University of Massachusetts Dartmouth
Graduate School
Department of Computer Science

MICROKERNEL OPERATING SYSTEMS

IN

PARALLEL ARCHITECTURES

A Thesis in

Computer Science

by

Joachim Blum

Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

December 1994

Abstract

MICROKERNEL OPERATING SYSTEMS IN PARALLEL ARCHITECTURES

by

JOACHIM BLUM

In the past few years operating systems' complexity has increased substantially because of the growing number of required services which it has to provide. Many users desire an operating system which supports several different interfaces. Companies desire to use workstations for real–time applications, and everyone needs a programming and working environment which is easy to use even in a distributed memory configuration.

These demands can not be met by the 'old' monolithic operating system architecture. Therefore several groups of researchers developed more structured way of designing an operating system. They structured the services into modules and removed non–critical *code out of the kernel into servers. As a result they achieved a minimized kernel along with a separate collection of cohesive operating system services.*

In addition, it has been demonstrated in this thesis that requirements of a contemporary operating system cannot be easily met based on the monolithic design when implementation and maintenance are taken into account.

Key attributes of modern operating system design, as well as the basic ideas of a new microkernel architecture, are presented.

This thesis describes concepts of some already existing microkernels *and a specification of a new microkernel operating system, denoted as* Object–Oriented Distributed Operating System*(ODOS).*

Topics of operating systems such as inter–process communication, process management, *and* operating system emulations *are discussed in details. An overview of how ODOS handles* load balancing, real–time facilities, fault–tolerance, memory management, *and* object orientation *is given.*

Acknowledgments

I am grateful and indebted to Professor Boleslaw Mikolajczak for his constant encouragement and for many rewarding discussions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Operating systems' complexity has increased in the last few years by the added functionalities of these systems. UNIX, as an example, has pulled away from its roots as a well structured, portable, and simple operating system. Operating systems' kernels are getting so complex that it is very difficult to maintain and to improve them.

To structure an operating system, services have been moduled and a small simple kernel has been developed. This system architecture is referred to as *Microkernel* Architecture. Oftentimes no more than message passing and context switching services are offered by these kernels. The majority of operating system services are implemented outside the kernel in a server's own address spaces [GG91, Gie91].

Microkernel operating systems do not reduce the amount of code to be written to perform a specific function. They essentially provide a structured framework to reduce the complexity, to design and develop new systems, and to integrate new open system standards. This structuring provides the possibility to distribute the services among several nodes. This is of interest to the growing market of distributed systems. Microkernels are therefore very good instruments of software engineering in an operating system design. A microkernel operating system offers systems' builders the tools to develop servers. These tools were formerly available only to application programmers. Mature microkernel designs also support structured integration of hardware specific capabilities.

Modern operating systems face three software engineering challenges [Gie90]:

- to support new multiprocessing and parallel hardware architectures,
- to incorporate their own value, adding operating system features within an open system framework,

- to extend the topology and functionality of operating systems into co-operative computing environments.

A system builder should be able to add new services to an existing system. In microkernel architectures, new operating system services can be introduced by providing new servers on top of the kernel. As shown in Figure 1.1, services of the operating system are in a layer above the protected kernel. In a monolithic kernel, services are in the protected kernel.

In monolithic operating systems, servers are integrated in the kernel and therefore a new kernel would have to be introduced every time a new service is necessary. A small change in a monolithic operating system can change the whole behavior of the system. As a result of the many interdependencies, an overview of the whole system is needed, before any modifications can be undertaken.

Microkernels offer different behaviors by only changing servers. If you change the code in one server, then there is nothing to be changed in another server. The exchange of two servers is even possible during runtime. Development of system servers should be possible at user–level. When servers are implemented and tested they should be able to run in the system space to gain an improvement in performance. This is possible in such systems as: Chorus[1], Mach[2][RJO+89, RBF+89], V[3], and Amoeba[4].

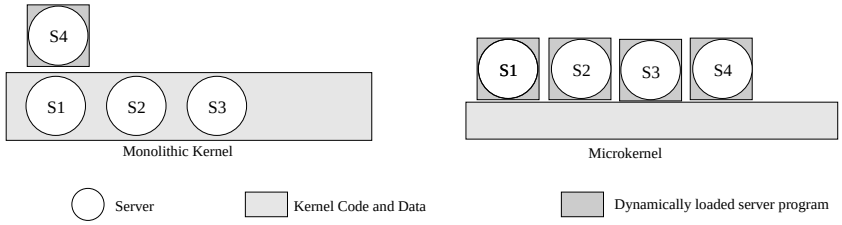

Figure 1.1: Comparison of a monolithic structure and a Microkernel

Microkernels also have the feature of executing concurrently a binary code from different systems. New personalities and existing operating systems' interfaces can be integrated without modifications into the kernel. Only the loader has to have knowledge of the new subsystem. The loader has to decide which subsystem is necessary for the process to run. If the subsystem is not

[1]Chorus is a registered Trademark of Chorus System
[2]Mach is a microkernel developed at Carnegie Mellon University, Pittsburgh
[3]V is a distributed operating system developed at Stanford University
[4]Amoeba is a distributed operating system developed at Vrige University Amsterdam

already loaded the loader loads its services as a new subsystem. For each subsystem process a runtime library is loaded into its address space.

The complexity of UNIX has increased with substantial growth in the scope of applications running on it. Since systems are merging together, UNIX has to execute all sorts of real–time applications and real–time systems need the easy interface of UNIX systems. These tendencies make improvement and maintainance of systems difficult.

New advanced computer architectures, such as multiprocessors and multicomputers, have brought a new challenge for operating system builders. Distributed environments need support for all levels of a system. The user does not want to be concerned with the hardware underlying his command interface. This interface needs to be standardized. Most users want a '*Single System Semantic*' [BGG+91], so that they only see one 'machine' even if they are working on several hundred processors.

Operating system services shall be distributed in a transparent manner. Requests for services, which are not provided on one site have to be sent to sites which provide the services. These sites process the request and send an appropriate answer back to the calling process. The calling process does not necessarily recognize that the service was provided by a remote site.

Many message passing microkernels provide location–transparent interprocess communication facilities, logical addressing, and naming which enable transparent distribution of ports and processes. The location–independent interprocess communication is the base for a transparent service distribution.

More and more systems need to be fault–tolerant. The distributed modular characteristic of microkernel architecture enables duplication of several servers for this purpose. Messages can be sent to several ports of different servers or to a port group. Even if one server crashes, other servers of this port group can substitute it. The server can recover and join the group of working servers again. For operations which need higher fault tolerance several servers can work on the same job. The results can then be compared and the most agreed solution can be delivered to the client. All these are hidden under the message passing primitives of a microkernel architecture.

In a microkernel operating system, the hardware–dependent parts of the system are only in the minimal kernel. Even though dependent code is rare, and it is well–structured into specific modules. Therefore porting to a new system is an easy and fast task. The portability allows platform builders to reduce their time to market. There is no need for developing a specific system for the new platform. Standard systems can be used on every platform. The user does not have to learn new commands and has a well–tested environment

from the beginning.

An operating system must be able to be scaled down to real–time embedded systems, to be scaled up to multi–processors and parallel architectures, and to be scalable to specialized network nodes in a fully distributed environment [Gie90]. All hardware architectures can be supported by one microkernel. This microkernel has little hardware dependent code and is well equipped to support specialized hardware.

Structure of The Thesis

In this thesis, the different components a microkernel operating system must support are described. It is shown how they are solved in mature microkernel developments such as Chorus and Mach. In addition, a new microkernel operating system is specified, which will be called Object–Oriented Distributed Operating System (ODOS). Especially, the parts of the system, which differ from other microkernels are described and some elements are implemented in pseudo–code.

In Chapter 2 new challenges which operating system builders face to support parallel architectures are described. Chapter 3 characterizes the organization and the abstraction of the microkernel. The importance and methods of implementing interprocess communication are shown in Chapter 4. New challenges for distributed operating systems like *load balancing* and *migration* are explored in Chapter 5. For success on the operating system market new operating systems have to support applications for existing systems. In Chapter 6, it is described how this is done in ODOS, Mach, and Chorus. Chapters 7 and 8 show what ODOS will support in the areas of real–time systems and fault tolerance. In Chapters 9 and 10, it is shown how memory management in distributed systems should be handled and how the operating system can support object orientation. A summary of the ODOS Operating System is given in Chapter 11. Finally in Chapter 12, the implementation status and expected future work on ODOS are described.

Chapter 2

Parallel Architectures

2.1 Hardware Architectures

In recent years, multiprocessor architectures became the answer to the design of high performance computing. Supercomputers and mainframes tend to have multiple CPUs.

There are two trends, one uses special purpose processors, which are designed for only this multiprocessor while the other uses standard components to speed up the computation. The prices of standard processors are very low and so more effort is done in the interconnection of these processors. There are two main designs of these kinds of architectures: loosely coupled multiprocessors, also called multicomputers, and shared memory multiprocessors.

The number of processors in a shared memory multiprocessor is limited since I/O and memory buses get bottlenecked as soon as the number reaches a few tens of processors. If a more advanced interconnection network [1] is used, which does not have a bus bottleneck, the interconnection then gets too expensive when more than a few processors are interconnected.

The bottleneck of communication bandwidth in buses can be limited in new designs, which give the processors their own memory. The communication network can then range from buses to hypercubes. The communication itself is an exchange of messages between the processors. Commercially available communication systems have up to thousands of processors and a good price versus performance ratio.

We are more interested in distributed multicomputer because this type of architectural design offers more scalability and fault–tolerance than shared memory multiprocessors. Processors in a multicomputer are called nodes or

[1] e.g. butterfly, or omega network

sites and there are two different kinds of nodes. First, there are nodes which have no external devices (basic nodes) and are only connected to other internal nodes through the internal network. Second, there are nodes which are connected to devices such as displays, disk drives, or external networks. These nodes are referred to as specialized nodes.

Each node can be a uni–processor or it can be an advanced multiprocessor with shared memory. This illustrates that a multicomputer is able to out–scale a multiprocessor with shared memory. The problem is to develop an operating system which takes advantage of this nearly unbounded computational power.

Such a multicomputer can be viewed as a network of workstations. The interconnections are usually more advanced than the Ethernet bus, but a cluster of workstations can also function as a multicomputer.

2.2 Operating Systems for Parallel Architectures

Operating systems have to be built to support the computational power of parallel machines. The users of these machines want to gain computational power without loosing the ease of programming and the comfort of a common programming environment. Because of the existing base of trained users and the easy use of standard operating systems like UNIX, these systems have been adapted to support multiprocessors, especially with shared memory. On these machines a parallelized instance of the kernel is running and manages load balancing among the processors. It is also easier on these machines to imitate a single system semantic.

In shared memory multiprocessors a high parallelization of the kernel has to be achieved, because several processors can be in a protected kernel state and they should not block each other. This is very difficult to achieve in a monolithic system, and also hard in a good structured microkernel.

Current operating systems for multicomputers are simple kernels which offer mainly communication features. User interaction and device accesses are handled by separate host computers, which are running their own operating system (Inmos,ICube,iPSC/2[Int87]). This makes a multicomputer a very fast 'coprocessor' for scientific computations. The goal should be, to change this view of a multicomputer. A modern operating system changes the 'coprocessor' to a real computer. A general purpose operating system is also necessary on multicomputers. This operating system must meet the needs of a multi–user environment such as UNIX offered on simpler architectures.

A single system semantic [AGP+91] enables a multicomputer to resemble a high performance computer running all kinds of applications on different

operating systems. To achieve this goal you need a *cooperative operating–environment* [Gie92], which is flexible enough to adapt to the different nature of the nodes of multicomputers (compute nodes, I/O nodes). This cooperative environment must support at least one uniform, standard operating system environment, which simplifies porting application programs to this new architecture. The final version should support binary compatibility with several operating systems and should manage the distribution of different tasks and threads without interaction with the user. The user should keep the possibility to determine the distribution, but in a typical case should not be aware of it.

The interface to the system is the same in all architectures (uniprocessors, shared memory multiprocessors, and multicomputers). The programs run on all systems without modifications. UNIX took the first step in this direction but the new requirements pulled UNIX away from it roots of simplicity and portability. The new requirements need a new underlying hardware abstraction layer [WindowsNT], which gives a common view of the hardware to the programmer and the user. The microkernel offers only a limited set of services. The main part of an operating system is built above this layer and is fully portable.

This main part should have a modular structure so that distribution [HP92] is very easy. Modules which are not necessary in one node, e.g. a device manager in a basic node, should not be loaded. This speeds up system load time, execution time and reduces the space which is necessary for the system. The nodes which do not have some modules offer the services of these modules in a location independent manner by sending messages to other nodes which offer the services. The user is not involved in the redirection of its call. Another advantage of this modular structure is the easy replacement of a server with a new or enhanced server, which provides improved or enhanced services. Also a multiplying of servers can improve fault–tolerance. Both benefits are due to the fact that all service calls are messages passed to the server.

Chapter 3

Microkernel Architectures

The whole operating system in a microkernel operating system is built of two levels. The crucial services are implemented in the lower level while operating system specific services are implemented in the higher level. The lower level is called Microkernel [Mach] or Nucleus [Chorus]. The higher level is called Subsystem Layer.

The kernel in a microkernel operating system is simpler than in a standard operating system, because of their ability to support standard programming features and services as application programs.

Operating system functions are broken down into modular pieces. These modular pieces can be configured in different ways. Some modules can even be installed separately by requirements of its service. This permits larger systems to be built in a structured way.

In microkernels most of the services that monolithic system provide are implemented in subsystem servers. The subsystem servers use the functionality of the underlying microkernel to offer their processes a common view of an operating system. One microkernel operating system can have several subsystems running at a time. The subsystems can use services of other subsystems by redirecting calls from processes of its own to other servers. For this purpose, a *Subsystem Independent Layer* is introduced in ODOS. This Subsystem Independent Layer offers services which are not crucial, but which are used by most of the subsystems. By offering this layer, a new subsystem does not have to implement all operating system functions, it can use the already provided services. Only the services, which are specific for this system have to be implemented.

Processes of different subsystems can run concurrently and send messages to each other. The processes of every subsystem assume that they are running

on a system which is controlled by a single operating system. Subsystems provide a *Virtual Single Operating System*. The processes place their system calls through system libraries, which are loaded into their address space during load–time. These system libraries redirect the system calls to the system server(s) by sending messages to the corresponding subsystem. It is not recognizable for a process of a subsystem if it runs its native operating system or if it runs on a subsystem on top of a microkernel.

The subsystems can be implemented as several servers [HP92, JCS+91] or by a 'monolithic' subsystem [GDFR90]. In case of several servers a *process manager* or a runtime library in the user space checks which server has to handle the request and sends a message to the server providing the service. In one block organization, calls are sent to the subsystem server and are handled similarly to calls in real monolithic systems.

The goal of a microkernel operating system is to build a system composed of several servers. In this case, the full range of possibilities that the microkernel provides is used. On some sites it may not be necessary to have all system servers running and to redirect calls to these servers to other sites. In addition, it is possible to reimplement one service without recompiling the whole subsystem. Mature microkernel developments try to module their services. A 'monolithic' subsystem can be a first step to provide the services by reusing already written code, but this should not be the final step.

This two–level organization provides an open operating system. Distribution is hidden from the user level. Distribution is implemented using the message passing primitive. To hide distribution Chorus, introduced *unique identifiers*. Unique Identifiers represent ports, where messages can be sent to. These ports can migrate from site to site and by knowing the unique identifier of a port a sending process does not have to know on which site the receiving process is running. ODOS adopts this idea of location independent communication through Unique Identifiers.

The ODOS Microkernel itself is responsible for Inter–Process– Communication, Scheduling, Virtual–Memory–Management, and Physical and Logical Resource Management. As it is shown in Figure 3.1, the ODOS kernel has a well–structured microkernel, which has the following services:

- a hardware–dependent **Supervisor** dispatches external events such as interrupts, traps, and exceptions delivered by the hardware to define routines or ports.

- a portable **Task Manager** handles the low–level operations upon tasks. A task is the basic entity to allocate resources such as memory and I/O–

ports. The facilities provided by this layer would normally be enhanced for applications by an intervening subsystem.

- a portable **Thread Manager** controls priority–based scheduling, thread creation, processor allocation, and provides fine–grained synchronization and low–level communication between threads. Threads are schedulable activities attached to tasks. Threads are scheduled across the available processors of a multiprocessor. To support different scheduling policies ODOS provides a way to change the scheduler.

- a mostly portable **Virtual–Memory–Manager** manages virtual memory hardware and local memory. The Virtual Memory System will be covered in chapter 9.

- a portable **Inter–Process–Communication Manager** offers asynchronous global message passing and remote procedure calls in a location–independent manner. Threads in different tasks communicate with each other by passing messages to the partner. The Inter–Process–Communication Manager is responsible for routing messages to the right receiver, which could be at a different site.

In a modern microkernel there should not be any interdependencies between different components. Chorus is a good example which has a totally modular structure even in its microkernel. Modular structure and hardware–independent code of the components make porting of a good microkernel to new hardware architectures an easy task. This includes tightly and loosely coupled multiprocessors, advanced distributed environments, and all kinds of different memory architectures. The specialized code to support hardware features is limited to several modules and is not distributed over the whole system code.

Microkernels also offer easier support of multiprocessors by creating a standard programming interface for systems. A program can take advantage of multiple processors if they are present, but the program is also able to run on uniprocessors. The specialized code for the support of multiple processors is limited to the kernel. The kernel also takes the responsibility for the network distribution of services. A multicomputer appears as a 'normal' multiprocessor. The user of this interface is not responsible to optimize the communication part of its program. He can concentrate on the implementation of a solution to his problem.

Microkernels provide a convenient method of supporting real–time features presently needed in a growing number of applications, such as multi–media,

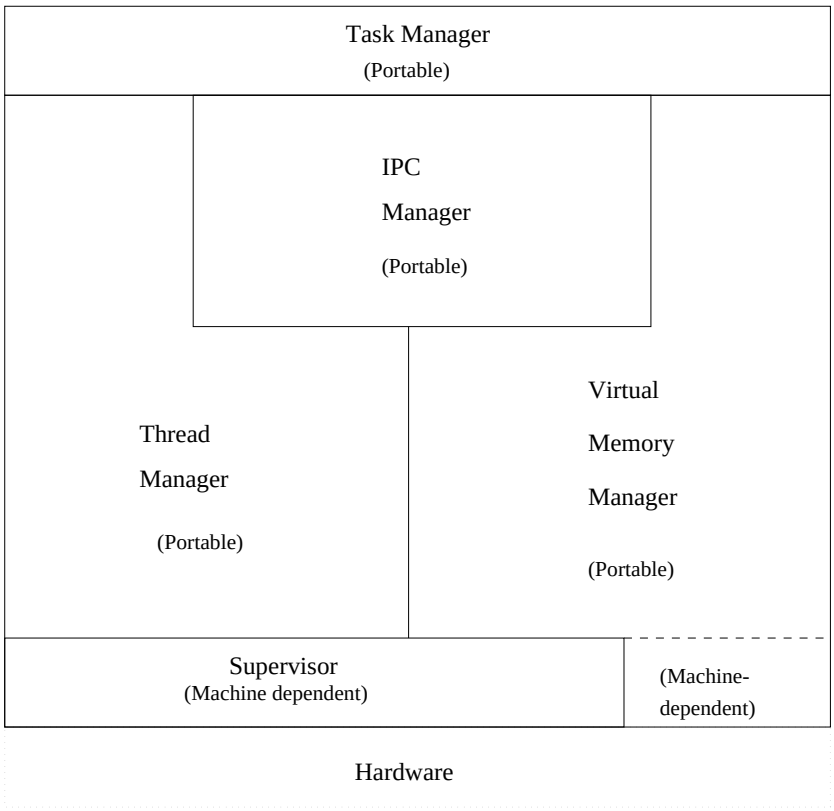

Figure 3.1: Microkernel Modularity

device control, and high speed communication. This is due to the relatively small amount of time microkernel based systems are spending in kernel mode. Real–time systems support is covered in Chapter 7.

3.1 Unique Identifiers

Unique identifiers (UI) are the addressing units for all objects in ODOS. They are unique in space and time. The idea of unique identifiers was inherited from Chorus where all objects (actors, ports, and segments) have unique identifiers.

In contrast to Chorus, the unique identifier in ODOS is the base class (Figure 3.2) in the new object–oriented system. In ODOS every object does not have a unique identifier, every object is a unique identifier. Every object is derived from the class *unique identifier* and inherits all the properties of

this base class. Figure 3.3 shows the definition of the class Unique Identifier. ODOS guarantees, similarly as Chorus, that a unique identifier only exists on one site and will never be reused.

