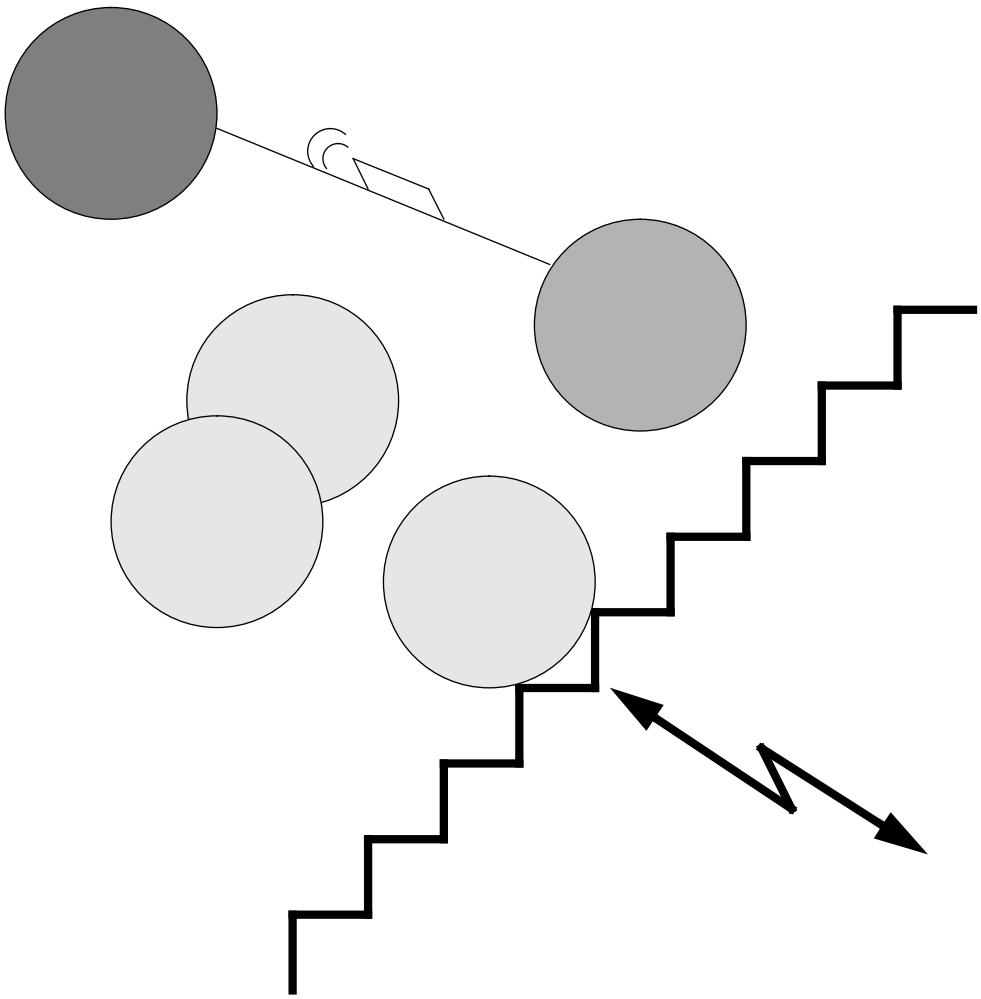
Background Analysis and Design of ABOS, an Agent-Based Operating System

Author: Mikael Svahnberg, pt94msv@student.hk-r.se



A Software Engineering Master's Thesis at the University of Karlskrona/Ronneby, 1998.

Abstract

Modern operating systems should be *extensible* and *flexible*. This means that the operating system should be able to accept new behaviour and change existing behaviour without too much trouble and that it should ideally also be able to do this without any, or very little, downtime. Furthermore, during the past years the importance of the network has increased drastically, creating a demand for operating systems to function in a distributed environment. To achieve this flexibility and distributedness, I have designed and evaluated ABOS, an Agent-Based Operating System. ABOS uses agents to solve all the tasks of the operating system kernel, thus moving away from traditional monolithic kernel structures. Early results show that I have gained in flexibility and modularity, creating a fault-tolerant distributed operating system that can adapt and be adapted to almost any situation with negligible decrease in performance. Within ABOS some tasks has been designed further, and there exists a demonstration of how the agent-based filesystem might work.

Keywords: Operating systems, Practical application of multi-agent systems

Author
Mikael Svahnberg, pt94msv@student.hk-r.se

Advisors
Håkan Grahn, hakan.grahn@ide.hk-r.se

Paul Davidsson, paul.davidsson@ide.hk-r.se

A Software Engineering Master's Thesis at the University of Karlskrona/Ronneby, 1998.

Table of Contents

1. Introduction

This chapter will give an overview and background for the thesis, as well as presenting a survey on what the current status is in the common operating systems. Futhermore the trends regarding operating systems in the research community will be investigated.

1.1 Overview

Requirements on modern operating systems are that they should be *extensible* and *flexible* [6]. This means that the operating system should be able to accept new behaviour and change existing behaviour without too much trouble. In truly distributed environments they should ideally also be able to do this without any, or very little, downtime. This is due to the trouble of for example migrating processes running on the machine in question. During the past years the importance of the network has increased drastically. Thus, operating systems also need to be more or less distributed.

