Process / Device Driver
- Ensuring the safe closure of a Process / Device Driver, verifying that its resources have been freed and that any subsequent jobs which must be created are started

5.6.1 Public Methods of the CProcessManager class

Certain functionality is allowed to be performed directly on a CProcess object An example of this would be to set the name of the process which is to be executed upon completion of the current process This would be performed by using the *SetExitProc()* method. However, in order for the operating system or a program to have access to a CProcess it must first request the pointer from the Process Manager Only when the request has been validated, will the program be able to perform actions on the CProcess This is demonstrated in the example portion of code in Figure 5 3

```
void ReturnToCLI()
{
// Select an arbitrary process number

unsigned long ProcNo = 3,

// Declare a pointer to a CProcess

CProcess *pAProcess,

// Retrieve a pointer to the process

pAProcess = ProcessManager->GetProcess(ProcNo),

// Check success of operation

if (pAProcess==NULL)
  xprintf("Process Manager reports no such process"),
else
  pAProcess->SetExitProc("c \\mmurtl\\cli run"),
}
```

Figure 5 3

Other functions, however, may only be performed by the Process Manager An example of this is when a process terminates This cannot be done manually - it may only be performed through the Process Manager A request is submitted to the Process Manager which, having been validated, deals with the request and the CProcess object directly Control only reverts to the program which invoked the recently completed process once the object representing that process has been deleted

Other public methods belonging to the Process Manager deal with setting and retrieving its attributes In summary, there are three categories of methods invocable on the Process Manager, they are

1 Process retrieval methods
2 Privileged process management methods
3 Attribute setting and retrieval methods

The methods belonging to each of these categories are described below

5.6.1.1 Process retrieval methods

*CProcess *GetProcess(unsigned int ProcNo)*

This method, having verified the validity of the ProcNo parameter, will return a pointer to the corresponding process object to the calling program.

*CProcess *GetCurrentProcess()*

This method will return a pointer to the currently executing process to the calling program. The index of the currently executing process is a kernel variable which is retrieved by the call *GetCurrProc()* There will always be at least one executing process (the OS Monitor)

5.6.1.2 Privileged Process Management Methods

long LoadProc(unsigned int ProcNo, long fhRunFile)

This method is responsible for loading a program into an existing Process object This operation is requested primarily by the *Chain()* method, however it may also be called by the *ExitProc()* method if a process to run following the completion of the current process is specified *ExitProc()* and *Chain()* are each responsible for opening and validating the run file[1] and setting up the run file variables The first step of the *LoadProc()* method is to allocate user memory for the new process' stack, code and data If this is completed successfully, the code and data are loaded from the run file

*long LoadNewProc(char *pFileName, long *pProcNumRet)*

This creates and loads a new process *LoadNewProc()* is responsible for reading and validating the run file It must identify the size of the code, data and stack segments required by the program and allocate these segments *LoadNewProc()* then loads the new process into the newly allocated segments It must also create a new CProcess object and add it to the existing collection of CProcess objects If the new process is successfully loaded and created, *LoadNewProc()* returns ErcOK and the new process number is stored in *pProcNumRet*, otherwise an error is returned

*long Chain(unsigned int ProcNo, char *pFileName)*

The purpose of this method is to allow a process or a service to terminate itself and replace itself by another process or service *Chain()* terminates all of the tasks belonging to the process which calls it and frees system resources which will not be used in the new process

[1] A run file is a file with the extension RUN - This represents an executable file analagous to a DOS EXE file

The location of the process which is to replace the current one is stored in the *ExitRunFile* attribute This field is set by the *SetExitProc()* method and retrieved by the *GetExitProc()* method The *ExitRunFile* field of every new process is initially empty One important aspect of the *Chain()* function is that the second process runs in the context of the first In other words, the second job uses the CProcess object which was previously home to the first process In this way, the Page Directory remains the same

void ExitProcess()
This is called from a user process or system service This cleans up all of the resources that the terminating process was allocated *ExitProcess()* also checks for an exit run file to load if one is specified If no exit run file is specified the process is completely terminated, and, if the video and keyboard were assigned to it, they are reassigned to the Monitor

5.6.1.3 Attribute Setting and Retrieval Methods

long GetProcessCount()
This function returns the current number of processes held in the CProcessManager container It is primarily used for administration and statistical purposes

5.7 The CProcess class

The primary purpose of the CProcess class is to define the basic attributes of a process in OO-MMURTL CProcess is an abstract class from which CJob and CSystemService are inherited, both of which are also abstract Note that application programmers will create process instances of either of these two subclasses, rather than of the base class CProcess, which will be inherited from when new types of executing entities are introduced to OO-MMURTL

CProcess is a generic class which defines the abstract behaviour of all subsequent subclasses of CProcess Further attributes which are specific to specialisations of CProcess are defined in the subclasses

5.7.1 Public Methods of the CProcess class

Many of the methods belonging to the CProcess class concentrate on setting and retrieving its data attributes Most of these are trivial such as the *SetExitJob() / GetExitJob()* pair whose code appears in Figure 5 4

```
void  CProcess  SetUserName(char *pUserRet)
{
  strcpy(sbUserName,pUserRet),
}

void CProcess  GetUserName(char *pUserRet)
{
  strcpy(pUserRet,sbUserName),
}
```

Figure 5 4 SetUserName() and GetUserName() methods

Due to the simplicity of their operation I will omit discussion of CProcess' more basic methods and instead concentrate on those which directly affect the operation of the OO-MMURTL Process Management Model. These are the constructor, destructor, and the *FreeResource()* method

CProcess() - [Constructor]

The CProcess constructor is only called from the Process Manager's *LoadNewJob()* method This is necessary in order that the new process request is first validated, the run file is then checked before code, data and stack segments are allocated and loaded Then the CProcessor constructor is invoked to create an object for the new process Finally the newly created CProcess object is added to the collection of CProcess objects maintained by the Process Manager If any other object tries to create a CProcess object, vital stages in its construction could be missed which could lead to errors

The CProcess constructor performs three basic tasks

1 It initialises attributes whose initial values are received through parameters to the constructor, for example process name, user name and process path
2 It sets the standard input to keyboard and the standard output to video
3 It sets the default values of the remaining variables which will always carry the same initial values for each task, for example cursor type, initial cursor position etc

The code for the CProcess constructor is shown in Figure 5 5

```
CProcess  CProcess(long Num, char *Name, char *User,
                   char *Path, char *CmdLine,
                   char *VidMem, char *VirtVid)
{
  /* Initialise variables as per parameters */
  ProcNum = Num,
  strcpy(sbProcName,Name),
  strcpy(sbUserName,User),
  strcpy(sbPath,Path),
  strcpy(ProcCmdLine,CmdLine);
  pVidMem = VidMem,
  pVirtVid = VirtVid,

  /* Set system input for this process to keyboard */
  strcpy(SysIn,"KBD"),

  /* Set system output for this process to video */
  strcpy(SysOut,"VID"),

  ExitError = 0,
  CrntX = CrntY = 0,       /* Initial cursor pos (0,0) */
  fCursOn = 1,             /* Cursor is on */
  fCursType = 1            /* Block cursor */
  ScrollCount = 0,
  NormVid = 7,             /* White on Black */
  strcpy(ExitRF,""),       /* No ExitRunFile initially */
}
```

Figure 5 5 CProcess constructor

An important point to note is that for each process, the default system input is always the keyboard and the default system output is always the monitor This may not always be the desire of the systems programmer For example in the case where a process must dump its entire output to a printer In such a case the programmer has two options

1 Reset the default system input value using the *CProcess->SetSysIn()* method This would have to be invoked immediately after the CProcess constructor to ensure that no output is missed

2 Derive a new subclass from the CProcess class The new class, called for example CPrinterDumpProcess, would have to create a new constructor, as shown in Figure 5 6 Now, the process will send its system output directly to the printer immediately following the call to its constructor creation This example, though trivial, shows how the CProcess hierarchy can be adapted to change the default behaviour of a set of processes

```
/* Class Definition */
/* Only a single method is rewritten the remainder are
   inherited from the CProcess class along with CProcess'
   attributes */

class CPrinterDumpProcess   public CProcess {
  public.
    CPrinterDumpProcess(long Num, char *Name, char *User,
                        char *Path, char *CmdLine,
                        char *VidMem,char *VirtVid),
},

/* Class Code */

CPrinterDumpProcess  CPrinterDumpProcess(long Num,
                  char *Name, char *User, char *Path,
                  char *CmdLine, char *VidMem,char *VirtVid)
    CProcess(Num,Name,User,Path,CmdLine,VidMem,VirtVid)

{
  /* Set system output for this process to printer */
  strcpy(SysOut,"PRN"),
}
```

Figure 5 6 The CPrinterDumpProcess class

~CProcess() - [Destructor]

The CProcess destructor is responsible for freeing any additional memory which was allocated by instantiations of the CProcess object Note that the underlying system resources used by the process - exchanges, page directories etc - should have been previously freed by the Process Manager by calling the CProcess method *FreeResources()*

FreeResources()

The purpose of the *FreeResources()* method is to deallocate all of the previously allocated system resources belonging to a process This involves

- Invoking the abstract method *FreeCurrentResources()* so that subclasses of CProcess may deallocate any additional system resources which they may have previously allocated
- Emptying the ReadyQueue of any tasks belonging to this CProcess object
- Freeing any exchanges used by this CProcess object

In this way, when control returns to the calling method, such as CProcessorManager's *Chain()* or *ExitJob()* methods, it can safely terminate this process knowing that all of its resources have been freed The *FreeResources()* method is shown in Figure 5 7

5.8 The CSystemService class

A system service is an installable program which provide system-wide message-based services for application programs and other services Each service is associated with an exchange Whenever a program wants a system service to perform an action, it sends a *Request* message to the appropriate exchange The service carries out any necessary actions before replying with a *Respond* message

In MMURTL, each system service is given an eight character name by which it is uniquely identified Examples of service names in OO-MMURTL include "FILESYSM" for the file service and "KEYBOARD" for the keyboard service

A service is only recognised by the system when it has called the function *RegisterSVC()* The following is the full set of steps required in order to set up a service

1 Allocate and initialise any resources required, including the main service exchange, additional exchanges, and additional memory
2 The function *RegisterSVC()* is called The name of the new service and the address of the function which will service the requests are passed as parameters
3 Wait for messages, service them, and respond

```
long CProcess  FreeResources()
{
  CExchange *pExch, *pCurrExch,
  long ercE, iE, ExchProc, CurrProc,

  // Allow subclasses to free additional system resources
  // which they may have allocated

  FreeSystemResources(),

  // Remove ALL tasks for this job that are at the ReadyQueue
  // This task won't be removed because it's Running'

  ReadyQueue->RemoveRdyProc(),

  /* Deallocate all exchanges for this process except the one
     belonging to current TSS  The Dealloc Exchange call will
     invalidate all TSSs found at exchanges belonging to this
     user, and will also free up RQBs and Link Blocks   The
     job will not be able to initiate requests or send messages
     after this */

  // Find out current TSS exchange so it isn't deallocated

  pExch = GetTSSExch(),
  CurrProc = GetCurrProcNum(),
  ercE = 0,
  iE = 0,

  // Loop through Exchanges, removing resources

  while (ercE != ErcOutOfRange) {
    ExchProc = ExchangeManager->GetOwner(iE),
    pCurrExch = ExchangeManager->GetExchange(iE),
    if (('ercE) && (ExchProc == CurrProc)&& (pCurrExch'= pExch))
      ercE = ExchangeManager->RemoveExch(iE),
    iE++,
    }

  /* Now that the user can't make anymore requests, Send Abort
     messages to all services  This closes all files that were
     opened by the Job and frees up any other resources held
     for this job by any services */

  SendAbort(),

 // Clear the exchange of abort responses (ignore them)

  TPacket *pPkt= pExch->CheckPacket(),
  while(pPkt==NULL)
    pPkt= pExch->CheckPacket(),
}
```

Figure 5 7 The CProcess FreeResources() method

The CSystemService class automates this behaviour in order to simplify the task of writing a system service and also to reduce the capacity for errors during the setting up of the service and the servicing of its requests

Each CSystemService class has two basic attributes - a main exchange, which will be used to route the requests, and a service name This remains the same as MMURTL In OO-MMURTL however, the creation and initialisation of the main exchange is automatically performed by the CSystemService constructor

In addition, the CSystemService class provides a specific abstract method, *ServiceRequest()*, which is used to service requests The private method, *Service()*, will monitor the system service's exchange, and call this function whenever a request is made The code for CSystemService's constructor and destructor are shown in Figure 5 8, while the code for its *Service()* method is shown in Figure 5 9

```
CSystemService  CSystemService(long Num, char *Name,
                        char *User, char *Path, char *CmdLine,
                        char *VidMem, char *VirtVid)
    . CProcess(Num,Name,User,Path,CmdLine,VidMem,VirtVid),
{
  SvcExch = CreateExchange(),
  strcpy(pSvcName,Name);
  RegisterService(pSvcName,SvcExch);
}

CSystemService  ~CSystemService()
{
  SvcExch->Remove();
  SvcExch->DeAllocate(),  // Free memory used by SvcExch
}
```

Figure 5 8 The CSystemService constructor and destructor

The CSystemService class has greatly simplified the process of writing a service The allocation of the service's exchange, the deallocation of its exchange, and the routing of its requests, are all performed by the class framework To create a new system service a programmer need only derive a new subclass of CSystemService and overwrite the *ServiceRequest()* message

```
void CSystemService  Service()
{
  unsigned long ErrorToUser,
  unsigned long Message[2],
  TRequest *pReqBlk

  while(1) {
    TPacket *pPkt = SvcExch->WaitPacket(),

    if('pPkt'=NULL) {
      pReqBlk = pPkt->Req,
      ErrorToUser = ServiceRequest(pReqBlk),
      Respond(pReqBlk,ErrorToUser),
      }
    }
}
```

Figure 5 9 The CSystemService Service() method

To demonstrate the simplicity of creating a system service, a trivial example is shown in Figure 5 10 The sole purpose of this service is to return unique ascending numbers in response to each *Request* The example device driver essentially consists of two lines of code One to initialise the counter, the other to assign the next value

```
class CNumbersService   public CSystemService {
  unsigned long NextNumber;

public·
   CNumbersService(long Num, char *Name, char *User,
                   char *Path, char *CmdLine, char *VidMem,
                   char *VirtVid),
    virtual unsigned long ServiceRequest(CReqBlock *pReqBlk);
}

CNumbersService  CNumbersService(long Num, "NUMBERS ",
                        char *User, char *Path, char *CmdLine,
                        char *VidMem, char *VirtVid)
      CSystemService(Num,"NUMBERS ", User, Path, CmdLine,
                        VidMem,VirtVid),
{
  NextNumber = 0,
}

unsigned long
CNumbersService· ServiceRequest(CReqBlock *pReqBlk)
{
  *pReqBlk pData1 = NextNumber++,
  return 0,
}
```

Figure 5 10 A sample CSystemService subclass

5.9 The CJob Class

This class forms the basis for future user applications and system programs It is an abstract class, which defines attributes and methods which are common to its subclasses, CUserJob and CSystemJob This common behaviour, which is abstracted in the CJob class, must be sufficiently generic so that it will also support further, currently unknown, types of jobs In other words, CJob should not be altered simply with CUserJob and CSystemJob in mind

Because the purpose of this design of OO-MMURTL is to replicate the existing behaviour of MMURTL, the CJob subframework is simple in its design Its sole purpose is to deal with the allocation of memory to the job in question The method AllocPage is an abstract one, which is implemented in concrete subclasses of CJob, in this case, CUserJob and CSystemJob

The class definitions for these three classes are presented in Figure 5 11

Future developments of OO-MMURTL can benefit from this class hierarchy in two ways, namely

1 Additional executing entities can be supplemented to the hierarchy, thus enabling them to inherit the basic behavioural traits from the existing classes, without affecting those classes or the programs which use them.
2 The existing classes can be augmented so that they provide further support for OO-MMURTL jobs

5.10 The CUserJob Class

This concrete class is a subclass of CJob Its role is to provide concrete implementations of CJob's abstract methods Currently, this involves providing an *AllocPage()* function which will allocate memory in an appropriate manner to a user job This becomes a simple matter of calling the CMemoryManager method *AllocPage()* (see *Chapter 7 The Memory Model*), which will attempt to allocate a specified amount of memory pages in user space, to the currently executing process

```
/************************************************
*                CJob Class Definition          *
************************************************/

class CJob : public CProcess {
public
    CJob(long Num, char *Name, char *User, char *Path,
         char *CmdLine, char *VidMem, char *VirtVid)
        CProcess(Num, Name, User, Path, CmdLine, VidMem,
                 VirtVid) {}0,

    virtual unsigned long AllocPage(unsigned long nPages,
                                    CPage *ppMemRet) = 0,
},


/************************************************
*              CUserJob Class Definition         *
************************************************/

class CUserJob : public CJob {
public·
    CUserJob(long Num, char *Name, char *User, char *Path,
             char *CmdLine, char *VidMem, char *VirtVid)
        CJob(Num,Name,User,Path,CmdLine,VidMem,VirtVid) {},

    virtual unsigned char *AllocPage(unsigned long nPages),
};


/************************************************
*             CSystemJob Class Definition        *
************************************************/

class CSystemJob . public CJob {
public·
    CSystemJob(long Num, char *Name, char *User,
               char *Path, char *CmdLine, char *VidMem,
               char *VirtVid)
        CJob(Num,Name,User,Path,CmdLine,VidMem,VirtVid) {},

    virtual unsigned char *AllocPage(unsigned long nPages),
};
```

Figure 5 11 The CJob, CUserJob, and CSystemJob class definitions

5.11 The CSystemJob Class

In the current implementation of OO-MMURTL, CSystemJob behaves in a similar manner to CUserJob This is reflected in the operation of system and user jobs by the operating system. The only difference between the two being that while CUserJob allocates pages from user memory, CSystemJob allocates pages from system memory As a result, CSystemJob's *AllocPage()* invokes the CMemoryManager method *AllocOSPage()* (see *Chapter 7 The Memory Model*), which will attempt to allocate a specified amount of memory pages in operating system space, to the currently executing process

The code for both of these implementations is shown in Figure 5 12

```
/************************************************
*              CUserJob Implementation          *
************************************************/

unsigned char *CUserJob:.AllocPage(unsigned long nPages)
{
  return MemoryManager->AllocPage(nPages),
}


/************************************************
*             CSystemJob Implementation         *
************************************************/

unsigned char *CSystemJob: AllocPage(unsigned long nPages)
{
  return MemoryManager->AllocOSPage(nPages),
}
```

Figure 5 12 Job allocation methods

5.12 Device Drivers

Device drivers can be loaded into the system during or following boot time As its position as a subclass of CProcess denotes, device drivers are programs in their own right They behave differently, however, from a conventional system or user program.

In MMURTL, in order for a device driver to be recognised by the system it has to perform the following steps

1 Allocate and initialise any system resources which are required
2 Allocate additional memory, if required
3 Allocate exchanges, if required
4 Set up interrupt service routines, if required
5 Check and initialise device
6 Initialise the DeviceControlBlock (DCB) structure
7 Enable any hardware interrupts which are being serviced
8 Call *InitDevDr()*, passing the DCB as a parameter

At this point the device driver task terminates but its code remains resident From this point on, the device driver is available to external applications The device driver must provide the following three public functions which will respond to calls from the operating system with device-specific actions

1 *DeviceInit()* - This function is called to initialise a device or reset one following a crash
2 *DeviceOp()* - This is used by services and programs to carry out operations pertinent to the device For example, a floppy disk driver's operations would include Read and Write operations Each device has a table of possible operations One of the parameters to the *DeviceOp()* function would specify which operation is to be performed while another would hold a pointer to data required by the operation
3 *DeviceStat()* - This function returns the status of the device controlled by the driver The value of the status is particular to each device When a device doesn't keep a status a zero should be returned to indicate there are no problems

There are two different types of device driver - sequential and random. A sequential driver deals with devices which handles data in fixed-size blocks A random device driver deals with variable sized data Device drivers can also be classified by their facility for re-entrancy Certain devices are capable of dealing with several programs at a time, others operate exclusively for a single program.