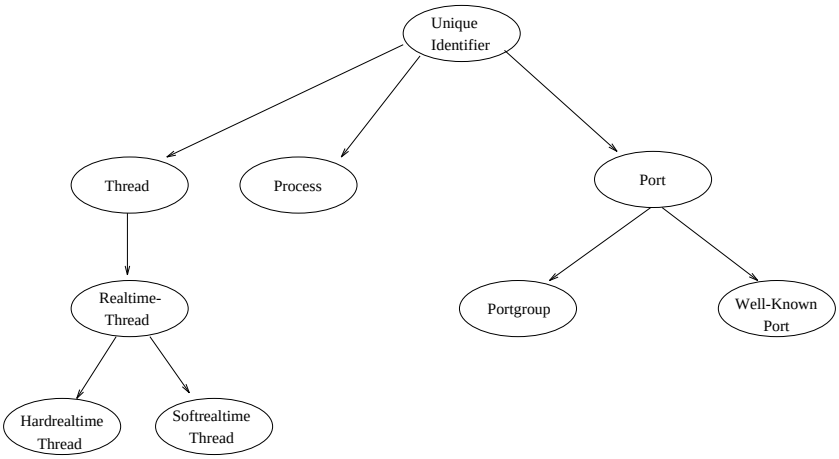

Figure 3.2: UI Class Structure

It uses the same length of 128 bits as the Chorus Nucleus. The construction of the identifier includes site number and time of creation. The site number in the identifier makes it easier for the location service algorithm to locate the actual site of UI. UIs can migrate from site to site. Each site keeps a list of last recently used UIs and their corresponding site numbers, if the number differs from the creation site of the UI. At least the actual site, the creation site, and usually the previous site know the exact location of the UI. If a UI migrates the location information saved in the creation site is changed. Even if some sites still think the UI is at the 'old' site, they can still send their message to the previous site, which recognizes that the UI is no longer present at the site and routes it to the site it thinks where the UI resides.

If the receiving site recognizes that the message was rerouted, it sends a broadcast message to all sites in the domain. All sites change the entry in their hash tables for that UI. Future messages do not have to be redirected.

Unique identifiers are used for addressing all objects in ODOS and for the communication between threads, ports, and tasks.

constructor

```
unique_identifier()
```

```
class     :
        unique_identifier
        // basic abstractions for communication and
        // processes
inherit   :

features  :
        ui_number_t ID (private)
        actual site number (private)
        messagesqueue (private)
methods   :
        unique_identifier()
        ~unique_identifier()
        migrate( siteno_t sitenumber)
        receive( header_t header, msg_t msg)
        send( header_t header, msg_t msg)
        getID()
        getsiteno()
        givesendright( ui_number_t ui)
        givesendonceright( ui_number_t ui)
        deletesendright( ui_number_t ui)
        deletesendonceright( ui_number_t ui)
```

Figure 3.3: Class unique-identifier

unique_identifier() initializes an object of class unique_identifier(). A 128 bit unique identifier is created out of the site number, where this object is created, and the creation time.

destructor

`~unique_identifier()`

~ unique_identifier() destroys an object of class unique_identifier. The unique identifier turns to a *deadname* and future messages to that UI are rejected and sent back as a *deadname* message.

migrate

```
migrate(siteno_t sitenumber)
```

migrate() migrates a UI to site *sitenumber* in the same domain. The site *sitenumber* must exist.

receive

```
kernelreturn_t receive(header_t header,msg_t msg)
```

receive() gets the first message from the message queue. Other objects in the domain can send messages to a UI if they have *sendright* to this UI. These messages are received by the *receive()* method. If no message is in the queue, then *receive()* waits until a message is delivered. When returning the *header* includes information about the sender and some other characteristics of the message and the *msg* is an untyped data–stream.

send

```
send(header_t header,msg_t msg)
```

send() sends a message to another UI in the domain. The receiver is named in the *header*. The *msg* is a untyped data–stream. Its length is mentioned in *header*.

getID

```
ui_number_t getID()
```

getID returns its own UI number. The UI number is a 128 bit number, which identifies the UI. It is unique in time and space and it is used for addressing a UI in ODOS.

getsiteno

```
siteno_t getsiteno()
```

getsiteno() returns the site number the UI is residing. A UI can *migrate* from site to site and so this number can be different from the number encoded in the UI number.

givesendright

```
givesendright(ui_number_t ui)
```

givesendright() gives the *ui sendright* to the own queue. The *ui* can send as many messages as it wants. The *sendright* can be deleted by *deletesendright()*.

givesendonceright

```
givesendonceright(ui_number_t ui)
```

givesendonceright() give the *ui sendonceright* to the own queue. The *ui* can send one message to the queue. After sending the message the *sendonceright* is deleted. The *sendonceright* can be deleted by *deletesendonceright()*.

deletesendright

```
deletesendright(ui_number_t ui)
```

deletesendright() deletes the *sendright* for *ui*. The *ui* is no more allowed to send messages to the UI.

deletesendonceright

```
deletesendonceright(ui_number_t ui)
```

deletesendonceright() deletes the *sendonceright* for *ui*. The *sendonceright* is deleted automatically if the *ui* sends a message. So, only if the object wants the *ui* removed from its list of UIs which has *sendoncerights*, it has to call *deletesendonceright()*.

3.2 Well–Known Ports

A capability is the unit of data access control in Chorus [RAA+90]. Some objects are not directly implemented in the Chorus Nucleus. They are implemented in external servers. To access these objects Chorus defines capabilities, which are global names for the services. A capability in Chorus is implemented as a concatenation of a UI and a key. The UI names the server, which manages the object, and the key addresses the object and access rights within this server.

ODOS uses a similar idea. It extends it to take fault–tolerance into account. Services which are not directly provided by the microkernel are not very widely used, because they are not known. This will change in ODOS.

Services which are usually offered by an operating system are known by all processes. This is not possible in a microkernel operating system, because service addresses can change. To minimize this deficiency ODOS provides *well–known ports*. Well–known ports are ports which UIs are fixed even after a restart of the service. In addition the service can be accessed by a name, which describes the service. UIs of these services can be asked similarly to the *gethostbyname* call of the internet protocol.

The greatest advantage of this well–known port service is the service–ports defined in ODOS, which are not directly provided by the kernel. The UI of these ports are always the same. Even if a system or a process crashes, as soon as a message is sent to a well–known port the service is recovered and the service is provided. The services of the basic operating system are extended without loosing the good structure.

Each device in ODOS is represented as a well–known port. So threads can send messages to devices, without knowing, which task will handle the request. Even if the task which is responsible for managing a request for a device is not loaded, ODOS loads the task and accepts the messages at its port.

The *well–known ports* do not have a site identifier. The first few bits are 0's. Well–known port numbers can be looked up by sending a message, which describes the service, to the well–known port naming process. The list of services and UIs is saved in a fault–tolerant manner. Some ports are fixed for all installations of the system. Others can be included into this list. Since ODOS offers a configuration of well–known ports, therefore it is possible to customize for services the system needs. Locations of well–known ports are distributed at every system's period. Therefore a direct communication can appear and the port naming process is not a bottleneck of the system.

Well–known ports derive the basic services from unique identifiers. The class definition is shown in Figure 3.4.

constructor

```
well_known_port(pathname_t afilename)
```

well_known_port() initializes an object of class well_known_port. It sets the *filename* to *afilename*. The file *filename* is called when a messages is sent to the well–known–port.

destructor

```
~well_known_port()
```

```
class     :
        well-known-port
inherit   :
        port
features  :
        filename
           // file name of server which
           // provides the service.
           // if the server is not loaded */
        status
           // status of server, which provides
           // the well-known service
methods   :
        well_known_port(pathname_t afilename);
        ~well_known_port();
        pathname_t getfilename();
        setfilename(pathname_t afilename);
        status_t getstatus();
```

Figure 3.4: Class well-known-port

~ *well_known_port()* destructs an object of class well_known_port.

getfilename

```
pathname_t getfilename()
```

getfilename() returns the name and path of the executable file which will be executed when someone calls the well–known port.

setfilename

```
setfilename(pathname_t afilename)
```

setfilename() sets the *filename* of the file which is loaded in case of a call to *afilename*.

getstatus

```
status_t getstatus()
```

getstatus() gives back the status of the server, which serves the well–known port. It can either be *busy, loaded, or notloaded.*

3.3 Sites

A site is a group of tightly coupled physical resources[1], which are controlled by one microkernel. Sites in one system are connected by a communication medium[2].

Every site is controlled by one microkernel. For a good cooperation among several sites the microkernel of each site declares some ports where messages are received by the microkernel. Theses ports are used for communication among the microkernels for establishing load balancing, location independent interprocess communication and other services.

Every kernel has as its own port ID, the site number and some well–known ports, which are fixed for all system implementations, but can be expanded if kernels offer new services to other kernels. These ports can not migrate.

Operating system services do not have to be established on each site. Due to the location–independent communication it is possible to load some services only on some sites without any changes to the application programs. Sites can be grouped into domains, which cooperate in load balancing and parallel execution of tasks. This grouping is configuration–dependent. A system administrator can configure it on request. Sites of one domain cooperate so closely that they seem to be a single system. This *single system semantic* is a feature which was introduced in Chorus/MiX to make the use of a parallel domain easier. When a user starts a program in one domain, he does not have to care on which site the program is executing. This is automatically done by the system. The kernels check for the most suitable site. This depends on the load of each site and the native binary code of the program. Different processor architectures can be in the same domain, so the kernel will route processes of one architecture to the appropriate site, which supports the required architecture.

3.4 Tasks and Threads

A modern operating system manages the execution environments for the programs by providing multiple tasks and threads [CD88, Dea93, DBRD91]. In

[1] one or more processors, central memory, and attached I/O devices

[2] e.g. Ethernet, token rings, transputer links, hypercubes, or high speed busses

earlier operating systems the unit of execution (threads) and the unit of resource allocation (tasks) were combined into a single process.

New architectures have capabilities to support parallel execution, therefore the operating systems have to support parallel execution too. To support multiple units of execution within one program the unit of execution (threads) is extracted from the unit of resource allocation (tasks).

Modern architectures also support the ability to distinguish between several address spaces. In mature microkernels such as Mach and Chorus each task has its own address space. The operating systems assign memory objects owned by the task to ranges of addresses within the address space.

The task is the unit of resource allocation and protection with tasks being assigned capabilities and access rights to the IPC facilities of the system.

In order to support parallel execution of a program within a single address space, the execution environment is separated from the actual streams of computation.

The streams of computation are called threads. Thus, a multi–threaded program can be loaded into a task and execute in several different places at the same time on a multiprocessor or parallel machine. This results in a better application performance.

For better load balancing, ODOS supports, in contrast to Chorus and Mach, task and thread migration. The effect of this feature is that an unbalanced system can move workloads from one site to another. The uniform address space provides an optimized memory management in the distributed environment. Non–local memory accesses are handled as page faults and the missing page is brought to the asking site on demand. A hardware delivered page fault is first checked by the microkernel and if it can not be served by the kernel the handler for this memory object is called. For more details see chapter 9.

3.4.1 Threads

Threads (Figure 3.5) are the basic units of computation. They run in the environment of exactly one task, which is the basic unit of resource allocation. For any manipulation of resources (except the virtual memory), threads have to call system traps and these traps manipulate the resources, if the calling thread has the authority to do so. A thread operates by executing instructions in sequential order. During execution of these instructions it can trap into the kernel. In the kernel mode, threads are allowed to execute privileged instructions.

Several threads can be created. The creation of a thread has very low

overhead. In multiprocessors threads are able to run in parallel with other threads of the same task [TRG+87, Dea93]. The scheduler releases threads depending on their priority. Every thread is an independent unit of execution. A thread can wish to run in parallel with other threads, but this is only guaranteed, if both threads have the highest priority. A thread context switch is much less expensive than a task context switch, because in a task context switch the address space has to be changed. Therefore the scheduler prefers threads of the same task to run in successive order.

```
class     :
        thread
inherit   :
        unique-identifier
features  :
        stack
        priority (private)
        register state (private)
        program counter
        execution state (waiting on, ready, running,..)
               (private)
        resource (scheduling) parameters (private)
        statistics (private)
        // accounting information
        owner
        // pointer to the task of its computational
        // environment(private)
        suspendcount (private)
methods   :
        thread()
        ~thread()
        suspend()
        migrate(siteno_t sitenumber)
```

Figure 3.5: Class thread

constructor

`thread()`

thread() initializes a new thread in the task, where the calling thread resides. The new thread becomes the same priority as the creating thread. All other variables become initial values. After creation the thread is put into the list of threads of the task. To create a thread from a remote task (parent or kernel) a message is sent to the control port. The control thread creates the new thread and puts its priority to the standard priority level of threads of this task.

It is visible from the class definition that the data which have to be manipulated during the creation of a thread are smaller in size than data whichhave to be manipulated during the creation of a task. Therefore the overhead of creating a new thread is very small compared to the overhead of creating a new task. In addition, scheduling of another thread of the same task is very inexpensive comparing to an address space change. This fact is taken into account when the kernel schedules a new thread; namely, local threads are scheduled preemptively.

destructor

`~thread()`

A thread is terminated by calling ~ *thread()*. All thread–specific variables should be removed. This can be executed by each thread of the task. A thread termination of a remote task is done by sending a message to the control port. The accounting information contained in *statistics* are passed to the task. During the runtime of a thread the accounting is done for the thread. When the thread is destroyed the accounting information is transferred to the task and the task sums up the accounting information for each of its threads.

suspend

`suspend()`

A thread can be suspended by calling *suspend()*. This suspends the thread from being scheduled. A thread can be suspended more than once. The *suspendcount* must be equal to zero to be scheduled again.

migrate

`migrate(siteno_t sitenumber)`

As mentioned ODOS supports migration of threads. This is done by calling *migrate()*. The actual working set of memory pages of the thread is moved with the thread to the new site *sitenumber*. After migration the thread is scheduled on the new site. All missing pages are brought in *on demand*.

3.4.2 Tasks

A task (Figure 3.6) is a collection of resources in a microkernel. It owns an address space in which its threads can execute. This address space is split into user and system space. The system space on a given site is shared by all tasks and these tasks usually can only manipulate this part of the address space through special system calls. ODOS supports three different access rights for tasks:

- *regular user task*: A regular user task resides in user address space. It is only allowed to call public microkernel operations. All manipulations of resources, except its own user address space, are done by system calls.

- *trusted user task*: A trusted user task is also resident in user address space. It is allowed to call more sensitive system calls. A trusted user task can be installed in system space and becomes in this configuration a system task.

- *system task*: A system task resides in the system space. It is a trusted task in the system space and it can directly manipulate the resources. To ensure the modularity of the system a *system task* should make calls through clearly defined interfaces. This makes it easier to reconfigure the system and to move a system task out of the system space. The main advantage of moving them into system space is an improvement in performance and the direct manipulation of hardware, such as I/O devices. In contrast to the kernel system tasks, although they reside in the system space, they can be paged out just like a regular user task.

In order to support access to I/O devices, the microkernel provides access to I/O resources such as memory–mapped devices, I/O ports and direct memory access (DMA) channels, as well as the ability to reflect interrupts to device drivers executing in user space. As mentioned above, a task can execute in a trusted manner. Therefore system tasks can own devices and are typically device driver servers.

Interrupts are typically delivered to the most privileged mode of a system. The microkernel has to handle them or transfer them to trusted tasks. ODOS offers a function for this delivery. Tasks that want to get interrupts must provide a thread which waits for the interrupt handling. As soon as the microkernel delivers the interrupt, the thread begins to process the interrupt.

```
class     :
        task
inherit   :
        unique-identifier
features  :
        trustlevel:(user,trusted,system) (private)
        list of threads (private)
        list of ports (private)
        address space (private)
        default execute parameter for include threads
                (private)
        semaphores or other synchronization objects
                (private)
        statistics
        // accounting information
methods   :
        task(vm_inherits)
        ~task()
        migrate(siteno_t siteno)
        suspend()
        resume()
        getsendports( ui_number_t *sendports,
                      ui_number_t *sendonceports)
        lock(semaphore_t semaphore)
        unlock(semaphore_t semaphore)
        getobject(ui_number_t ui)
```

Figure 3.6: Class task

constructor

`task(vm_description_t vm_inherits)`

A new object of class task is created by *task()*. The creation of a task can be divided into three stages.

- choose the site on which the new task shall be created,
- create a new execution environment: create a new address space, and a new control port, and
- create a new control thread for the control port.

The choice of the creation site is dependent on hardware requirements of a task and the load of sites in a domain. The new address space can share address pages with the creating task. Other pages can be copied. The information which pages are copied or shared is passed in the parameter *vm_inherits*. The returned value indicates a failure if the parameter is invalid or there are not enough kernel resources available.

destructor

`~task()`

A thread in a task can terminate its own task by calling *~ task()*. The kernel or the parent task can terminate the task by sending a message to its control port and the control thread will call the *~ task()*. When a task is terminated all threads which are running inside the task are terminated too. The ports are deleted and appropriate actions[3] are done for all terminating objects.

migration

`migrate(siteno_t siteno)`

As mentioned previously it can be necessary to migrate one task to another site. This is done by *migrate()*. This method can be called by any thread in the task. Kernel and parent task can send a migration message to the control port, which initiates the control thread to call this method. A task migration moves all entities which are built during task creation and all ports, where the migrating task has *receiverights*. Virtual memory is migrated only in parts, since the execution of threads on the old site is still possible (See Chapter 5 for details).

[3] e.g. deadname notifications for ports are sent to tasks, which wanted to be notified

suspend

`suspend()`

All threads of a task can be suspended by calling *suspend()*. If a task is suspended the corresponding method *thread::suspend()* of each thread is called. The only thread which runs in a suspended task is the control thread. It still receives messages which are sent to the control port and can act in an appropriate way. Therefore the kernel and parent can suspend and resume the task again. A task can be suspended more than once. No thread is allowed to run until it is resumed as often as it was suspended.

resume

`resume()`

All threads of a task which were suspended can resume again by calling *resume()*. The corresponding method *thread.resume* is called for every thread. The *suspendcount* of each thread is decremented and if it is zero the threads are allowed to run again. Resuming can only be done by the kernel or parent, because the only thread, which is able to call resume is the control thread, which listens to the control port where only parent and kernel have *sendrights*.

send port information

```
getsendports(ui_number_t *sendports,
             ui_number_t *sendonceports)
```

As mentioned a task is a unit of resource allocation. Ports are very important resources. There are *sendrights* and *receiverights* of ports. To get information about which ports the task has sendrights, *getsendports()* is called. This method returns two lists of ports, one for *send* the other for *sendonce* rights. A list of *receiveright* ports is included in the task data structure.

lock

`lock(semaphore_t semaphore)`

lock() locks the *semaphore*. A semaphore is an object which is protected by the kernel. *lock()* will block until the semaphore is available. All threads which try to lock a locked semaphore are kept in a list of *blockedthreads*. The programmer is responsible for unlocking the semaphore again.

unlock

`unlock(semaphore_t semaphore)`

unlock() unlocks the locked *semaphore*. If there are threads in the list of *blockedthreads* the first thread of this list is resumed and this thread will lock the semaphore again. If there are no threads blocked on the *semaphore* the semaphore is free.

getobject

`ui_object_t getobject(ui_number_t ui)`

getobject() returns a language specific pointer to the object with number *ui* if the object is part of the task. If it is not part of the task, then *getobject()* returns NULL. The lists held in the task are lists of ui_numbers. To get the address of the real object you can call *getobject()*.

3.5 Load–time Compilation

ODOS is an operating system for heterogeneous distributed systems. Usually in such systems you need system binaries for all different kinds of hardware architectures. ODOS is designed to reduce this unnecessary redundancy by defining generic RISC architectures on which system binaries can be compiled. Disk access time is very slow comparing with processor execution time. If a process is loaded into memory, then the processor can translate this abstract code into hardware binary code. The generic RISC architecture, which ODOS defines, is very similar to most RISC architectures available currently on the market. Therefore the translation (between generic and specific RISC architecture) is an easy and fast task. Time is not lost during the translation and the abstract code can be used for all architectures.

The above feature is similar to Taos [Pou94] operating system, which defines an abstract RISC architecture to run on heterogeneous networks. Taos expands the idea, assuming that even the kernel is not written in a native code. The only module which is in a native code in Taos is the loader. All other modules are translated during the load time. Taos' developers need on average six man–months to port Taos onto a new platform. Most of this time is spent creating a new loader, which translates all binaries into the native code of the hardware.

This way of loading binaries can be extended to binaries that are compiled for other hardware architectures. During the load time, the binary code is

translated into the binary code of the actual hardware. If both hardware architectures are similar then the translation is not difficult. The speedup which is achieved by translating binaries before running is high comparring with the speedup that can be achieved during the step by step translation during the run time. Since it runs the real native code, therefore the only difference is that the program is possibly not optimized for the specific hardware.

The modules which offer the service of load time translation can be loaded separately and be extended to all varieties of hardware combinations.

Chapter 4

Inter–Process Communication

The microkernel IPC system provides the basic mechanism that allows threads running in different tasks to communicate with each other. The IPC system supports the reliable delivery of messages on ports. Ports are protected channels between tasks and are addressed by their unique identifier. Therefore ODOS–IPC is location–independent. ODOS provides three different kinds of port rights. First, only one task can hold *receive right* for a port. Any thread of the task which is holding the receive right can receive messages from this port. Second, there can be several tasks which have *sendrights* to a task. *Sendrights* can also be given to *world*, which means that every thread in the system has the right to send messages to that port. Third, there exists the possibility to give tasks the right to send once to a port. This is called *sendonceright* and is very useful in client–server architectures. The information, which tasks have *sendrights* to a port is held in the task itself. This is in contrast to Mach [BBB+90, Bar91, Ber92, Dra90, WT88a, WT88b], where a sending task stores its own *sendrights*. Only one task keeps the *receive right* on a port, but several tasks can have *sendrights* to a port.