As always, performance is also an issue. The traditional way to solve performance problems is to embed everything into one single, monolithic kernel, while still using a microkernel design and object-oriented programming to achieve flexibility. This enables modules to communicate via shared memory and simple procedure calls, instead of using the overhead of inter process communication (IPC). Obviously, this embedding is in conflict with the requirement of flexibility. When every new function has to be imported or implemented in the kernel space, one cannot easily add new services on-the-fly. Indeed, it is also the exact opposite of what one wishes to achieve by using a microkernel design. The cause of this embedding is, as stated, the overhead of using IPC calls. However, recent studies [1] have shown that the overhead for IPC calls has decreased enough to make this a practical approach.

The benefits one can gain by using IPC to communicate within the kernel are substantial. Modules can be exchanged during run-time and new ones can be added without having to reboot or recompile, which sometimes is the case. New strategies and resources can be changed and added as easy as they should be. By using IPC, one can also easily and transparently run some of the services on another machine, thus creating a truly distributed system.

A concept that is beginning to see the light and that can address the questions above is *agents*. Agents are small software components with certain qualities, described later in this paper. Interesting about agents is that they are autonomous, uses IPC and can adapt over time. This makes them highly suitable for employing within an operating system kernel. My idea is to explore whether agents can be used to facilitate the tasks in an operating system. I aim to present a model where agents reside within the kernel of an operating system. Furthermore, I will present a design solution for some common tasks that a distributed operating system performs.

This thesis will present a brief overview of the most commonly known operating systems, followed by a presentation of some of the research performed in operating systems. Succeeding this there is a presentation of agents and agent technologies to clarify what an agent is. There is also a survey showing what attempts has been made to apply agents in operating systems. To get some understanding of what the requirements and problems are in a modern operating system the section following this survey will deal with what an operating system should perform. Once this is clear we can move on to the presentation of ABOS, an agent-based operating system. Some examples of more top-level tasks are also given, after which it is time to evaluate the agent operating system. This is done and topped of with some concluding comments.

1.2 Current Praxis

In this section, I will present some of the more well-known operating systems. Some of them are maybe not so well known, but they are representatives of a group of operating systems. No one can write anything about objects and agents without mentioning CORBA, so I will present this as well.

UNIX

UNIX is the gathering name of a number of operating systems that share some equalities. UNIX systems usually rely on one of the two "grandfathers of all UNIX's", Berkley UNIX and System V. This may sound as if there exists many different implementations of UNIX, but these are basically just different variations of the overall architecture.

UNIX is structured as a number of layers, with the hardware at the bottom. On top of this is the parts that run in kernel mode like process management, memory management, I/O and the file system. The bottom level in the user mode consists of standard libraries like open, close, read, write, fork etc. The top-layer consists of the programs. These can be anything from shells, editors and compilers to databases and advanced applications.

Distribution is not a part of the original UNIX, even though networking was an early part of it. The distribution is, in fact, limited to the ability to remotely execute programs and a file system that allows transparent mapping of remote disks. The kernel layout in itself varies much from different versions but generally one cannot exchange kernel parts without restarting the system. From this I draw the conclusion that the various tasks of the kernel is highly intertwined. Some parts, like time synchronization and, in fact, most of the other functionality runs in user mode.

Windows NT

Windows NT[1] is a single-user multi-tasking operating system developed by Microsoft. This is one of the few operating systems around that was developed commercially from the beginning. The basic assumption for the design of Windows NT is that people will be using the same machine, probably residing on their desktop, but they might want to run more than one application simultaneously. [8]

The structure of the kernel is rather complex, and is easier explained by an image (Figure 1, as presented by Stallings [8]) than by text. Unlike UNIX the system is not truly layered and unlike Aegis, Amoeba, and Mach (see below) much of the control code resides in kernel mode. The Hardware Abstraction Layer makes the implementation of Windows NT platform independent and the subsystem architecture makes it client independent.

The layout of the NT Executive enables a great flexibility in the choice of for example process manager. Much of the kernel resides in DLLs, dynamically linked libraries, which makes it easy to exchange parts like the security manager. Novell utilizes this in their directory service for Windows NT [12]. It is a pity that Microsoft has not taken care of this flexibility and support the exchange of kernel modules by providing open interfaces and explicit support for third-party developers.

---

[1]Windows NT is a registered trademark of Microsoft Corp.