In order for a device driver to operate in non-reentrant mode, its programmer must include code which will guarantee mutual exclusion within each of the drivers three public functions

OO-MMURTL provides three classes to describe its device drivers facilities They are CDeviceDriver, CNonReentrantDeviceDriver and CReentrantDeviceDriver

5.13 The CDeviceDriver class

This is a base class which defines the attributes which are held by each object The CDeviceDriver class also provides a pair of constructors One initialises the class attributes as a sequential device driver, the other for a random one The code for each of these is shown in Figure 5 13

```
CDeviceDriver  CDeviceDriver(char *DevName, unsigned int BPB,
                             int Blocks,    int SingleUser)
{
  strcpy(Name,DevName);
  nBPB = BPB,                  // Bytes per block
  nBlocks = Blocks,
  fSingleUser = SingleUser,    // Is device assignable?

  Type = 2,                    // Sequential device driver

  LastDevErc = 0,
  wJob = 0,
}


CDeviceDriver  CDeviceDriver(char *DevName, int SingleUser)
{
  strcpy(Name,DevName),
  fSingleUser = SingleUser,    // Is device assignable?

  Type = 1,                    // Random device driver
  nBPB = 0,                    // Does not apply
  nBlocks = 0,                 // Does not apply

  LastDevErc = 0,
  wJob = 0,
}
```

Figure 5 13 The CDeviceDriver constructors

CDeviceDriver also defines three public abstract classes and three abstract private classes The three public functions are

- Operation()
- Status()
- Initialise()

Each of these corresponds to the original *DeviceInit()*, *DeviceOp()* and *DeviceStat()* functions When an operation is requested by a program or service, the message is sent to the Device Driver Manager This in turn forwards the message to the appropriate CDeviceDriver object

Each of CDeviceDriver's three abstract public functions also have a corresponding abstract private function

- DevOperation()
- DevStatus()
- DevInitialise()

Although this may seem like unnecessary redundancy, the purpose of these mirror functions becomes clear in further subclasses of CDeviceDriver

5.14 The CNonReentrantDeviceDriver class

This class provides intrinsic mutual exclusion in order to ensure non-reentrancy The reason for providing two functions for each action now becomes clear The public methods act as wrappers providing protection in order to ensure mutual exclusion exists Once mutual exclusion has been granted, the corresponding private method is invoked The private methods remain abstract in this class This should be obvious since the initialisation, operation, and status code must be written by the device driver programmer

Mutual exclusion is achieved by using an exchange as a semaphore Each non-reentrant device driver must initialise an exchange in its constructor Before entering each critical section, the object performs a WaitMsg on the exchange If there is already another process in the device driver's critical section, the others will wait until a dummy message is sent to the exchange

The code for the three public CDeviceDriver methods are presented in Figure 5 14

```
long CNonReentrantDeviceDriver  Operation(unsigned long dOpNum,
                    unsigned long dLBA,unsigned long dnBlocks,
                    unsigned char  *pData)
{
  long erc,

  TPacket *pSemPkt = SemExch->WaitPacket(), // Wait for MutEx

  /* Mutual exclusion has been achieved, perform operation */
  erc = DevOperation(dOpNum,dLBA,dnBlocks,pData),

  SemExch->SendDummyPacket();  // Signal

  return erc,
}

long CNonReentrantDeviceDriver  Status(char *pStatRet,
                            unsigned long dStatusMax,
                            unsigned long *pdSatusRet)
{
  long erc,

  TPacket *pSemPkt = SemExch->WaitPacket();

  /* Mutual exclusion has been achieved, retrieve status */
  erc = DevStatus(pStatRet,dStatusMax,pdStatusRet),

  SemExch->SendDummyPacket(),  // Signal

  return erc,
}

long CNonReentrantDeviceDriver..Initialise(char *pInitData,
                                  unsigned long sdInitData)
{
  long erc,

  TPacket *pSemPkt = SemExch->WaitPacket(),

  /* Mutual exclusion has been achieved, initialise */
  erc = DevInit(pInitData,sdInitData),

  SemExch->SendDummyPacket(),

  return erc,
}
```

Figure 5 14 The CNonReentrantDeviceDriver class implementation

5.15 The CReentrantDeviceDriver Class

This class is similar to, but simpler than, its sibling, CNonReentrantClass The reason for this is that mutual exclusion is not required by the device driver, hence the three public functions become trivial. they are presented in Figure 5 15

```
long CReentrantDeviceDriver  Operation(unsigned long dOpNum,
                    unsigned long dLBA, unsigned long dnBlocks,
                    unsigned char  *pData)
{
    return DevOperation(dOpNum,dLBA,dnBlocks,pData),
}

long CReentrantDeviceDriver  Status(char *pStatRet,
                                    unsigned long dStatusMax,
                                    unsigned long *pdStatusRet)
{
    return DevStatus(pStatRet, dStatusMax,pdStatusRet),
}

long CReentrantDeviceDriver  Initialise(char *pInitData,
                                        unsigned long sdInitData)
{
    return DevInit(pInitData,sdInitData),
}
```

Figure 5 15 The CReentrantDeviceDriver class

5.16 Summary

This chapter introduced the OO-MMURTL process management frameworks This involved abstracting the entire functionality of MMURTL's original job management model, and encapsulating that behaviour into a set of classes The classes that were introduced in this chapter are CProcessManager, CProcess, CSystemService, CJob, CUserJob, CSystemJob, CDeviceDriver, CNonReentrantDeviceDriver and CReentrantDeviceDriver

The most important advantage gained from implementing the process management class hierarchy was in providing a framework which facilitates enhancements and further specialisations of the current model.

The main disadvantage which results from using the newly implemented process management model is caused by the introduction of the Object Manager Since tables have been disposed of, speed of access to processes has slowed, although not by a significant amount This, however, has been counterbalanced by the advantages provided through the use of the CProcessManager class, primarily that secure access has at last been introduced to the process model In addition, the CProcessManager is capable of introducing advanced process management mechanisms and policies which would have been difficult to implement under the previous table-based process management system in MMURTL Also, because the operation of this subsystem has been centralised around a single component, the process manager, MMURTL's fragmentation of code which manipulated the job tables has been eliminated

With reference to the subframeworks within OO-MMURTL's process management system, there have been similar advantages provided as a result of object-orientation Taking the CDeviceDriver framework as an example, the device driver programmer need no longer deal with problems regarding re-entrancy Mutual exclusion can be easily achieved through the use of a CNonReentrantDeviceDriver instead of a CReentrantDeviceDriver Other than the point at which the device driver programmer creates an instance of CNonReentrantDeviceDriver, the mechanism which provides non-reentrancy is hidden from the programmer

In summary, the process management framework in this chapter provides several promising advantages over the original implementation in MMURTL There remains however, much scope for development (see *Chapter 10 Conclusions*) The opportunity for this development is facilitated by the object-oriented hierarchy which has been designed with future enhancements in mind

Chapter 6

The Messaging Model

6.1 Overview

This chapter introduces the OO-MMURTL messaging model Firstly, the implementation of messaging in the original MMURTL operating system is explained Following this the class which is central to the messaging model, the CExchangeManager class, is introduced and explained Finally, each of the three classes which comprise the OO-MMURTL message model are described - CExchange, CServiceExchange, and CMessageExchange

6.2 Messaging in MMURTL

MMURTL is a message-based operating system. Messages are an integral part of many operating system functions They form the basis for interprocess communication, process synchronisation, and for communicating requests to system services In addition, they can be used for transferring data between processes

An integral part of MMURTL's messaging system is the concept of an *Exchange* Messages are never sent directly from one process to another This would require prior knowledge of the existence and address of the receiving process Instead, messages are sent to an exchange which behaves like a virtual post office box where a message is sent until a process retrieves it

In order for a task to use an exchange, it must request from the operating system that one be allocated to it. It is the responsibility of this task to make sure that the exchange is returned to the operating system to be deallocated when it has finished with it

To ensure optimum flexibility, few assumptions are made in the messaging system. When a message is sent to an exchange, a process may or may not be expecting it Similarly, it is possible that an unretrieved message is already at an exchange when another one is sent It is also feasible for a process to be waiting for a message which has yet to be sent

Each exchange has two queues associated with it - a message queue and a task queue Both operate under the first come first served principle If a message arrives at an exchange and there are no waiting tasks, it is placed on the message queue If another message arrives, it is placed behind it on the queue The reverse is true for tasks If a task arrives at an exchange and no messages are there, the task is placed in the task queue Items can be on either the message queue, the task queue or neither queue It would be contradictory, and therefore is impossible, for items to be on both queues

There are two different message types in MMURTL The smallest and most common form is simply known as a *Message* A message is comprised of two double-word (32 bytes) sized variables The usage of these fields is defined by the tasks which are communicating with it Messages are sent and received using the *SendMsg() / WaitMsg()* function pair

The second form of a message is a *Request* A request is only used when a task or system service communicates with another system service Specifically, requests are made when a service is required to be performed by the system service on behalf of the calling process Requests are comprised of information required by the system service, such as the address of an exchange where it is to post its response, a service code defining which service is to be performed, along with six fields which hold data whose usage is defined by the service in question and the operation to be performed

Communication with system services is performed using the *Request() / Respond()* pair To avoid confusion, a communication in OO-MMURTL, that is either a basic message or a request, is referred to as a packet

6.3 The CExchangeManager Class

As with OO-MMURTL's other Object Managers, the CExchangeManager is a subclass of CObjectManager, inheriting all of the methods and attributes which are required to store and access the objects under its protection

In addition to its role as an object container, the purpose of the CExchangeManager class is to manage OO-MMURTL's message processing system. Due to the nature of messaging, this can at times prove to be a more complicated task than is performed by other Object Managers The nature of this complexity is apparent when it is considered that OO-MMURTL's messaging component is one of only two places in the operating system which performs task switching (the other being the timer management component)

6.3.1 Responsibilities of the CExchangeManager Class

The activities of the CExchangeManager can be as complex as they are diverse The following is a list of actions for which the CExchangeManager is responsible

Maintaining Exchanges - As Exchange Manager, the CExchangeManager class must allocate exchanges to processes upon receipt of correctly formatted requests In OO-MMURTL, exchanges are always owned by the Exchange Manager, never by the task which requested their allocation or by any other tasks which may use them. This is for security reasons The alternative scenario involves the Exchange Manager returning a pointer to each newly allocated exchange to the requesting process Each task which wants to retrieve (or send) messages to the exchange would need to be given a copy of this pointer In a common situation where several different tasks require access to a given exchange, the Exchange Manager has no control over use, and more importantly misuse, of the many copies of the exchange pointer Instead the Exchange Manager retains all copies of pointers to CExchange objects When a new Exchange is allocated, a unique Exchange Key is returned to the task which requested its allocation All requests to Exchanges must be made through the CExchangeManager class, quoting the relevant Exchange Key In this way possible misbehaviour can be monitored by the Exchange Manager and repercussions kept to a minimum.

Routing Packets - As packets are received by the Exchange Manager, the destination CExchange object must be determined and the message forwarded to it Because the data contained within the packet is defined by the communicating tasks, the Exchange Manager cannot verify the correctness of its contents This must be done by the receiving Exchange

Looking-up services - Much of the time, several of the Exchanges will be allocated to system service tasks Each of these will expect Request type messages, instead of the more basic message type It is the responsibility of the Exchange Manager to differentiate messages from requests, and route messages to the appropriate CExchange object based upon the Exchange Key Due to the dynamic nature of system services, the same service will most likely be allocated different Exchange Keys each time the system is started For this reason, the use of keys proves inadequate in locating their exchanges Instead, the Exchange Manager is responsible for managing the System Service Naming Directory Each service is given a unique eight character name by its system programmer This name remains constant It is used by application programmes to access the system service's exchange in order to send a request to it Each time a system service is added to the operating system, the Exchange Manager will associate its name with the appropriate exchange in the Name Directory Upon receipt of a request, the Exchange Manager will then look up the Name Directory, before forwarding the request

Securing Task Switches - This relates to the security provided by the Exchange Manager Because task switches can be performed as a result of messages receiving a message or tasks waiting for one, security is an issue of particular importance in the Exchange Manager Although not every transaction can be fully vetted, such as the contents of a message, the Exchange Manager must ensure to the best of its abilities that a secure operating environment exists at all times

6.4 The Exchange Hierarchy

In order to accommodate both message types, basic and request, the OO-MMURTL exchange hierarchy provides two exchange classes - CServiceExchange and CMessageExchange The basic behaviour of each of these is defined by the CExchange class The hierarchy is displayed in Figure 6 1

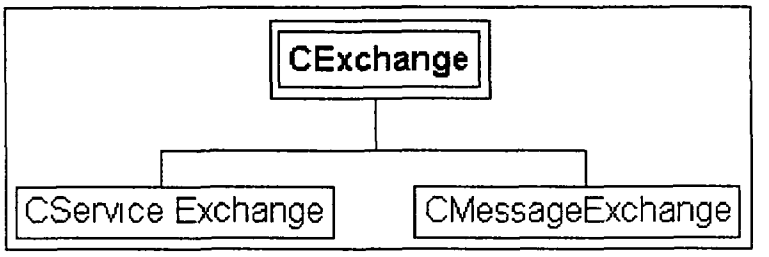


Figure 6 1 The CExchange class framework

The CExchange class can encapsulate the functionality to control the two queues which belong to each Exchange - one for waiting tasks, the other for waiting messages It also serves to receive messages from either type of exchange and to check if requests exist at an exchange Due to the different nature of messages and requests, however, the generic CExchange class is insufficient to send messages to an exchange - the *SendMsg()* method is inherently different from the *Request()* method Each of the classes in the hierarchy are described below

6.5 The CExchange Class

This class is the base class in the exchange hierarchy It is also an abstract class CExchange has three roles Firstly, it must provide functions to manage the task and message queues for the exchange object These functions are private to the CExchange class They may only be accessed by CExchange objects and instantiations of subclasses of CExchange This is an important security feature The four functions which access the queues - *deQueueTSS()*, *enQueueTSS()*, *deQueueMsg()* and *enQueueMsg()* - are capable of modifying them. Modifying either queue however, could result in a task switch Therefore it is essential that none of the methods be interrupted midway through processing The code to clear the interrupt flag (prevent interrupts) does not lie within any of these methods Instead the responsibility belongs to the methods which call them. For this reason, the four queue methods are private They are called by public methods of the CExchange class and its subclasses

The second role of the CExchange class is to provide functionality so that processes can check to see if a message is at an exchange and also to wait until a message is at an exchange These two operations make use of the link block construct The code for this simple structure is shown in Figure 6 2

```
struct TPacket {
  struct TRequest *Req,
  struct TMessage *Msg,

  struct TPacket *next,
},
```

Figure 6 2

In essence it consists of two data pointers One points to a request structure, the other to a message structure These are shown in figure 6 3 A TPacket record will store either a pointer to a request or a pointer to a message, but not both The definition of a packet enables CExchanges to deal with packets of information, without having to unnecessarily determine if the packet is a message or a request Although by definition, this means that one pointer belonging to each packet will always be wasted, the advantage comes from increased processing speed Only when the packet reaches its final destination will a check to determine the nature of its contents have to be made The third field in a packet is a pointer to another packet for use when packets are stored on queues

```
struct TRequest {
  CExchange *RespExch,   // Exchange to respond to
  long RqOwnerProc,      // JobNum of Owner
  int  ServiceCode,      // Sys Service Command Number
  long dData0,           // Srvc Defined (No Pointers)
  long dData1;           // Srvc Defined (No Pointers)
  long dData2,           // Srvc Defined (No Pointers)
  char *pData1,          // Srvc Defined
  long cbData1,          // Length of data in pData1
  char *pData2,          // Srvc Defined
  long cbData2,          // Length of data in pData2

  struct TRequest *next,
},

struct TMessage{
  long dData1,                          , Data field 1
  long dData2,                          , Data field 2
},
```

Figure 6 3 Messaging structures

The final role of the CExchange class is to send packets of data to its exchange The *SendPacket()* method is only called by subclasses of CExchange It may only be called once a packet has been created and formatted For this reason, *SendPacket()* is only called by *Request()* in the CServiceExchange class and by *SendMsg()* in the CMessageExchange subclass

The methods belonging to each of these groups are described below

6.5.1 Queue Management Methods

To support the queue management methods, CExchange has four attributes, two for each queue The attributes are pointers to the head and tail of each queue They are - *pPacketQueueHead*, *pPacketQueueTail*, *pTSSQueueHead*, and *pTSSQueueTail*

*TTSS *deQueueTSS()*
This method returns a pointer to the task at the top of the queue If no task is present a NULL is returned The code for this method is shown in Figure 6 4

*void enQueueTSS(TTSS *pTSS)*
This method adds a pointer to a task to the task queue The code for this method is also shown in Figure 6 4

```
TTSS *CExchange  deQueueTSS()
{
  TTSS *pTSS,

  pTSS = pTSSQueueHead,
  pTSSQueueHead = pTSSQueueHead->next;

  return pTSS,
}

void CExchange  enQueueTSS(TTSS *pTSS)
{
  if (pTSSQueueHead == NULL) {
    pTSSQueueHead = pTSSQueueTail = pTSS;
    pTSSQueueHead->next = NULL,
    }
  else {
    pTSSQueueTail->next = pTSS,
    pTSSQueueTail = pTSS,
    }
}
```

Figure 6 4 CExchange's enQueueTSS() and deQueueTSS() methods

*TPacket *deQueuePacket()*

This is similar to the deQueueTSS method described above It returns a pointer to the packet at the top of the queue If no packet is present a NULL is returned

*void enQueuePacket(TPacket *pPacket)*

Again, this method is similar to enQueueTSS It adds a packet pointer to the packet queue

6.5.2 Packet Retrieval Methods

OO-MMURTL provides two functions with which to access packets at an exchange - *CheckPacket()* and *WaitPacket()* They differ in one respect - *CheckPacket()* is a non-blocking function, while *WaitPacket()* performs blocking This is described in detail below

*TPacket *CheckPacket()*

This is the simpler of the packet retrieval methods *CheckPacket()* checks to see if a packet is waiting at an exchange If there is a waiting packet, it is removed and returned to the task which invoked the method If no packet is waiting, the calling task is notified and it continues processing The code for *CheckPacket()* is shown in Figure 6 5

```
TPacket *CExchange· CheckPacket()
{
  TPacket *pPacket,

  // Disable interrupts
  #asm
  CLI
  #endasm

  pPacket = deQueuePacket();

  // Reenable interrupts
  #asm
  STI
  #endasm

  return pLB,
}
```

Figure 6 5 The CExchange CheckPacket() method

*TPacket *WaitPacket()*

Like *CheckPacket()*, this method also checks to see if there are any waiting packets at the exchange queue As before, if a packet is found, it is returned to the calling task and it continues running However, if no packet is found, then the current task is queued at the exchange The task with the highest priority in the Ready Queue is removed If there are no tasks on the Ready Queue, then interrupts are enabled and the processor is halted The processor will be restarted automatically when an interrupt is caused Interrupts will then be disabled and a check is made to see if there is a task on the Ready Queue as a result of the interrupt being serviced Only when a task is retrieved from the Ready Queue will the *WaitPacket()* method continue At this stage a task has been identified that can continue running If this task was the same as the original task which called WaitPacket(), then interrupts are reenabled and control returns to it from the method If the task is different from the original one, then a task switch is performed before continuing The code is shown in Figure 6 6a and Figure 6 6b

```
TPacket *CExchange. WaitPacket()
{
  TPacket *pPacket;
  TTSS *pRunTSS, *pPriorityTSS,

  #asm
  CLI
  #endasm

  pPacket = deQueuePacket(),

  if (pPacket==NULL) {
    // Add the current task to the TSSQueue of this exchange
    pRunTSS = GetpRunTSS(),
    pRunTSS->next = NULL,
    enQueueTSS(pRunTSS);

    // Get the next TSS to run (if there is one)
    pPriorityTSS = ReadyQueue->deQueueRdy(),
```

Figure 6 6a The CExchange WaitMsg() method

```
    // If none were ready, loop until one is
    while (pPriorityTSS == NULL) {
      #asm
      STI
      HLT
      CLI
      #endasm

      pPriorityTSS = ReadyQueue->deQueueRdy(),
      }

    if (pPriorityTSS != pRunTSS) {
      // Tasks are switched by performing a 386 task switch
      #asm
      MOV EAX, [pPriorityTSS]
      MOV BX,[EAX.Tid]
      MOV TSS_Sel,BX
      JMP FWORD PTR [TSS]
      #endasm
      }
    }
    // A task has finished "Waiting"  Now in the new task
  }

  /*  we have either switched tasks and are delivering a
      packet to the new task, or there was a packet waiting
      at the exch of the first caller and we are delivering
      it  */

  #asm
  STI
  #endasm

  return pPacket,
}
```

Figure 6 6b The CExchange WaitMsg() method

6.5.3 Packet Routing Methods

The CExchange class has a single packet routing method It is a private method which is only invoked by subclasses of CExchange

*void SendPacket(TPacket *pPacket)*
The purpose of this method is to forward a packet to an exchange If no task is waiting at the exchange, the packet is queued If a task is waiting, the packet is given to that task and the current task is placed on the Ready Queue The Ready Queue is then re-evaluated and the highest priority task is removed from it The reason this occurs is because it is possible that a task with a higher priority than the current one was waiting for the packet that was delivered If this is so then the current task is queued and the highest priority task on the Ready Queue regains control of the processor The code for this method is shown in Figure 6 7 Note that interrupts must be disabled before a call is made to this method