For multiple receives of different tasks, ODOS supports port–groups. These port–groups have a receiving mode, which defines how incoming messages are handled. Messages can be delivered to exactly one port–group member or to all of them to provide broadcast and fault tolerance. The members of a port–group can either be ports or other port–groups.

A task which holds the receive right for a port informs other tasks what kind of *sendrights* they have for this port. The information about *sendrights* of a task is held in the port rights table. This is done to reduce the network traffic. The port rights table is of limited size and therefore the last recently used UIs are stored only.

A thread in a task can inquire at the local port rights table whether it has *sendright* to a specific port. Alterantively, it can issue a request to receive indentifications of all ports to which it has *sendrights*. The response is limited only to the information stored in the task. The method used to determine all ports to which a task has *sendright* checks only the local port rights table. No broadcast operation is performed to receive information about the *sendrights* of the task. To get all valid information about *sendrights* ODOS provides an extra broadcast message, which asks all receivers to send back a message if the task has a *sendright* to a port of its members. These two functions are separated because the broadcast messages and the reply messages cost a lot of communication time; this would increase the network traffic.

A task does not have to know all ports to which it has *sendrights*. Tasks can have *sendright*, but there can be no entry in this list. This is especially happens when the receiving ports have *sendrights* for *world*. The list is implemented as a cache, which keeps the last recently used UIs. Messages are only sent if the task has *sendright* to the destination port or if there is no entry in the list. If a task tries to send a message to a port, where it does not have a *sendright* the receiving task sends back a *noauthorization* message. After receiving this message the *sendright* table in the sending task is checked and invalid entries are corrected.

The kernel also provides a high performance method of passing large data areas in messages. In shared memory systems ODOS does not copy the data, because the message contains a pointer to them: this is referred to as a pointer to *out–of–line data*. A more detailed description is shown in the section describing the optimization techniques for IPC/RPC.

To provide modularity, the kernel is defined as a task with several threads. These threads communicate with each other mostly by using IPC. This makes it much easier to configure a system for its specialized services. Most of the ODOS system services are implemented as IPCs to the kernel rather than as direct system calls.

4.1 Communication in ODOS

4.1.1 Messages

A message is a collection of data, which is passed between two entities. By using messages for communication between entities, ODOS can provide the same syntax both for local and remote communications. The local communication is optimized in the kernel and therefore invisible to the user.

A message contains a small fixed size header and a data part of varying size. Large messages in shared memory are passed by marking the pages where the data of the message resides as *copy on write*. The message body can contain regular data, references to *out of line data*, and a port right, which are passed to a task to get a *sendright* for a specific port.

The header includes:

- *notification bit*: A bit which is set if the sending thread wants to be notified, when the message is actually received by the receiving thread. The IPC sends this notification back to the sender.
  It is also used if a port dies. In this case all senders having messages in the queue and having set the notification bit, will be notified with a dead name notification.
- sender's UI
- receiver's UI
- response's UI (port where a response is expected)
- indication what kind of send rights the receiver has for this port
- message id: necessary to identify duplicated messages and to identify the message for future communication between the sender and the receiver.
- kind of message: out of line data, regular message, port right
- size of message
- checksum : corrupt data is identified by comparing the received checksum with the computed checksum of the received messages. If a message has been corrupted, then the sender will resend the message after getting informed that the message was received incorrectly.

4.1.2 Ports

A port is a destination address of a send operation. Messages are sent to that port. A port is often called a mailbox. Associated with the port is a message queue, where received messages are stored until a thread in the task with a receive right for that port wants to receive a message. Therefore *asynchronous* message exchanges are possible in ODOS. ODOS guarantees the delivery of the messages to the message queue. It does not guarantee the receiving of

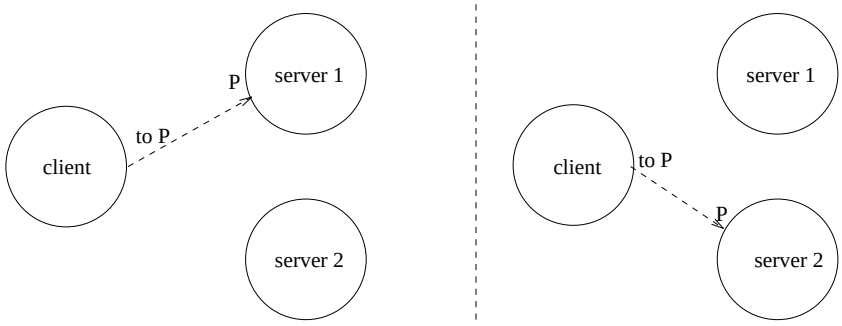

Figure 4.1: Port Migration

messages by the receiving task. A task can possess several receive rights for ports. The message queues in a port are *first in first out* buffers.

The class port is derived from class' *unique identifier* and therefore derives all properties of a UI. Due to the migration of UIs a port can migrate from site to site. The sending tasks do not have to know the exact location of a port.

The port can even migrate from one task to another (Figure 4.1). This happens by transferring the *receive right* of a port to another task. During the port migration both tasks have to participate actively in this operation. This port–migration offers an opportunity to dynamically reconfigure a system without a *shutdown* of the whole system. A port can migrate from one server to another. All the messages are kept in the message queue. Therefore the task which holds the receive right can start to receive messages, which are already in the queue. Figure 4.1 shows a port–migration in a graphical way. Figure 4.2 shows the class definition of class port.

```
class     :
        port
inherit   :
        unique-identifier
features  :
        ui_number_t* sendonceports
        // list of ports, which have sendoncerights
        ui_number_t* sendports
        // list of ports, which have sendrights
        thread_t* waitingthreads
        // list of threads, which are sleeping,
        // waiting for a receive msg
        ui_number_t* ownertask
        // task which owns the port
methods   :
        port()
        ~port()
        receive( header_t header, msg_t msg)
        send( header_t header, msg_t msg)
        status( long num_msg, long maxsize_of_msgqueue,
                ui_number_t* sendright,
                ui_number_t* sendoncerights)
        migrate(ui_number_t sitenumber)
```

Figure 4.2: Class port

Constructor

`port()`

port() creates an object of class *port*. The task where the calling thread resides becomes the *ownertask*, which has the *receive right*. The list of *sendonceright* ports and the list of *sendright* ports is empty. The list of waiting threads is empty.

```
port::port() {
   initialize sendonceports to empty
   initialize sendports to empty
   initialize waitingthreads to empty
   initialize ownertask to owner of calling thread
}
```

Destructor

`~port()`

~ port() destroys an object of class port. A *dead–name message* is sent to all ports, which are in the lists *sendonceports* and *sendports* and have set the *notification bit*. All threads which are waiting to receive messages from the port are released. The returned value they get is *deadname*. The port is deleted from the ports' list of the *ownertask*.

The destructor of the port class has to check several states. As mentioned above a check for the value of the notify bit among messages of the message queue has to be performed and appropriate actions have to be undertaken.

```
port::~port() {
   set state of port to dying
   while waitingthreads not empty
      release thread with return value deadname
   mark UI as dead-name
   for all msgs in message queue
      if notify-bit is set
         send msg-sending port a dead-name message
      remove msg out of msg queue
   notify all ports which have sendoncerights
      with dead-name-message
   notify all ports which have sendrights with
```

```
      dead-name-message
   remove port of port-list of ownertask
}
```

receive

```
kernelreturn_t receive(header_t header,msg_t msg)
```

receive() gets the first message from the message queue. Other objects in the domain can send messages to a port if they have *sendright* to this port. These messages are received by the *receive()* method. If there is no message in the queue, then the *receive()* method waits until a message is delivered. When returning, the *header* includes information about the sender and some other characteristics of the message. The *msg* is an untyped data–stream.

```
kernelreturn_t port::receive(header_t header,msg_t msg){
   while not getmessage(msg){
      /* checks the local queue for messages. If there
         are no messages and the port is a member of a
         port-group it tries to get a message from the
         port-group queue */
      put thread in waitingthreads
      sleep on event msgreceive
   }
   if notifybit(header)
      send received notification to sender
}
```

send

```
send(header_t header,msg_t msg)
```

send() sends a message to another UI in the domain. The receiver is named in the *header*. The *msg* is an untyped data–stream. Its length is mentioned in *header*. If the *send* method (Figure 4.2) in ODOS returns a successful value, it guarantees the acceptance of the msg in the message queue of the receiving process, but it does not guarantee that the receiving process is really receiving the message. A receiving process has to call the *receive()* to get the next message in the queue. If it does not call *receive()* before it dies, the message is never delivered. Before a message queue is emptied during the quitting phase,

the messages are checked for the notify bit and a *dead–name* message is sent to the tasks, which want to be notified (notify bit is set).

The *send()* method is used to send messages to another port in the system.The caller is blocked until the message is queued into the message queue of the recipient. For better performance a hash table for the most recently used UIs is kept in each kernel. It is not guaranteed that the entries in this table are correct, but usually at least the table of the site, where the UI resided last, and the actual residing site have a correct entry. A site reroutes the message if it is not the receiving site. If a UI was not used for a long time, it can happen, that even the table on the site, where it last resided, might not have an entry for this UI any longer. In this case a broadcast message is sent to all sites in the domain. The site, where the UI resides, answers to this message and the original message can be sent to the receiving site. For more information see the explanation of the thread class, which checks for incoming messages.

The *send()* waits for the *acceptance message* of the receiving site. If the message is accepted the *send()* call returns with a value of success, otherwise it returns with an error.

The *send()* method of the port class works as follows:

```
kernel_return_t port::send( header_t header, msg_t msg)
{
   if ownertask.sendright(header.destination_ui)
      /* returns true if no entry in the sendright
         table of the task exists, which implies that the
         task has no sendright to the UI. If it has
         sendonceright this call automatically removes the
         right*/
     then
       siteaddress= UI_hashtable (header.destination_ui)
          /* returns the entry for the UI in hash-table,
             otherwise the Site number, which is
             encoded in the UI */
       send msg to siteaddress
         /* send message to the site address */
       wait until receiving site sends answer.
         /* waiting for a response from the receiving
            site. If there is no response after a
            certain period of time, the send is
            repeated. */
```

```
        if msg accepted by the receiving site
           return(success)
        else
           return(deadname or noright)
   else
        return(noright)
}
```

status information

```
status( long num_msg, long maxsize_of_msgqueue,
        ui_number_t* sendright,
        ui_number_t* sendoncerights)
```

status() returns information about the status of the port. The *num_msg* keeps the actual number of messages in the queue. The queue has the maximum size of *maxsize_of_msgqueue*. The *sendright* and *sendoncericht* is a list of UIs which has *sendright* and *sendoncericht* to the port.

migrate

```
migrate(siteno_t sitenumber)
```

migrate() migrates a port to site *sitenumber* in the same domain. The site *sitenumber* must exists.

receiver thread

On the receiving site the microkernel runs a thread which checks for incoming messages. This thread is called when the kernel receives a message. This thread has a loop with the following body:

```
while true do
   wait until a message arrives
     /* here this thread waits after it has queued a
        message */
   get message header
   if destination_UI is resident
     if msg was rerouted because of wrong hash table
        broadcast actual site of UI
     if sendingUI has sendright
```

```
      if sendingUI has sendonceright
         remove sending UI from sendonceright list
      if typeof(destination_UI) is portgroup
         if receiving mode is broadcast
            broadcast to all member ports
         else
           store msg in address space. If fault
              tolerance is required then store
              msg in redundant spaces
      else
         queue msg into the msg-queue
         notify sending thread about queuing
         wake up threads sleeping on msgreceive on
            this port
   else
      send back noright
else
   if UI dead
      send back deadname
   else
      look in hashtable for actual site of UI
      send msg to site
```

4.1.3 Port–groups

The exchange of single messages from one thread in a task to a group of tasks is not the best model of communication. This kind of communication is often necessary to provide fault tolerance in a system. For this purpose a *multicast message* is more appropriate. Sometimes threads should not be aware how and on how many servers a specific service is performed. This redundancy should often be invisible to the user process. A multicast message is a message which is sent from one sender to a group of receivers. Multicast messages provide a useful tool to provide fault tolerance by replicating servers.

ODOS offers for this purpose a class *port–group* (Figure 4.3), which has the ability to broadcast messages to other ports. For a sending thread this class looks like a regular port, which is addressed by its UI. Therefore sending threads do not have to know what kind of structure is underlying the unique identifier.

The class of port–groups is also used to offer a runtime reconfiguration. Ports can dynamically join and leave port–groups. If a service is associated

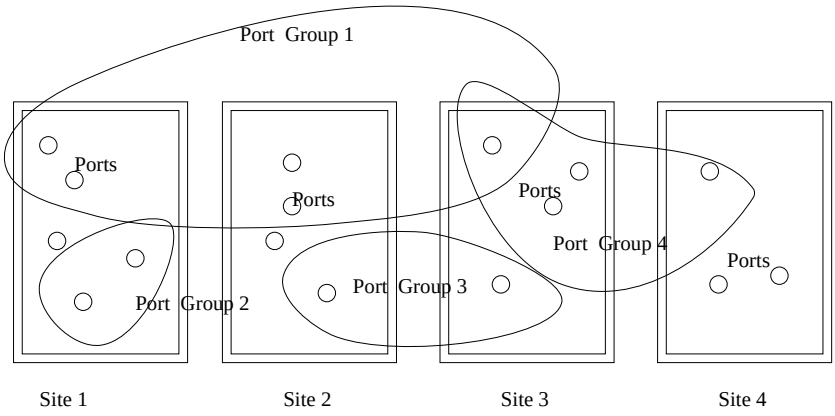

Figure 4.3: Ports can be members of Port–groups

with the UI of a port–group, then the servers which have receive rights for port–group members, will receive request for this service. A newly developed server can join this group of workers and if they have shown reliability during a test phase, then the old servers can leave the group and the new generation of servers offers the service without a noticeable change for clients.

Port–groups (Figure 4.4) also offer a second receive mode. Not all services need fault tolerance. Sometimes clients are only interested in quick responses to their requests. To provide a quick response a server can be replicated several times. Each of these servers work on different requests and when they are ready they respond and get further requests for service from the port–group. For this purpose port–groups can have a receive mode which indicates that only one member gets the message. The messages are kept in a queue until a server requests the receiving of a message.

```
class     :
        portgroup
inherit   :
        port
features  :
        ui_number_t* sendonceports
        // list of ports, which have sendoncerights
        ui_number_t* sendports
        // list of ports, which have sendrights
        ui_number_t* portgroupmembers
        // list of ports, which have members of the group
        receivemode
        // receiving mode:
        // broadcast (fault-tolerance), single delivery
        address space
methods   :
        portgroup ()
        ~portgroup ()
        add_port(ui_number_t ui)
        remove_port(ui_number_t ui)
        changemode(portgroup_rec_right_t mode)
        status( long num_msg, long maxsize_of_msgqueue,
                ui_number_t* sendright,
                ui_number_t* sendoncerights,
                ui_number_t* memberports,
                portgroup_rec_right_t  receivemode)
```

Figure 4.4: Class portgroup

Constructor

`portgroup()`

port() initializes an object of class *portgroup*. The inherited features are initialized by constructing the *port* class. The *portgroupmembers* is empty. The *receivemode* is set to single delivery.

Destructor

`~portgroup()`

An object of class *portgroup* is destroyed by ~ *portgroup*. All lists are destroyed and for messages, which are still in the queue, appropriate actions are performed.

adding ports

`add_port(ui_number_t ui)`

add_port() adds a port to the list of ports which are receiving messages from this port–group. It depends on the *receivemode* if this added port will now get all messages or if it is allowed to ask for messages.

removing ports

`remove_port(ui_number_t ui)`

remove_port removes a port *ui* from the port–group.

changemode

`changemode(portgroup_rec_right_t mode)`

changemode changes the *receivemode* of a port–group. The *receivemode* is changed to *mode*. *Mode* can either be *broadcast* or *singlereceive*.

status information

```
status( long num_msg, long maxsize_of_msgqueue,
        ui_number_t* sendright,
        ui_number_t* sendoncerights,
        ui_number_t* memberports,
        portgroup_rec_right_t  receivemode)
```

status() returns information about the status of the port–group. *num_msg* keeps the actual number of messages in the queue. The queue has the maximum size of *maxsize_of_msgqueue*. The *sendright* and *sendonceright* is a list of UIs which has *sendright* and *sendonceright* to the port–group. Also *memberport* is a list of ports, which can receive messages which are sent to the port–group. The *receivemode* is the mode which controls how incoming messages are handled.

4.2 Remote Procedure Calls

A special kind of interprocess communication activities are *Remote Procedure Calls(RPC)*. Remote procedure calls have many similarities to ordinary procedure calls. Whenmaking procedure calls, the caller gives control to the called procedure and gets the control back, when the procedure has finished and returns its result. The difference between an ordinary and a remote procedure call is that the remote procedure is part of another thread. This thread normally runs in another task, which has a different address space. The call and the parameters have to be packed into a message and send to the remote task. A microkernel operating system is built out of independent blocks. Services of one block are needed by other blocks. This makes a remote procedure call very important for a microkernel architecture. Badly specified or implemented RPC would decrease the performance of the whole system.

Problems with Heterogeneity

ODOS is designed to be an operating system which can unify several different hardware architectures into one system. Unfortunately, different processor types and even different programming languages use different representations for data they manipulate. A RPC protocol has to define single representation, which is common for all communications throughout the system.

Server and client stub procedures are implemented in the server and client, respectively. The stub procedures offer to the program the same interface as they would have for local procedure calls. The program itself is not involved in the remote procedure call. The stub procedures format the parameters and send them via the IPC to the partner stub. The partner stub reassembles the parameter into its own representation and returns as it would return if it would be a regular procedure call. The calling thread is not aware that the call is not handled locally.

At run–time the client stub will use a named server to locate the location of the procedure it wants to call. After it knows the location it can communicate

directly with the stub of the server. On the server, each module which exports a procedure for RPC must be ready to receive calls. Each remote procedure call is registered at the *name server* to enable clients to ask for their location.

Problems Resulting from Partial Failure

A big difference between an ordinary procedure call and a remote procedure call is that in the remote case the callers environment can fail independently from the callee's environment. One system may crash during a call and therefore a mechanism must be implemented to inform the calling thread that its call was not successful. In languages like Ada the system can raise an exception, but in languages like C the system can only inform the thread by returning with an error return value.

4.3 Optimization for IPC and RPC

4.3.1 Out of Line Data

For shared memory parallel computers the kernel optimizes the transfer of huge amounts of data between two address spaces by transferring the data as *out of line data*. When the kernel transfers the message from the sender to the receiver, it makes the memory region being passed, to disappear from the sender's address space and to appear in the receiver's address space. If the kernel transfers messages by out of line data it does not have to copy the data to the protected kernel address space and then back to the other user address space. So, the speed of the data transfer is increased, because there is at most one copying of the data from internal data spaces to an address space, which can change the ownership. An even greater speedup can be achieved if pages are marked as *copy on write*.

4.3.2 Copy on Write

Most virtual memory systems support a shared ownership of a page of memory. In this case the page where data are stored can be marked as *copy on write*, which means that the page is copied as soon as one of the owners try to write on this page. This is done by the virtual memory system. The kernel only has to set the bit, which marks the page as copy on write. This reduces the cases in which the data has to be copied. Since each copying would cost a significant amount of time, the speed of communication between two address spaces is increased dramatically by this method.

4.3.3 Light–weight Remote Procedure Calls In Shared Memory

Bershad *et al* [BAEL90] have shown in a study of some systems that most RPCs are cross–address space invocations within one shared memory. Especially modular operating system design such as in multi–server personalities increases the number of local cross–address space calls. In their study they developed an optimization of the RPC in shared memory called *Light–weight Remote Procedure Call*. They showed that most RPCs are cross–domain and not cross machine RPCs, and most parameters in RPCs are rather simple than complex data structures. Their implementation is based on optimizations in data copying and thread scheduling.

They use four techniques to achieve a performance increase compared to regular RPC:

- *Simple control transfer.* The client's thread executes the requested procedure in the server's domain.
- *Simple data transfer.* The parameter passing mechanism is similar to that used by procedure call. A shared argument stack, accessible to both client and server, can often eliminate redundant data copying.
- *Simple stubs.* LRPC uses a simple model of control and data transfer, facilitating the generation of highly optimized stubs. Usually, RPC has to be much more general than a cross domain RPC, since it takes all cross–machine RPCs into account. This stub overhead is eliminated in LRPC. LRPC checks at the first statement in the stub if it is a local or remote call. If it is a remote call it jumps to the regular RPC otherwise it uses LRPC.
- *Design for concurrency.* LRPC avoids shared data structure bottlenecks and benefits from the speedup potential of a multiprocessor.

The data has to be copied four times in a regular RPC. From client to kernel, from kernel to server, from server to kernel, and from kernel to client. The LRPC minimizes this copying of data to one copy. The parameters are copied to a part of a region in the memory, which is shared by the client and the server. The same region is used by the client and the server stub. There can be more than one communication channel in one shared region. Several client threads can call the server at a time.

Unfortunately, the location–independent IPC is sacrificed in Bershad's implementation of LRPC. In ODOS, the *stubs* have to check through a system

call if the call can be satisfied locally or on a remote site. It sets a bit to know for future invocation how a RPC will be handled. Due to the fact that UIs can migrate from site to site the situation can change. If a call is unsatisfactory the bit has to be changed and the invocation method has to performed as a regular RPCs.

4.3.4 URPC in Shared Memory

The overhead of switching into the protected kernel mode is not necessary. Particularly in multiprocessors with shared memory, where both the client and the server can run on different processors, a processor reallocation to the kernel and back is an overhead, which should be eliminated. Bershad, Anderson, Lazowska, and Levy developed a new *Userlevel Remote Procedure Call*[BAEL90] for Shared Memory Multiprocessors. They came to the conclusion that the communication on the user level is in most cases faster then the communication through the kernel.

If you build the communication above the kernel then the kernel becomes even smaller as it would be without the URPC. This increases the throughput, because if one processor is in kernel mode all others would be blocked at the kernel entrance. The second advantage is the higher speed of user–level communication. A user–level communication increases flexibility and speed of a multiprocessor operating system.

These researchers mentioned in their work devoted to LRPC [BAEL90] that the majority of Light–weight Remote Procedure Call's Overhead (70 percent) is due to the fact of involving the kernel in the communication. The time is a little bit more than invoking twice the kernel with necessary reallocation of processors. If this overhead could be avoided, then it would be a great success. Cross–address space communication at the user–level has higher performance because of:

- Unnecessary processor reallocation is eliminated. Direct (the call itself) and indirect (future failures in cache and TLB) overhead is reduced.
- Messages are sent between address spaces directly without copying them first into the protected kernel space. This idea was also applied in LRPC.
- When processor reallocation does prove to be necessary, its overhead can be amortized over several independent calls of different threads of a process.
- The inherent parallelism of sending and receiving a message can be exploited, thereby improving call performance.

Another aspect of operating systems is a secure communication. This secure communication is provided by the user–level remote procedure call (URPC) as described below. URPC divides the communication in three installments:

- *processor reallocation* : ensuring that there is a physical processor which can handle the client's call in the server and the servers response in the client.
- *thread management* : blocking the caller's thread, running a thread through the remote procedure in the address space of the server, and resuming the caller's thread when returning.
- *data transfer* : moving the data between the two address spaces.

Out of these three steps only the processor reallocation requires kernel interaction. Thread management and data transfer are done by the application level libraries. This approach is even more radical than the 'traditional' microkernel approach, where thread management and interprocess communication are handled in the kernel. The kernel is therefore responsible only for allocating processors to address spaces (tasks). In conventional message systems the three components are located together under the kernel interface. Even for uniprocessors with multi–threaded processes, the URPC is faster than traditional communication.

Implementation

In traditional operating systems, communication between different address spaces is only possible through well–defined narrow channels in the kernel. RPC is supported through these channels.

URPC also has logical channels. These channels guard a pairwise shared address space between server and client. The channels are mapped once for every client/server pair and are used for all communications between these two address spaces. Thread management is implemented entirely at the user level. A thread sends messages to a thread in another address space by queueing the message into the queue of the common address space. The calling thread is blocked until the answer is received. The cross–address space procedure call is a synchronous operation in the programer's view. As long as the calling thread is waiting for an answer, the thread management system switches to another thread, which also could use the URPC facility. The processor is kept busy until the answer from the called thread in the other address space is received.

Switching within one address space has a very small overhead compared with processor reallocation. Processor reallocation requires changing the mapping registers that define the virtual address space context in which the process is executing. There is additional overhead incurred as a time needed to reload entries in TLB and cache, and the scheduling time to make a decision to which address space the processor should switch. Processor reallocation is sometimes necessary in URPC. This happens only if a server does not have enough processor resources or the client does not have enough threads to fulfill the time break when the kernel switches a processor to the server thread, which handles this service. In URPC the special low priority threads in the runtime library are responsible for detecting incoming messages. These threads can then unblock the blocked calling thread.

In URPC the cross–address space call is divided into three independent components. In the following subsection these three components 'Processor Reallocation', 'Data Transfer', and 'Thread Management' are described.

Processor Reallocation

Using an optimistic reallocation policy the URPC attempts to reduce the frequency by which the OS reallocates the processors. URPC assumes the following:

- The client exists as several threads, out of which at least one is ready to run. Therefore the client can run in another thread and does not have to be deallocated from the processor.
- The server has or will have soon a processor to handle the client's call.

This optimistic policy leads to a very high performance, because of inexpensive switches within a single address space during a cross–address space call, and because of the parallelism of the client and server paradigm. It is even possible that the already running thread will also call a procedure in the server. Therefore no time is wasted. If the server does not have a processor then there can be several calls from the client before the OS makes a processor reallocation to the server, who also can work for several requests before deallocation.

Kernel–level communication and thread management facilities use a pessimistic processor reallocation policy. They are unable to exploit concurrency within an application. Especially in multiprocessors with shared memory aggressive multi–threading is an approach for higher speedup. In this pessimistic view of the scheduling there is a high overhead of communication, because of

this reallocation of the processor to the server (Hand–off scheduling). In traditional implementation of the kernel, the resources are physically distributed over several processors but at the same time are logically centralized. This distribution can be used by the URPC for higher throughput. The bottleneck of centralized kernel data structures is due to the contention for logical resources (locks) and physical resources (e.g. memory). By distributing the communication and thread management functions to the address space that uses them directly, the contention is reduced. This approach is also appropriate in uniprocessors with multi–threaded applications.

Unfortunately, the optimistic assumptions do not always hold (e.g. single–threaded applications, real time applications). In this case URPC has to involve the kernel to reallocate the processor to another address space. In this case, URPC allows a client to cause a processor reallocation, even if there are runnable threads in the clients address space.

URPC implements the following return policy in the server: return the processor to the client, when all outstanding messages have generated replies or when the server determines that the clients' address space is underpowered.

Load balancing is also done by the URPC runtime library. The processor's reallocation policy is that no high priority thread waits for a processor while a lower priority thread is running.

Data Transfer Using Shared Memory

Safe interaction can be guaranteed between mutually suspicious subsystems by the use of shared memory message channels. Client and server can still overload each other. The safety of a communication based on shared memory in URPC is the responsibility of the stubs.

The arguments of a call are passed in message buffers that are allocated and pair–wise mapped during the binding phase, that precede the first cross–address space call. By the use of stubs and standard runtime facilities it is no longer necessary to have the kernel copy data from one address space to the other. This was necessary when programmers dealt with raw data in the form of messages. Pair–wise shared memory can be used to transfer data between address spaces more efficiently, and as safe as messages through the kernel. These points motivate the use of pair–wise shared memory for cross–address space communication.

Data flow in URPC through bidirectional shared memory queues with Test–and–Set locks on either end. These locks are non–spinning to prevent processors from waiting indefinitely.

Cross–Address Space Procedure Call and Thread Management

Cross–address space communication and thread management are very closely related. They need to interact because when calling a remote procedure the thread has to be stopped and restarted when the answer is delivered. High performance thread management facilities are necessary for fine–grained parallel programs. While thread management facilities can be provided either in the kernel or on the user level, high performance thread management can only be provided at the user level. The close interaction between communication and thread management can be exploited to achieve extremely good performance for both, when both are implemented together at the user level.
There are three different cost categories of thread management in programs:

- *heavy–weight threads*: they are kernel supported. There is no distinction between heavy–weight threads, the dynamic component of a program, and its address space, the static component. Threads and address spaces are created, scheduled and destroyed as single entities. Operations on heavy–weight threads are costly because of the amount of data that has to be manipulated.

- *middle–weight threads*: Address space and thread are decoupled, but the thread is still managed by the operating system. A single address space can have several middle weight threads. This is the way by which threads are handled in ODOS.

- *fine–grained threads*: Many threads in one address space. Managed by the runtime system of the process itself. Very low overhead, because very often no saving of several registers and no integrity check is necessary.

A kernel–based thread management system has to protect itself from malfunctions. The kernel has to check arguments and access privileges. Light–weight threads are scheduled on two levels. First by the kernel in their middle or heavy–weight context and then by the runtime library within their context. The cost of thread management operations in a user–level scheduler can be orders of magnitude less than for kernel–level threads.

The performance of URPC is dependent on several factors. In general, we can see that latency and throughput are load–dependent. When the number of caller threads (T) is greater than the total number of processors (number of server processors (C)+number of client processors (S)), then call latency increases, since the additional threads have to wait for a processor. Throughput is less sensitive to T, but it increases until $T = C + S$, because then all processors are busy and cannot service other threads.

As long the number of threads is sufficiently large to keep the processors busy the message processing can be done in parallel like in a two–stage pipeline. Contention in the critical section in which the thread management system and the message queue is handled, limits the throughput over a single channel. This critical section is half of the instructions which are required for a round trip. Therefore with four processors the critical section is nearly always occupied.

With URPC functionality is pushed out of the kernel and performance and flexibility are both improved. In some cases, like in a *NULL* procedure call, the LRPC is better than URPC. This is due to two facts:

- Processor reallocation in URPC is based on LRPC. LRPC is the general cross–address space procedure call facility for kernel–level threads.
- URPC is integrated with two–level scheduling. LRPC does not make any scheduling decisions. Performance is decreased if you use both layers of scheduling. But we can assume that the first layer scheduler is seldom called. Only in the worst case studies it is advantageous to use a scheduling mechanism, which only uses one level.

URPC shall be implemented in ODOS. It is an extension which seems to be very important to achieve the RPC performance. Therefore ODOS, the microkernel operating system which has to have more RPCs than monolithic operating systems, should have a level of performance which will be at worst the same as of a monolithic system and which offers much better structure and extensibility than its monolithic counterpart.

Chapter 5

Load Balancing and Migration

This chapter describes the relationship between load–balancing and migration of processes. Load–balancing is used to get an equal load on all sites of a domain. For this purpose it is sometimes necessary to migrate one or more threads/tasks to another site. Therefore load balancing uses migration to achieve a fair load among all sites. Migration can also be used for another purpose, which will be explained shortly.

There are several reasons why a process should have the possibility to change its execution site. Migration can be used for the purpose of [SG94]:

- *Load balancing*: Distributing tasks and threads across sites of a system to even the load of each site.
- *Hardware preferences*: In heterogeneous systems some processes can be coded as being hardware–dependent and will not be able to run on other hardware configurations or otherwise face unacceptable decreases in speed.
- *Data access*: It is sometimes more efficient to bring the computation to the data than the data to the computation. This is the case if the data which is being used is enormous in size; then the transferring of the data is more expensive than the transferring of computation.
- *Computation speedup*: If a single process can be divided into several subprocesses, then these subprocesses can execute in parallel on different processors. This minimizes the runtime of the process. This speedup has its source in the design of the process. ODOS therefore supports several

threads in one task. The execution can be done in parallel to speedup of the whole task.

ODOS is a highly portable microkernel. It shall support all kinds of different hardware architectures. Also heterogeneous systems are supported. Binary code from one microprocessor architecture has to be executed on a processor which can execute this code. Other processors need an extra translation and have therefore an overhead which can be eliminated if the binary code is executed on the native processor. By migrating the process to the native processor a better throughput can be achieved.

On parallel and distributed systems there is always the problem that some processors have more work to do than others. A load balancing algorithm must be efficient enough that no processor is idle at any time. Therefore a load balancing algorithm should result in an improveed system performance. If there is work to do in the specific domain, then the work should be migrated to the idle sites being able to execute the code.

Different Scheduling Strategies

There are three different load balancing strategies which an algorithm can follow [Mul93]:

- *Throughput*: One can optimize the throughput of a system. Throughput is the amount of useful work, which could be done in a time unit. It is maximized by minimizing *overhead, communication, and waiting time*. Overhead is the time which is spent by allocating and deallocating processes plus the processor time used to make the scheduling decision. Communication time is the time which is spent to wait for data transmission. Waiting time is the time spent idle waiting for work while there are processes waiting for a processor elsewhere in the system.

  This policy tries to schedule especially longer jobs.

- *Response time*: In interactive systems we are interested in a good response time. This is the time between an input or creation of a process and an output or termination of a process. A scheduling algorithm tries to minimize the sum of response times and so attempts to run especially short processes.

- *Fairness*: Tries to divide the processor resources fairly to all processes. This policy tries to switch the context as often as possible to give each process a fair part of processor resources. This maximizes overhead and increases response time.

These three strategies are not simultaneously possible. A system has to check which policy is the best for most of its purposes and get the policy which is best for it. Many schedulers are often able to combine all three methods.

Placement of Communicating Processes

There is also another job for the scheduler. The scheduler has to place communicating processes on processors which are directly connected. Therefore the scheduler has to know the topology of the Interconnection Network and also the communicating partners of a process.

In a multi–threaded system like ODOS the communicating partners are at least the threads of one task. All these threads use the same virtual memory. It is therefore clear that they can run on different processors of one shared memory multi–processor [Bla90b, Bla90a, Wen87]. They can also migrate from one processor to another. In the shared memory system they can access memory pages randomly.

A difficult situation is when threads are spread out onto different sites and not all memory pages are in the local memory. Threads which operate on the same memory page have to be placed on one site. In opposite case every time a page is demanded it has to be transmitted from one site to the other. This is too expensive and gaining a little parallelization is not worth the extra cost incurred. Threads which access different memory pages do not have to be placed on the same site. Their interaction is limited to message exchanges and therefore they can use the regular IPC of ODOS.

The memory management in ODOS provides a distributed shared memory. This means that pages of one task can be spread out on several sites. If a page is demanded on one site, where it does not reside, the memory management handles this request in the same way as it handles the request for an *outpaged* page. A closer description of distributed shared memory is given in the chapter 9.

Memory Size and Process Clustering

A scheduling system has to take the size of primary memory into account. If the schedulable processes cannot fit into primary memory then outpaged processes have to be brought in each time when they are scheduled which reduces the response time dramatically. A scheduler should therefore prefer processes which are already in memory. Processes can be clustered into classes which are scheduled separately. First processes of the first class, then processes of the second, and so on. This separates page faults to mostly the time when

the clusters are changed. At that time there are peaks of page faults, but for other times the page fault ratio is much smaller than on not clustered systems. User–interactive processes can be members of several clusters and cluster changes can therefore be done infrequently. This expands the time when one cluster is executing. The number of cluster switches and the number of page faults during cluster switches is therefore reduced.

A fair distribution of execution time is only provided in long terms. If this is not acceptable the time between cluster switches has to be decreased. The cluster switch interval is configurable during runtime.

Migration

If it is necessary to move one process to another processor then this has to be decided carefully. The rule must always be that the benefit of migration must outweight the cost of the migration. It does not make sense to migrate one process to another processor if this process does not still have to execute for quite a while.

Most processes in present operating systems, like UNIX, run for a very short time. These processes are not worthy of migration. For this class of processes the distribution algorithm has to place the processes to the best suitable processor at the creation time. For few processes, which run for a longer time then regular processes, special migration policies can be developed. The question is how to identify the long running processes. One good estimate is the elapsed run time so far [Mul85]. Another method is that the user tells the system the estimated runtime. In ODOS this is done by hand. During a task/thread creation one can tell the system that a task/thread will run for a longer time and the scheduler shall migrate the task/thread to another site if load balancing requires a migration.

There are two different ways a process and its memory can migrate from one site to another. The goal of both is that the process can execute as long as the memory is transmitted. This results in a smaller migration time and therefore the process does useful work during that time. The strategies are quite different, but the effect is very similar. The two strategies are:

- Researchers at CMU [Zay87] showed by transmitting only the state of a process and the current working set of pages that the amount of memory, which has to be transmitted can be reduced and that the process can continue to execute after a very short time. Pages which are needed later were transmitted on demand. This method leaves the data of one process distributed over different sites and is therefore sensitive to processor

crashes. The advantage is that only the pages which are needed have to be transmitted.

- Researchers at Stanford [TLC85] did the same from the opposite direction. The process continues to run on the old host and pages are copied to the new one. When a page is copied it is marked as clean; when the running process modifies this page then it is marked as dirty. When the first round of copying is completed the dirty pages are copied again. After this round the state of the process and the process itself are sent to the new host. All pages which were marked dirty during the second round of copying are now transferred and the process can start again. Using this method the suspended time is very small and the process can start running after a small break. The advantage here is that the data are not distributed across several processors and the failure of one processor does not result in a lost of data. The disadvantage is that pages which are no more needed are copied and the pages in the active working set can be copied up to three times.

Memory shortage is one reason for a process migration. Zayas's (CMU) method keeps memory occupied much longer in the old host than Theimer's (Stanford) method.

A process migration is always very expensive. Therefore some researchers doubt that a migration is worthwhile. In their opinion a good load balancing can be achieved by a careful selection of the initial executing processor. ODOS supports a process migration in special cases.

- First, if a processor has to be freed, because of a service situation or that the processor is needed for other purposes such that a user is logging onto a workstation.

- Second, a process is expected to be running for a long time, e.g. server processes of subsystems. In this case processes are marked as processes which can be moved. This has to be done carefully because the communication overhead should not increase in the same amount as the time won by migration.

Load Balancing at Startup Time

Initial load balancing can be done during the creation of new threads or tasks. In practice, operating systems usually execute processes for a short period of time. Therefore a good initial placement of processes is very important for an equal load on all processors.

In Taos operating system load balancing is done during thread/task creation. When a new thread/task is created the load balancing algorithm checks if any of the surrounding processors have lower load than the processor where the creation is initiated. If so, the processor does not demand to load the task/thread and sends the thread to the less loaded processor. This processor also checks again, if any surrounding site has a lower load. The task/thread is finally created at the site, where the load is the lowest. The task/thread goes down a *'load mountain'* like water running down to the lowest point in its environment.

Chorus handles load balancing in a much simpler way. Implementation of Chorus/MiX for multi–computers [HP92] introduces two methods by which processes execute on a remote site.

At programming level each process has a default execution site. This default site can be changed with the *csite* call. When a new process is created the process will be created on the site being the execution site of the process; this site executes the *fork or exec* system call. Parallel programs can therefore be written by using *csite, execv*, and *fork*. All the processes communicate by using the Chorus IPC.

At the shell level the user can execute processes on a remote site by a newly introduced command. The command

remex -s[sitenumber] command

executes the command on the remote site. A user can execute a second shell on a remote site.

This enables a process to do load balancing. It can check the load of a remote site and can send remote executions to the sites, which are rarely in use. It is the easiest and the least expensive method of implementing load balancing support by an operating system. The location–independent IPC of Chorus supports all kinds of load balancing and migration, but it is not yet implemented.

The *Single Site Semantic* [AGP+91](see Chapter 6.4.1) is a good example how to achieve the advantage of a multiprocessor performance without changing user's skills in using the system.

Chorus/Mix uses the IPC to execute servers on remote sites. This enables the system to support processes of one subsystem, even if the demanded service is not available, on the local site. This reduces redundant servers and decreases the memory shortage of one site.

ODOS inherits the initial load balancing strategy from Taos. If a site execution number is not specified then the site with the lowest load will be

chosen and the new process will be created on that site. Threads are created preemptively on sites where other threads of the task already reside. If a thread placement results in a high page fault rate then this thread will be replaced by the system. The system tries to group together threads which use the same memory pages.

Ports are created on sites, where threads of the receiving task reside. In opposite case, the network load would increase and the throughput would decrease.

Migration in ODOS

As described in earlier chapters ODOS supports a migration of Unique Identifiers. Due to the fact that all ODOS' classes are derived from UI, all objects are movable from one site to another. ODOS uses (for task migration) a method which is similar to the method of Zayas [Zay87]. Due to the modular structure of ODOS this migration policy must be replaceable. Migration in ODOS is done differently if an object is a task, thread or a port. For ports the whole port is migrated, for threads the actual working set of pages is migrated, and for tasks the task specific information is migrated.

Each ODOS kernel has a *well–known port*, where neighboring kernels send their actual load ratio. If the domain is unbalanced the involved kernels can decide to migrate tasks or threads. In the case of memory shortage on one site ports/port–groups can be migrated to sites, which have plenty of memory accessible. The migration decision must involve the location of the threads of the task which hold the receive right for a port. A migration of a port away from the sites, where the receiving threads reside, can result in a higher network load and can therefore decrease throughput.

Tasks which provide well–known services usually run for an extended period of time. Even if such tasks are suspended or not used at all they remain in memory. These tasks are very good examples of long–term processes; therefore it is worth to migrate them from site to site. The IPC in ODOS is location–independent. There is therefore no change for threads which use the service of a well–known port. A well–known port is usually a port group, which can stay on the site, even if the task is migrating. Not only well–known servers but also servers which deliver services for well–known ports are good examples of processes suitable for migration.

Chapter 6

Support for Personalities

Users do not want to change the way they use computers. Even if they change to new computers or new operating systems they still want to use their old programs and have their familiar user interfaces. Users are trained on special programs and it would be too expensive for them to learn new programs. Changes in the user interface can therefore not be too radical. Programs often exist only for special platforms and porting them to other platforms is not possible. Companies react in the way: *'never change a winning team'*. Since there are so many different user and system interfaces on the operating system market, it would be beneficial for a new operating system to support several different 'personalities'.