6.6 The CServiceExchange Class

This class is a subclass of CExchange Instantiations of the CServiceExchange class may only be created by system services CServiceExchanges are designed to process requests for action by the system service which owns it System services may also own non-service exchanges, but not to process requests CServiceExchange has three request-specific methods The *Request()* method is called by tasks which require a service to be performed, the *Service()* method is used to respond to such a request, and the *MoveRequest()* method is used by system services to move a request from one of its exchanges to another Each of these is described in turn below

```
void CExchange  SendPacket(TPacket *pPacket)
{
  TTSS *pWaitTSS,*pPriorityTSS,

  // Remove task from the exchange's queue
  pWaitTSS = deQueueTSS(),

  // If no task is waiting, queue the packet
  if (pWaitTSS==NULL) {
    enQueuePacket(pPacket),
    return,
    }

  // If a task was waiting notify it of the received packet
  pWaitTSS->pLBRet = pPacket,

  // Reevaluate the Ready Queue in case a higher
  // priority task is available
  ReadyQueue->enQueueRdy(pWaitTSS),
  pPriorityTSS = ReadyQueue->deQueueRdy(),

  // If the highest priority task is the current one,
  //   no task switch is required
  if (pPriorityTSS == pWaitTSS)
    return;

  /* Perform a 386 processor task switch */
  #asm
  MOV EAX,pPriorityTSS
  MOV BX,[EAX Tid]
  MOV TSS_Sel,BX
  INC _nSwitches
  JMP FWORD PTR [TSS]
  #endasm
}
```

Figure 6 7 The CExchange SendPacket() method

*void Request(int code, CExchange *respexch, long data0, long data1, long data2, char *pdata1, long cbdata1, char *pdata2, long cbdata2)*

This method creates a TRequest record and assigns its data based on the parameters which have been passed to it The request is then placed into a TPacket structure Interrupts are disabled before the *SendPacket()* method is called This method is inherited from CServiceExchange's base class, CExchange Upon return from *SendPacket()*, interrupts are enabled before returning from the method The code for this method is shown in Figure 6 8

```
void CServiceExchange::Request(int code, CExchange *respexch,
            long data0, long data1, long data2, char *pdata1,
            long cbdata1, char *pdata2, long cbdata2)
{
  TPacket *pPacket,
  TRequest *pReq,

  /* Create request structure */
  pReq = new TRequest,

  pReq->ServiceCode = code,
  pReq->RespondExch = respexch,
  pReq->RqOwnerProc = GetCrntProcNum(),
  pReq->dData0 = data0,
  pReq->dData1 = data1,
  pReq->dData2 = data2,
  pReq->pData1 = pdata2,
  pReq->cbData1 = cbdata1,
  pReq->pData2 = pdata2,
  pReq->cbData2 = cbdata2,

  /* Create the packet */
  pPacket = new TPacket,
  pPacket->Req = pReq,
  pPacket->Msg = NULL,

  /* Disable interrupts */
  #asm
  CLI
  #endasm

  SendPacket(pPacket),

  /* Reenable interrupts */
  #asm
  STI
  #endasm
}
```

Figure 6 8 The CServiceExchange Request() method

*void Respond(TRequest *pReq)*

This method is used by a system service to notify the task which requested a service that the task has completed (or that the service was unable to be completed) The full steps in the Request/Respond process are as follows

1 A task sends a request to a system service The request is composed of a service code, data which may be required to perform the service, and an exchange owned by the calling task at which it will wait for a response

2 The system service retrieves the request from its queue and deals with it
3 Upon completion, the system service invokes the *Respond()* method of the exchange object whose pointer was passed as a parameter to the *Request()* method It places the original Request structure that, which has been modified to reflect function performed by the service and a status reflecting the success of its action
4 The calling task will either wait or else periodically check the respond exchange, for the reply by the system service

The first task performed by this method is to dealias the memory pointers in the TRequest structure if the current process is different from the process which created the TRequest This is necessary otherwise the pointers would point to the wrong segment, albeit at the correct address The dealiasing is performed by the *DeAliasMem()* kernel primitive As before, the exchange is checked for waiting tasks, if there are none, the request structure is queued Otherwise, the current task is placed on the ReadyQueue The highest priority task is then removed from the Ready Queue If the two tasks are different, a processor task switch is performed The code for this method is shown in Figure 6 9a and Figure 6 9b

```
void CServiceExchange.:Respond(TRequest *pReq)
{
  long dCurrProc,dReqProc;
  TTSS *pTSS,*pPriorityTSS,
  TPacket *pPacket;

  dCurrProc = GetCurrProcNum(),
  dReqProc = pReq->RqOwnerProc,

  /* Perform memory aliasing if required */
  if (dReqProc'=dCurrProc) {
    if (pReq->cbData1 > 0) && (pReq->pData1 '= NULL)
      DeAliasMem(pReq->pData1,pReq->cbData1,dCurrProc),

    if (pReq->cbData2 > 0) && (pReq->pData2 '= NULL)
      DeAliasMem(pReq->pData2,pReq->cbData2,dCurrProc),
    }

  /* Disable interrupts */
  #asm
  CLI
  #endasm
```

Figure 6 9a The CServiceExchange Respond() method

```
  /*Create packet structure */
  pPacket = new TPacket,
  pPacket->Req = pReq;
  pPacket->Msg = NULL;
  pPacket->next = NULL,

  /* Remove waiting task (if any) */
  pTSS = deQueueTSS(),
  if (pTSS == NULL) {
    enQueuePacket(pPacket),
    #asm
    STI
    #endasm
    return,
    }

  /* Store request in the dequeued task */
  pTSS->pLBRet = pPacket,

  /* Reevaluate the ready queue */
  ReadyQueue->enQueueRdy(pTSS),
  pPriorityTSS = ReadyQueue->deQueueRdy();

  /* If the highest priority task is the same as the
     original task, return */
  if(pPriorityTSS == pTSS) {
    #asm
    STI
    #endasm
    return,
    }

  /* Switch task if the highest priority task is
     not the original task */
  #asm
      MOV EAX,pPriorityTSS
      MOV BX,[EAX.Tid]
      MOV TSS_Sel,BX
      INC _nSwitches
      JMP FWORD PTR [TSS]
      STI
  #endasm
}
```

Figure 6 9 The CServiceExchange Respond() method

6.7 The CMessageExchange Class

This class is a subclass of CExchange Any active task may request the creation of a CMessageExchange class This class, combined with the methods inherited from CExchange, is designed to assist interprocess communication, data transfer, and task synchronisation CMessageExchange provides two methods to facilitate sending messages to exchanges They are *SendMsg* and *ISendMsg()*, each of which is described below

void SendMsg(long dMsgData1, long dMsgData2)
This method accepts two doubleword parameters, both of which hold generic data The nature of each variable is determined by the communicating processes Using this data, *SendMsg()* creates a TMessage structure This message is then stored in a TPacket structure before interrupts are disabled and the *SendPacket()* method which was described above is called. Upon returning, *SendMsg()* reenables interrupts before exiting The code is shown in Figure 6 10

```
void CMessageExchange:.SendMsg(long dMsgData1, long dMsgData2)
{
  TPacket *pPacket,
  TMessage *pNewMsg,

  /* Create & fill the TMessage structure */
  pNewMsg = new TMessage;
  pNewMsg->dData1 = dMsgData1,
  pNewMsg->dData2 = dMsgData2,

  /* Create & fill the TPacket structure */
  pPacket = new TPacket,
  pPacket->Req = NULL,
  pPacket->Msg = pNewMsg,
  pPacket->next = NULL;

  /* Disable interrupts */
  #asm
  CLI
  #endasm

  SendPacket(pPacket),

  /* Reenable interrupts */
  #asm
  STI
  #endasm
}
```

Figure 6 10 The CMessageExchange SendMsg() method

void ISendMsg(long dMsgData1, long dMsgData2)

This method is extremely similar to *SendMsg()* It differs in that *ISendMsg()* is designed to be called from within an interrupt service routine For this reason no task switch is performed and interrupts remain cleared The *ISendMsg()* method, having created a TMessage structure and stored it in a TPacket, checks to see if a task is waiting at the exchange If so, the packet is assigned to the task and it is returned to the Ready Queue Otherwise, the packet is simply added to the packet queue at the exchange

```
void CMessageExchange  ISendMsg(long dMsgData1, long dMsgData2)
{
  TPacket *pPacket;
  TMessage *pNewMsg,*pMessage,
  TTSS *pWaitTSS,

  /* Disable interrupts */
  #asm
  CLI
  #endasm

  /* Create the message structure */
  pMessage = new TMessage;
  pMessage->dData1 = dMsgData1,
  pMessage->dData2 = dMsgData2,

  /* Create the packet */
  pPacket = new TPacket,
  pPacket->Msg = pMessage,
  pPacket->Req = NULL;
  pPacket->next = NULL,


  pWaitTSS = deQueueTSS();
  if (pWaitTSS==NULL) {
    enQueuePacket(pPacket);
  else {
    pWaitTSS->pLBRet = pPacket,
    ReadyQueue->enQueueRdy(pWaitTSS),
    }
}
```

Figure 6 11 The CMessageExchange ISendMsg() method

6.8 Conclusions

This chapter introduced the OO-MMURTL messaging model The role of the class CExchangeManager was described in detail This is the central component of the messaging model, which itself is an integral part of object-oriented MMURTL, providing a means of interprocess communication, process synchronisation, and system service messaging

The classes which encapsulate the behaviour of the messaging system were also introduced, CExchange, CServiceExchange, and CMessageExchange Further concrete subclasses of the abstract CExchange class could be provided by the system programmer, allowing a more diverse implementation of messaging in MMURTL

The messaging component of the original MMURTL system was a suitable candidate for migration to object-orientation, thanks to the modular nature of its design. This is not always the case as is demonstrated in the next chapter

Chapter 7

The Memory Model

7.1 Overview

This chapter introduces the design of OO-MMURTL's memory model Before this is detailed, however, an introduction to some of the advanced concepts found in MMURTL's original memory model is provided These are explained in order to provided a clearer depiction of their role in the design of the new model further into the chapter Next, the reasoning underlying the design of the new memory model is provided as an introduction to the detailed description of the CMemoryManager class Finally, the conclusions drawn from the work in this chapter are presented

7.2 MMURTL's Memory Model

MMURTL is a paged memory operating system. It makes use of the Intel hardware-based paging facilities for memory allocation and management The design and implementation of the memory model is quite complex, however this complexity at the system design level alleviates the burden on application and system programmers, by ensuring that the memory programming interface is as simple as possible

The basic concepts of the MMURTL memory model have been discussed previously in *Chapter 2 The MMURTL Operating System* This chapter explains the operation of the MMURTL memory model in further detail This includes the discussion of advanced topics such as memory aliasing and shadow memory

The complexity of the memory model provoked a difficult decision in the design of OO-MMURTL, the design options, along with the chosen design of the new memory model are also presented in this chapter

Following this, the implementation of the OO-MMURTL memory model is presented This includes descriptions of the memory model classes, methods, attributes and behaviour

Finally, possible future developments of the OO-MMURTL memory model are offered These developments are presented in light of the design decisions described earlier in the chapter and further possibilities for development of the memory model in light of this decision

7.3 Memory Model Advanced Concepts

Before discussing the design of the OO-MMURTL memory model, it is necessary to describe an important underlying concept of OO-MMURTL's low-level memory operations, Shadow Memory In addition, the use of Memory Aliasing in MMURTL is also described in this section

7.3.1 Shadow Memory

Each MMURTL process has a single page directory This page directory holds 1024 page directory entries, each of which is four bytes long, which points to a page table In turn, each page table also contains 1024 entries, each of which points to a page of linear memory Each page is four kilobytes in size Thus

> Total size of addressable linear memory by a MMURTL process
> = 1024 (Page Directory Entries) x 1024 (Page Table Entries) x 4Kb (Pages)
> = 4,294,967,296 (4Gb)

The operating system code and data are mapped into the bottom of every process's address space As a result, each process's own memory actually begins at the 1Gb linear memory mark

The memory schema for every process is shown in Figure 7 1

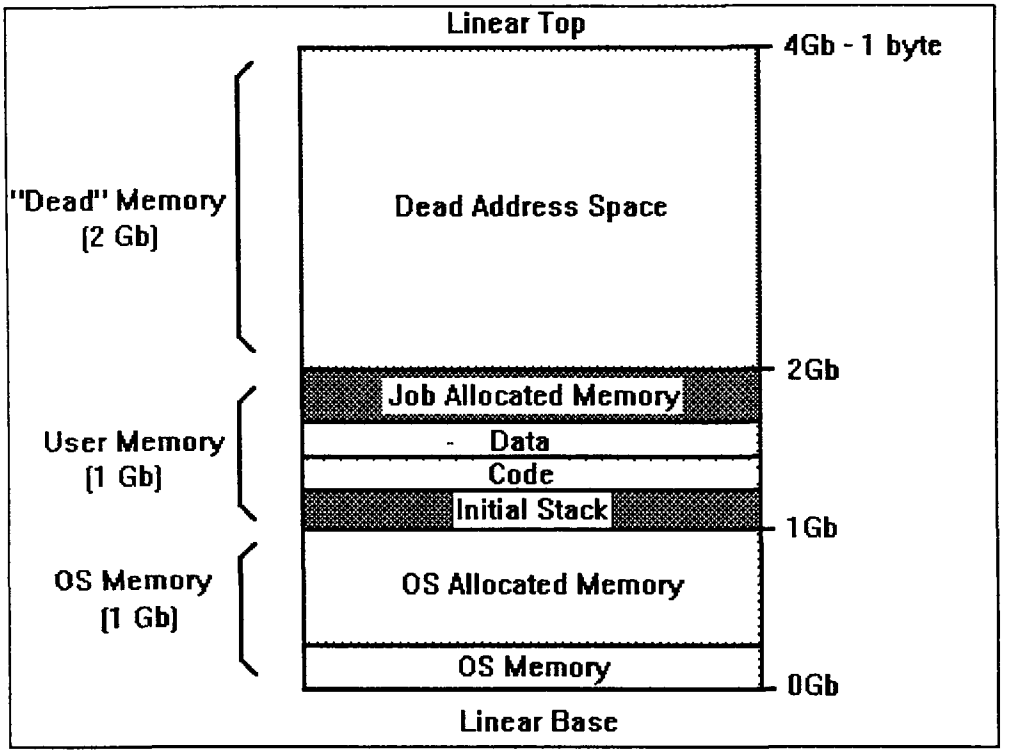


Figure 7 1 The MMURTL memory map

At first appearance, it seems that the upper 2Gb is wasted, however the purpose of this area of memory will soon become clear The problem confronted by MMURTL's designers was as follows Each process refers to its memory space in terms of linear memory Before carrying out its instructions, it is necessary for the operating system to map the memory addresses from the process's linear address space to the processor's physical address space In order to do this, the operating system accesses the page tables for the relevant process and retrieves the physical offset for the page in question

Each linear memory address is 32 bits long The operating system uses the upper ten bits to reference the page directory of the appropriate process in order to retrieve the physical address of the relevant page table (The physical address of the process' page directory is stored by the operating system upon creation of the process and thus may be easily retrieved) The next lower ten bits in the linear address are used as an index into the page table in order to retrieve the physical address of the page being referenced Finally, the remaining twelve bits combined with the address of the page offset combine to produce the relevant physical address

A problem arises when the operating system needs to amend or delete a page table entry The operating system has no special privileges as far as addressing physical memory goes, so it must make use of linear address space in the same way that each application program does So the question becomes, how does the operating system find the linear address of each page table ?

The solution chosen by MMURTL's developers was to make use of only half of each page table to store the physical addresses of the page tables The upper two kilobytes of each page directory is a shadow of the lower two kilobytes The shadow area, contains the linear address of the corresponding physical address from the lower half of the page directory

In this way, Page Directory Entry 512 holds the linear address of the Page Table whose physical address is held in Page Directory Entry 0 This example is shown in Figure 7 2

To ensure that shadow memory is not accidentally appropriated, the relevant Page Directory entries are marked as being non-existent

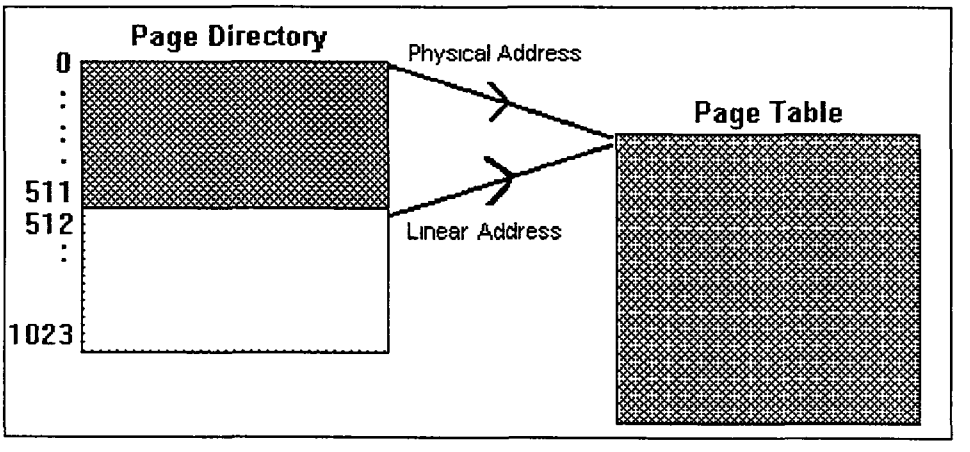


Figure 7 2 Shadow memory in practice

7.3.2 Memory Aliasing

MMURTL allows two processes to share some or all of their memory address space This is most commonly used as a result of interprocess communication such as a Request / Respond transaction between an application and a system service

As with all memory activities, the transaction unit of shared memory is a page In order to share a page, the operating system must copy the relevant Page Table Entry from the Page Table belonging to the process which originally owned the page, to the Page Table belonging to the process which is seeking to sharing the page This is known as Memory Aliasing

The process which is sharing the memory doesn't realise that it is not the page's owner This is due to the fact that the page now appears in its own linear address space An aliased Page Table Entry is marked as such and can't be deallocated or swapped until the alias is dissolved

7.4 Design Decisions

Because the MMURTL memory model relies to such a large extent on Intel processor memory management support, in particular Intel's hardware-based paging, its current implementation does not endear itself easily to the object-oriented paradigm. As a result, there were two possible directions to take in the design of the OO-MMURTL memory model - implementing an object-oriented programming interface, and implementing a complete object-oriented class framework Each of these is described below, before a description of the actual design is given

7.4.1 Object-Oriented Programming Interface

This design option leaves much of the original architecture unchanged The system designer retains the original memory management functionality, however an object-oriented class is introduced which will encapsulate the system in question, i e the memory system, and provide an object-oriented programming interface to be used by the application programmer

The new class will hide much of the underlying low-level functionality from the programmer, making available only the essential high-level calls which are required to manage the memory of a given application

The use of this option provides several advantages to the implementation of OO-MMURTL

- Speed is an essential factor in the design of the memory subsystem of any operating system, particularly a system such as MMURTL which uses a virtual paged memory model Each linear memory address must be checked in a Page Directory, then a Page Table, to retrieve the physical address of the page in question Since the majority of the memory system is still written in assembly language, the speed of the original model is retained
- The object-oriented class serves to hide the low-level code of the memory model. To allocate several pages of memory requires only a single call to the memory manager class Mutual exclusion, searching for a free Page Tables and free Page Table Entries and other necessary actions are performed by the memory manager class
- The memory manager class provides an additional level of protection to the system tables, by ensuring that errant application programs will be unable to gain access to restricted areas of memory

There are however, inherent disadvantages to this type of implementation

- When a complete operating system module, such as the memory subsystem is represented by a single class, it is unlikely that the operating system is using the optimal object-oriented class hierarchy
- The primary role of the memory model class is to provide a seemingly cosmetic application interface

7.4.2 Complete Object-Oriented Class Framework

The second design option is to completely redesign the memory model so that it facilitates a complete object oriented class framework In such a design, each Page Directory would be an instantiation of a CPageDirectory class which encapsulated the behaviour of the directory Similarly each Page Table would be an instantiation of a CPageTable class which encapsulated the behaviour of a Page Table A new class would be introduced to manage the various table objects this class would also be responsible for the allotment of Page Table Entries, and the allocation and deallocation of memory pages

The advantages of this type of implementation are

- The memory subsystem is designed to be a completely object-oriented component of the operating system. As a result, the memory subsystem can reap the advantages of an object-oriented design and implementation
- The memory subsystem may be optimally redesigned from the bottom-up, without being restricted by the previous implementation of the memory subsystem.
- Using a complete object-oriented hierarchy also provides the additional protection and hiding of low-level functionality described among the advantages of providing just an object-oriented programming interface

The main disadvantage of using a complete object-oriented class framework is that the additional overhead can lead to a reduction in processing speed in the operating system.

7.5 OO-MMURTL Memory Subsystem Architecture

OO-MMURTL makes use of the former option, that is, an object-oriented class is designed which provides an application programming interface in order to encapsulate the functionality of the memory subsystem.