Personalities are not only user interfaces or shells which have familiar commands, they are also whole 'emulations' of existing systems. System calls have to be handled in the same fashion as they are done in the original operating system. The user can run programs from his old operating system without recompilation. That means that a user can run Macintosh[1], Windows[2], or UNIX[3] programs on various platforms. These operating systems have different traps into their kernels and are totally incompatible. The goal shall be that all programs of all operating systems can run concurrently and be able to exchange information between each other.

Many contemporary operating systems support different emulations. These emulations are libraries which redirect system calls of the emulated operating system to system calls of the native operating system. The redirections are time consuming and often difficult. An emulation is usually slower than the

[1] Macintosh is a trademark of Apple Computers
[2] Windows is a trademark of Microsoft
[3] UNIX is a tradmark of USL

native operating system and therefore is of limited usefulness.

A support of different personalities in microkernel–based operating systems is provided in different manner. A microkernel does not have a native operating system interface. The microkernel only offers basic abstractions and most of the services of an operating system have to be implemented in the user space. What seems to be a disadvantage is actually an advantage. All services have to be programmed for all operating system interfaces. Emulations do not use any additional time. Personalities on top of a microkernel are equally fast as personalities in traditional OS. Although personalities can share services, they do not need to (see Personality–Independent Layer below). Special services of one operating system do not have to be implemented in another one. For examples the *print service* can be shared by all personalities on top of one microkernel.

The biggest problem is binary compatibility to existing systems. All binary files compiled for a conventional version of an operating system should run correctly and without modification. To achieve this goal the following requirements have to be met:[CDK94]

- *Address space layout*: The emulation has to provide the regions expected by the program. If the code is non–relocatable, the machine instructions assume that regions such as the program text and the heap occupy certain expected address ranges. Address space regions such as the stack must grow as necessary.
- *System call processing*: All correct system calls must be handled in the same way as the emulated system does.
- *Error semantics*: All error case responses have to be the same as in the emulated system. If a system call is used with wrong parameters then the emulation has to pass back the predefined messages. Even if the error would raise an exception in the emulating system it must handle errors as they are handled in the emulated system.
- *Failure semantics*: The emulation must not introduce a new system call failure modes.
- *Protection*: User data and the emulation system itself must not be compromised.
- *Signals*: Signals must be generated and the user level handlers have to be called as appropriate when a program causes an exception such as an address space violation.

Emulations are only required to run on those nodes which are currently running processes of the emulated operating system.

6.1 Subsystems

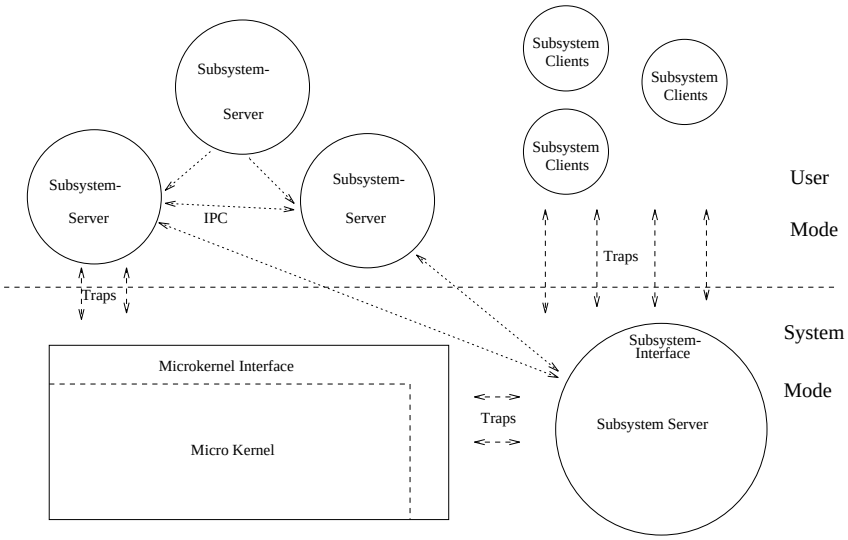

Figure 6.1: Integration of a server into system space

Subsystems are representations of personalities in a microkernel architecture. Subsystems are located in a layer between the microkernel and the application programs. They have to handle the system calls of the applications which are normally handled by the native operating system. Subsystems have a complete standard operating system interface to standard application programs. They export high–level operating system abstractions, such as process objects, protection modes, and data providing objects.

Different Subsystem Concepts

There are different concepts for subsystems in existing microkernels like Chorus or Mach (Section 6.4). In the Chorus architecture (Figure 6.1) subsystems are sets of system servers that use the generic services of the Chorus Nucleus to provide higher–level services. The servers, within a set, communicate via the IPC facility of the nucleus. The servers are well–structured and give system builders a well–defined way to manage operating system complexity.

Servers can be replaced during runtime. The first versions of Mach–UNIX personalities were 'monolithic' subsystems (section 6.4.3) on top of the microkernel. In Mach version 3.0 the subsystem grew out into the user space and a new approach was a multi–server concept (section 6.4.4), similar to Chorus (section 6.4.1).

Subsystems manage physical and logical resources like files, devices and high level communication services.

6.2 Several Personalities on One System

The requirement that all operating systems have to be emulated on a microkernel scheme should not be considered as a disadvantage. Since every operating system's personality is only a subsystem of the whole, therefore one would conclude that a system, which has been specifically designed to support one operating system, must be faster than a specific subsystem solution. Such a conclusion is false since improvements in microkernels like Chorus and Mach have shown that there is no performance deterioration related to this design.

It is a big advantage not to have a native operating system. This is due to the fact, that system calls do not have to be transformed into native operating system calls. Every system call in the subsystem is 'native'. So even when you are running several operating systems, none of them will show a performance deterioration. These operating (sub–) systems coexist on top of a microkernel (see Figure 6.2). They can exchange messages and share data regions.

To change the behavior of a system only the subsystem on top of the microkernel has to be changed. Personalities like MS–DOS [FM91, Mal92, MRGB91], UNIX [HP92, JCS+91, GDFR90] can coexist on top of the same microkernel.

6.3 Personality–Independent–Layer

All operating system personalities have a common set of services. Since for many of these services it is not necessary to execute in the protected space, therefore it is not necessary for them to be implemented in the microkernel. The sets of common services vary from operating system to operating system. However, certainly it is not necessary to implement all services in all subsystems. ODOS provides therefore the personality–independent layer. Figure 6.3 shows how several subsystems can use services of the personality–independent–layer and do not have to implement separately these services.

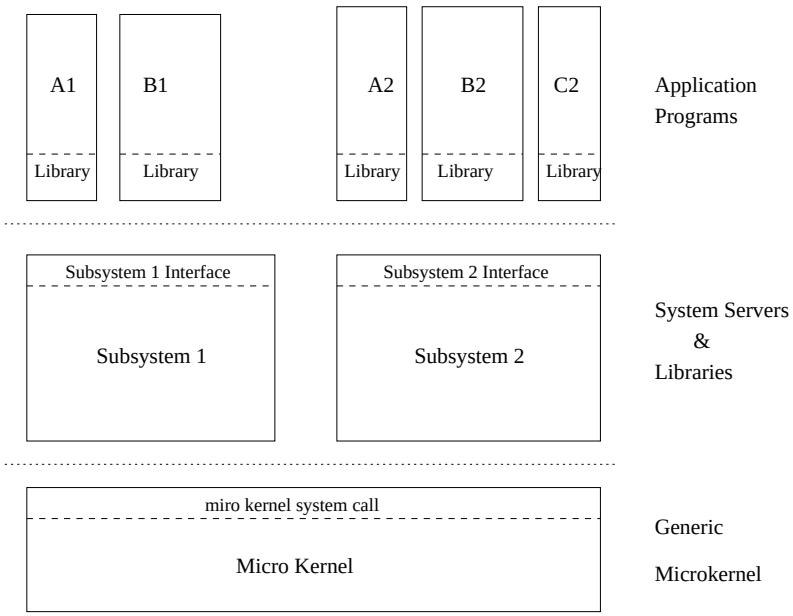

Figure 6.2: Several Subsystems on one System

The personality–independent layer resides immediately on top of the microkernel and is implemented as a number of application servers that provide general purpose system services. These servers depend only on the microkernel and themselves. Several of the common personality–independent servers are:

- default pager,
- hardware resource manager that assigns available device hardware to device drivers,
- a master server that loads other personality–independent servers into memory,
- several device drivers,
- general purpose file server,
- general netware server,
- *tty* manager that handles the output to the screen for the sites notconnected to such a device (e.g. general purpose windows' manager)[GBB93].

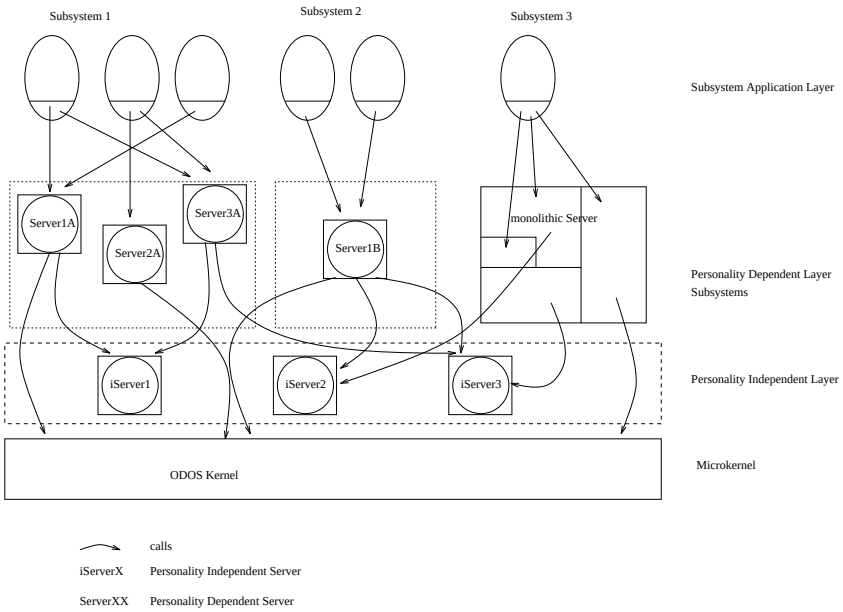

Figure 6.3: Personality–Independent Layer

These servers can be replaced during runtime. Therefore the view of the system as a whole is changeable, although the system always has to offer services which are necessary to provide a good operating system environment.

Personality Implementations

The microkernel and the personality–independent servers are common for all ODOS installations independently of the locally running personality. This system architecture offers simple extensions to various operating system personalities. Each subsystem has to implement a *process manager* or a system library, which handles the traps of the original operating system calls. These traps are usually packed into messages to be sent to the appropriate server. The server can be one of the ODOS personality–independent servers or it can be a server that is directly implemented for this personality.

As it is shown in Figure 6.3 the implementation of the subsystem can be accomplished in several different ways. System A is a subsystem with a runtime library which redirects the calls directly to several servers. Some of the servers can be servers from the *Personality–Independent Layer* (PIL). The servers are processing the call and communicate with the kernel or other servers as needed. System B is a subsystem where the runtime library directs

all calls to a process manager. The process manager sends messages to servers that can serve the request. In this kind of design, servers can be specific for the subsystem or they can be general servers of the PIL. The subsystem C is an example of a monolithic subsystem. A monolithic subsystem can implement all services or only a subset of them. The other services are handled by the PIL servers. The figure shows only a few combinations of how subsystems can be implemented. The full range of permutations is possible. The decision how system traps are handled in the subsystem is done in the implementation of the subsystem. ODOS does not restrict this design variety.

Dominant Personality

All ODOS installations have one personality, which is dominant among available personalities. The dominant personality is chosen during installation. It emulates a standard operating system such as UNIX and should provide a startup view of the system. Since the dominant personality is an application server, it is possible to have multiple servers for different personalities executing programs written for different operating systems running concurrently. However, several operating system services, such as an error message translation, are not provided in the personality–independent servers. Since it is the best not to duplicate such services, therefore the dominant personality provides them not only to its client applications but also to any other personalities running on the machine.

6.4 UNIX Subsystems

6.4.1 Chorus MiX

The Chorus system is composed of the small Nucleus and a number of system servers. The Chorus/Mix is the subsystem to provide the services of UNIX to the users. Chorus/Mix is a UNIX System V based subsystem on top of the nucleus microkernel. It provides a standard–based, real–time, transparently distributed UNIX environment. It is binary compatible to System V release 3.2 and release 4.0. Existing device drivers can be integrated into Chorus/Mix with minimal effort.

As mentioned above Chorus/Mix is a multi–server subsystem [BGH+92]. Each server is implemented as a Chorus actor. Therefore it has its own context and is usually multi–threaded. Each request to a server is processed by one thread, which manages the context of the request until a response has been delivered. The subsystem exists out of several different managers like process

manager, file manager, socket manager, device manager, IPC manager, key manager and user–defined servers. The type of services they offer can be described as:

- The **Process Manager** maps UNIX processes onto Chorus abstractions like tasks, threads, and regions. It also receives all system calls and distributes them to other servers via the IPC of the nucleus. Many of the receiving servers can be located on other sites, due to the transparent IPC of the nucleus.

  The Process Manager itself satisfies requests for process and signal management, such as fork, exec, and kill. Each process manager cooperates with other sites for enabling remote execution and remote signaling. Therefore they create a dedicated port where they receive remote requests, which are then handled by process manager threads. Process operations that do not apply to the local site are sent to a functional address of the appropriate process manager. This functional address is an entry in the static port group representing the dedicated port of the process manager.

- The **File Manager** runs on each site which supports a disk. It supports a disk and provides UNIX file system management. UNIX processes running on diskless sites have access to disk abstractions by transparently communicating with distant File Managers. The Program Manager, which manages the process does not need to know where the actual location of the File Manager is on the network, because Chorus IPC is location–independent.

  The Chorus virtual memory management uses service of the File Manager as an external mapper. In addition, Chorus/Mix file system cache uses Chorus virtual memory mechanism. Those two services interact very closely in a well–defined manner.

  Chorus introduced a new file type called *symbolic port*. A symbolic port is a filename which is directly connected to the Unique Identifier. Symbolic ports are inserted into the UNIX file tree. If during a file name analysis, a File Manager encounters a symbolic link, it redirects the request to the port, represented by the Unique Identifier.

- A **Device Manager** is only on the sites that have any kind of devices like ttys, pseudo–ttys, or bitmaps. These devices can be connected to several Device Managers loaded on one site.

- The **Socket Manager** manages high–level network protocols such as TCP, UDP and IP accessed through the BSD socket interface. It only has to be present on sites which are connected to external networks.
- The **IPC Manager** provides services that are usually offered by a UNIX Inter–Process Communication facility. The services include messages, semaphores, and shared memory. The IPC Manager interacts with the Process Manager, the Key Manager, and the File Manager.
- The **Key Manager** is an internal server that does not offer any user service. It handles requests from the Process Manager and the IPC–Manager in context of System V IPC services. It provides creation and maintenance of a mapping between user provided keys and Chorus internal descriptors. It ensures the uniqueness and coherence of the mapping across a distributed system. The Key Manager is unique in a system.

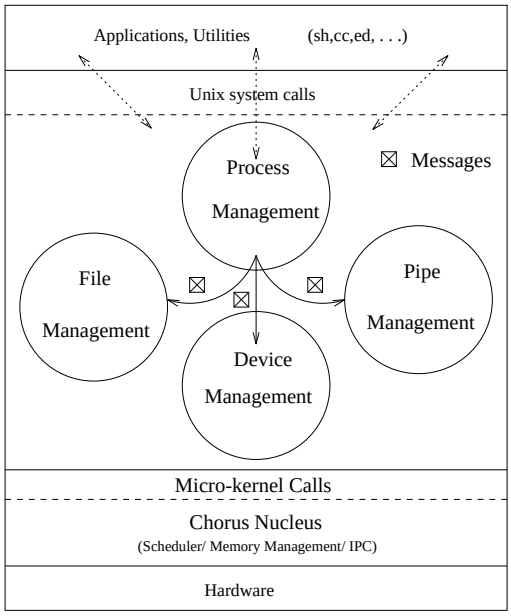

Figure 6.4: Chorus MiX Modular Design

Subdivision into a set of cooperative servers (see Figure 6.4) provides functional and distributed modularity. Each server implements a particular well–defined set of services. Each type of system resources, like processes and files, is handled separately by a system server.

This modular structure with its object–oriented approach enables debugging and testing to be handled more efficiently.

The well–defined homogeneity of interfaces to servers makes it easy for system builders to implement new servers. Therefore Chorus/MiX offers an easy integration of new user–defined servers. This easy integration of new servers enables application developers to implement their own servers like fault–tolerant file manager or window manager. All servers can run in user or system space. Developers can implement and debug their server in user space by using powerful debbuging tools and integrate them into system space for a better performance. The clarity of the module interface definition and the message–based communication permits a greater independence among system services, which simplifies and speeds up integrations of new services.

Distribution of Servers

Not all servers have to run on all sites of a distributed system. Servers can be dynamically loaded and unloaded. Only Process Manager and Object Manager are required on all sites which are running UNIX processes. Servers of the Chorus/Mix subsystem can be distributed across different sites of a network (see Figure 6.5)[GGT+91]. Servers communicate using the IPC facilities provided by the nucleus. Servers' distribution across the network can be dynamically reconfigured to adapt to changing network topologies or for system load balancing. Replication of servers can be used for fault tolerance. Several servers on different sites can join a port group and therefore they will be aware of hardware and software failures for the specific application program.

UNIX Services of MiX

Chorus/MiX handles all traditional UNIX functions such as: managing signals, creating processes, and process communication [WIK92]. UNIX services have been extended in Chorus to manage real–time, multi–threaded and distributed processes. Real–time facilities and distribution are completely transparent to UNIX application programs. These extensions allow the creation and manipulation of processes on other sites of a system while respecting the rules of UNIX (such as open files).

The location–independent IPC of Chorus has been exported to UNIX. From the perspective of UNIX processes, they seem to communicate with processes on their own sites while exchanging messages with processes on other sites of the same system. Chorus/MiX also offers other Chorus services to UNIX processes; such as: virtual memory management and device drivers.

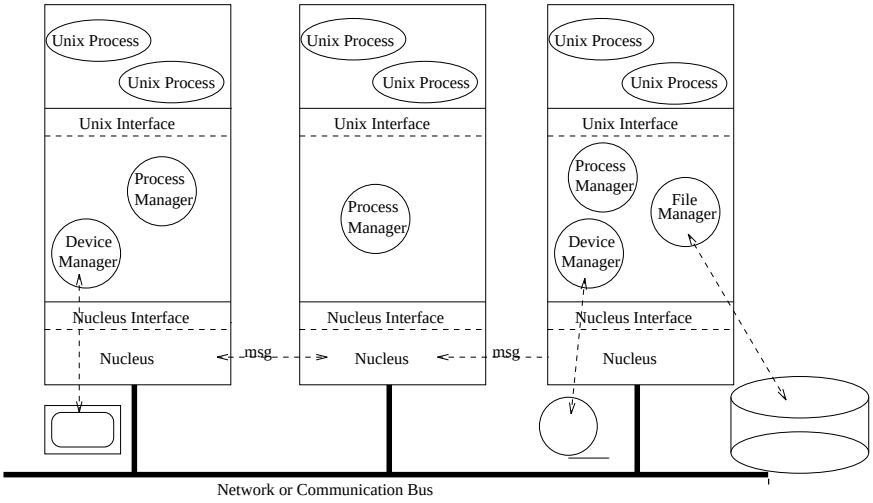

Figure 6.5: Distribution of Server through the System

Powerful real–time facilities make it possible to dynamically connect handlers to hardware interrupts. The scheduling is handled by the real–time executive. UNIX processes benefit from the preemptive scheduling.

Parallel Execution Support

These extensions enable programmers to implement parallel applications. The multi–threaded approach for local parallelization and the easy IPC for distributed parallelization makes this task very easy. The *exec* system call has been extended to execute processes on remote sites. Load balancing can be done by hand. A process can ask for the utilization of remote sites and can execute processes on sites with the lowest utilization level in the system. Work has to be done in the distribution of a process. Processes in Chorus are tied to one site. This is a restriction from parallelization (concurrency) and load balancing points of view.

Multi–threaded UNIX has some restrictions in Chorus. The interface semantic and syntax to the UNIX calls have been adapted [RAA+90]]. The *fork* and *exec* system calls produce only a single–threaded process. The interface includes methods to create and delete threads, to suspend and resume their execution, and to modify their priority.

Each thread has its own stack and can have its own signal context, consisting of signal handler, signal stack, and blocked signal. Signals are delivered to processes which have expressed interest in receiving them.

There are two types of signals in a Chorus/Mix to determine which thread should receive the signal.

- First, there are signals for which the target thread is not ambiguous. They are sent directly to the thread. If that thread has not declared a handler for the signal, then the default handler for the whole process is called.
- Second, there are signals which are sent to the entire process or signal for which the target thread can not be determined. These signals are delivered to all the threads in the process. If no thread has declared a handler for that signal, then the default action for the process is called.

Structure of Processes

The Process Manager attaches to all UNIX processes a *control–port*, which is not visible to the user. It also creates a thread that controls this control port. The *control–thread* receives and processes all requests on the control port. The control–thread resides in the address space of its UNIX process and therefore can access and manipulate all data structures proceeding a signal to the control port. It is also able to handle asynchronous events on the control port, which are implemented as Chorus messages. For debbuging sessions a debug port is assigned to a process. Its identifier is sent to debugger. The whole communication between debugger and process is based on Chorus IPC and therefore they can reside on different sites.