Although this may not be the most desirable option in a completely object-oriented operation system, the basic criteria of this thesis is to migrate MMURTL's existing functionality to an object-oriented paradigm. This dictated that a complete redesign of the memory model was not an option

OO-MMURTL introduces a single class, CMemoryManager, whose primary purpose is to provide an object-oriented programming interface to the operating system's underlying low-level memory functionality This class has been designed as generically as possible so that future development and redesign of the memory model can take place

In other words, by accessing CMemoryManager's *AllocPage()*, the operating system currently performs several error checks, before enforcing mutual exclusion on the memory subsystem critical section and calling a kernel function which will perform the actual allocation

In the future, OO-MMURTL's underlying memory model could be redesigned using object-oriented techniques If this happens, although CMemoryManager's implementation will change, its interface should not Its method *AllocPage()* will still allocate memory pages, but by interacting with CPageDirectory and CPageTable classes Programs which currently call

```
MemoryManager->AllocPage( ..)
```

will still continue to do so Other calls will similarly be re-implemented without affecting the interface

Section 7 4 of this chapter suggests a possible class hierarchy which could be introduced in future revisions of OO-MMURTL to fully encapsulate the memory subsystem in an object-oriented framework

7.6 The CMemoryManager Class

The purpose of the CMemoryManager class is to provide a programming interface to the underlying low-level code which controls the management of memory in the operating system. Unlike many of the manager classes in OO-MMURTL, the CMemoryManager class does not own the data it provides access to (i e the Page Directories, Page Tables etc) Instead, it acts as a buffer between application programs, and the kernel functions which access this data

The CMemoryManager class therefore owns just a single attribute, an exchange This is used by the memory manager to provide mutual exclusion to certain kernel functions, in particular those which modify the Page Directory or Page Tables This exchange is initialised in the CMemoryManager constructor

The remaining CMemoryManager methods fall into three categories, each of which is described below

1 Allocation of memory
2 Deallocation of memory
3 Aliasing of memory

7.6.1 Memory Allocation Methods

There are two different types of methods capable of allocating memory pages *AllocOSPage()* and *AllocPage()* Each of these represents one of the types of memory recognised by OO-MMURTL, namely operating system memory and user memory

*char *AllocOSPage(long nPages),*
This method attempts to allocate *nPages* of contiguous operating system linear memory to the user If the operation succeeds, a pointer to the memory is returned If the operation fails, a NULL pointer is returned

AllocOSPage() searches through the page tables for the current process, attempting to find enough contiguous Page Table Entries to satisfy the request If the current Page Table doesn't have enough contiguous entries, additional Page Tables are added to the operating system Page Directory (if possible), until either enough page tables are available to satisfy the request or the request fails The allocation may span several Page Tables The code for *AllocOSPage()* is shown in Figure 7 3

*char *AllocPage(long nPages),*
This method is very similar to the preceding one, with the exception that AllocPage attempts to allocate pages of application memory to the calling process If the operation succeeds, a pointer to the memory is returned If the operation fails, a NULL pointer is returned The code for *AllocPage()* is shown in Figure 7 4

Note that both *AllocPage()* and *AllocOSPage()* make use of CMemoryManager's exchange to ensure mutual exclusion If this is not enforced, it would be possible that while one process is creating additional Page Tables to accommodate a large allocation, another process could step in and inadvertently allocate pages which have been included by the first process in its tally of contiguous pages, thus causing a reduced number of pages to be allocated to the calling process

```
char *CMemoryManager  AllocOSPage(long nPages)
{
  char *RunPages,

// Steps
// ------
// 1) See if we have enough physical memory (check nPagesFree)
// 2) Find a contiguous run of linear pages to allocate (PTEs)
// 3) Allocate each physical page placing it in the run of PTEs


  // Must request > 0 pages for allocation
  if (nPages<=0)
    return NULL,

  // Ensure mutual exclusion
  TPacket *pPacket = MemExch->WaitPacket(),

  // Verify sufficient pages exist to satisfy request
  if (nPages> nPagesFree)
    return NULL;

  // Find contiguous run of OS PTEs in current PT
  RunPages = FindRun(OS_BASE,nPages);

  // Allocate further PTs until request is satisfied
  While(RunPages==NULL) {
    if (AddOSPT()!=0)
      return NULL,
    RunPages = FindRun(OS_BASE,nPages);
    }

  // Mark these pages as allocated
  AddRun(RunPages,nPages);

  // Leave critical section
  MemExch->SendDummyMsg(),

  return RunPages,
}
```

Figure 7 3 The CMemoryManager AllocOSPage() method

7.6.2 Memory Deallocation Methods

A single method is provided to deallocate memory previously allocated by either *AllocOSPage()* and *AllocPage()* This is because in deallocating both types of memory, the same transaction occurs, namely that the relevant Page Table Entries are removed and the relevant bits in the Page Allocation Map are cleared This is performed in the *UnMarkPTEs()* kernel function.

```
char *CMemoryManager· AllocPage(long nPages)
{
  char *RunPages,

// Steps
// ------
// 1) See if we have enough physical memory (check nPagesFree)
// 2) Find a contiguous run of linear pages to allocate (PTEs)
// 3) Allocate each physical page placing it in the run of PTEs


  // Must request > 0 pages for allocation
  if (nPages<=0)
    return NULL,

  // Ensure mutual exclusion
  TPacket *pPacket = MemExch->WaitPacket(),

  // Verify sufficient pages exist to satisfy request
  if (nPages> nPagesFree)
    return NULL,

  // Find contiguous run of user PTEs in current PT
  RunPages = FindRun(USER_BASE,nPages);

  // Allocate further PTs until request is satisfied
  While(RunPages==NULL) {
    if (AddUserPT()!=0)
      return NULL,
    RunPages = FindRun(USER_BASE,nPages);
    }

  // Mark these pages as allocated
  AddRun(RunPages,nPages),

  // Leave critical section
  MemExch->SendDummyMsg(),

  return RunPages,
}
```

Figure 7 4 The CMemoryManager AllocPage() method

int DeAllocPage(unsigned char pOrigMem, int nPages),

Before mutual exclusion is enforced, DeAllocPage() drops unnecessary bits from the linear address of the memory to be deallocated Only the upper twenty bits are required, since memory is deallocated in blocks of Pages, four kilobytes in size Mutual exclusion is enforced around the critical section of the call to the kernel function *UnMarkPTEs()*

```
void CMemoryManager  DeAllocPage(unsigned char pOrigMem,
                                 int nPages)
{
  int ProcNum,

  // Discard unnecessary bits
  pOrigMem = (pOrigMem % 4096),

  ProcNum = GetCrntProcNum(),

  // Enforce mutual exclustion
  TPacket *pPacket = MemExch->WaitPacket(),

  UnMarkPTEs(ProcNum,pOrigMem,nPages),

  // Leave critical section
  MemExch->SendDummyPacket(),
}
```

Figure 7 5 The CMemoryManager DeAllocPage() method

7.6.3 Memory Aliasing Methods

There are two co-operating methods dealing with memory aliasing One to perform the aliasing, the other to perform the dealiasing These are described below

*unsigned char *AliasMem(unsigned char *pMem, unsigned long dcbMem, unsigned long dJobNum)*

This method aliases pages in the current process' Page Directory / Page Tables, providing its Page Directory is different to the Page Directory belonging to the job specified in the parameter *dJobNum* In such cases no aliasing is needed as both processes belong to the one program, and are therefore operating in the same linear address space

Note that even if the length of the memory to be aliased is only two bytes, if it crosses page boundaries, both pages must be aliased The code for *AliasMem()* is shown in Figure 7 6

```
unsigned char *CMemoryManager  AliasMem(unsigned char *pMem,
                unsigned long dcbMem, unsigned long dProcNum)
{
  char *RunPages,
  unsigned int base,
  unsigned long CurrProc = GetCurrProcNum(),

  // Check Page Directories (ie Processes)
  if (CurrProc==dProcNum) return ,

  // Ensure mutual exclusion
  TPacket *pPacket = MemExch->WaitPacket(),

  // Calculate number of pages required
  pMem = pMem % 4096,
  dcbMem += pMem,
  unsigned long nPages = GetAliasReqSize(pMem,dcbMem),

  if (CurrProc == OP_SYSTEM) base = OS_BASE;
  else base = USER_BASE,

  // Find contiguous run of PTEs in current PT
  RunPages = FindRun(base,nPages),

  // Allocate further PTs until request is satisfied
  while (RunPages==NULL) {
    if (CurrProc == OP_SYSTEM)
      if (AddOSPT()==NULL) return;
    else
      if (AddUserPT()==NULL) return,
    RunPages = FindRun(base,nPages);
    }
  // Perform aliasing
  AddAliasRun(pMem, RunPages, nPages, dProcNum);

  // Leave critical section
  MemExch->SendDummyMsg(),
  return RunPages,
}
```

Figure 7 6 The CMemoryManager AliasMem() method

There are essentially four steps to the *AliasMem()* method

1 Check to see if the current Page Directory is the same as the Page Directory for the specified process In such a case aliasing is not required This is analogous to checking to see if the current process is the same as the specified process

2 The number of pages which need to be aliased is calculated

3 A check is made to see if there are sufficient Page Table Entries to satisfy the request If not, further Page Tables are allocated until either the request is satisfied or else no more Page Tables are available, in which case the alias fails

4 The Page Table Entries are aliased

*unsigned int DeAliasMem(unsigned char *pAliasMem, unsigned long dcbAliasMem, unsigned long JobNum)*

This method dealiases memory which was previously aliased using the *AliasMem()* method *pAliasMem* is the linear address which was previously passed as a parameter to the *AliasMem()* method There is no need to use a semaphore since Page Tables are being dealiased one at a time This would not interfere with any concurrent memory allocation routines

The behaviour of *DeAliasMem()* is much simpler than its companion AliasMem() It consists of only two steps

1 Calculate number of pages which need to be dealiased
2 Call the system kernel *RemoveAliasRun()* to dealias the necessary Page Table Entries

The code for *DeAliasMem()* is shown in Figure 7 7

```
unsigned int CMemoryManager· DeAliasMem(unsigned char *pAliasMem,
                                        unsigned long dcbAliasMem)
{
  // Calculate number of pages to be dealiased
  unsigned long nPages = GetAliasReqSize(pAliasMem,dcbAliasMem);

  // Retrieve id number of current process
  unsigned long CurrProc = GetCurrProcNum(),

  // Perform call to kernel function to perform dealiasing
  return RemoveAliasRun(pAliasMem,nPages,CurrProc),
}
```

Figure 7 7 The CMemoryManager DeAliasMem() method

7.7 Conclusions

This chapter described in detail the design of the OO-MMURTL memory model. The design served to highlight a significant problem in the migration of MMURTL to OO-MMURTL, namely that some components of the original MMURTL operating system cannot be easily encapsulated by an object-oriented framework The cause of this is due to either the original design, and possibly the original implementation, of the component in question

Instead of providing a complete class framework which would have meant redesigning the entire memory model, OO-MMURTL encapsulates the interface between user applications and MMURTL's original memory model. This still provided some advantages over the original model, but it is not an optimal design when the capabilities of the object-oriented paradigm are taken into account Future development should involve a complete redesign of the OO-MMURTL memory subsystem so that the entire subsystem is modelled as a set of object-oriented classes

Chapter 8

Additional Classes

8.1 Overview

This chapter introduces three additional classes in OO-MMURTL These classes are responsible for encapsulating the behaviour of the Ready Queue, system interrupts, and the system timer, which itself is an interrupt

The Ready Queue
As described in previous chapters, MMURTL uses a prioritised queue to manage the execution of tasks Task switching occurs only between processes of the same priority In fact, MMURTL implements the Ready Queue as an array of thirty-two queue structures Each of these structures holds a queue of the running tasks of a single priority In this way, each of the thirty-two queues represents each of the thirty-two possible priorities in the operating system.

This results in a small overhead in terms of storing the Ready Queue, however this is acceptable when the reduction in the duration of a task switch is considered Whenever a timer interrupt occurs, notifying the operating system that the time slice of the current executing task has elapsed, the operating system can go directly to the relevant queue without having to evaluate tasks of a differing priority

Interrupts
The Intel processor design allows two types of interrupt - one which executes as an independent task, or one which executes within the context of the task that is interrupted MMURTL uses the latter due to its speed, the former requiring a context switch each time an interrupt occurs and each time an interrupt completes The execution of interrupts in the context of the interrupted task is easily facilitated by the MMURTL architecture which dictates that the operating system is mapped into the lower gigabyte of linear memory of every process

Timer Interrupts

As with most time-slicing operating systems, the timer interrupt plays a special role in MMURTL. Every ten milliseconds, the timer interrupt fires. When a task is running for thirty milliseconds, the Ready Queue is re-evaluated to check if another task of the same or higher priority is waiting to run. If so, the current task is placed on the Ready Queue and the next task is given control.

8.2 The CReadyQueue Class

This class encapsulates the operating system's Ready Queue mechanism. It is an entirely abstract class, however. The purpose of the CReadyQueue class is to define a template for future implementations of ready queues in OO-MMURTL. To this end, CReadyQueue defines four basic methods which may be performed on all ready queues, regardless of their implementation. They are:

- *void enQueueRdy(TTSS * pTSS)*
  This method adds a task to the Ready Queue.

- *TTSS *deQueueRdy()*
  This method removes a task from the Ready Queue.

- *TTSS *ChkRdyQ()*
  This method returns the task at the top of the Ready Queue, without removing it.

- *void RemoveRdyProc(CProcess *pProc)*
  This method removes all tasks belonging to a given process from the ready queue.

OO-MMURTL provides a single implementation of the Ready Queue, CPrioritisedReadyQueue, which mimics the original thirty-two queue implementation of MMURTL. The importance of the role of object-oriented techniques in OO-MMURTL's CReadyQueue class framework lies in the ability of the system designer to completely redesign the implementation of the Ready Queue, without altering the public interface of the class. Figure 8.1 presents a simple class hierarchy of the Ready Queue framework as it stands in OO-MMURTL, along with another possible different implementation CFIFOReadyQueue, which is discussed later.

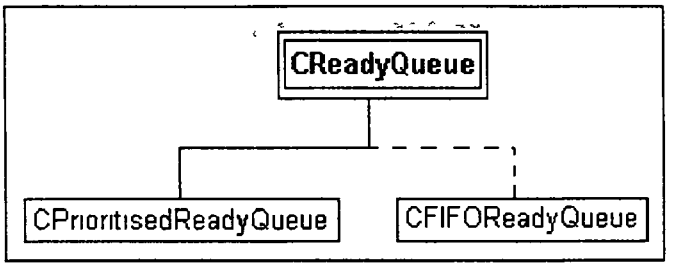


Figure 8 1 A possible CReadyQueue framework

Note that the dashed line in the above diagram represents the possible introduction of the CFIFOReadyQueue class Although the implementation of this class is discussed below, it is not currently implemented in OO-MMURTL Its use in the class hierarchy above is merely to show further possible implementations of the CReadyQueue

8.3 The CPrioritisedReadyQueue Class

CPrioritisedReadyQueue is a concrete subclass of the abstract CReadyQueue class It overrides CReadyQueue's four abstract methods with its own methods which duplicate the functionality of MMURTL's original Ready Queue implementation

CPrioritisedReadyQueue has a single attribute, an array of thirty-two queues, each of which contains a head and a tail pointer to a task structure Each of these task structures contains a pointer to another task structure, and it is through these links that the queue is threaded

The implementation of each of the CPrioritisedReadyQueue methods is described below

*void enQueueRdy(TTSS *pTSS)*
The *enQueueRdy()* method retrieves the priority of the currently running process which owns the task in question This priority represents the index of the prioritised queue to which the task should be added The remaining step is to adjust the relevant queue's pointers so that the task structure is added to its tail The code for this method is shown in Figure 8 2

Note that the actual task switching does not occur in any descendants of the CReadyQueue class Their sole responsibility is the management of the Ready Queue

```
void CPrioritisedReadyQueue::enQueueRdy(TTSS *pTSS)
{
  // Verify the task structure pointer is valid
  if (pTSS==NULL)
    return;

  // Retrieve the priority of the current process
  unsigned long dPriority = GetCurrentPriority();

  pTSS->next = NULL;

  // If the queue is empty, this TSS becomes head and tail
  if (ReadyQ[dPriority].Head == NULL) {
    ReadyQ[dPriority].Head = pTSS;
    ReadyQ[dPriority].Tail = pTSS;
    }
  else { // Otherwise adjust the tail of the queue
    (ReadyQ[dPriority].Tail)->next = pTSS;
    ReadyQ[dPriority].Tail = pTSS;
    }
}
```

Figure 8.2. The CPrioritisedReadyQueue::enQueueRdy() method

*TTSS *deQueueRdy()*

This method must find, remove and return the highest priority task on the Ready Queue. In terms of the implementation of the CPrioritisedReadyQueue class, this involves going to the highest priority queue, and removing the task at its head. If this queue is empty, then *deQueueRdy()* goes to the queue with the next lowest priority, and checks to see if there are any tasks waiting with that priority. This continues until a task is found on one of the queues or else all of the ready queues are found to be empty.

Upon finding the highest priority task, this method returns a pointer to the task structure to the calling program. If no tasks are found on the Ready Queue, a NULL pointer is returned.

This method demonstrates one of the disadvantages of the prioritised queue implementation of the ReadyQueue as used by MMURTL. Few tasks actually execute at the highest priority. This means that almost all of the calls to the *deQueueRdy()* method result in stepping through each of the queues looking for a waiting task. The further down the array of queues the highest priority task exists, the bigger the delay before a task-switch is made.

In the case of using a FIFO implementation as shown later, the *deQueueRdy()* method need only check a single queue to find the highest priority task This results in a faster execution, which is of importance particularly during the course of a context switch MMURTL sacrifices this speed in return for providing a more intelligent ready queue The code for the *deQueueRdy()* method is shown in Figure 8 3

```
TTSS *CPrioritisedReadyQueue. deQueueRdy()
{
  TTSS *cHighPri,

  // Loop through all of the queues, starting with the
  // queue with the highest priority, until a waiting
  // task is found. When one is found, adjust the queue
  // and return a pointer to it
  for (int i=0, i<PRIORITYCOUNT, i++) {
    if (ReadyQ[i] != NULL) {
      pHighPri = ReadyQ[i] Head,
      ReadyQ[i].Head = (ReadyQ[i] Head)->next;
      return pHighPri,
      }
    }

  // If no waiting task is found, a NULL pointer is returned
  return NULL,
}
```

Figure 8 3 The CPrioritisedReadyQueue deQueueRdy() method

*TTSS *ChkRdyQ()*

This purpose of this method is related to that of the *deQueueRdy()* method which is described above *ChkRdyQ()* also attempts to retrieve the highest priority task on the Ready Queue, but instead of removing it once it has been found, *ChkRdyQ()* leaves the highest priority task where it is As in *deQueueRdy()*, *ChkRdyQ()* returns a pointer to the highest priority task it finds, or a NULL pointer if none were found

Because of the similarities between the behaviour of the two methods, the operation of *ChkRdyQ()* is almost identical to *deQueueRdy()* It first goes to the highest priority queue, and attempts to remove the task at its head If there is no task there, then *ChkRdyQ()* goes to the queue with the next lowest priority, and checks to see if there are any tasks waiting there This continues until a task is found on one of the queues or else all of the ready queues are found to be empty

The code for this method is shown in Figure 8 4

```
TTSS *CPrioritisedReadyQueue::ChkRdyQ()
{
  TTSS *cHighPri;

  // Loop through all of the queues, starting with the
  // queue with the highest priority, until a waiting
  // task is found
  for (int i=0; i<PRIORITYCOUNT; i++) {
    if (ReadyQ[i].Head != NULL) {
      pHighPri = ReadyQ[i].Head;
      return pHighPri;
      }
    }
  // If no waiting task is found, a NULL pointer is returned
  return NULL;
}
```

Figure 8.4. The CPrioritisedReadyQueue::ChkRdyQ() method

*void RemoveRdyProc(CProcess *pProcess)*

The purpose of this method is to remove all of the tasks belonging to a given process from the Ready Queue. In terms of the prioritised Ready Queue implementation of OO-MMURTL, this involves removing all of the process' tasks on all of the queues which comprise the Ready Queue. This method is called from the methods *Chain()* and *ExitJob()* belonging to the *CProcess* class (see Chapter 5 : The Process Management Model). It is used to remove any remaining tasks belonging to a process which is about to be exited.

The *RemoveRdyProc()* method begins at the queue with the highest priority and loops through every queue. This is necessary since each process may have multiple tasks, possibly with the same priority. For this reason every node on every queue must be checked. Even if a task belonging to the current process is found on a given queue, the remainder of the nodes must be checked.

Providing the head of each queue is not NULL, *RemoveRdyProc()* checks to see if that node belongs to the parameter *pProcess*. If it does, the task in question is removed. The next node in the queue is then evaluated and removed if it belongs to the process *pProcess*. This continues until the end of the queue, when a NULL pointer is reached.