Chorus identifies the UNIX processes by a global unique 32 bit process identification number [RAA+90]. This number is created out of two integers that represent creation site and the traditional UNIX process id. In addition, Chorus maintains for the UNIX process a child process creation site for each process. This number identifies the site on which a child process would be executed. By default this site number is the same as the current site number.

The file system is fully distributed; and file access is location–independent. File trees are automatically interconnected with symbolic links to provide a file name space, which is accessed using regular UNIX semantics.

Single System Semantic

Chorus tries to realize a single system semantic on a multicomputer. It eliminates the heavy–weight socket communication by introducing the light–weight IPC of the nucleus. It implements also a light–weight remote procedure call on shared memory multiprocessors [BAEL90]. Chorus hides the difficulties of

this problem in the kernel and so the user does not have to be aware of it. Chorus/MiX introduced the remote execute commands and made it easier to manage the multiprocessor by having a single process identification space and file name space, device naming, time management, resource accounting, and swap space management. These are the first and the most important steps to a real single system semantic.

The single system semantic is very difficult to manage for a large amount of processors. Chorus therefore offers the possibility to restrict the number of processors that are in one parallel domain on a parallel machine [AGP+91]. Due to the restricted input/output capacity, it is unlikely to scale much beyond several tens of processors. Parallel applications use the whole computational power of certain number of processors that are available in the parallel domain. The utilization of processors in parallel machines varies from time to time and from application to application. In addition, some parallel applications may need more processors. Therefore a fixed number of processors would be restrictive on modern parallel architectures. A modern system has to provide a possibility to reconfigure dynamically to the single system semantic. An individual processor should be able to join and leave the parallel domain at any time.

The boundary of such single system semantic system does not have to be a a single machine. Different supercomputers and file servers could join the computing system to increase the computing power of a system. It is very important to keep the system operational and coherent even in the case of failure of several sites. Fault tolerance will be discussed later.

UNIX processes can communicate with all processes in the system. An exchange of data between different subsystems is accomplished using the IPC provided by the nucleus. The processes can create and manipulate Chorus ports, and can also issue remote procedure calls.

The real–time facilities of the Chorus Nucleus have been exported to UNIX processes. Processes can therefore dynamically connect handlers to hardware interrupts. This is necessary for various UNIX device drivers. The second benefit of these real–time facilities is the priority–based scheduling which is provided by the real–time executive of the Nucleus.

The Chorus implementation of UNIX is well–suited to multicomputers because of its modular dynamical structure. The single system semantic is important for all kinds of multiprocessors.

6.4.2 UNIX on Mach

Mach is a microkernel developed as a research project at Carnegie Mellon University. The purpose of the research was to develop a structured operating system [FGB91, TR87]. Researchers utilized the existing code from the BSD UNIX and tried to eliminate unnecessary services from the kernel. The project went through several stages. In each of them the system became more structured. The first versions layered the system into different priority levels and kept UNIX in the protected space. A working version is Mach UX, which is an UNIX–personality implemented outside of the kernel. Several people are still working on Mach US, which will be a modular UNIX implementation on top of the Mach microkernel.

6.4.3 Mach–UX

A UNIX process in Mach is implemented as a single–threaded Mach task. To achieve binary compatibility an emulation library is linked into UNIX process' address space. The library exists in a region which is inherited by every UNIX process from the directory */etc/init*. On Mach UX, there is one 4.3BSD server for every computer, which is running the UNIX subsystem.

System calls of the process are redirected by the kernel. The kernel stores a system call redirection table. This table indicates which emulation handler it has to call for specific system call. The handler is then called and the emulation sends a message to the BSD server or in some cases it handles the request on its own.

The 4.3 BSD server is a multi–threaded Mach task (Figure 6.6). Several threads are dedicated to routines that, in the BSD kernel, would be driven by hardware interrupts. The threads are waiting for receiving and processing requests from various emulation libraries. When a request arrives at the BSD server, then a thread is dispatched from a pool of threads to handle this operation. Communication with hardware devices is done through the Mach IPC protocol.

UNIX processes share two memory regions with the local BSD server. The first one is for the *read only* process. It contains information such as: process ID or user ID. Calls to get information which is stored in this region are handled by the library directly without interaction with the server. The second shared region contains signal–related information and can be written by the process.

An optimization in read/write system call offers great performance. At the open system call the file is mapped into a 64 kByte region in the process. The

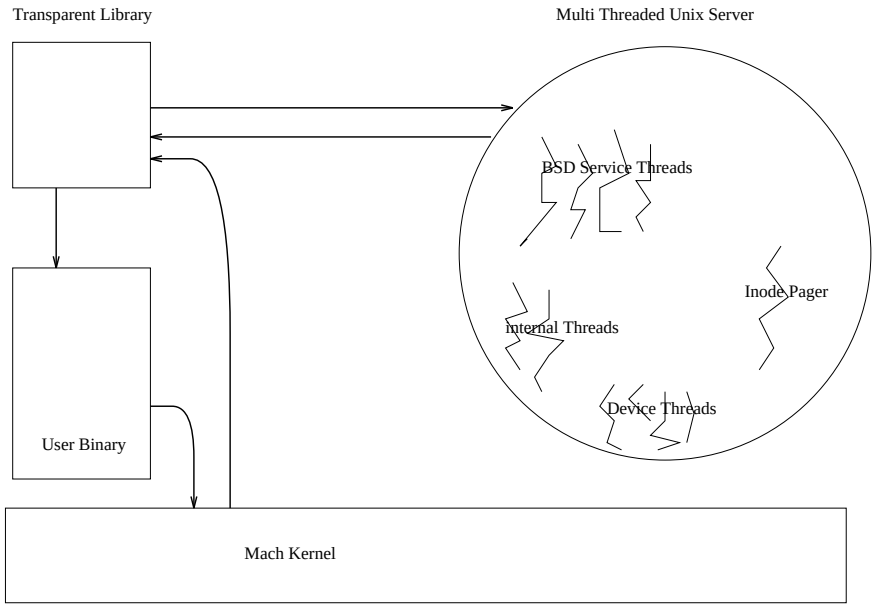

Figure 6.6: Mach BSD 4.3 UNIX Server

BSD server acts only as an external mapper. If the file is larger than the 64 kByte, then the region behaves like a window to the file. Read/write system calls can therefore be handled by the library and they do not have to send a message to get a small amount of bytes. A read/write system call is a copy between regions. A move of the file window requires a synchronization with the server because a file can be shared and it would need consistent updates. Mach uses a token to obtain the mutual exclusion. The emulation library is responsible for handling the token inside the process.

6.4.4 Mach–US

Mach–US (Figure 6.7) is a project to implement the scheme of personality–independent layers. Currently it supports only BSD–UNIX, but the design is a base for all operating systems. UNIX is only a test operating system for the implementation.

Mach–US is referred to as a symmetric multi–server system. It consists of two layers. First, there is a *service layer*, which has a set of separate servers supplying generalized system services (file systems, network server, process–manager, tty server,...). It is as generic as possible so that it can be used for the emulation of several operating systems. Second, there is an emulation library

which is loaded into each user process' address space. This library uses the services to generate the semantics of the emulated industry standard operating system. Some of the system servers are specific to the operating system being emulated while most are meant to be fully generic. Even servers which are developed for a specific operating system can be reused in the implementation of another.

In contrast to Chorus there is no 'central' server like the *Process Manager* in Chorus/MiX; neither for emulating a specific operating system nor for a general traffic control. All multi–service actions are controlled by the emulation library. This reduces the communication between different tasks, but also introduces a less secure system. Faulty tasks can access system data, which are kept in the emulation library. In Chorus this information is not accessible to the user task.

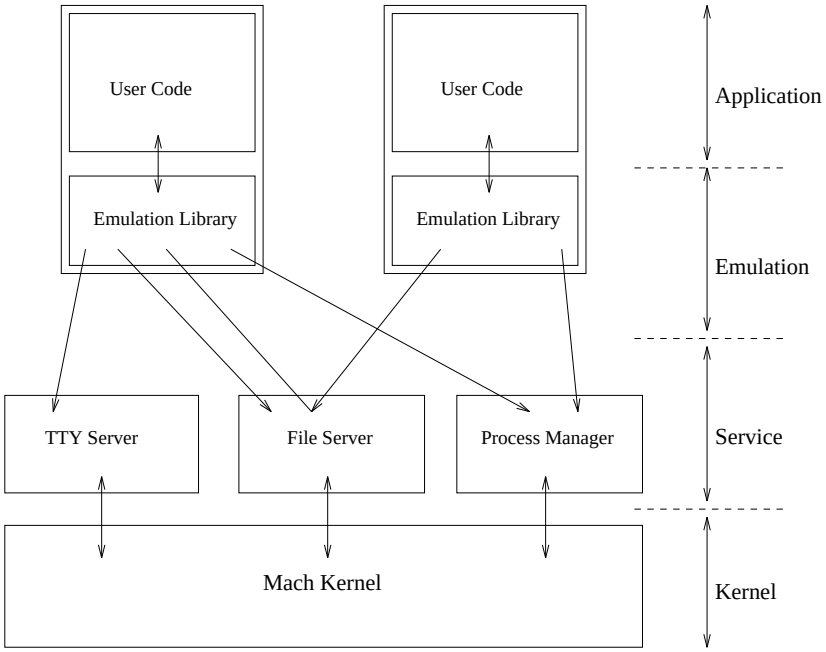

Figure 6.7: Mach Multi–Server System Architecture

The biggest feature of Mach–US is its flexibility. It offers a whole new, highly modifiable operating system architecture without significant structural impediments to performance. The reusability of the servers is very important for a wide range of different operating system personalities.

Object–Oriented Generic Operating System Interfaces

Most of the high–level functions are defined in terms of the object–oriented framework. Various servers support a combination of C++– based object–oriented interfaces to do their work and an emulation library uses them to emulate the target operating system.

This uniformity of access makes it easy for new servers to supply additional functionality by sliding it into the name–space under the known interfaces. This functionality is generally available to system users through their pre–existing software. Many primitives of a standard operating system can be implemented by using an equivalent primitive of the standard system interface.

For object–oriented design it is trivial to make necessary refinements to interfaces via inheritance. In order to maximize flexibility, complex operations are often a concatenation of several simpler primitives in the generic interface.

Specific services of an operating system (such as the process–manager), may supply specific interfaces within the same basic C++ virtual class model.

To take advantage of the reusability, the services are implemented in as many servers as possible. These servers can be used as building blocks for the construction of new personalities. Different functionalities are separated into various servers: configuration, authentication, root–name, diagnostic, pipenet, ufs, process–manager, tty, and network[4] [BB92, Jul89, MB92, MB93]. This separation makes it simple to develop and debug operating system services. One can add and subtract services as needed for a given invocation of the system.

Building the system as being composed of separate servers has the following advantage: a bug in one service does not crash or corrupt the entire system.

There is an extensive object library for support/implementation of various generic interfaces as well as for general server building blocks. This enables faster prototyping of a new server and eases creation of servers from foreign code.

To avoid the unnecessary duplication of code, Mach–US offers a common library that provides a set of functions which have to be implemented on several servers. This library contributes to the extensibility and ease of modification of the emulated services.

- *Interrupts/Signals*: There is a package that enables the author of a server to handle the interrupts and UNIX signals in a consistent and relatively painless manner. It has been used to make interrupt–ignorant code useful in the UNIX environment, and easy to use for development. The por-

[4] xKernel protocol engine developed at University of Arizona

tion of the interrupt system that the servers use is not specific to UNIX signals. It works also for any source of interrupts (pending invocation cancellations). Most of the emulation–library side of this mechanism is either operating system–independent or easily modifiable for a different signaling semantic.

- *Remote Method Invocation*: There is a remote method invocation system that enables clients of the system services to simply invoke a method against an operating system item. The appropriate method is correctly invoked on the appropriate server. This occurs with the needed authentication and protection guarantees.

Chapter 7

Real–Time Facilities

Real–time operating systems are used if there are special time constrains imposed on an operation of a process or the flow of data. They are used to control devices for which sensors deliver data to the processor. Processor evaluates the data and makes possible changes in the control of a device. Real–time systems are used in factory automation, military control and command, robotics, and in continuous media.

To guarantee the correct function of a device, a control system has to process data within a well–defined, fixed time. Otherwise, the result could be total failure of the whole system. In some systems, like the control system of a nuclear power plant, this could be a disaster. This is in contrast to a time–sharing operating system, where the correct result is important at any time.

Because of time constraints the real–time applications frequently require greater time precision and flexibility. Time stamps, exact delays, and timeouts are facilities required from a real–time operating system. To support real–time applications an operating system needs:

- time measurement with a high degree of precision.
- synchronization support from high precision time source.
- integration of scheduler and synchronization; in this manner computations can be modified when timing faults occur.

Merging of RT System and General Purpose Operating System

Real–time operating systems have been very specialized. Years ago real–time application developers built their own real–time operating systems to support

their application. System builders exploited every hardware feature to increase the speed of the applications. The result of this has been a non–portable code. When introducing new hardware the whole system had to be rewritten.

Nowdays the hardware is less specialized and also other tasks have to be supported by standard operating systems. In real–time systems some services are built in, which formerly were offered by standard operating systems [Gui90]. These systems became too complex to develop and maintain for a single purpose only. The answer to this merging problem (regular OS + RT system) is also the microkernel approach.

In the same way standard operating systems, such as UNIX, have adopted features from real–time executive operating systems. All operating systems should support everything else. The existing monolithic operating systems are too complex to support real–time applications in a satisfying way. With such complexity an operating system cannot serve as a real–time executive, since it can not offer the required performance level.

Computers are used for factory automation and process control. Therefore there are several reasons why companies are interested in having the real–time operating system the same as for all other applications.

- Training is much simpler and less expensive if the operating system meets the requirements of both.
- Programmers are usually familiar with standard operating systems such as UNIX.
- Applications and development can be shared between the two areas.
- It is very likely that one can purchase third party products for a standard operating system such as UNIX.

Compared to traditional real–time executives, the microkernel adds distribution and subsystem support to real–time systems.

Types of Real–Time Systems

We can distinguish two types of real–time systems [SG94].

- *Hard real–time systems*, which are required to complete a task within an exact amount of time. This type of real–time systems requires special purpose hardware, since general purpose hardware cannot guarantee an exact timing. For this type of systems special operating systems are still needed, but a general purpose operating system can also meet some of

the requirements that a hard real–time application imposes. It can not be guaranteed, in a standard operating system, that the deadline is met exactly. Therefore the general *thread* class is extended to a class that offers real–time features.

- *Soft real–time systems*, where computing is less restrictive. In these systems, it is necessary to increase the priority of critical tasks so that they are scheduled with a higher probability than less critical ones. These systems are general purpose operating systems that support multi–media and other tasks, which usually run on a non real–time system.

Scheduling

An important part of a soft real–time system is the scheduler, which policy observes real–time requirements. This scheduler must be a priority scheduler, where real–time applications have the highest priority. Mature microkernel developments like Mach [RT90, TNR90, Tok91, ST93] and Chorus [Gui90] support priority–based scheduling.

Process aging feature should be disabled for real–time threads and therefore real–time threads do not change their priority. A problem in most operating systems is that context switches can only occur after a system call or a given time period. The dispatch latency in such a system is too long, since in monolithic operating systems a system call can be very complex and therefore needs a large amount of time. In microkernel systems, calls are less complex and the latency is reduced. This reduces the necessity of preemption point in system calls of long duration.

For a proper scheduling, real–time threads need additional timing attributes comparing to attributes of regular threads. A real–time thread can be a *hard* or *soft* real–time thread. Hard real–time threads have a hard deadline time, which is the time when all threads have to be completed. Soft real–time threads have a *softer timing*. It is not so critical for them to complete very fast; it is more important to respond very fast. This happens if their priority is very high. Such threads wait only if there is nothing to do for them.

In many advanced operating systems threads enter a critical section in FIFO order. This is not appreciated in a priority–based real–time system. If low priority threads are waiting on a *mutex* when a higher priority thread is arriving at this FIFO, then the high priority thread is blocked by these low priority threads. This is called the*priority inversion*. This can result in missing of a deadline by the high priority thread.

An advanced real–time kernel should allow to analyze the process runtime

during design and to predict the exact runtime for a thread that has various types of system interactions. This analysis is not possible if waiting time is unpredictable such as in the case of priority inversion.

To avoid priority inversion, a buffer in front of a *mutex* should be in priority–based order. The high priority threads do not have to wait for the completion of all threads which arrived before. To control the order of execution of a critical section the Real–Time Mach [TNR90] supports three different levels [TN91] of preemption:

- *Non Preemptable* does not support preemption while a thread is executing in a critical section.
- *Preemptable* does allow preemption of the currently running thread by a higher priority thread. The high priority thread will be blocked, if it wants to enter a critical section.
- *Restartable* does abort a running thread when a higher priority thread arrives and puts it back into the waiting queue. The higher priority thread executes its critical section without further delay.

ODOS will inherit this idea of preemption from Mach.

Priority Inheritance

If a low priority thread owns a resource which is needed by a high level thread and medium priority threads are ready to run, then the medium threads are scheduled and the low priority thread never frees the resource. This is also a condition which should not appear in a real–time operating system. The blocking time is not predictable. To bound the blocking time a simple scheme, called *priority inheritance*, has been developed [Raj89, SRL87, TMIM89]. When using priority inheritance a low priority thread, which is blocking a *mutex*, inherits the priority of a high priority thread which is blocked on a *mutex*. The blocking thread runs again and frees the *mutex* and the blocked thread can enter the critical section. The worst case blocking time is a function of the duration of the critical region. The presence of medium priority threads doesn't influence the waiting time.

In selecting a suitable queuing method and preemption, the RT–Mach defines the following synchronization methods:

- *Kernelized Monitor Protocol* uses no preemption.
- *Basic Priority Protocol* uses a priority–based queuing and a preemption.

- *Basic Priority Inheritance Protocol* uses a priority–based queuing and a priority inheritance for the running thread. This allows the blocking thread to free the *mutex* faster, and the higher priority thread can enter the critical section earlier.

- *Priority Ceiling Protocol* blocks a thread at a lock if the ceiling priority of the lock (ceiling priority is the priority of the highest priority thread which can lock the lock variable) is not higher than priorities of all locks which are owned by another threads. This prevents deadlock and chained blocking.

- *Restartable Critical Section Protocol* restarts a thread and puts it in the queue again if a higher priority thread tries to lock a *mutex*.

The costs of entering and exiting a critical section are very different. The programmer has to select a suitable synchronization protocol for his application.

Due to the basic feature of ODOS, that the whole system should be modularized, all components of the kernel are also modularized. This enables a personalized configuration of a site or even of a domain. The parallel LRPC module (see Chapter 4) is an example of the module that is not loaded in the kernel when a uniprocessor is present. This makes the loaded kernel smaller and faster, since only a few of the possible cases for RPC have to be checked. One of the modules which are exchangeable is also the scheduler [Bla89]. This central part of an operating system can be customized in ODOS. The integration of the scheduler server into system space is done like any other integration of a server. This enables configuration and customization of a system for user's needs. This flexibility is an important feature offered by many operating systems with one underlying microkernel.

The standard scheduler in ODOS is a preemptive priority–based scheduler. If the system needs more advanced real–time scheduling, ODOS can still serve as a kernel but with an alternate scheduler.

The ODOS kernel supports user–defined exception handling. Exception handling can easily be done in *user threads*. Therefore ODOS can be asked to declare a port in the task, where the kernel sends description of exceptions. These exceptions can then be handled by user–defined threads which listen to that port. If no such port is declared, the exceptions are handled by the standard exception handler and this usually causes the exception generating thread to terminate.

Supervisor treads can directly access system data structures and therefore they have access to hardware exceptions and low level events. They are in

protected system space and no access control support is done by any of the hardware. This is a dangerous way of implementing low level real–time management. To keep control of interrupt handling and to retain the underlying supervision, ODOS provides the possibility of user level interrupt handling. User level threads can be bound to interrupts, so that they are called for interrupt handling. When an interrupt occurs the kernel saves the context of the interrupted thread and checks the interrupt table for an interrupt handler assigned to that interrupt. When this happens then the kernel calls this handler; otherwise it calls the standard handler. After return from the interrupt procedure, the kernel checks rescheduling.

This facility makes interrupt handling for subsystems efficient and offers protection for the microkernel interface.

Chapter 8

Fault–Tolerance

ODOS is a distributed operating system. Distribution over several sites increases the probability of a fault. Existence of redundant servers offers a better method to recover from these faults. Therefore fault–tolerance and distribution are interrelated.

Fault–Tolerance in ODOS

ODOS offers fault–tolerance by multiple execution of servers on different sites. These servers listen to a port–group. The *receiving mode* of the port–group depends on the kind of fault–tolerance the servers have to provide.

There are two methods how fault–tolerance can be provided.