When a task is removed from one of the queues, the queue's pointers must be updated, the task structure must be added to the system heap of free task structures, and the system statistics must be updated to reflect the freeing of the task.

```
void CPrioritisedReadyQueue  RemoveRdyProc(CProcess *pProcess)
{
  TTSS *pTSS,*pQueue,*pQueuePrev,

  // pQueuePrev is used to point to the node traversed
  // immediately prior to the current one  It is
  // required when a task is deleted from the middle of
  // a queue
  pQueuePrev = NULL,

  // Check every queue
  for (int i=0, i<PRIORITYCOUNT, i++) {
    pQueue = ReadyQ[i] Head,

    // If the queue is not empty, check every task within
    // the queue
    while(pQueue'=NULL) {

      // If the the current node represents a task which
      // belongs to the process pProcess, it must be
      // removed from the Ready Queue and its former links
      // must be updated
      if (pQueue->pProc==pProcess) {
        pTSS = pQueue,

        if(pPrevQueue==NULL)
          ReadyQueue[i].Head = pQueue->next;
        else
          pPrevQueue->next = pQueue->next,

        // The task structure is replaced on the heap of free
        // task structures and the system statistics are
        // updated
        pTSS->next = pFreeTSS,
        pFreeTSS = pTSS,
        pTSS->pProc = NULL,
        nTSSLeft++,
        }

      pPrevQueue = pQueue,
      pQueue = pQueue->next,
      }
  }
}
```

Figure 8 5 The CPrioritisedReadyQueue RemoveRdyProc() method

The code for the *RemoveRdyProc()* method is shown in Figure 8 5

8.4 The CFIFOReadyQueue Class

In this section the entire code for a different implementation of the Ready Queue is presented This class is not implemented in OO-MMURTL, however it does show how a different subclass of the abstract CReadyQueue class can be implemented resulting in a different Ready Queue mechanism. This serves to highlight the flexibility of the Ready Queue object-oriented framework and its capacity for future development The complete code for this class is shown in Figures 8 6a and 8 6b

```
CFIFOReadyQueue. CFIFOReadyQueue()
{
    ReadyQ.Head = NULL,
    ReadyQ.Tail = NULL,
}

void CFIFOReadyQueue .enQueueRdy(TTSS *pTSS)
{
  if (pTSS==NULL)
    return,
  pTSS->next = NULL,

  if (ReadyQ Head == NULL) {
     ReadyQ Head = pTSS;
     ReadyQ Tail = pTSS,
     }
  else {
     (ReadyQ Tail)->next = pTSS,
     ReadyQ Tail = pTSS,
     }
}

TTSS *CFIFOReadyQueue. deQueueRdy()
{
  TTSS *pHighPri,

  if (ReadyQ[i] Head != NULL) {
    pHighPri = ReadyQ[i] Head,
    ReadyQ[i] Head = (ReadyQ[i] Head)->next,
    return pHighPri,
    }

   return NULL,
}

TTSS *CFIFOReadyQueue .ChkRdyQ()
{
  if (ReadyQ.Head != NULL) return ReadyQ.Head,
  return NULL,
}
```

Figure 8 6a The CFIFOReadyQueue class implementation

```
void CFIFOReadyQueue  RemoveRdyProc(CProcess *pProcess)
{
  TTSS *pTSS,*pQueue,*pQueuePrev,

  pQueuePrev = NULL,

  pQueue = ReadyQ[1] Head,

  while(pQueue!=NULL) {

    if (pQueue->pProc==pProcess) {
      pTSS = pQueue,

      if(pPrevQueue==NULL)
        ReadyQueue Head = pQueue->next,
      else
        pPrevQueue->next = pQueue->next,

      pTSS->next = pFreeTSS,
      pFreeTSS = pTSS,
      pTSS->pProc = NULL,
      nTSSLeft--,
      }

    pPrevQueue = pQueue,
    pQueue = pQueue->next,
    }
}
```

Figure 8 6b The CFIFOReadyQueue class implementation

8.5 The CInterrupt Class

The CInterrupt class is a simple encapsulation of the basic functionality required to write an interrupt service routine in OO-MMURTL CInterrupt is an abstract class which performs the default behaviour of an interrupt To create a new interrupt, a programmer need only create a subclass of the CInterrupt class, overriding the *Service()* method with the appropriate interrupt service routine

This greatly simplifies the writing of an interrupt service routine, which now becomes a two-step process

1 Create a subclass of the CInterrupt class, overriding the abstract *Service()* method with the interrupt service code
2 Create an instance of the class

When the new class is instantiated, the operating system will be notified and will insert an interrupt vector into the operating system's Interrupt Descriptor Table

This approach to writing interrupt service routines has several advantages

- The number of steps to creating an ISR is reduced, thus simplifying the creation process
- Previously it was possible for an application to place an errant vector address into the Interrupt Descriptor Table The effect of this was to cause a system crash as soon as the interrupt was first called Now, unless a valid *Service()* method is provided to create a concrete subclass of CInterrupt, the compiler will not allow instantiations of the subclass (because without overriding the *Service()* method they remain abstract classes)
- Another frequent error which previously occurred was the accidental omission by the interrupt service routine programmer to call the EndofIRQ() kernel function, resulting in further system misbehaviour The new CInterrupt class includes this behaviour by default, removing the onus, and therefore the opportunity to make mistakes, from the programmer

The CInterrupt class presented here is simple in idea and implementation, however it still provides several advantages over the non object-oriented implementation In addition, further subclassing of the CInterrupt class will provide the system designer with a more specialised interrupt mechanism. Each of CInterrupts methods are described below

CInterrupt()

CInterrupt's constructor serves two purposes Firstly it assigns the IRQ number to the class attribute *IRQNum* In addition, the Constructor calls the kernel primitive *SetIRQVector()* This creates a vector to represent the new interrupt and adds the vector to the Interrupt Descriptor Table The *SetIRQVector()* call requires as a parameter the address of the routine which will service the interrupt This routine *ISR()*, which is described below, is responsible for ensuring that the users interrupt service routine is called The *CInterrupt* code is shown in Figure 8 7

Figure 8 7 The CInterrupt constructor

```
CInterrupt  CInterrupt(unsigned long Num)
{
  IRQNum = Num,
  ..SetIRQVector(IRQNum,&ISR),
}
```

void interrupt ISR()

The address of this method is placed into the Interrupt Descriptor Table when this interrupt object is first invoked. It serves two purposes. Firstly it must call the abstract method *Service()* which must be written by the user to ensure that the interrupt subclass is a concrete one. Secondly, the kernel primitive *EndOfIRQ()* is called, which notifies the Programmable Interrupt Controller Units that the interrupt service routine has completed.

```
void interrupt CInterrupt::ISR()
{
  if (Service())
      EndOfIRQ(IRQNum);
}
```

Figure 8.8. The CInterrupt::ISR() method

*void GetIRQVector(char *pVectorRet)*

This method calls the kernel primitive *GetIRQVector()*. This returns the address of the interrupt service routine which is currently serving the interrupt represented by this class. Note that this is a 32-bit offset address in the operating system address space. The code for this method is shown in Figure 8.9.

void MaskIRQ()

This method calls the kernel primitive *MaskIRQ()*. This masks the hardware interrupt request represented by this class. Once masked, the CPU will not be interrupted by this interrupt, even if interrupts are enabled. The code for this method is shown in Figure 8.9.

void UnMaskIRQ()

This method reverses the effect of the *MaskIRQ()* method. Interrupts representing this class will now be serviced again by the operating system, unless interrupts as a whole are disabled. The code for this method is shown in Figure 8.9.

```
void CInterrupt::GetIRQVector(char *pVectorRet)
{
  ::GetIRQVector(IRQNum, pVictorRet);
}

void CInterrupt::MaskIRQ()
{
  ::MaskIRQ(IRQNum);
}

void CInterrupt::UnMaskIRQ()
{
  ::UnMaskIRQ(IRQNum);
}
```

Figure 8.9. Additional CInterrupt methods

8.6 The CTimer Class

The timer plays an important role in most operating systems, particularly those which operate using some form of time slicing technique. In MMURTL the timer class fires every 10 milliseconds. It is the responsibility of the timer to check if any alarms must be triggered, or if the current task has been running for 30ms or more. If so, the Ready Queue is checked to see if a task of equal or higher priority is awaiting execution.

Because the timer is a form of interrupt, we must derive a concrete subclass from the abstract CInterrupt class. The new class, CTimer, must provide a *Service()* method in order that timer interrupts will be serviced and that the class will become a concrete one.

The CTimer class makes thirty-two timer blocks available. Each of these blocks may be used to trigger an alarm. A timer block consists of a flag indicating whether the block is in use or not, a pointer to an exchange, and a count indicating how many timer ticks remain before the alarm will trigger. A block is created by calling the method *Alarm()*, which takes two parameters. Firstly an exchange must be provided at which the CTimer class will send a notification when the alarm is triggered. The exact time at which the alarm is triggered is decided by the second parameter. Alarms can be removed using the *KillAlarm()* method or temporarily postponed using the *Sleep()* method. The *MicroDelay()* method provides the facility for small-value timing delays in increments of 15 microseconds. Each of CTimer's methods are described below in detail.

CTimer(unsigned long Num, unsigned int TickCount)

The CTimer constructor is responsible for two things. Firstly, it must immediately call the CInterrupt constructor, which sets up the timer as a valid interrupt in the Interrupt Descriptor Table. Secondly, the *TimerTick* attribute is initialised. This measures the number of timer ticks which have elapsed since the timer was created. The code for this method is shown in Figure 8.10.

```
CTimer::CTimer(unsigned long Num) : CInterrupt(Num)
{
  TimerTick = 0;  nTmrBlocksUsed = 0;
}
```

Figure 8.10. The CTimer constructor

virtual unsigned int Service()

This method is called by the CInterrupt interrupt method *ISR()*. It is responsible for servicing the timer interrupt. If no timer blocks have been created then the sole action the timer interrupt need take is to increment the count of ticks which have elapsed since the system started. If this is the case, *Service()* returns the value *ENDOFIRQ_NOTPERFORMED*. This tells the IRQ() method to perform the *EndOfIRQ()* method itself. This is the default behaviour for most interrupts.

If, however, there are timer blocks set up, *Service()* must attend to them. Before it does this, *Service()* calls *MaskIRQ()* to ensure that no further timer interrupts occur. Next it calls *EndOfIRQ()*, before setting the interrupt flag. The net effect of this is to allow other (non-timer) interrupts to perform an interrupt during the servicing of the timer.

The CTimer class only retains a single variable, *nBlocksUsed*, to indicate whether any of the blocks are being used or not. If the value of this variable is above zero, the *Service()* method must check the *fInUse* flag of each of the thirty-two timer blocks in the timer array in order to discover which of the timer blocks are in use.

If a timer block is in use one of two things will happen. If the value of the timer block's *Tick* variable has reached zero, the timer has elapsed and the process which set up the timer must be notified. This is done by sending a dummy message to the exchange pointed to by the timer block. If the *Tick* count has not reached zero, it is reduced by one. Finally, the *Service()* method clears the interrupt flag before unmasking its interrupt. The code for this method is shown in Figure 8.11.

```
void CTimer::Service()
{
  // Increase the count of elapsed ticks since the
  // timer was created
  TimerTick++;

  // Return if no timer blocks are in use, reporting
  // that the EndOfIRQ() primitive was not called
  if (nBlocksUsed==0) return ENDOFIRQ_NOTPERFORMED;

  // Enable all interrupts except the Timer interrupt
  MaskIRQ();
  EndOfIRQ();

  #asm
  STI
  #endasm

  // Check every timer block, notifying the appropriate
  // processes through the use of an exchange if an
  // alarm has elapsed, otherwise decrement the timer
  // block tick
  for(int i=0; i<nTMRBLKS; i++) {
    if (TmrBlks[i].fInUse) {
      if (TmrBlks[i].Tick == 0) {
        (TmrBlks[i].RespondExch)->ISendDummyPacket();
        TmrBlks[i].fInUse = FALSE;
        }
      else
        TmrBlks[i].Tick--;
      }
    }

  #asm
  CLI
  #endasm

  // Reenable timer interrupts
  UnMaskIRQ();

  // Signal that primitive has already been performed
  return ENDOFIRQ_PERFORMED;
}
```

Figure 8.11. The CTimer::Service() method

void Sleep(unsigned long nDelay)

This method results in the suspension of the process which called it for *nDelay* ticks of the timer. Having verified that the required delay is greater than zero, *Sleep()* searches the timer block array searching for a free array. If none are found, the method returns, reporting a failure to perform the delay.

If an empty timer block is found, all interrupts are disabled while its variables are set. The *fInUse* flag is set to true, and its *Tick* is set to the value of *nDelay*. The exchange used by the timer block is the calling task's default exchange which is held in its TTSS structure. Next, the number of timer blocks in use is incremented before interrupts are reenabled. Finally the task waits for a message to arrive at its exchange before continuing. This message is a dummy message which will be sent by the CTimer object's *Service()* method when the task's timer block ticks have been reduced to zero. Once the current task receives the dummy message, the *nDelay* delay will have elapsed and it will continue processing. The code for *Sleep()* is shown in Figure 8.12.

```
unsigned int CTimer::Sleep(unsigned long Delay)
{
  if (Delay==0)
    return FAIL;

  // Find an empty timer block
  int i=0;
  while ((i<nTMRBLKS)&&(TmrBlks[i].fInUse))
    i++;

  // Return if there are no free timer blocks
  if (i>=nTMRBLKS)
    return FAIL;

  #asm
  CLI
  #endasm

  // Setup the timer block structure, using the tasks
  // default exchange as the timer block's exchange
  TmrBlks[i].Tick = AlarmDelay;
  TmrBlks[i].fInUse = TRUE;
  nTmrBlksUsed++;
  TmrBlks[i].RespondExch = GetTSSExch();

  #asm
  STI
  #endasm

  // Wait here until the dummy message is received from
  // Service, notifying that the delay has elapsed
  TLinkBlock *lb = (TmrBlks[i].RespondExch)->WaitMsg();

  // Continue processing once the delay has elapsed
  return SUCCESS;
}
```

Figure 8.12. The CTimer::Sleep() method

*void Alarm(CExchange *AlarmExch, unsigned long AlarmDelay)*

Along with the *Delay()* method, This is the only other method which results in a timer block being set up However, *Alarm()* differs in that it allows the user to specify an exchange to which the dummy message will be sent as soon as the alarm is triggered An example of the advantage of this is exemplified in the implementation of a hardware controller which must detect if a given hardware event has occurred in a specified amount of time The controller can set up an alarm to be sent to the same exchange as used by the hardware device itself Whichever message comes first, the alarm or the message from the hardware, will show whether the action was carried out successfully or whether a time-out error has occurred

Another difference between the two methods is that the *Alarm()* method will not wait until the alarm is triggered As a result of these similarities, the code for the *Alarm()* method is very similar to that of *Delay()*, the main difference being that *Alarm()* returns to the calling task as soon as the timer block has been set up The code for this method is shown in Figure 8 13

```
unsigned int CTimer. Alarm(CExchange *AlarmExch,
                           unsigned long AlarmDelay)
{
  if (AlarmDelay==0) return FAIL;

  // Find an empty timer block
  int i=0,
  while ((i<nTMRBLKS)&&(TmrBlks[i] fInUse))
    i++,

  // Return if there are no free timer blocks
  if (i>=nTMRBLKS) return FAIL;

  #asm
  CLI
  #endasm

  // Setup the timer block structure, using the tasks
  // default exchange as the timer block's exchange
  TmrBlks[i] Tick = Delay,
  TmrBlks[i] fInUse = TRUE,
  nTmrBlksUsed++,
  TmrBlks[i] RespondExch = AlarmExch,

  #asm
  STI
  #endasm

  return SUCCESS,
}
```

Figure 8 13 The CTimer Alarm() method

void MicroDelay(unsigned long dDelay)
This method is used for creating very brief delays, in multiples of fifteen microseconds The timing for this delay is based on the toggle of the refresh bit from the System Status port The toggle is approximately fifteen microseconds This means this call will not be very accurate for values less than three or four *dDelay* units

unsigned long GetTick()
This method returns the number of timer ticks which have occurred since the timer was created (at system boot time)

8.7 Conclusions

This chapter introduced additional object-oriented classes which are required by the main components of OO-MMURTL The CReadyQueue is an abstract class which provides a template for implementations of the ready queue Its design makes no presumptions about the scheduling policy of the operating system. This allows system developers to easily derive a concrete subclass of CReadyQueue which implements specific scheduling policies

The CInterrupt class provides a simple mechanism for programmers to set up a hardware interrupt in the operating system. To create an interrupt, a programmer need only create a subclass of CInterrupt, providing a single method to service the interrupt, and then instantiate the object to incorporate the interrupt into the operating system. This greatly simplifies the steps previously required by MMURTL

Finally, the CTimer class, which is a concrete subclass of CInterrupt, was described This provides a flexible timer and alarm mechanism which can be easily manipulated by the application programmer

Chapter 9

Design Testing

9.1 Overview

This chapter briefly describes my experiences in designing and testing OO-MMURTL, As an introduction, the steps taken by MMURTL's original developers are discussed This is followed by a description of the problems I encountered and the solutions I implemented

In addition, an overview is presented of the OO-MMURTL Simulator This is an MS-DOS based program which I developed to provide further testing capabilities of the OO-MMURTL source code which is presented in this thesis Finally, a brief description of the shortcomings of the testing which I have conducted is presented, along with proposed solutions to overcome these shortcomings

9.2 Development of MMURTL

As with any other operating system, MMURTL required a suitable programming environment from an early stage Initially the designers sought to use tools written by third-party developers However as MMURTL evolved, the need for a custom made set of tools became apparent

During the initial development of MMURTL, the Microsoft Assembler (MASM) v5 1 was used Subsequently the team changed to Borland's Turbo Assembler (TASM), before the need for a custom assembler became necessary The developers designed and implemented a new assembler, DASM, which was custom written for the MMURTL operating system.

The next logical step was to write a high level language which could produce assembly compatible with the DASM assembler MMURTL's designers selected the C language as the most appropriate and wrote a MMURTL-specific implementation of the language which they called C Minus 32 This 32-bit compiler does not support the entire set of features of the C language (hence the "minus"), but those that are supported are ANSI-C compliant The assembly language code produced by C Minus 32 is compatible with both DASM and TASM

9.3 Development of OO-MMURTL

The intention of my research was to suggest an appropriate design for an object-oriented version of the MMURTL operating system. This did not include an implementation of the design, which would necessitate a research project in itself. However, during the course of my research it was desirable to validate my proposed implementation of the OO-MMURTL class frameworks which are presented in this thesis

The primary purpose of this section is to describe the approach I took in testing my proposed object-oriented designs and frameworks Due to the low-level nature of much of the functionality of OO-MMURTL, and also due to the movement from a conventional to an object-oriented paradigm, the testing and debugging of the code was a non-trivial task This section will document the problems which I encountered during the testing and debugging phase, along with the solutions which I developed to overcome them. In addition, this section provides a summary of the outstanding testing and debugging issues which must be dealt with at a later stage of OO-MMURTL's implementation

There are two aspects to the testing phase of the OO-MMURTL operating system proposed in this thesis Firstly, there is the unit testing of the code underlying each distinct framework, for example the Messaging Model, or the Process Management Model. Secondly, a DOS-based program was written which would act as a simulator of the OO-MMURTL microkernel, allowing further testing of the behaviour and interactions between the various frameworks to take place Each of these are described below in further detail.

9.4 Testing the OO-MMURTL Class Frameworks

The first and most obvious problem which presented itself was the lack of a C++ compiler The only high level language which is currently supported by the MMURTL development kit is C This problem was further compounded by the fact that MMURTL's C compiler, C Minus 32, was released as freeware, not public domain This resulted in a decision by the MMURTL's designers not to distribute the compiler's sources

In late 1995 the source code of the latest version of C Minus 32 was made available on the CD-ROM which accompanies Richard Burgess' book *Developing Your Own 32-Bit Operating System* [Burgess 95] However, these sources arrived too late to have a reasonable impact on my research

Since the possible use or adaptation of C Minus 32 proved to be unfeasible, due to the lack of source code availability the only remaining choice was to make use of a third-party C++ compiler, as MMURTL's designers had originally done In doing this, I chose to use Borland's C++ compiler v4 0 The main reason for selecting this compiler was so that I could remain as faithful as possible to the current MMURTL development process Similarly to C Minus 32, Borland C++ v4 0 compiles TASM compatible assembly language

Given this solution, it became possible to compile both the C++ code and the assembly code suggested in my research Although the resulting output is neither MMURTL specific nor compatible, the solution allowed me to fulfil the goal of verifying the syntactic and semantic structure of OO-MMURTL's class designs It was worth noting that additional work had to be performed in order to overcome some of the incompatibilities between the C Minus 32 and Borland's C++ compiler For example there are minor syntactic differences between the way assembly code may be integrated into programs by both compilers

This, however, has only guaranteed semantic and syntactic correctness of the class frameworks The behavioural correctness of the classes has yet to be verified This proves to be a difficult task, given that the executable code provided by Borland C++ was incompatible with the C Minus 32 executables, and thus could not be integrated with the rest of the operating system.

In order to provide a compromised solution to this problem, I attempted to simulate the behaviour of the classes by integrating them with an existing MMURTL program. I achieved this by redesigning the program, MMURTL's user-interface module, using the new class frameworks However, in order to execute the program I replaced the object-oriented method invocations with non-object-oriented function calls, creating global variables to replace the object's public and private variables Although this proved to be an incomplete and crude test, it did serve to show that it remained possible to integrate the new frameworks with an existing MMURTL program.

In summary, I have attempted to verify the syntactic, semantic, and behavioural correctness of the proposed OO-MMURTL class frameworks While I succeeded in asserting the first two, a full investigation of the behavioural correctness of the frameworks will not be possible until an executable implementation, integrated with the rest of the operating system, is achieved

9.4.1 The MMURTL Debugger

During the course of my research it was necessary for me to experiment with, and delve deeper into, the low-level workings of the MMURTL operating system. In order to do this in any operating environment requires the use of a low-level tool which executes close to the system kernel.