- If it is acceptable that the service request, which the crashing server is answering, is not delivered, a *broadcasting* is not necessary. The task which is requesting the service has to be notified. The other servers are still providing the services and therefore future requests will be answered.
- If it is necessary that a requested service is answered correctly, even in case of a hardware failure, the service requests have to be broadcast to all servers. The servers then calculate their local results and communicate them to all other servers. They negotiate their common result and send it back to the service requesting task. Servers that are wrong in computing their local results are checked off; if their recovery is not possible such servers are removed from the *member list* of the port–group. Another server can be started, which replaces the old faulty server.

The second method relies on servers which are participating actively in fault–tolerance. The servers have to negotiate their results and react on the

presence of faulty servers. Methods for accomplishing the above are covered in detail in recent literature [SBB94, LS94, PBB+94, UR94].

Since both methods rely on a working port–group, fault–tolerant port–groups are needed. In ODOS this is solved by replicating the whole port–group. All of them have the same *receive UI*. Messages are sent to the *receive UI* and are delivered to one of the port–group replications. In addition to the *receive UI*, they have their *real UI* to communicate with other replications. All messages, which are sent to one of the replications, are copied to the other replications too. The order of the messages has to be preserved. Therefore an enqueuing operation into the message queue is not possible before all replications have negotiated the order. This is done by assigning replication priorities. If two replications attempt to insert a message then the higher priority replication will succeed.

When a thread wants to receive a message on a port–group, then the port–group replications have to negotiate their outgoing message. The negotiation is performed in the same way as for receiving messages. Requests to the higher priority port–group replications get the first message, and requests to lower priority replications get later messages.

A task can not decide to which replication it sends the request or message. The decision is made by the routing algorithm and is dependent on the location of the sending task. Routing table entries for fault tolerant port–group *receive UIs* can differ from site to site. As long as the port–group replications send a broadcast of the message to all replications the message will go to all of them.

Well–Known Ports

Another fault–tolerant feature of ODOS is an automatic reconfiguration of *well–known ports*. Well–known ports are ports where requests for services, usually offered by an operating system, are sent (Section 3.2). If a server, which provides services for a well–known port, dies and a message is sent to the *deadname*, then ODOS reloads this server and the server can then provide the requested service. As shown in Figure 3.4 the well–known port class has a *filename* describing an executable file which should be loaded in case of a request.

Chapter 9

Memory Management

ODOS is a distributed operating system. In a distributed operating system the design of a virtual memory management [ARS89, ARG89] has two different aspects:

- an operating system needs a backing storage if a processor has to *page out* data because the main memory is needed for other tasks.
- a distributed system, such as ODOS, supports distributed shared memory [AD92, AJL92]. The memory pages of one address space can be distributed over several sites. The access to remote sites is handled by the virtual memory system.

Memory Management Unit

ODOS allows several tasks to be run together in the same memory. Therefore, it depends on runtime mapping from virtual to physical addresses. In a computer system, this is done by the *memory management unit* (MMU), which is a hardware device. ODOS manages memory protection and sharing on a memory object basis using an abstract layer (above hardware), independently of any particular processor address translation hardware.

Swapping

To execute a process, a part of a process has to be in main memory, but some other parts can be swapped out to free space for other parts of the same task or for different tasks. The address space of a task is divided into pages, which all have the same size. These pages are mapped onto memory frames. One memory region can be used for different pages. The mapping from a logical

address (in the program) to a physical address (in memory) is done by the MMU.

Communication Support

The virtual memory system of a microkernel has to support optimization of the communication. Quality of communication in a microkernel is very crucial for the performance of the whole system, because all services are called by sending messages to the appropriate server. Sometimes huge amounts of data have to be sent. In this case a support for the communication by the virtual memory system is very useful. This support can be:

- *copy on write*: a method to reduce copying of pages between different address spaces. If large amounts of data are transferred, e.g. in messages or when creating a new task, then pages can first be marked as copy on write; and then if one task tries to modify the page, this page is copied. As long as both tasks only read data, the copy does not have to be made. In many cases this reduces the number of pages to be copied.

- *page sharing*: two processes can share a page to reduce the number of pages allocated and to share some pages for common use. Examples of common use are:

  - shared text regions: text regions are usually not modified. Two tasks which use the same code can share the same page.
  - IPC optimizations: LRPC (Section 4.3.3) and URPC (section 4.3.4) use the fact that memory pages can be shared. These protocols are optimizations of the regular RPC–protocol.

Customized Pager

A virtual memory system should be personalizable. In Mach [Dra91, FBYR89, GD91, RTY+87, Tev87] each task can determine which virtual memory pager is responsible for *page in/out* pages of this task. If ODOS encounters a *page fault*(two main reasons are: lack of specific page in memory or removal of pages from memory due to the requirements of other programs executing on the same machine), then it notifies the pager of the task in which the fault occurs using the IPC mechanism. It is then up to the pager (Figure 9) and an application server, to determine how to provide or to store the data. This permits the system to define different semantics for the paging strategies based on the needs of the programs that use them. The default pager, which is used

if no specific pager is named, is a pager for memory pages which do not require anything other than the usual semantics of virtual memory.

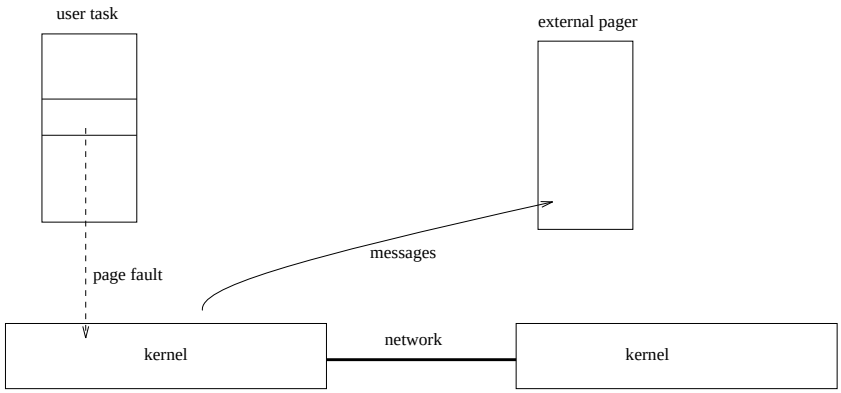

Figure 9.1: Pager of Memory Objects

Unified Address Space

To support concurrent execution of several threads of the same task in a multiprocessor system without shared memory, an operating system has to unify logically the address space [BZ91, BZS93]. If a thread demands to read from/write to a page which resides in a non–local memory, then the access is handled similarly to a *page fault*. The threads are paused until the page can be brought in. An optimization is possible for read access. The page can still reside in the secondary memory. As soon as one site tries to write into the page, the page on the second site is removed and future accesses are handled as if the page had never been on that site. As long as both sides only read from this page, the page does not have to move from site to site when the sites demand data from this page.

Interaction Between Fileserver and VM–System

To speed up file access the *file server* reads in whole pages of data. For this purpose it allocates a page of memory which will be shared with the task which accesses the file [TRY+87]. Both tasks have read/write access. Due to locality of the data access from files, a file access is faster than without data page sharing. It is also faster than regular caching, because data can be accessed directly. The page is owned by the requesting task, so threads within

this task can access the data of the page in the same way as they access data of all other pages. The right mapping is done by the memory manager.

Process Integration into System Space

As mentioned in Chapter 3, tasks can be integrated into system space. To speed up server response time, it is better to allow frequently used servers to be integrated into the system space. When services of these servers are called, the kernel does not have to switch between address spaces. It can serve the request directly in the system space without switching to the servers address space. This reduces the context switch overhead. For this purpose ODOS offers the integration of trusted servers into system space.

For device drivers and other tasks which have to deal directly with the hardware [BGR+88], it is necessary to run them in the most privileged mode of the processor. Device drivers have to execute privileged instructions and to control data. Therefore they have to be integrated into the system space. ODOS supports this integration.

Chapter 10

Object Orientation

Microkernels such as Mach [OJ91, JR86] and Chorus [HMA90, ALJ92, LJ92] are microkernels where object orientation can be put on top of the kernel. The fundamental requirement of ODOS is to implement a microkernel as an object–oriented structure with some degree of inheritance. Therefore a support for object–orientation on the user level is much better.

Requirements for Object–Orientation Support

There are some features which an operating system should provide to support object–oriented programming. These features are:

- *object management*: creation and termination of objects, and up–calls for user–defined classes.
- *object communication support*: location–independent inter–object communication.
- *resource management*: allocation of resources (memory and processors).
- *uniform access*: all objects should be accessed in the same way.

In the design of an operating system the system builders have to decide what kind of objects should be supported directly by the operating system. There are different sizes of granularity. An operating system can support:

- *large grain*: every object is managed by the operating system. Each object has its own protected resources. Switching from one object to another is very expensive. Systems have limits on the number of address spaces and therefore limits on the number of large grain objects.

- *medium grain*: some objects are grouped together. They share their resources such as memory. There is no protection against objects of the same group but protection against objects from other groups exists. Concurrency is increased; however with the growing number of objects to manage them is more expensive.

- *fine grain*: small objects which are data types in non object–oriented languages. Having small objects increases the number of objects in the system and every operation is an object invocation. These objects are not protected by the operating system.

COOL on top of Chorus Nucleus

Chorus implementation of an object–oriented layer on top of the Nucleus is called COOL [AJJ+92]. This object–oriented layer consists of three functionally separate blocks:

- The *COOL–base* is the base level of the object–oriented layer. It acts as a microkernel on which other levels can be built. The COOL–base is highly dependent on the Nucleus and its services. It supports object sharing in distributed memory and it supports the distribution of threads across different sites.

- The *COOL generic runtime* implements object management. This management includes creation, dynamic link, and destruction of objects. Two types of object identification, unique domain wide and local language references, are supported. The unique domain–wide references are location independent. The language references are pointers to the object in the residing context.

- The *language specific run–time* is the language interface to the COOL system. It maps a language specific object model to the generic run–time model. The generic run–time has to call the language specific run–time to manage the language specific implementations of objects. Therefore, the compiler has to generate enough information for the generic run–time to satisfy this need. This is done by creating an up–call table associated with the object.

Object–Orientation Support in Mach

The interface of the Mach–US–multi–server system is object–oriented. The server and the emulation library are written in a object–oriented language and

export their functionality to their interface. The kernel itself is not object–oriented; when writing code for a new subsystem the object–oriented interface of the multi–server is used.

Object Management in ODOS

The object management in ODOS is similar to that in COOL. The main difference is that *COOL's base* and parts of *COOL's generic runtime* are directly supported by the ODOS kernel. As outlined in other chapters [3, 4, 11], the o bject orientation is fundamental for the concept of ODOS. All ODOS entities are objects derived from the class *unique identifier*, which is a location–independent addressing reference. This is comparable to the *unique domain–wide* reference of the *generic runtime* in COOL. Similarly to COOL, the objects in the task are addressed by language specific references. Mapping from unique references to language–specific references is provided by the task class.

Objects in ODOS can be passive or active. Threads are active objects manipulating on tasks which are passive objects. Large–grained and medium–grained granularity is directly supported by the kernel. The medium–grained active threads are members of the large–grained passive task. Protection is provided only on a task level. All threads of a task can manipulate all resources of that task. The operating system protects only against interaction with resources being outside the allocated resources of the task.

Chapter 11

Object Oriented Distributed Operating System— ODOS a New System

This chapter shows the most important elements of the new object–oriented distributed operating system named ODOS. ODOS will support all the features listed below. These items are described in details in the preceding chapters.

11.1 Tasks and Threads

- *multi–threaded*

  The ODOS system is a multi–tasking system. The unit of execution and the unit of resource allocation are separated by thread and task. A task can have several threads running in its address space. Each tread is scheduled separately and threads of the same task can have different priorities. Due to the possibility of having several threads in one task, ODOS provides real concurrent execution of threads on different processors of the same domain. The distribution can be decided by the user. Tasks are usually placed on processors of the system, which have the smallest load at start time of the task. It is possible that the system or the user decides that a task or threads of a task should migrate from one site to another. There may be various reasons for such decision. One reason is that one site has to be shut down; another reason is that the load is to high on the original site.

For each task the kernel creates a control thread, which handles requests from the kernel and other tasks which have *sendright* to the control port of the task. The task is manipulated through this control port.

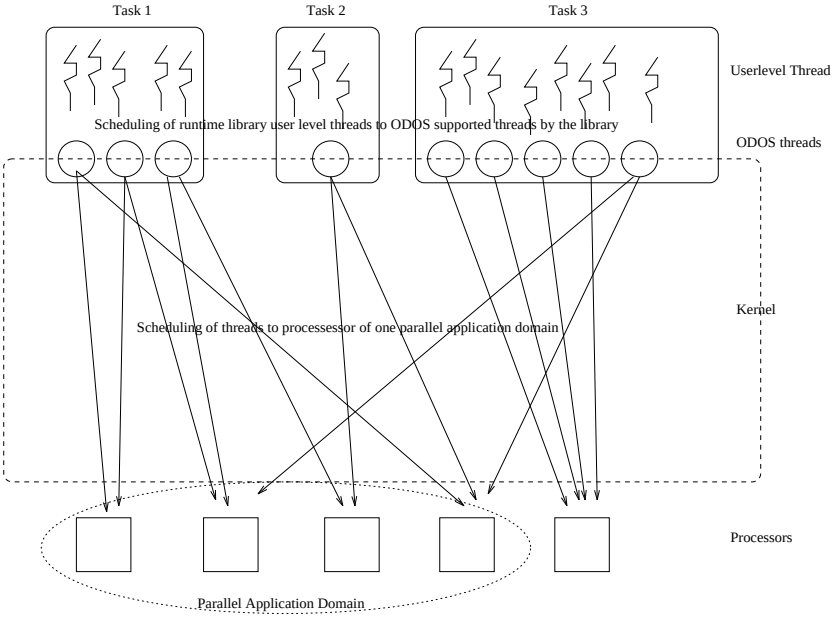

Figure 11.1: Thread Scheduling

In Figure 11.1, it is shown how parallel application domains, in which ODOS threads are scheduled, can be designed. If a processor needs to be rescheduled, it takes the highest priority thread of the ready queue for this parallel application domain. On top of the multi–threaded task environment, offered by the kernel, the runtime systems can declare and schedule lightweight user–level threads. These user–level threads are scheduled onto ODOS threads by the runtime system. This provides a facility, where the overhead of ODOS–threads scheduling is minimized or thread scheduling decisions are customized.

- *Real parallel application*

  In ODOS threads can migrate from site to site. In multiprocessors with shared memory, the threads are automatically scheduled on a free processor. Therefore a real parallel execution of a thread of the same task is possible.

- *Load Balancing During Execution*

  ODOS provides load balancing during execution of a task/ thread. A task/ thread can mention at creation time that it wants to be subject of a migration. All these tasks/threads have an expected runtime, which is longer than the runtime in average case. On average, tasks/ threads run such a short time that benefits of migration (better load balancing) is not worth the expenses of the migration. To implement load balancing the kernels have to exchange their actual load and make a decision if a migration is appropriate. To exchange these messages, ODOS defines for each kernel a well–known port to which kernels of other sites can send their load balancing messages.

  If there is an uneven balance in the domain, then the kernels have to check if this situation is temporary (many short term tasks/threads are running) or if it can hold for a longer time. In the long term case the kernels pick suitable tasks/threads to migrate.

  If a thread is migrated, its actual working set of pages is migrated with the thread. The other pages are brought in *on demand*. If the page fault rate of the thread is too high or pages are swapping from one site to another, then the thread either has to migrate back or other threads of the task (executing on the same set of pages), have to migrate to this site too.

  If a task is migrated, then the control port and the control thread are also migrated to the new site. It depends on the available processor resources how many other threads of the task are moved to the new site. For each thread the migration is described above.

- *Load Time Compilation*

  ODOS is an operating system for heterogeneous distributed systems. In such systems one needs system binaries for all different types of hardware architectures. ODOS is designed to reduce this unnecessary redundancy. ODOS uses a general RISC architecture to which system binaries are compiled. Access to the disk is very slow comparing with processor execution time. If a program is loaded into memory, then the processor can translate this abstract code into hardware binary code. The general RISC architecture which ODOS uses is very similar to most of the RISC architectures available on the market. Therefore the translation is easy and fast. Time is not lost during this translation and the generic code can be used for all architectures.

- *User–decided Pager*

  ODOS offers, in the same way as Mach, the option for a task to choose its own pager. The pager can be optimized in a way suitable for the task.

- *Expandable Virtual Memory System*

  The virtual memory system of ODOS is expandable over site boundaries. This is necessary because threads of the same task are able to run on different sites. Each thread has a working set of pages which resides on the same site. If a thread accesses a page outside of its site, then it is handled similarly to a regular *page fault*.

- *copy on write*

  Before pages of memory are copied within the same physical memory, the pages are marked as *copy on write*. As soon as one of the tasks tries to write in these pages, they are copied and the task which tries to write gets the copy and the other tasks get the original page. Before modifying a page both tasks can access the page for reading. As long as they do not modify this page is not duplicated. This reduces the number of pages to be duplicated. The startup time for a task is reduced dramatically, since the pages which a task inherits from its creator do not have to be copied immediately. Text memory pages are therefore never copied. To start a new task from an executable file in operating systems such as UNIX, the creator first *fork*s the task and starts the new program in the child task by loading (*exec*) the program text from disk into memory. If a system does not *copy on write*, then it first copies the whole address space of the creating task and then loads the data from the disk into these pages. The copy of pages is unnecessary and the number of copying operations is reduced to a minimum in ODOS.

- *Process Integration Into System Space*

  ODOS supports the integration of tasks into system space. This is a useful feature:

  - to speed up server response time. It is better to allow frequently used servers to be integrated into system space. Therfore context switch overhead time is reduced.
  - to offer some tasks the direct hardware access. All tasks in system space can access hardware and can execute privileged instructions.

One example of tasks which need direct hardware access and execute privileged instructions are device drivers. Therefore they have to be integrated into the system space.

11.2 Object Orientation

- *Object Oriented System*

  Object orientation is a very fundamental concept in ODOS. In contrast to other operating system kernels, which support object–oriented language on top of the kernel in extra layers, ODOS supports object–orientation directly in the kernel. ODOS' basic abstractions are objects. Everything in ODOS is founded on the concept of the addressing unit *Unique Identifier*, which is the base class for all objects in the system. All other objects are derived from *Unique Identifier* class and inherit its properties.

11.3 Personalities

- *Binary Compatibility*

  For a success on the market a new operating system has to have binary compatibility with operating system which presently dominates the kind of hardware on which the new system should run. Binary compatibility is broadly accepted feature by the users and administrators. Old programs do not have to be recompiled and users do not have to learn new system interfaces. Microkernel operating system can replace the existing monolithic structures, since it is more flexible in supporting different operating systems. ODOS will support at least the native operating system. The upgrade therefore to the new ODOS does not have any disadvantages for a user or an administrator of a domain.

- *Mounting*

  Mounting of a new file systems is similar to the *symbolic ports* of Chorus. If in a pathname evaluation the fileserver gets to a mount point, it transfers the request to the fileserver, which handles this part of the filesystem tree.

- *Modular Design*

Due to the modularity of the ODOS system the complexity is decreased and the design process for new releases is supported.

- *Single System Semantic*

  ODOS is designed to be a distributed operating system. The use of conventional operating system on a distributed system was not flexible. Chorus introduced the idea of a *single system semantic* which offers the user the view of one system. The user operates the system in the same way as having only one uniprocessor. The operating system distributes the work in a way that offers a decent load balance. Processor pools, workstations, and supercomputers can work together to offer the user a performance, which he could never achieve with conventional kernels. There is no need to log into a remote machine of the same domain and export some work to that machine and exonerate the own host. This is done automatically.

- *Parallel Application Domain*

  ODOS defines parallel application domains in which load balancing is performed. The sites of one parallel application domain cooperate for the purpose of load balancing during startup time of a new task and after each period. After each period the loads of all sites are compared and a load balancing is done if appropriate. The parallel application domain is necessary because a system can have several thousand sites, which would keep the system busy by only performing administrative services. The membership of a processor in a parallel application domain is decided by the system administrator and should take the topology of the communication network into account.

- *Dynamic Reconfiguration*

  All services of ODOS are dynamically reconfigurable. Services which are not needed are first swapped out and later when the space is needed they are subject to unloading. System services are usually accessed through well–known ports so that a reload is automatically caused when a message is sent to that well–known port.

- *Personality Independent Layer*

  ODOS implements a personality–independent layer, which offers general services such as fileserver, netwareserver, pipebuffering, devicedrivers,

and displaying. Subsystems can use general servers or implement the service for their specific needs.

11.4 Inter–Process Communication

- *Message Passing*

  ODOS is a purely message passing operating system kernel. Even communication with the kernel is often done by passing messages to the kernel and the kernel passes back the reply. To influence the behavior of a task the parent or the kernel sends messages to the *control port*. In each task a *control thread* receives messages on this port and reacts in the way as it is described in the message. Messages such as remote *thread creation* and *task destruction* are possible. Messages to this control port have a strong impact on the behaviour of the task; therefore only a few tasks have sendright to this port.

- *send–once rights*

  To support the client–server concept with a better security, ODOS allows ports to give *sendonce* rights. If a task gives another task a *sendonce* right to a port this task has the right to send exactly one message to this port. The right is automatically deleted after being used. A second message would result in a *noright* messages.