Fortunately, MMURTL could provide this facility through its debugger which is an integral component of the operating system. The availability of a debugging tool provided two advantages to my research presented in this thesis Firstly, the MMURTL debugger allowed me to perform step-through debugging into the low-level components of the new operating system. Secondly, it allowed me to investigate in detail the behaviour of the low-level behaviour of the original operating system.

Problems arose, however, in attempting to trace and test system behaviour during boot-up MMURTL's debugger could only be invoked when all of the system constructs and modules required by it had been initialised This occurred quite late in the boot cycle As a result it was not possible to use the debugger to trace through the system boot-up The only solution available to me was to study in detail the system sources prior to the point at which the debugger could be invoked

9.5 The OO-MMURTL Simulator

The methods described thus far have enabled me to test and debug OO-MMURTL's object-oriented code and frameworks on a syntactic and semantic level I have been able to verify the correctness of the code through the use of a non-native C++ compiler I have also been able to trace through the low-level code using a debugger

Although these methods served to provide an increased level of confidence in my designs, there remained a question mark over the actual behaviour of the OO-MMURTL's framework in an active environment Because the OO-MMURTL sources had been compiled in a non-system-compatible compiler, they could not be integrated with the remainder of the system for complete behavioural and system testing

This led to the need for the implementation of a new environment where OO-MMURTL's object-oriented class frameworks could be tested I have called the resulting environment simply 'The OO-MMURTL Simulator' The simulator is a DOS-based executable program compiled with Borland C++ The purpose of this simulator was not to perform MMURTL's operating system activities, but to provide an environment in which it was possible to test and validate the behaviour of the class frameworks

9.5.1 Activity of the OO-MMURTL Simulator

Perhaps the most important aspect of the performance of the OO-MMURTL simulator is that it attempts to simulate the behaviour of both the user and the system kernel. In attempting to simulate the kernel, OO-MMURTL makes the subset of the kernel API used by the class frameworks available Because the simulator is merely attempting to simulate the operating system behaviour, and not replace it, this subset of API functions have been rewritten So instead of actually allocating a contiguous block of memory, for example, the simulator will report the request (on the screen, for example), and simulate the allocation by updating the internal system variables which keep track of memory availability Thus although no memory has actually been allocated, the Memory Manager object does not realise this Similarly, the simulator's internal variables reflect that the memory has been allocated.

In addition to simulating the behaviour of the operating system kernel, the simulator also performs the role of system user In this way, the simulator program can make a set of calls and requests to the various class frameworks, thus simulating actual user requests upon these objects An example of this would be to invoke the Memory Manager object and to request a contiguous block of memory from it

The purpose of this role of the simulator is not in the calling of the OO-MMURTL class frameworks alone, but in the observation of the sequence of calls which occur in the class frameworks in response to the request Thus, by simulating a user request upon the class frameworks, and then observing the resulting kernel API calls, and the inter-framework calls, it is possible to adjudge the correctness of the class framework behaviour Thus, the simulator has succeeded in providing an additional layer of system testing

9.5.2 Strengths of the OO-MMURTL Simulator

- The simulator complements the previous testing and debugging by providing an additional level of syntactic checking of the OO-MMURTL class framework source code
- The introduction of the simulator has provided the first opportunity to examine and test the class frameworks communicating together and operating within a single executing environment Previous testing concentrated upon testing the individual class frameworks, for example the Memory framework, distinct from the rest of the frameworks
- The simulator provides the ability to observe the interactions between the class frameworks and the system kernel. This allowed comparisons to be performed between the expected kernel interactions and the actual ones
- The simulator is capable of monitoring the behaviour of the class frameworks in response to the simulation of a user request, allowing the system designer to verify the response is appropriate to the request

9.5.3 Weaknesses of the OO-MMURTL Simulator

- Because of its nature, the Simulator remains a step away from a system test It is merely an extension of the unit testing phase A complete system test will not be possible until the class frameworks have been fully integrated with the native MMURTL kernel.
- Similarly, it is impossible to judge the system performance based upon the behaviour of the simulator
- Because the simulator abstracts the behaviour of the kernel, it is impossible to examine how the OO-MMURTL operating system will manage its resources in response to the object-oriented entities, for example the Memory Manager, whose responsibility they are

9.6 Outstanding Issues

My research presents a possible design of an object-oriented implementation of MMURTL Before OO-MMURTL can be implemented, however, the developers must first provide an object-oriented derivative of the C Minus 32 programming language This implementation must be MMURTL-specific and DASM-compatible Once this object-oriented programming environment has been provided, system developers can begin to integrate the class frameworks presented here with the current MMURTL system. Only at that stage will a full system test be possible

9.7 Conclusions

This chapter presented the methods I used in validating the syntactic and semantic correctness of the object-oriented frameworks proposed during my research The methods used to test the behaviour of the classes were also presented, although a true test will only be possible once the class designs have been implemented and integrated with the operating system. Finally, a brief summary of outstanding issues was presented

Chapter 10

Conclusions

10.1 Overview

The purpose of this thesis was to attempt to migrate a conventional operating system to an object-oriented design and to observe the problems and benefits of such a migration This chapter recaps the main points which were observed, in addition to suggesting future directions which may be taken by Object-Oriented MMURTL

10.2 Benefits of the Migration to Object-Orientation

10.2.1 Introduction of Object Managers.

The concept of using an object manager (see section 4 4) in which to store objects provided many benefits to the new operating system. Each of the major system components, such as process management, memory management, and messaging, rely on a specialisation of the base object manager class

All object-oriented operating system must keep a repository of the objects which comprise it However, MMURTL's object managers perform a much greater role The behaviour of each of these specialised object managers has been designed in order to provide maximum support to the objects within each particular subsystem. The advantages of using object managers are summarised below

Centralised Processing - All processing of the objects belonging to each subsystem becomes the responsibility of a single entity, the object manager As a result, the behaviour of each subsystem, and all accesses to the components of that subsystem, may be regulated by the object manager This results in a stronger system design with tighter controls on privileged objects and operations

Protection - This is related to the previous point Since access to each subsystem's objects are through a centralised processing area, better protection is offered to the store of objects No task is allowed to gain access to an object, or perform an action on one, without full error checking having been performed in advance by the object manager in question Only when the object manager is satisfied, will each requested access be allowed

Intelligence - This is possibly the biggest difference between MMURTL's object managers and normal object repositories Each object manager has some specific knowledge of the nature of the data it holds and the operations they perform. For example, the exchange manager (see Section 6 3) can receive a communication from a task and detect whether it is a message or a request Having verified the correctness of the communication, the exchange manager will then forward it to the exchange belonging to the intended destination's task if it's a message, or in the case of a request then the intended system service will be looked up, before the communication is forwarded to it

Dynamic Allocation - This feature ensures that system resource usage is optimised Subsystem objects are created upon request by object managers and deallocated upon completion All of the objects which are allocated at any given time are in use by some component of the operating system.

Extensibility - The flexibility of the object-oriented paradigm in conjunction with the design of the object managers ensure their flexibility, thus rendering them capable of growth and expansion

10.2.2 Improved Classification of System Entities

The design of Object-Oriented MMURTL, as presented in this thesis, uses a hierarchy of objects to encapsulate the behaviour of each subsystem. By definition, each object-oriented hierarchy must adequately classify the various entities within the subsystem it represents This ensures the components of every object-oriented subsystem in OO-MMURTL are fully represented in, and classified by, a class hierarchy Taking the process management subsystem as an example (see Section 5 5), the components of this subsystem are represented by two hierarchies The base of the first class is the abstract CProcess class, while the base of the second is the CDeviceDriver class

Immediately every executing entity in the operating system must be classified by one of these categories Each of these base classes are inherited by subclasses and so on, resulting in five leaf nodes between the two hierarchies, namely CSystemService, CUserJob, CSystemJob, CReentrantDeviceDriver, and CNonReentrantDeviceDriver Now, each object in the process management subsystem must be classified as being one of these objects

This detailed classification of system entities allows the system designer to present a different set of behaviour, by designing a unique set of attributes and methods, for each of these classes This facility is not present in MMURTL, where the same code is responsible for dealing with entities of a different nature

10.2.3 Simpler Programming Interface

Object-oriented programming provides several advantages over conventional programming languages, these advantages are universal and are much documented This section deals with the advantages resulting from the design of OO-MMURTL in particular This is shown by taking examples from different areas of the operating system.

Programming Interrupts - Thanks to the CInterrupt class, much of the low-level work involved in setting up an interrupt has been encapsulated, and therefore hidden from the programmer (see Section 8 5) OO-MMURTL does this by abstracting the default behaviour required by every interrupt, namely, setting up an interrupt vector, responding to an interrupt call, and by ensuring that the operating system is signalled with an end of interrupt message To create an interrupt in OO-MMURTL, the system programmer need only derive a subclass of CInterrupt and in doing so, provide a method which will service the interrupt Immediately this class is instantiated, the interrupt will be added to the Interrupt Descriptor Table and will be ready to perform.

Programming System Services - Similar to programming interrupts, the CSystemService class encapsulates much of the default behaviour which is performed by every system service, this even extends to creating the exchange through which the system service will receive its requests In addition, subclasses of the CSystemService class provide more specialised system services while still hiding the code from the service programmer

In this way, the programmer need only decide whether to derive a subclass of CReentrantDeviceDriver or CNonReentrantDeviceDriver If the latter is chosen mutual exclusion, and thus non-reentrancy, will be enforced without any extra programming required by the developer

10.2.4 Multiple Personalities

The use of object-oriented class hierarchies enables the system designer to create abstract classes which define the default behaviour of classes which implement system policy decisions This is exemplified by the abstract CReadyQueue class (see Section 8 2) In this case, the default behaviour of this class indicates that the CReadyQueue class must be capable of adding items to the queue, removing items from the queue, and checking what the item at the top of the queue is No details are given, however, describing how the queue should be implemented

This mechanism allows the system developer to provide multiple differing implementations of the ready queue, each enforcing a different scheduling policy, for example a prioritised queue and a FIFO queue (see Sections 8 3 and 8 4 respectively) It would then be possible to create different installations of the same operating system, whose policies are customised for the installation in question, without affecting the rest of the system.

10.2.5 Extensible Frameworks

The OO-MMURTL class hierarchies have been designed to be as generic as necessary at the base level, and specialised in further levels This provides for an extensible operating system which may be easily augmented or enhanced at a later date For example, the messaging hierarchy provides for two type of exchange, and two types of messages Future development of MMURTL could necessitate the introduction of a new form of exchange or a new type of message In this case, the existing hierarchy need only be extended at an appropriate level, depending on the nature of the new class

10.3 Problems Encountered During Migration

Several problems were encountered during the migration of MMURTL to an object-oriented paradigm. These should serve to warn future migrations, particularly those involving more complex operating systems, of the problems which may be encountered

Inappropriate Implementations - The single most worrying problem arose in the redesign of the memory subsystem as an object-oriented component of OO-MMURTL Unlike the rest of the object-oriented components in OO-MMURTL, the memory subsystem did not adapt easily to the new paradigm. These difficulties are described in detail previously (see Section 7 4) In summary, the only reasonable option was to provide a single class which encapsulated the low-level code, thus providing an object-oriented programming interface which provided at least some advantages over the original implementation

Trivial Class Behaviour - Other subsystems of MMURTL proved less troublesome to migrate to object-orientation In the process management subsystem, for example, many different entity classifications were easily identified, as was mentioned earlier in this chapter This did allow the behaviour of classes to be differentiated and classified in detail, however, due to the original implementation of MMURTL, the object-oriented implementation of some of these entities proved to be close to trivial.

This is best exemplified by the CUserJob / CSystemJob pair Both of these are subclasses of CJob, however their only behavioural differences are in terms of their memory allocation routines[1] If OO-MMURTL had been designed from bottom-up as an object-oriented operating system, a more diverse set of behaviour for each class would have been expected Another reason for this triviality lies in the fact that much of the low-level task management routines remain in the kernel, thus reducing the amount of functionality performed by the CJob subclasses

[1] Note that matters such as loading the jobs into the appropriate address space, ie user or system, and the setting of the appropriate system protection level, again either user or system, is the responsibility of the process manager class when it loads new jobs

Widespread Hardware Dependencies - MMURTL was designed explicitly with the Intel 386/486 architecture in mind In designing MMURTL, its creators attempted to streamline the system in terms of speed as much as possible This was often done by taking advantage of low-level code Unfortunately, this resulted in a microkernel whose implementation was hardware dependent to a big extent This caused problems in designing OO-MMURTL since these hardware-dependent routines had to remain within the microkernel. Again if OO-MMURTL had been designed from the bottom-up, the hardware dependent code would be minimised and centralised thus allowing the remainder of the operating system to reap the full rewards of object-orientation

10.4 Future Directions

There are many possible directions in which Object-Oriented MMURTL can be extended The most obvious of these would be to create a machine-independent version of the operating system by reducing the hardware-dependent components to a minimum level within a single area of the microkernel. This could then allow a future development of OO-MMURTL to become a distributed operating system. (Migrating a single machine object-oriented operating system to a distributed operating system could prove to be a very interesting thesis) Naturally, a project such as this would introduce a wide range of new factors to the operating system design Naming, persistency, distributed object-invocation and distributed linking are but the tip of the iceberg

The most important move for OO-MMURTL now would be a re-implementation of the components which did not migrate easily, in particular the memory management system. This would involve redesigning the memory model, probably away from its non-object-oriented origins, and providing a class hierarchy of component objects Due to the current design of the OO-MMURTL memory model, this should not affect existing components This is because the interface of the memory manager will remain the same in that it will continue to allocate and manage pages of memory The underlying implementation of the memory manager need only be re-implemented to support the new design This, once again, highlights the advantages of the design of Object-Oriented MMURTL

Bibliography

[Balter 91] R. Balter, J Bernadat, D Decouchant, A. Duda, A. Freyssinet, S Krakowiak, M Meysembourg, P Le Dot, H. Nguyen Van, E Paire, M Riveill, C Roisin, X Rousset de Pina, R. Scioville, G Vandome *Architecture and Implementation of Guide, an Object-Oriented Distributed System*, Computing Systems, 1991

[Burgess 95] Richard Burgess *Developing Your Own 32-bit Operating System* Sams Publishing, 1995

[Campbell 95] Roy H. Campbell and See-Mong Tan μ*Choices An Object-Oriented Multimedia Operating System* IEEE Computer Society Fifth Workshop on Hot Topics in Operating Systems, Orcas Island, Washington, May 1995

[Campbell 91] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris and Peter Madany *Choices, Frameworks and Refinement* Proceedings International Workshop on Object Orientation in Operating Systems, October 17-18 1991

[Campbell 92] Roy H. Campbell and Nayeem Islam. *A Technique for Documenting the Framework of an Object-Oriented System* Proceedings Second International Workshop on Object-Orientation in operating Systems

[Campbell 93a] Roy H Campbell, Nayeem Islam, David Raila, Peter Madany *Experiences designing and implementing an object-oriented system in C++* University of Illinois 93

[Campbell 93b] Roy H. Campbell and Nayeem Islam. *Choices A Parallel Object-Oriented Operating System* Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993

[Cheriton 88] David Cheriton *The V Distributed System* Communications of the ACM, pg 314-334, 1988

[Deutsch 87] Peter Deutsch *Levels of Reuse in the Smalltalk 80 Programming System* IEEE Computer Society Press, Cambridge, Mass , 1987

[Deutsch 89] Peter Deutsch *Design Reuse and Frameworks in the Smalltalk-80 Programming System* ACM Press, Cambridge, Mass , 1989

[Guedes 92] Paolo Guedes and Daniel P Julin *Object-Oriented Interfaces in the Mach 3 0 Multi-Server System* Proceedings Second International Workshop on Object-Orientation in operating Systems

[Hamilton 93] Graham Hamilton and Panos Kougiouris *The Spring nucleus A microkernel for objects* Proceedings of the 1993 Summer Usenix Conference, June 1993

[Julin 89] D Julin and R. Rashid *MachObjects* Internal document, Mach project, Carnegie Mellon University, 1989

[Krakowiak 93] S Krakowiak *Issues in Object-Oriented Distributed Systems* 1993 Guide Project, University of Grenoble, France

[Mitchell 93] James G Mitchell, Jonathon J Gibbons, Graham Hamilton, Peter B Kessler, Yousef A. Khalidi, Panos Kougiouris, Peter W Madany, Michael N Nelson, Michael L Powell, and Sanjay R. Radia *An Overview of the Spring System* Sun Microsystems, 1993

[Ritchie 75] Dennis M Ritchie and Kenneth Thompson *The UNIX Time-Sharing System* AT&T Bell Laboratories Technical Journal, 57(6) 1905, 1975

[Rashid 86] Richard Rashid *Threads of a New System* Unix Review, 1986

[Tannenbaum 90] Andrew S Tannenbaum, Robbert van Renesse, Hans van Staveren, Gergory J Sharp, Sape J Mullender, Jack Jensen, and Guide van Rossum. *Experiences with the Amoeba distributed operating system* Communications of the ACM 33(12), December 1990

[Talligent 93] *Leveraging object-oriented frameworks* Taligent Inc white paper, 1993

[Weiner 90] L Weiner, B Wilkinson, R. Wirfs-Brock *Designing Object-Oriented Software*, Prentice-Hall, 1990

Appendix

C++ Implementations of OO-MMURTL Classes

Process Definitions

```
// CProcess Class
//---------------

class CProcess {
private
  long ProcNum,
  char sbProcName[14],  // 13 bytes  First byte is length
  unsigned char *pPD,   // Linear add of Job's PD (0=unused)

  char sbUserName[30],  // User Name for Job
  char sbPath[70],      // Path name
  char ExitRF[80],      // Exit Run file (if any)
  char ProcCmdLine[80], // Command Line string
  char SysIn[50],       // Standard input
  char SysOut[50],      // Standard output

  long ExitError,       // Error Set by ExitJob
  char *pVidMem,        // pointer to crnt video buffer
  char *pVirtVid;       // Virtual Video Buffer Address
  long CrntX,           // Current cursor position
  long CrntY;
  long NormVid,         // 7 = WhiteOnBlack

  char fCursOn,         // 1 = Cursor is visible
  char fCursType,       // 0 = Underline, 1 = Block
  long ScrlCnt,         // Count since last pause
  char fVidPause,       // Full screen pause (Text mode)

public:
    CProcess(long Num, char *Name, char *User, char *Path,
             char *CmdLine,char *VidMem,char *VirtVid);

    void Chain(char *pFileName, long dExitError);
    void ExitProc(long dError),
    void FreeResources(),
    virtual long FreeSystemResources() = 0;
    long GetProcNum(),

    void SetUserName(char *pUser);
    void GetUserName(char *pUserRet),
    void GetCmdLine(char *pCmdRet),
    void GetPath(long JobNum, char *pPathRet);

    void SetExitJob(char *pRunFile),
    void GetExitJob(char *pRunRet),
    void SetSysIn(char *pFile);
    void GetSysIn(char *pFileRet),
    void SetSysOut(char *pFile),
    void GetSysOut(char *pFileRet),

    void KillTask(),
},
```

```
// CJob Class
//-----------

class CJob · public CProcess {
public
  CJob(long Num, char *Name, char *User, char *Path, char *CmdLine,
       char *VidMem,char *VirtVid)
        CProcess(Num,Name,User,Path,CmdLine,VidMem,VirtVid) {};

  unsigned long AllocPage(unsigned long nPages,CPage *ppMemRet) = 0;
},



// CUserJob Class
//---------------

class CUserJob   public CJob {
public
  CUserJob(long Num, char *Name,char *User,char *Path, char *CmdLine,
           char *VidMem,char *VirtVid)
         : CJob(Num,Name,User,Path,CmdLine,VidMem,VirtVid) {};

  virtual unsigned char *AllocPage(unsigned long nPages),
},



// CSystemJob Class
//-----------------

class CSystemJob : public CJob {
public
  CSystemJob(long Num, char *Name, char *User, char *Path,
             char *CmdLine, char *VidMem, char *VirtVid)
      : CJob(Num, Name, User, Path, CmdLine, VidMem, VirtVid) {};

  virtual unsigned char *AllocPage(unsigned long nPages);
},


// CSystemService Class
//---------------------

class CSystemService   public CProcess{
private
  char SvcName[12],
  CExchange *pSvcExch,
public
  CSystemService(long Num, char *Name, char *User, char *Path,
                 char *CmdLine,char *VidMem,char *VirtVid),
  void Service();
  virtual unsigned long ServiceRequest(TRequest *pReqBlk) = 0;
},
```