- *out of line data*

  To support large message transfers in a shared memory the kernel does not transfer the data, it marks the page where the data resides as *copy–on–write*. Only a message header is transfered, which indicates that the message is an *out of line data* message and it also indicates where the actual data of the message reside.

- *Unique Identifier*

  Ports are addressed by their unique identifier. The unique identifier is unique in space and time. By knowing an UI and having a *sendright* to this UI a location–independent communication is performed. The *Unique Identifier* in ODOS is addressed by a 128 bit number, which includes the creation site and the creation time.

- *Well–Known Ports*

The *well–known ports* do not have a site identifier. The first few bits are 0's. Well–known port numbers can be looked up by sending a message, which described the service, to the well–known port naming process. The list of services and UIs is saved in a fault–tolerant manner. Some ports are fixed for all systems. Others can be included into this list by sending a message to the *well–known port server*. Therefore ODOS offers a configuration of well–known ports and a customization for services a system requires. Locations of well–known ports are distributed every period. Therefore a direct communication can appear and the port naming process is not a bottleneck of the system.

- *Subsystem own IDs*

  Subsystems can have their own ids. The subsystem decides if a subsystem process gets a global or local portname. All processes have to have a UI but beside this UI it can have several local ids. Processes even do not have to know their UIs if they interact only directly with the subsystem. For scheduling and for communication with the *control thread* the kernel uses the UI. For interaction with the subsystem the task can use internally the id of the subsystem. The local ids are mapped to the global UIs in the subsystem. Therefore it is possible for UNIX subsystems to keep their file descriptors and to process ids as integers. Files in ODOS are accessed through ports. A file descriptor of a UNIX process is mapped onto the global port. File memory objects are used to speed up access to data that are stored in files.

- *Kernel Ports*

  Every kernel has a set of its own ports through which threads of other kernels can communicate with threads of the kernel. The UI numbers of these ports are created out of the site number and some well–known ports which are fixed for all systems, but can be expanded if kernels offer new services to other kernels. These ports cannot migrate.

- *Networking*

  ODOS is a distributed operating system. There is the necessity to expand communication over system boundaries. In ODOS one or more servers are responsible for inter–domain communication. These servers are well–known servers listening to a port–group. Port–group receives all messages which shall be sent to the outside. The servers are usually running on sites, which are connected directly with the outside world.

Messages to a task in the system are received by other threads of these tasks and are transferred into local IPC. For the outside world the whole domain looks like one computer and all networking is done through a task running on this *single system*. There can be different servers for different protocols used on various networks.

- *Port Migration*

  Ports are derived from UIs and so they are able to migrate from site to site. The location–independent IPC guarantees that the UIs will still get the messages which are send to them.

- *Local and Remote Servers*

  Due to the location–independent IPC, the server realizing a request of a client, does not have to be on the same site as the client. This reduces the number of servers which has to be loaded. More memory is available for other tasks and this speeds up computation because of the extension of the time when tasks reside in memory without being swapped out.

- *LRPC and URPC*

  Optimized IPC protocols are used in shared memory.

11.5 Fault Tolerance

- *Well–Known Port Reconfiguration*

  If a message is sent to a well–known *deadname*, then the server which usually realizes the requests is reloaded automatically.

- *Easy Fault Tolerance Extension*

  It is easy to create a fault–tolerant server due the fact that ODOS offers port–groups. If the *receiving mode* is set to broadcast, then every task which is a member of that port–group gets a copy of the request. All servers can compute and after completion they can compare their results. The result on which most servers agree is sent back as the result of the service request.

11.6 Kernel

- *Modular Design*

The modularity of all system components decreases the complexity of the design process and supports better the design process for new releases of the system.

- *Modular Kernel Design*

  In ODOS even the kernel is designed in a modular way. If some specific hardware is not present, then some modules do not have to be loaded. It is faster if, at this address, the appropriate module is called which simulates the non existing hardware.

- *Exchangeable Scheduler*

  The kernel offers the option to exchange a scheduler. It offers an advantage that a user–defined scheduler is integrated into the kernel. Therefore special purpose schedulers, which deliver better real–time facilities, are possible with ODOS.

Chapter 12

Conclusion

This thesis shows the reason why operating system builders had to explore new ways of building operating systems. The operating system requirements for parallel hardware and real–time system have been presented.

The need for several personalities in one system has been expressed. In the thesis it is shown how different personalities are implemented in existing systems such as Mach or Chorus. The multi–server concept of the independent layer in ODOS is the most efficient way to support the implementation of new personalities.

Memory management in a distributed operating system has many new challenges which have been described and design decisions which have to be made have been explained. A distributed shared memory concept has been applied to support the parallel execution of several threads of one task.

The object–orientation of ODOS is shown by defining classes which handle interprocess communication and process management. The advantages of a fully object–oriented operating system have been pointed out and specifications of the most fundamental classes have been given. RPC optimization protocols have been described and their incorporation into ODOS has been explored.

The result of this thesis is a specification of ODOS' crucial concepts and class definitions of basic object classes. A future project will be an implementation of these requirements and specification to show that these concepts and classes can work efficiently. The first step in this direction will be a simulation of the kernel on top of an existing operating system. Due to the portable code of ODOS an integration into system space will be a very late step.

Appendix A

Comparison between Microkernels

Item	ODOS	Mach	Chorus
Target system	single CPU up to **distributed system**	single CPU, multiprocessor	single CPU, multicomputer
Hardware environment	Workstations, PCs, networks	Workstations	Workstations, Multicomputer
System calls	some system calls	many system calls	some system calls
Optimized for	remote case and some optimizations for local case	Local case	remote case and some optimizations for local case
Multithreading	Yes	Yes	Yes
Thread managed by	Kernel	Kernel	Kernel
Load Balancing	**available**	none	remote execution possible

Table A.1: Some differences between ODOS, Mach, and Chorus

Item	ODOS	Mach	Chorus
Address space	page based	page based	segment based
Mapped objects	pages, but memory objects in work	memory objects	segments
Demand paging	yes	yes	yes
Copy on write	yes	yes	yes
Distributed shared memory	page based	page based	region based
Capability based	Yes	Yes	Yes
Capabilities for	services(through ports)	ports	services (through ports)
Capabilities in	Kernel	Kernel	Kernel
Communication Model	message passing, RPC, group communication	message passing, RPC	message passing, RPC, group communication
Msgs addressed to	UIs (ports, portgroups)	Ports	UIs (ports, portgroups)
Intermachine msgs	kernel space (outside a domain: userspace)	user space	kernel space
Transparent Heterogeneity	**Yes, based on a unique identification of all items in a domain**	No	Yes, Unique Identifiers for ports
Low level protocol	own protocol	IP	
reliable group communication	**Yes**	No	
Stub generator	Yes	Yes	Yes
User level server	Yes	Yes	Yes
UNIX emulation	Yes	Binary BSD 4.4	Binary System V
Distribution of UNIX server	Yes	BSD 4.3 No; Multiserver Yes	Yes

Table A.2: Comparison of Memory Management, Communication, Capabilities, and Emulations in ODOS, Mach, and Chorus

Bibliography

[AD92] Francois Armand and Robert Dean. Data Movement in Kernelized Systems. *In: Proc. of the Usenix Workshop on Micro-kernels and Other Kernel Architectures, Seattle, WA, April 27-28*, 1992.

[AGP+91] L. Albinson, D. Grabas, P. Piovesan, M. Tombroff, C. Tricot, and H. Yassaie. UNIX on loosely coupled architecture: the CHORUS/miX approach. Technical Report CS-TR-91-49, The Chorus Operating system, 1991. (Proc. EIT Workshop on Parallel and Distributed Workstation Systems, Florence, Italy, 26-27 September 91.).

[AJJ+92] Paulo Amaral, Christian Jacquemot, Peter Jensen, Rodger Lea, and Adam Mirowski. Transparent Object Migration in COOL-2. *In: Proc. of Workshop on Dynamic Object Placement and Load-Balancing in Parallel and Distributed Systems, ECOOP'92, Utrecht, The Netherlands, June 29*, 1992.

[AJL92] Paulo Amaral, Christian Jacquemot, and Rodger Lea. A model for persistent shared memory addressing in distributed systems. *In: Proc, of IWOOOS'92, Paris, France, September 24-25*, 1992.

[ALJ92] Paulo Amaral, Rodger Lea, and Christian Jacquemot. Implementing a modular object oriented operating system on top of Chorus. In *OpenForum*, pages 193–204, Utrecht (NL), November 1992.

[ARG89] V. Abrossimov, M. Rozier, and M. Gien. Virtual Memory Management in Chorus. *In: Proc. of Progress in Distributed Operating Systems and Distributed Systems Management, Springer Verlag, Berlin, April*, 1989.

[ARS89] V. Abrossimov, M. Rozier, and M. Shapiro. Generic Virtual Memory Management in Operating Systems Kernels. *12th ACM Sym-*

posium on Operating Systems Principles, Litchfield Park, AZ, December 3-6, 1989.

[BAEL90] Brian Bershad, T. Anderson, E.Lazowska, and E. Levy. Lightweight Remote Procedure Call. *ACM Trans. Computer Systems*, 8 No 1, 1990.

[Bar91] Joseph S. Barrera. A Fast Mach Network IPC Implementation. Usenix Mach Symposium, 1991.

[BB92] Jose C. Brustoloni and Brian N. Bershad. Simple Protocol Processing for High-Bandwith Low-Latency Net. Technical Report CMU-CS-93-132, School of Computer Science, Carnegie Mellon University, 1992.

[BBB+90] Robert V. Baron, David Black, William Bolosky, Jonathan Chew, Richard P. Draves, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Wayne Young. Mach Kernel Interface Manual. August 1990.

[Ber92] Brian N. Bershad. The Increasing Irrelevance of IPC Performance for Microkernel Operating Systems. 1992.

[BGG+91] A. Bricker, M. Gien, M. Guillemont, J. Lipkis, D. Orr, and M. Rozier. A new look at micro-kernel-based UNIX operating systems: Lessons in performance and compatability. Technical Report CS-TR-91-7, The Chorus Operating system, 1991. (Proc. of the EurOpen Spring'91 Conference, Tromsoe, Norway, 20-24 May 1991.).

[BGH+92] Nariman Batlivala, Barry Gleeson, Jim Hamrick, Scott Lurndal, Darren Price, James Soddy, and Vadim Abrossimov. Experience with SVR4 over CHORUS. In *Proc. of Usenix Workshop on Microkernels and Other Kernel Architectures*, pages 223–241, Seattle, WA, April 1992. Usenix Association.

[BGR+88] David L. Black, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael W. Young. The Mach Exception Handling Facility. Technical Report CMU-CS-88-129, School of Computer Science, Carnegie Mellon University, 1988.

[Bla89] David L. Black. The Mach cpu-server: An Implementation of Processor Allocation. Unpublished manuscript from the School of Computer Science, Carnegie Mellon University, 1989.

[Bla90a] David L. Black. Scheduling and Resource Management Techniques for Multiprocessors. Master's thesis, School of Computer Science, Carnegie Mellon University, 1990.

[Bla90b] David L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. Technical Report CMU-CS-90-125, School of Computer Science, Carnegie Mellon University, 1990.

[BZ91] Brian N. Bershad and Matthew J. Zekauskas. Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie Mellon University, 1991.

[BZS93] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. Technical Report CMU-CS 93-119, School of Computer Science, Carnegie Mellon University, 1993.

[CD88] Eric C. Cooper and Richard P. Draves. C Threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, 1988.

[CDK94] George Coulouris, Jean Dollimore, and Tim Knidberg. *Distributed Systems, Concepts and Design.* Addison-Wesley, second edition, 1994.

[CS94] Thomas L. Casavant and Mukesh Singhal, editors. *Distributed Computing Systems.* IEEE Computer Society Press, 1994.

[DBRD91] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. Technical Report CMU-CS-91-115, School of Computer Science, Carnegie Mellon University, 1991.

[Dea93] Randall Dean. Using Continuations to Build a User-Level Threads Library. Third USENIX Mach Conference, 1993.

[Dra90] Richard P. Draves. A Revised IPC Interface. USENIX Mach Conference, 1990.

[Dra91] Richard P. Draves. Page Replacement and Reference Bit Emulation in Mach. Usenix Mach Symposium, 1991.

[FBYR89] Alessandro Forin, Joseph Barrera, Michael Young, and Richard Rashid. Design, Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach. Winter USENIX Conference, 1989.

[FGB91] Alessandro Forin, David Golub, and Brian Bershad. An I/O System for Mach. Usenix Mach Symposium, November 1991.

[FM91] Alessandro Forin and Gerald R. Malan. An MS-DOS File System for UNIX. Usenix Mach Symposium, 1991.

[GBB93] Michael Ginsberg, Robert V. Baron, and Brian N. Bershad. Using the Mach Communication Primatives in X11. Technical Report CMU-CS-93-121, School of Computer Science, Carnegie Mellon University, 1993.

[GD91] David B. Golub and Richard P. Draves. Moving the Default Memory Manager out of the Mach Kernel. Usenix Mach Symposium, 1991.

[GDFR90] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. UNIX as an Application Program. USENIX Summer Conference, 1990.

[GG91] M. Gien and L. Grob. Micro-kernel based operating systems: Moving UNIX onto modern system architectures. Technical Report CS-TR-91-81, The Chorus Operating system, 1991. (Proc. UniForum'92 Conference, San Francisco, CA, 22-24 January 1991.).

[GGT+91] D. Grabas, M. Guillemont, M. Tombroff, C. Tricot, and H. Yassaie. CHORUS on H1: UNIX System V on Networks of Transputers. *In: Proc. of TRANSPUTING'91, Sunnyvale, CA, April 22-27*, 1991.

[Gie90] Michel Gien. Micro-kernel architecture - key to modern systems design. Technical Report CS-TR-90-42, The Chorus Operating system, 1990. (Unix Review, November 1990.).

[Gie91] Michel Gien. Next Generation Operating Systems Architecture. *In: Proc. International Workshop on Operating Systems of the 90s and Beyond, Dagstuhl Castle, Germany, July*, 1991.

[Gie92] Michel Gien. From Operating Systems to Cooperative Operating Environments. *Chorus System, Technical report*, 1992.

[Gui90] M. Guillemont. Microkernel design yields Real Time in a Distributed Environment. *Computer Technology Review, Winter*, pages 13–22, 1990.

[HMA90] S. Habert, L. Mosseri, and V. Abbrosimov. COOL:Kernel support for object oriented environments. *In: Proc. of OOPSLA'90, Ottawa, Canada*, 1990.

[HP92] B. Hermann and L. Philippe. CHORUS/MiX, a Distributed UNIX, on Multicomputers. *In: Proc. of Transputers'92, Arc et Senans, France, May 20-22*, 1992.

[Int87] Intel. ipsc/2 system. *Product and Market Information*, 08 1987.

[JCS+91] Daniel P. Julin, Jonathan J. Chew, J. Mark Stevenson, Paulo Guedes, Paul Neves, and Paul Roy. Generalized Emulation Services for Mach 3.0: Overview, Experiences and Current Status. Usenix Mach Symposium, 1991.

[JR86] Michael B. Jones and Richard F. Rashid. Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems. Technical Report CS-CMU-88-129, School for Computer Science, Carnegie Mellon University, 1986.

[Jul89] Dan Julin. Network Server Design, Mach Networking Group. September 1989.

[LJ92] Rodger Lea and Christian Jacquemot. The COOL architecture and abstractions for object–oriented distributed operating systems. *In: Proc. of 5th ACM SIGOPS European Workshop on Models and Paradigms for Distributed Systems Structuring, Rennes, France, September 21-23*, 1992.

[LS94] Sunggu Lee and Kang G. Shin. *Probabilistic Diagnosis of Multiprocessor Systems*, chapter 5, pages 207–222. In Casavant and Singhal [CS94], 1994.

[Mal92] Gerald Malan. Mach DOS Update. *Open Software Foundation Newsletter*, 1992.

[MB92] Chris Maeda and Brian N. Bershad. Networking Performances for Microkernels. Third Workshop on Workstation Operating Systems(WWOS-3), 1992.

[MB93] Chris Maeda and Brian N. Bershad. Protocol Service Decomposition for High-Peformance Networking. 14th ACM Symposium on Operating Principles, 1993.

[MRGB91] Gerald Malan, Richard Rashid, David Golub, and Robert Baron. DOS as a Mach 3.0 Application. Usenix Mach Symposium, 1991.

[Mul85] Sape Mullender. *Principles of Distributed Operating System Design.* PhD thesis, Vrije Universiteit Amsterdam, 1985.

[Mul93] Sape Mullender. *Distributed Systems.* Addison-Wesley, second edition, 1993.

[OJ91] Paulo Guedes (Open Software Foundation) and Daniel Julin. Object-Oriented Interfaces in the Mach 3.0 Multi-Server System. IEEE Second International Workshop on Object Orientation in Operating System, 1991.

[PBB+94] David Powell, Peter Barrett, Gottfried Bonn, Marc Chereque, Douglas Seaton, and Paulo Verissimo. *The Delta-4 Distributed Fault Tolerant Architecture*, chapter 5, pages 223–248. In Casavant and Singhal [CS94], 1994.

[Pou94] Dick Pountain. Parallel Course. *Byte*, July, 1994.

[RAA+90] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating Systems. Technical Report CS-TR-90-25, Chorus Systems, 1990.

[Raj89] R. Rajkumar. *Task Synchronization in Real-Time Systems.* PhD thesis, Carnegie Mellon University, 1989.

[RBF+89] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr, and Richard Sanzi. Mach: A Foundation for Open Systems. Second Workshop on Workstation Operating Systems(WWOS2), 1989.

[RJO+89] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, and Michael Jones. Mach: A System Software kernel. 34th Computer Society International Conference COMPCON 89, 1989.

[RT90] Richard F. Rashid and Hideyuki Tokuda. Mach: A System Sofware Kernel. Symposium on Computational Technology for Flight Vehicles, 1990.

[RTY+87] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. Technical Report CMU-CS-87-140, School of Computer Science, Carnegie Mellon University, 1987.

[SBB94] M.K. Saxana, K.K. Biswas, and P.C.P. Bhatt. *Problem Solving with Distributed Knowledge*, chapter 5, pages 267–283. In Casavant and Singhal [CS94], 1994.

[SG94] Abraham Siberschatz and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, fourth edition, 1994.

[SRL87] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. Technical Report CMU-CS-87-181, Carnegie Mellon University, 1987.

[ST93] Stefan Savage and Hideyuki Tokuda. Real Time Mach: Exporting Time to the User. Third Usenix Mach Symposium(Machnix), 1993.

[Tev87] Avadis Tevanian. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1987.

[TLC85] M.M. Theimer, K.A. Lantz, and D.R. Cheriton. Preemtable remote execution facilities for the v-system. In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 2–12, Orcas Island, WA, 1985.

[TMIM89] H. Tokuda, C.W. Mercer, Y. Ishikawa, and T.E. Marchok. Priority Inversion in real-time communication. 10 th IEEE Real-Time System Symposium, 1989.

[TN91] Hideyuki Tokuda and Tatsuo Nakajima. Evaluation of Real-Time Synchronization in Real-Time Mach. USENIX 1991 Mach Workshop, 1991.

[TNR90] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-Time Mach: Towards Predictable Real-Time Systems. USENIX 1990 Mach Workshop, 1990.

[Tok91] Hideyuki Tokuda. RT-Thread Model for Real-Time Mach. RT-SOSS, 1991.

[TR87] Avadis Tevanian and Richard F. Rashid. Mach: A Basis for Future UNIX Development. Technical Report CMU-CS-87-139, School of Computer Science, Carnegie Mellon University, 1987.

[TRG+87] Avadis Tevanian, Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, and Michael W. Young. Mach Threads and the Unix Kernel: The Battle for Control. Technical Report CMU-CS-87-149, School of Computer Science, Carnegie Mellon University, 1987.

[TRY+87] Avadis Tevanian, Richard F. Rashid, Michael Young, David B. Golub, Mary R. Thompson, William Bolosky, and Richard Sanzi. A Unix Interface for Shared Memory and Memory Mapped Files Under Mach. Technical report, School of Computer Science, Carnegie Mellon University, 1987.

[UR94] Shambhu J. Upadhyaya and Aravindan Ranganathan. *Rollback Recovery in Concurrent Systems*, chapter 5, pages 249–266. In Casavant and Singhal [CS94], 1994.

[Wen87] James W. Wendorf. Operating System/Application Concurrency in Tightly Coupled Multi-Processor Systems. Master's thesis, School of Computer Science, Carnegie Mellon University, 1987.

[WIK92] Jonathan Walpole, Jon Inouye, and Ravindranath Konuru. Modularity and interfaces in micro-kernel design and implementation: a case study of Chorus on the HP PA-RISC. pages 71–82. Usenix, April 1992.

[WT88a] Linda R. Walmer and Mary R. Thompson. A Programmer's Guide to the Mach System Calls. Technical report, School of Computer Science, Carnegie Mellon University, 1988.

[WT88b] Linda R. Walmer and Mary R. Thompson. A Programmer's Guide to the Mach User Environment. Technical report, School of Computer Science, Carnegie Mellon University, 1988.

[Zay87] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the Eleventh Symposium on Operating System Principles*, pages 13–24, Austin, TX, 1987.