Process Class Implementations

```
CProcess ·CProcess(long Num, char *Name, char *User,
                   char *Path, char *CmdLine,
                   char *VidMem, char *VirtVid)
{
  /* Initialise variables as per parameters */
  ProcNum = Num,
  strcpy(sbProcName,Name),
  strcpy(sbUserName,User),
  strcpy(sbPath,Path),
  strcpy(ProcCmdLine,CmdLine),
  pVidMem = VidMem,
  pVirtVid = VirtVid,

  /* Set system input for this process to keyboard */
  strcpy(SysIn,"KBD"),

  /* Set system output for this process to video */
  strcpy(SysOut,"VID");

  ExitError = 0,
  CrntX = CrntY = 0,       /* Initial cursor pos (0,0) */
  fCursOn = 1,             /* Cursor is on */
  fCursType = 1            /* Block cursor */
  ScrollCount = 0,
  NormVid = 7;             /* White on Black */
  strcpy(ExitRF,""),       /* No ExitRunFile initially */
}
```

```
void CProcess. Chain(char *pFileName, long dExitError)
{
  CExchange *pExch, *pCurrExch,
  long ercE, iE, ExchProc, CurrProc, ExitError,

  ercE =  GetRunFile(pFileName, cbFileName, &job_fhE),
  if (ercE)
    return(ercE),

  ExitError = dExitError,

  // Remove ALL tasks for this job that are at the ReadyQue
  // This task won't be removed because its RUNNING'

  RemoveRdyJob(),

/* Deallocate all exchanges for this job except the one belonging to
   current TSS' The Dealloc Exchange call will invalidate all TSSs
   found at exchanges belonging to this user, and will also free up
   its resources   The job will not be able to initiate requests or
   send messages after this unless it is done with the
   TSSExchange because it will get a kernel error  */

// Find out TSS exchange so it isn't deallocated */

  pExch = GetTSSExch(),
  CurrProc = GetCurrProcNum(),

  ercE = 0,
  iE = 0;
  while (ercE != ErcOutOfRange) {
    ExchProc = ExchangeManager->GetOwner(iE);
    pCurrExch = ExchangeManager->GetExchange(iE),
    if (('ercE) && (ExchProc == CurrProc) && (pCurrExch'= pExch))
      ercE = ExchangeManager->RemoveExch(iE),
    iE++;
    }

/* Now that the user can't make anymore requests, Send Abort messages
   to all services. This closes all files that were opened by the Job
   and frees up any other resources held for this job by any
   services  */

  SendAbort(),

  TPacket *pPacket = pExch->CheckPacket(),
  while(pPacket != NULL)   // clear the exchange of abort responses
    pPacket = pExch->CheckPacket(),
}
```

```
void CProcess  ExitProc(long dError)
{
  CExchange *pExch, *pCurrExch,
  long ercE, iE, ExchProc, CurrProc, ExitError,

  ExitError = dError;

  // Remove ALL tasks for this job that are at the ReadyQue.
  // The task we are in won't be removed because its RUNNING!

  ReadyQueue->RemoveRdyProc(ProcNum),


/* Deallocate all exchanges for this job except the one belonging to
   current TSS! The Dealloc Exchange call will invalidate all TSSs
   found at exchanges belonging to this user, and will also free up
   its resources   The job will not be able to initiate requests or
   send messages after this unless it is done with the
   TSSExchange because it will get a kernel error  */

// Find out TSS exchange so it isn't deallocated */

  pExch = GetTSSExch();
  CurrProc = GetCurrProcNum();

  ercE = 0;
  iE = 0,
  while (ercE != ErcOutOfRange) {
    ExchProc = ExchangeManager->GetOwner(iE);
    pCurrExch = ExchangeManager->GetExchange(iE),
    if ((!ercE) && (ExchProc == CurrProc) && (pCurrExch!= pExch))
      ercE = ExchangeManager->RemoveExch(iE),
    iE++;
    }

/* Now that the user can't make anymore requests, Send Abort messages
   to all services. This closes all files that were opened by the Job
   and frees up any other resources held for this job by any
   services. */

  SendAbort(),

  TPacket *pPacket = pExch->CheckPacket();
  while(pPacket != NULL)    // clear the exchange of abort responses
    pPacket = pExch->CheckPacket(),
}
```

```
void CProcess  FreeResources()
{
  CExchange *pExch, *pCurrExch,
  long ercE, iE, ExchProc, CurrProc,

  // Allow subclasses to free additional system resources
  // which they may have allocated

  FreeSystemResources();

  // Remove ALL tasks for this job that are at the ReadyQueue.
  // This task won't be removed because its Running'

  ReadyQueue->RemoveRdyProc(),

  /* Deallocate all exchanges for this process except the one
     belonging to current TSS  The Dealloc Exchange call will
     invalidate all TSSs found at exchanges belonging to this
     user, and will also free up RQBs and Link Blocks   The
     job will not be able to initiate requests or send messages
     after this */

  // Find out current TSS exchange so it isn't deallocated

  pExch = GetTSSExch(),
  CurrProc = GetCurrProcNum(),

  ercE = 0;
  iE = 0,
  while (ercE != ErcOutOfRange) {
    ExchProc = ExchangeManager->GetOwner(iE),
    pCurrExch = ExchangeManager->GetExchange(iE),
    if ((!ercE) && (ExchProc == CurrProc) && (pCurrExch'= pExch))
      ercE = ExchangeManager->RemoveExch(iE),
    iE++,
    }

  /* Now that the user can't make anymore requests, Send Abort
     messages to all services  This closes all files that were
     opened by the Job and frees up any other resources held
     for this job by any services.*/

  SendAbort(),

 // Clear the exchange of abort responses (ignore them)

  TPacket *pPacket = pExch->CheckPacket();
  while(pPkt==NULL)
    pPacket= pExch->CheckPacket(),
}
```

```
void CProcess  KillTask(void)
{
  CExchange *pExch,

  FreeResources(),

  pExch = GetTSSExch(),

  pPacket = NULL,
  while(pPacket == NULL)
    pPacket = pExch->CheckPacket(),

  pExch->ISendPacket(ErcOpCancel,ErcOpCancel),
  ProcessManager->SetPriority(ProcNum,31),
  TPacket *pPacket = WaitPacket(),

  while(1), /* in case we get scheduled again */
}


long CProcess· GetProcNum()
{
  return ProcNum,
}


void CProcess  SetUserName(char *pUserRet)
{
  strcpy(sbUserName,pUserRet);
}


void CProcess  GetUserName(char *pUserRet)
{
  strcpy(pUserRet,sbUserName);
}


void CProcess· GetCmdLine(char *pCmdRet)
{
  strcpy(pCmdRet,JobCmdLine),
}


void CProcess  GetPath(char *pPathRet)
{
  strcpy(pPathRet,sbPath),
}


void CProcess  SetExitJob(char *pRunFile)
{
  strcpy(JcbExitRF,pRunFile);
}
```

```
void CProcess  GetExitJob(char *pRunRet)
{
  strcpy(pRunRet,JcbExitRF),
}


void CProcess: SetSysIn(char *pSysIn)
{
  strcpy(JcbSysIn,pSysIn);
}


void CProcess  GetSysIn(char *pSysInRet)
{
  strcpy(pSysInRet,JcbSysIn),
}


void CProcess  SetSysOut(char *pSysOut)
{
  strcpy(JcbSysOut,pSysOut),
}


void CProcess  GetSysOut(char *pSysOutRet)
{
  strcpy(pSysOutRet,JcbSysOut),
}
```

```
// CUserJob Class

unsigned char *CUserJob  AllocPage(unsigned long nPages)
{
  return MemoryManager->AllocPage(nPages),
}


// CSystemJob Class

unsigned char *CSystemJob ·AllocPage(unsigned long nPages)
{
  return MemoryManager->AllocOSPage(nPages),
}
```

```
CSystemService  CSystemService(long Num, char *Name, char *User,
                               char *Path, char *CmdLine,
                               char *VidMem, char *VirtVid)
    . CProcess(Num,Name,User,Path,CmdLine,VidMem,VirtVid),
{
  SvcExch = CreateExchange(),
  strcpy(pSvcName,Name);
  RegisterService(pSvcName,SvcExch),
}


CSystemService  ~CSystemService()
{
  SvcExch->Remove(),
  SvcExch->DeAllocate();  // Free memory used by SvcExch
}


void CSystemService  Service()
{
  unsigned long ErrorToUser,
  unsigned long Message[2];
  TRequest *pReqBlk

  while(1) {
    TPacket *pPkt = SvcExch->WaitPacket();

    if(pPkt'=NULL) {
      pReqBlk = Message[0],

      ErrorToUser = ServiceRequest(pReqBlk);

      SvcExch->Respond(pReqBlk,ErrorToUser),
      }
    }
}
```

Device Driver Definitions

```
class CDeviceDriver {
private.
  char Name[12],
  unsigned int Type,     // 0 = No Device, 1=Random,   2=Sequential
  unsigned int nBPB,     // Sequential Bytes per block (1-65535)
  long LastDevErc;       // Last operation error code
  int nBlocks;           // Number of blocks in device
  int fSingleUser,       // Is device assignable?
  long wJob,             // If assignable, is it assigned?

  virtual long DevOperation(unsigned long dOpNum,
                            unsigned long dLBA,
                            unsigned long dnBlocks,
                            unsigned char *pData) = 0,

  virtual long DevStatus(char *pStatRet,
                         unsigned long dStatusMax,
                         unsigned long *pdSatusRet) = 0;

  virtual long DevInitialise(char *pInitData,
                             unsigned long sdInitData) = 0;

public
  // Constructor for sequential device driver
  CDeviceDriver(char *DevName, unsigned int BPB, int Blocks,
                int SingleUser);

  // Constructor for non-sequential device driver
  CDeviceDriver(char *DevName, int SingleUser);

  virtual long Operation(unsigned long dOpNum,
                         unsigned long dLBA,
                         unsigned long dnBlocks,
                         unsigned char  *pData) = 0,

  virtual long Status(char *pStatRet,
                      unsigned long dStatusMax,
                      unsigned long *pdSatusRet) = 0,

  virtual long Initialise(char *pInitData,
                          unsigned long sdInitData) = 0;
},
```

```
class CReentrantDeviceDriver {
private
  virtual long DevOperation(unsigned long dOpNum, unsigned long dLBA,
                            unsigned long dnBlocks,
                            unsigned char *pData) = 0,

  virtual long DevStatus(char *pStatRet, unsigned long dStatusMax,
                         unsigned long *pdStatusRet) = 0,

  virtual long DevInitialise(char *pInitData,
                             unsigned long sdInitData) = 0,

public
  // Constructor for sequential device driver
  CReentrantDeviceDriver(char *DevName, unsigned int BPB,
                             int Blocks, int SingleUser)
      : CDeviceDriver(DevName, BPB, Blocks, SingleUser) {} ;

  // Constructor for non-sequential device driver
  CReentrantDeviceDriver(char *DevName, int SingleUser)
      : CDeviceDriver(DevName, SingleUser) {} ;

  virtual long Operation(unsigned long dOpNum, unsigned long dLBA,
                         unsigned long dnBlocks,
                         unsigned char *pData),

  virtual long Status(unsigned long dDevice, char *pStatRet,
                      unsigned long dStatusMax,
                      unsigned long *pdSatusRet),

  virtual long Initialise(unsigned long dDevNum, char *pInitData,
                          unsigned long sdInitData),
},
```

```
class CNonReentrantDeviceDriver {
private
      CExchange    *SemExch,    // Exchange for device semaphore
      CMessage     *SemMsg;     // CMsg holder for WAITs from queued
CTasks

  virtual long DevOperation(unsigned long dOpNum, unsigned long dLBA,
                            unsigned long dnBlocks,
                            unsigned char *pData) = 0,

  virtual long DevStatus(char *pStatRet, unsigned long dStatusMax,
                         unsigned long *pdStatusRet) = 0,

  virtual long DevInitialise(char *pInitData,
                             unsigned long sdInitData) = 0,

public
  // Constructor for sequential device driver
  CNonReentrantDeviceDriver(char *DevName, unsigned int BPB,
                            int Blocks, int SingleUser)
        CDeviceDriver(DevName, BPB, Blocks, SingleUser) {} ;

  // Constructor for non-sequential device driver
  CNonReentrantDeviceDriver(char *DevName, int SingleUser)
      : CDeviceDriver(DevName, SingleUser) {} ;

  virtual long Operation(unsigned long dOpNum, unsigned long dLBA,
                         unsigned long dnBlocks,
                         unsigned char *pData),

  virtual long Status(unsigned long dDevice, char *pStatRet,
                      unsigned long dStatusMax,
                      unsigned long *pdSatusRet);

  virtual long Initialise(unsigned long dDevNum, char *pInitData,
                          unsigned long sdInitData),
},
```

CDeviceDriverClass Implementation

```
CDeviceDriver  CDeviceDriver(char *DevName, unsigned int BPB,
                             int Blocks,    int SingleUser)
{
  strcpy(Name,DevName),
  nBPB = BPB,                  // Bytes per block
  nBlocks = Blocks,
  fSingleUser = SingleUser,    // Is device assignable?

  Type = 2,                    // Sequential device driver

  LastDevErc = 0,
  wJob = 0,
}


CDeviceDriver :CDeviceDriver(char *DevName, int SingleUser)
{
  strcpy(Name,DevName);
  fSingleUser = SingleUser,    // Is device assignable?

  Type = 1,                    // Random device driver
  nBPB = 0;                    // Does not apply
  nBlocks = 0;                 // Does not apply

  LastDevErc = 0,
  wJob = 0,
}
```

CReentrantDeviceDriver Class Implementation

```
long CReentrantDeviceDriver· Operation(unsigned long dOpNum,
                    unsigned long dLBA, unsigned long dnBlocks,
                    unsigned char  *pData)
{
    return DevOperation(dOpNum,dLBA,dnBlocks,pData),
}

long CReentrantDeviceDriver .Status(char *pStatRet,
                                    unsigned long dStatusMax,
                                    unsigned long *pdStatusRet)
{
    return DevStatus(pStatRet, dStatusMax,pdStatusRet),
}

long CReentrantDeviceDriver  Initialise(char *pInitData,
                                        unsigned long sdInitData)
{
    return DevInit(pInitData,sdInitData),
}
```

CNonReentrantDeviceDriver Class Implementation

```
long CNonReentrantDeviceDriver.:Operation(unsigned long dOpNum,
                    unsigned long dLBA,unsigned long dnBlocks,
                    unsigned char  *pData)
{
  long erc,

  TPacket *pSemPkt = SemExch->WaitPacket(), // Wait for MutEx

  /* Mutual exclusion has been achieved, perform operation */
  erc = DevOperation(dOpNum,dLBA,dnBlocks,pData),

  SemExch->SendDummyPacket(),  // Signal

  return erc,
}

long CNonReentrantDeviceDriver::Status(char *pStatRet,
                              unsigned long dStatusMax,
                              unsigned long *pdStatusRet)
{
  long erc,

  TPacket *pSemPkt = SemExch->WaitPacket();

  /* Mutual exclusion has been achieved, retrieve status */
  erc = DevStatus(pStatRet,dStatusMax,pdStatusRet),

  SemExch->SendDummyPacket();  // Signal

  return erc,
}

long CNonReentrantDeviceDriver.:Initialise(char *pInitData,
                                    unsigned long sdInitData)
{
  long erc;

  TPacket *pSemPkt = SemExch->WaitPacket();

  /* Mutual exclusion has been achieved, initialise */
  erc = DevInit(pInitData,sdInitData),

  SemExch->SendDummyPacket(),

  return erc,
}
```

Messaging Definitions - Structures

```
// TTSS Structure
// --------------

struct TTSS {
  CProcess *pProc,
  TPacket *pLBRet,

  struct TTSS *next,
},



// TRequest Structure
// ------------------

struct TRequest {
  CExchange *RespExch;  // Exchange to respond to
  long RqOwnerProc;     // JobNum of Owner of the CRequestBlock
  int  ServiceCode;     // System Service Command Number
  long dData0;          // User fill / Srvc Defined (No Pointers)
  long dData1,          // User fill / Srvc Defined (No Pointers)
  long dData2;          // User fill / Srvc Defined (No Pointers)
  char *pData1;         // User fill / Srvc Defined
  long cbData1;         // Length of data in pData1
  char *pData2,         // User fill / Srvc Defined
  long cbData2,         // Length of data in pData2

  struct TRequest *next;
},


// TMessage Structure
// ------------------

struct TMessage{
  long dData1,          // Data field 1
  long dData2;          // Data field 2

  struct TMessage *next,
},


// TPacket Structure
// -----------------

struct TPacket {
  struct TRequest *Req,
  struct TMessage *Msg;

  struct TPacket *next,
};
```

Messaging Definitions - Classes

```
// CExchange Class
// ---------------

class CExchange {
  TPacket *pPktQueueHead, *pPktQueueTail,
  TTSS *pTSSQueueHead, *pTSSQueueTail;
  CProcess *pOwnerProc;

public
  CExchange(),

  CProcess *GetOwner() { return pOwnerProc, },

  TTSS *deQueueTSS(),
  void enQueueTSS(TTSS *pTSS);
  TPacket *deQueuePacket(),
  void enQueuePacket(TPacket *NewPacket);
  TPacket *WaitPacket(),
  void SendPacket(TPacket *pPacket);
  void SendDummyPacket(),
  void ISendDummyPacket();
  TPacket *CheckPacket();
},



// CServiceExchange Class
// ----------------------

class CServiceExchange · public CExchange {
public.
  CServiceExchange()   CExchange() {},

  void MoveRequest(TRequest *pReq, CExchange *pExch);
  void Respond(TRequest *pReq),
  void Request(int code, CExchange *respexch, long data0, long data1,
               long data2, char *pdata1, long cbdata1, char *pdata2,
               long cbdata2),
},


// CMessageExchange Class
// ----------------------

class CMessageExchange   public CExchange {
public.
  CMessageExchange() : CExchange() {};

  void SendMsg(long dMsgData1, long dMsgData2);
  void ISendMsg(long dMsgData1, long dMsgData2),
};
```

CExchange Class

```
CExchange:.CExchange()
{
  pPktQueueHead = pPktQueueTaıl = NULL,
  pTSSQueueHead = pTSSQueueTaıl = NULL,
  pOwnerProc = GetpRunProc(),
}



TPacket *CExchange  deQueuePacket()
{
  TPacket *pPacket,

  pPacket = pPktQueueHead,
  pPktQueueHead = pPktQueueHead->next;

  return pPacket;
}



voıd CExchange  enQueuePacket(TPacket *pPKT)
{
  ıf (pPktQueueHead == NULL) {
    pPktQueueHead = pPKTQueueTaıl = pPKT,
    pPKTQueueHead->next = NULL,
    }
  else {
    pPKTQueueTaıl->next = pPKT,
    pPKTQueueTaıl = pPKT,
    }
}



TTSS *CExchange..deQueueTSS()
{
  TTSS *pTSS,

  pTSS = pTSSQueueHead,
  pTSSQueueHead = pTSSQueueHead->next;

  return pTSS,
}
```

```
void CExchange :enQueueTSS(TTSS *pTSS)
{
  if (pTSSQueueHead == NULL) {
    pTSSQueueHead = pTSSQueueTail = pTSS,
    pTSSQueueHead->next = NULL;
    }
  else {
    pTSSQueueTail->next = pTSS,
    pTSSQueueTail = pTSS,
    }
}


void CExchange  SendPacket(TPacket *pPacket)
{
  TTSS *pWaitTSS,*pPriorityTSS;

  // Remove task from the exchange's queue
  pWaitTSS = deQueueTSS();

  // If no task is waiting, queue the packet
  if (pWaitTSS==NULL) {
    enQueuePacket(pPacket),
    return,
    }

  // If a task was waiting notify it of the received packet
  pWaitTSS->pLBRet = pPacket,

  // Reevaluate the Ready Queue in case a higher
  // priority task is available
  ReadyQueue->enQueueRdy(pWaitTSS),
  pPriorityTSS = ReadyQueue->deQueueRdy(),

  // If the highest priority task is the current one,
  //   no task switch is required
  if (pPriorityTSS == pWaitTSS)
    return,

  /* Perform a 386 processor task switch */
  #asm
  MOV EAX,pPriorityTSS
  MOV BX,[EAX Tid]
  MOV TSS_Sel,BX
  INC _nSwitches
  JMP FWORD PTR [TSS]
  #endasm
}
```

```
TPacket *CExchange· WaitPacket()
{
  TPacket *pPacket,
  TTSS *pRunTSS, *pPriorityTSS,

  #asm
  CLI
  #endasm

  pPacket = deQueuePacket(),

  if (pPacket==NULL) {
    // Add the current task to the TSSQueue of this exchange
    pRunTSS = GetpRunTSS(),
    pRunTSS->next = NULL,
    enQueueTSS(pRunTSS);

    // Get the next TSS to run (if there is one)
    pPriorityTSS = ReadyQueue->deQueueRdy();

    // If none were ready, loop until one is
    while (pPriorityTSS == NULL) {
      #asm
      STI
      HLT
      CLI
      #endasm

      pPriorityTSS = ReadyQueue->deQueueRdy(),
      }

    if (pPriorityTSS != pRunTSS) {
      // Tasks are now switched by performing a 386 task switch
      #asm
      MOV EAX, [pPriorityTSS]
      MOV BX,[EAX.Tid]
      MOV TSS_Sel,BX
      JMP FWORD PTR [TSS]
      #endasm
      }
    }
    // A task has just finished "Waiting"  Now in the new task
  }

  /* we have either switched tasks and we are delivering a packet to
     the new task, or there was a packet waiting at the exch of the
     first caller and we are delivering it  */

  #asm
  STI
  #endasm

  return pPacket;
}
```

```
TPacket *CExchange  CheckPacket()
{
  TPacket *pPacket,

  // Disable interrupts
  #asm
  CLI
  #endasm

  pPacket = deQueuePacket(),

  // Reenable interrupts
  #asm
  STI
  #endasm

  return pLB,
}
```

CServiceExchange Class

```
void CServiceExchange  MoveRequest(TRequest *pReq, CExchange *pExch)
{
  TPacket *pPacket;
  TTSS *pTSS, *pPriorityTSS,

  pPacket = (TPacket *)malloc(sizeof(TPacket)),
  pPacket->Req = pReq;
  pPacket->Msg = NULL,
  pPacket->next = NULL,

  #asm
  CLI
  #endasm

  pTSS = pExch->deQueueTSS();
  if (pTSS==NULL) {
    pExch->enQueuePacket(pPacket),
    #asm
    STI
    #endasm
    }

  // Store link block in dequeued task
  pTSS->pLBRet = pPacket,

  ReadyQueue->enQueueRdy(pTSS);
  pPriorityTSS = ReadyQueue->deQueueRdy(),

  if(pPriorityTSS == pTSS) {
    #asm
    STI
    #endasm
    return,
    }

  #asm
  MOV EAX,pPriorityTSS     ; Make the TSS in EAX the Running TSS
  MOV BX,[EAX Tid]         , Get the task Id (TR)
  MOV TSS_Sel,BX           ; Put it in the JumpAddr
  JMP FWORD PTR [TSS]      , JMP TSS
  STI
  #endasm
}
```

```
void CServiceExchange. Request(int code, CExchange *respexch,
            long data0, long data1, long data2, char *pdata1,
            long cbdata1, char *pdata2, long cbdata2)
{
  TPacket *pPacket,
  TRequest *pReq,

  // Create request structure
  pReq = new TRequest,

  pReq->ServiceCode = code,
  pReq->RespondExch = respexch,
  pReq->RqOwnerProc = GetCrntJobNum(),
  pReq->dData0 = data0,
  pReq->dData1 = data1,
  pReq->dData2 = data2,
  pReq->pData1 = pdata2;
  pReq->cbData1 = cbdata1;
  pReq->pData2 = pdata2,
  pReq->cbData2 = cbdata2,

  // Create the packet
  pPacket = new TPacket;
  pPacket->Req = pReq,
  pPacket->Msg = NULL;

  // Disable interrupts
  #asm
  CLI
  #endasm

  SendPacket(pPacket);

  // Reenable interrupts
  #asm
  STI
  #endasm
}


void CServiceExchange·.Respond(TRequest *pReq)
{
  long dCurrProc,dReqProc,
  TTSS *pTSS,*pPriorityTSS,
  TPacket *pPacket,

  dCurrProc = GetCurrProcNum(),
  dReqProc = pReq->RqOwnerProc;
```

```
  // Perform memory aliasing if required
  if (dReqProc!=dCurrProc) {
    if (pReq->cbData1 > 0) && (pReq->pData1 '= NULL)
      DeAliasMem(pReq->pData1,pReq->cbData1,dCurrProc),

    if (pReq->cbData2 > 0) && (pReq->pData2 '= NULL)
      DeAliasMem(pReq->pData2,pReq->cbData2,dCurrProc),
    }

  // Disable interrupts
  #asm
  CLI
  #endasm


  pPacket = new TPacket,
  pPacket->Req = pReq,
  pPacket->Msg = NULL,
  pPacket->next = NULL,

  // Remove waiting task (if any)
  pTSS = deQueueTSS(),
  if (pTSS == NULL) {
    enQueuePacket(pPacket),
    #asm
    STI
    #endasm
    return;
    }

  // Store request in the dequeued task
  pTSS->pLBRet = pPacket,

  // Reevaluate the ready queue
  ReadyQueue->enQueueRdy(pTSS);
  pPriorityTSS = ReadyQueue->deQueueRdy(),

  // If the highest priority task is the same as the original, return
  if(pPriorityTSS == pTSS) {
    #asm
    STI
    #endasm
    return,
    }

  // Switch task if the highest priority task is not the original one
  #asm
      MOV EAX,[pPriorityTSS]
      MOV BX,[EAX Tid]
      MOV TSS_Sel,BX
      INC _nSwitches
      JMP FWORD PTR [TSS]
      STI
  #endasm
}
```

CMessageExchange Class

```
void CMessageExchange  SendMsg(long dMsgData1, long dMsgData2)
{
  TPacket *pPacket,
  TMessage *pNewMsg;

  /* Create & fill the TMessage structure */
  pNewMsg = new TMessage,
  pNewMsg->dData1 = dMsgData1,
  pNewMsg->dData2 = dMsgData2,

  /* Create & fill the TPacket structure */
  pPacket = new TPacket,
  pPacket->Req = NULL,
  pPacket->Msg = pNewMsg,
  pPacket->next = NULL,

  /* Disable interrupts */
  #asm
  CLI
  #endasm

  SendPacket(pPacket),

  /* Reenable interrupts */
  #asm
  STI
  #endasm
}
```

```
void CMessageExchange  ISendMsg(long dMsgData1, long dMsgData2)
{
  TPacket *pPacket,
  TMessage *pNewMsg,
  TTSS *pWaitTSS,

  /* Disable interrupts */
  #asm
  CLI
  #endasm

  /* Create the message structure */
  pMessage = new TMessage,
  pMessage->dData1 = dMsgData1,
  pMessage->dData2 = dMsgData2,

  /* Create the packet */
  pPacket = new TPacket,
  pPacket->Msg = pMessage,
  pPacket->Req = NULL,
  pPacket->next = NULL,


  pWaitTSS = deQueueTSS(),
  if (pWaitTSS==NULL) {
    enQueuePacket(pPacket),
  else {
    pWaitTSS->pLBRet = pPacket,
    ReadyQueue->enQueueRdy(pWaitTSS),
    }
}
```

Memory Manager Definitions

```
class CMemoryManager {
  CExchange *MemExch,
  unsigned int nPagesFree,

public
  CMemoryManager(),

  unsigned char *AllocPage(long nPages),
  unsigned char *AllocOSPage(long nPages),
  void DeAllocPage(unsigned char *pOrigMem, int nPages),

  unsigned char *AliasMem(unsigned char *pMem, unsigned long dcbMem,
                          unsigned long dProcNum);

  unsigned int DeAliasMem(unsigned char *pAliasMem,
                          unsigned long dcbAliasMem),

  unsigned int QueryMemPages() { return nPagesFree, },
},
```

CMemoryManager Implementation

```
unsigned char *CMemoryManager  AllocOSPage(long nPages)
{
  char *RunPages,

// Steps·
// ------
// 1) See if we have enough physical memory (check nPagesFree)
// 2) Find a contiguous run of linear pages to allocate (PTEs)
// 3) Allocate each physical page placing it in the run of PTEs


  // Must request > 0 pages for allocation
  if (nPages<=0)
    return NULL,

  // Ensure mutual exclusion
  TPacket *pPkt = MemExch->WaitPacket()

  // Verify sufficient pages exist to satisfy request
  if (nPages> nPagesFree)
    return NULL,

  // Find contiguous run of OS PTEs in current PT
  RunPages = FindRun(OS_BASE,nPages),

  // Allocate further PTs until request is satisfied
  While(RunPages==NULL) {
    if (AddOSPT()!=0)
      return NULL,
    RunPages = FindRun(OS_BASE,nPages),
    }

  // Mark these pages as allocated
  AddRun(RunPages,nPages);

  // Leave critical section
  MemExch->SendDummyMsg(),

  return RunPages,
}
```

```
unsigned char *CMemoryManager  AllocPage(long nPages)
{
  char *RunPages,

// Steps
// ------
// 1) See if we have enough physical memory (check nPagesFree)
// 2) Find a contiguous run of linear pages to allocate (PTEs)
// 3) Allocate each physical page placing it in the run of PTEs


  // Must request > 0 pages for allocation
  if (nPages<=0)
    return NULL,

  // Ensure mutual exclusion
  TPacket *pPkt = MemExch->WaitPacket()

  // Verify sufficient pages exist to satisfy request
  if (nPages> nPagesFree)
    return NULL,

  // Find contiguous run of user PTEs in current PT
  RunPages = FindRun(USER_BASE,nPages);

  // Allocate further PTs until request is satisfied
  While(RunPages==NULL) {
    if (AddUserPT()!=0)
      return NULL,
    RunPages = FindRun(USER_BASE,nPages);
    }

  // Mark these pages as allocated
  AddRun(RunPages,nPages),

  // Leave critical section
  MemExch->SendDummyMsg();

  return RunPages,
}


void CMemoryManager: DeAllocPage(unsigned char pOrigMem,
                                 int nPages)
{
  int ProcNum,

  // Discard unnecessary bits
  pOrigMem = (pOrigMem % 4096);
  ProcNum = GetCurrProcNum(),

  // Enforce mutual exclustion
  TPacket *pPkt = MemExch->WaitPacket()
  UnMarkPTEs(ProcNum,pOrigMem,nPages),

  // Leave critical section
  MemExch->SendDummyPacket(),
}
```

```
unsigned char *CMemoryManager  AliasMem(unsigned char *pMem,
                unsigned long dcbMem, unsigned long dProcNum)
{
  char *RunPages,
  unsigned int base,
  unsigned long CurrProc = GetCurrProcNum(),

  // Check Page Directories (ie Processes)
  if (CurrProc==dProcNum)
    return ,

  // Ensure mutual exclusion
  TPacket *pPacket = MemExch->WaitPacket()

  // Calculate number of pages required
  pMem = pMem % 4096,
  dcbMem += pMem,
  unsigned long nPages = GetAliasReqSize(pMem,dcbMem),

  if (CurrProc == OP_SYSTEM)
    base = OS_BASE,
  else
    base = USER_BASE,

  // Find contiguous run of PTEs in current PT
  RunPages = FindRun(base,nPages),

  // Allocate further PTs until request is satisfied
  while (RunPages==NULL) {
    if (CurrProc == OP_SYSTEM)
      if (AddOSPT()==NULL) return NULL;
    else
      if (AddUserPT()==NULL) return NULL,

    RunPages = FindRun(base,nPages),
    }

  // Perform aliasing
  AddAliasRun(pMem, RunPages, nPages, dProcNum),

  // Leave critical section
  MemExch->SendDummyMsg(),

  return RunPages,
}

unsigned int CMemoryManager  DeAliasMem(unsigned char *pAliasMem,
                                        unsigned long dcbAliasMem)
{
  // Calculate number of pages to be dealiased
  unsigned long nPages = GetAliasReqSize(pAliasMem,dcbAliasMem),

  // Retrieve id number of current process
  unsigned long CurrProc = GetCurrProcNum();

  // Perform call to kernel function to perform dealiasing
  return RemoveAliasRun(pAliasMem,nPages,CurrProc),
}
```

Ready Queue Definitions

```
#define PRIORITYCOUNT 32


// TTaskQueue Structure
// --------------------

struct TTaskQueue {
  TTSS *Head,
  CProcess *pProcess,
  TTSS *Tail,
},


// CReadyQueue Abstract Class
// --------------------------

class CReadyQueue {
  TTaskQueue *ReadyQ[32],
public
  CReadyQueue() {},
  virtual void enQueueRdy(TTSS *pTSS)=0,
  virtual TTSS *deQueueRdy()=0,
  virtual TTSS *ChkRdyQ()=0,
  virtual void RemoveRdyProc(CProcess *pProcess)=0,
};


// CPrioritisedReadyQueue Concrete Class
// -------------------------------------

class CPrioritisedReadyQueue : public CReadyQueue {
  TTaskQueue ReadyQ[PRIORITYCOUNT];

public
  CPrioritisedReadyQueue(),
  virtual void enQueueRdy(TTSS *pTSS),
  virtual TTSS *deQueueRdy(),
  virtual TTSS *ChkRdyQ(),
  virtual void RemoveRdyProc(CProcess *pProcess),
},
```

CReadyQueue Class Implementation

```
CPrioritisedReadyQueue  CPrioritisedReadyQueue()
{
  for(int i=0, i<PRIORITYCOUNT, i++) {
    ReadyQ[i] Head = NULL,
    ReadyQ[i] Tail = NULL,
    }
}

void CPrioritisedReadyQueue  enQueueRdy(TTSS *pTSS)
{
  // Verify the task structure pointer is valid
  if (pTSS==NULL)
    return,

  // Retrieve the priority of the current process
  unsigned long dPriority = GetCurrentPriority();

  pTSS->next = NULL,

  // If the queue is empty, this TSS becomes head and tail
  if (ReadyQ[dPriority] Head == NULL) {
    ReadyQ[dPriority] Head = pTSS;
    ReadyQ[dPriority].Tail = pTSS,
    }
  else { // Otherwise adjust the tail of the queue
    (ReadyQ[dPriority].Tail)->next = pTSS,
    ReadyQ[dPriority] Tail = pTSS,
    }
}


TTSS *CPrioritisedReadyQueue ·deQueueRdy()
{
  TTSS *cHighPri;

  // Loop through all of the queues, starting with the
  // queue with the highest priority, until a waiting
  // task is found  When one is found, adjust the queue
  // and return a pointer to it
  for (int i=0, i<PRIORITYCOUNT, i++) {
    if (ReadyQ[i] Head != NULL) {
      pHighPri = ReadyQ[i] Head,
      ReadyQ[i] Head = (ReadyQ[i] Head)->next,
      return pHighPri,
      }
    }

  // If no waiting task is found, a NULL pointer is returned
  return NULL;
}
```

```
TTSS *CPrioritisedReadyQueue  ChkRdyQ()
{
  TTSS *cHighPri,

  // Loop through all of the queues, starting with the
  // queue with the highest priority, until a waiting
  // task is found
  for (int i=0, i<PRIORITYCOUNT, i++) {
    if (ReadyQ[i] Head != NULL) {
      pHighPri = ReadyQ[i] Head,
      return pHighPri,
      }
    }
  // If no waiting task is found, a NULL pointer is returned
  return NULL,
}

void CPrioritisedReadyQueue  RemoveRdyProc(CProcess *pProcess)
{
  TTSS *pTSS,*pQueue,*pQueuePrev;

  // pQueuePrev is used to point to the node traversed
  // immediately prior to the current one. It is
  // required when a task is deleted from the middle of
  // a queue
  pQueuePrev = NULL,

  for (int i=0, i<PRIORITYCOUNT; i++) {
    pQueue = ReadyQ[i] Head,

    // If the queue is not empty, check every task within
    // the queue
    while(pQueue!=NULL) {
      // If the the current node represents a task which
      // belongs to the process pProcess, it must be
      // removed from the Ready Queue and its former links
      // must be updated
      if (pQueue->pProc==pProcess) {
        pTSS = pQueue;

        if(pPrevQueue==NULL)
          ReadyQueue[i] Head = pQueue->next,
        else
          pPrevQueue->next = pQueue->next,

        // The task structure is replaced on the heap of free
        // task structures and the system statistics are
        // updated
        pTSS->next = pFreeTSS,
        pFreeTSS = pTSS,
        pTSS->pProc = NULL,
        nTSSLeft++,
        }
      pPrevQueue = pQueue;
      pQueue = pQueue->next;
      }
  }
}
```

Interrupt Definitions

```
#define ENDOFIRQ_PERFORMED     0
#define ENDOFIRQ_NOTPERFORMED  1



// CInterrupt Abstract Class
// -------------------------

class CInterrupt {
  unsigned long IRQNum,
public
  CInterrupt(unsigned long Num),
  void GetIRQVector(char *pVectorRet),
  void MaskIRQ();
  void UnMaskIRQ(),
  void EndOfIRQ(),

  virtual void interrupt ISR(),
  virtual unsigned int Service() = 0,
},
```

CInterrupt Class Implementation

```
CInterrupt  CInterrupt(unsigned long Num)
{
  IRQNum = Num,
    SetIRQVector(IRQNum,&ISR),
}


void interrupt CInterrupt  ISR()
{
  if (Service())
      EndOfIRQ(IRQNum),
}


void CInterrupt  EndOfIRQ()
{
    EndOfIRQ(IRQNum),
}


void CInterrupt  GetIRQVector(char *pVectorRet)
{
    GetIRQVector(IRQNum, pVictorRet),
}


void CInterrupt  MaskIRQ()
{
    MaskIRQ(IRQNum),
}


void CInterrupt: UnMaskIRQ()
{
    UnMaskIRQ(IRQNum),
}
```

Timer Definitions

```
#define nTMRBLKS 32
#define ENDOFIRQ_NOTPERFORMED 1
#define ENDOFIRQ_PERFORMED    0



// TTimerBlock Structure
// --------------------

struct TTimerBlock {
  int fInUse,
  CExchange *RespondExch,
  unsigned long Tick,
},



// CTimer Concrete Class
// --------------------

class CTimer   public CInterrupt {
  struct TTimerBlock TmrBlks[nTMRBLKS],
  unsigned int nTmrBlocksUsed, TimerTick,
public
  CTimer(unsigned long Num),

  virtual unsigned int Service(),
  unsigned int Sleep(unsigned long Delay);
  unsigned int Alarm(CExchange *AlarmExch, unsigned long AlarmDelay),
  void KillAlarm(CExchange *pExch),
  void MicroDelay(unsigned long dDelay) { : MicroDelay(dDelay), },
  void GetTick() { return TimerTick, } ,
},
```

CTimer Class Implementation

```
CTimer  CTimer(unsigned long Num)   CInterrupt(Num)
{
  TimerTick = 0,
  nTmrBlocksUsed = 0,
}

unsigned int CTimer: Service()
{
  // Increase the count of elapsed ticks since the
  // timer was created
  TimerTick++,

  // Return if no timer blocks are in use, reporting
  // that the EndOfIRQ() primitive was not called
  if (nTmrBlocksUsed==0)
    return ENDOFIRQ_NOTPERFORMED,

  // Enable all interrupts except the Timer interrupt
  MaskIRQ(),
  EndOfIRQ();

  #asm
  STI
  #endasm

  // Check every timer block, notifying the appropriate
  // processes through the use of an exchange if an
  // alarm has elapsed, otherwise decrement the timer
  // block tick
  for(int i=0; i<nTMRBLKS; i++) {
    if (TmrBlks[i] fInUse) {
      if (TmrBlks[i].Tick == 0) {
        (TmrBlks[i].RespondExch)->ISendDummyPacket();
        TmrBlks[i] fInUse = FALSE,
        }
      else
        TmrBlks[i].Tick--,
      }
    }

  #asm
  CLI
  #endasm

  // Reenable timer interrupt
  UnMaskIRQ(),

  // Signal that the EndofIRQ primitive has already been
  // performed
  return ENDOFIRQ_PERFORMED;
}
```

```
unsigned int CTimer  Sleep(unsigned long Delay)
{
  if (Delay==0)
    return FAIL,

  // Find an empty timer block
  int i=0,
  while ((i<nTMRBLKS)&&(TmrBlks[i] fInUse))
    i++,

  // Return if there are no free timer blocks
  if (i>=nTMRBLKS)
    return FAIL,

  #asm
  CLI
  #endasm

  // Setup the timer block structure, using the tasks
  // default exchange as the timer block's exchange
  TmrBlks[i] Tick = Delay,
  TmrBlks[i].fInUse = TRUE,
  nTmrBlksUsed++;
  TmrBlks[i] RespondExch = GetTSSExch();

  #asm
  STI
  #endasm

  // Wait here until the dummy message is received from
  // Service, notifying that the delay has elapsed
  TPacket *pPkt = (TmrBlks[i].RespondExch)->WaitPacket(),

  // Continue processing once the delay has elapsed
  return SUCCESS,
}


unsigned int CTimer  Alarm(CExchange *AlarmExch,
                           unsigned long AlarmDelay)
{
  if (AlarmDelay==0)
    return FAIL,

  // Find an empty timer block
  int i=0;
  while ((i<nTMRBLKS)&&(TmrBlks[i] fInUse))
    i++,

  // Return if there are no free timer blocks
  if (i>=nTMRBLKS)
    return FAIL;

  #asm
  CLI
  #endasm
```

```
  // Setup the timer block structure, using the tasks
  // default exchange as the timer block's exchange
  TmrBlks[i] Tick = AlarmDelay,
  TmrBlks[i] fInUse = TRUE;
  nTmrBlksUsed++,
  TmrBlks[i] RespondExch = AlarmExch,

  #asm
  STI
  #endasm

  return SUCCESS,
}



void CTimer:.KillAlarm(CExchange *pExch)
{
  if(nTmrBlksUsed==0)
    return;

  #asm
  CLI
  #endasm

  for(int i=0; i<nTMRBLKS; i++)
    if (TmrBlks[i] fInUse)
      if (TmrBlks[i] RespondExch==pExch) {
        TmrBlks[i] fInUse = FALSE,
        nTmrBlksUsed--;
        }

  #asm
  STI
  #endasm
}
```