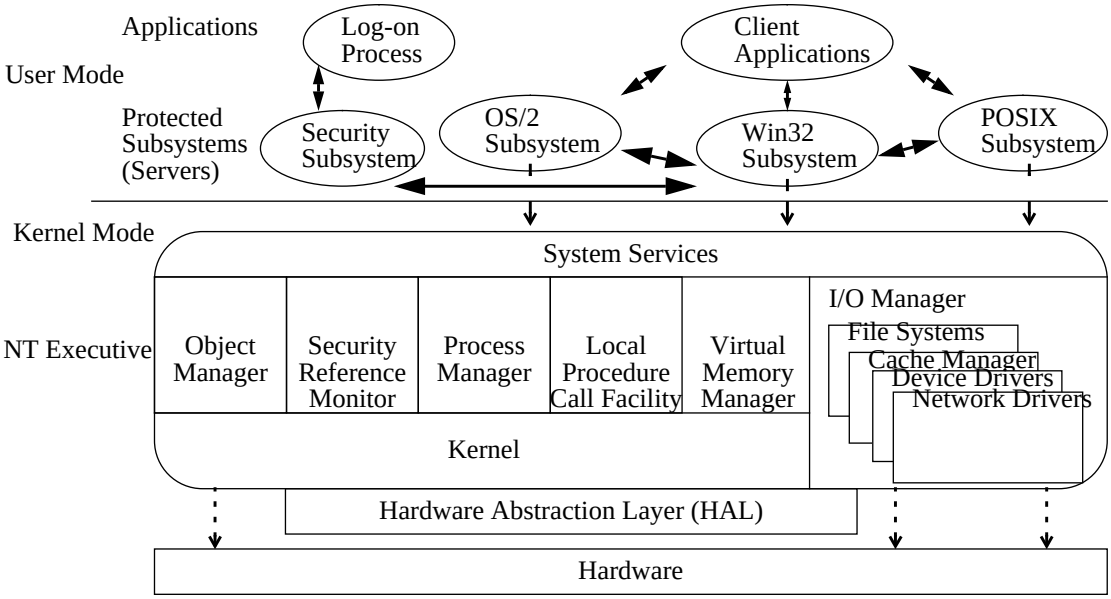


Figure 1. Windows NT Structure

Windows NT has many shortcomings like not being able to distribute load over the network, not even in the crude way UNIX does. Having most of the system running in kernel mode makes it very hard to adapt new services without rebooting. As mentioned, the subsystem layout ensures that the kernel is essentially 'client independent' since both OS/2 and POSIX applications can run on top of the NT kernel. This is used in some UNIX ports for Windows NT [13].

Amoeba

Amoeba [7] is a distributed operating system, developed as a research project by Andrew S. Tanenbaum *et al.* at the Vrije Universiteit, Amsterdam. The goals of Amoeba is to be a transparent distributed operating system. This means that the user should not be aware that he is using more than one computer while working. In contrast to many other approaches the load is balanced across the entire system without preference to a specific machine. Two assumptions are made about the hardware, that both limits and aids Amoeba: Systems will have a very large number of CPUs and each CPU will have tens of megabytes of memory. Based on these assumptions Amoeba is designed to use a pool of processors, accessed via X-terminals.

The basic layout of Amoeba is that of a microkernel that manages processes, threads, memory, communication, and low-level I/O. Clients and servers rest on top of this microkernel. As with the case of Aegis everything from file systems to resource allocation are managed through servers that run in user space. The notion of objects is central to Amoeba. Everything is encapsulated into an object and managed by a server. Objects are accessed using a cryptographically protected capability, a handle to the object. All access to servers is done by using a system-global port number. This port number is present in all object capabilities so that one can easily find the responsible server process. The port number is not machine specific so if a server migrates to another CPU or system the port number will remain the same.

Amoeba is based on many outdated assumptions, that have sprung from the correct assumptions about many CPUs and lots of memory. The processor pools in the style of an ancient mainframe that Tanenbaum assumes will, I argue, not happen again. My prediction is that the ongoing trend with more CPUs sharing the same memory space in servers as well as in some clients will continue and even though Sun, Oracle, and

IBM are making a great show about their thin clients, so called NCs [14], no one is talking about removing the processing power from the clients. The alternative, to have dedicated workstations that are part of the processor pool, is visible in today's clustering techniques [15], but only organizations with high requirements on servers will use clustering. In the case of memory, Tanenbaum *et al* were correct in that machines will have tens of megabytes of memory, but their assumption that this excess would be theirs for the taking to achieve performance is proven wrong by the applications of today that sometimes requires even larger amounts of memory for themselves.

Be that as it may, Amoeba shows many bright ideas as well. To start with, it is distributed in the deepest sense of the word with process migration and communication primitives supported by the kernel. As with Aegis, most of the system runs in user mode, thus enabling fast and easy switching of functionality.

Mach

Mach [9] [10] is perhaps the most well-known microkernel operating system. It was initially developed as a research project at the university of Rochester and continued by Carnegie Mellon University. The goal of Mach is to demonstrate that you can structure operating systems in a modular way.

Mach has a microkernel that performs process and thread management, memory management, communication, and I/O services. On top of this microkernel rests, in user space, a software emulator layer. In this layer other operating systems are emulated like UNIX, Windows NT, or even another Mach kernel. File systems and other handy stuff are managed by these emulators. Mach supports communication between processes at kernel level using the concept of ports. Ports reside in the kernel and acts as message queues.

The microkernel architecture in Mach is limited to relying on an operating systems emulator running on top of the kernel. The advantages one can gain by exploiting the microkernel is thus left to the capriciousness of the emulator. The fact that IPC is supported in the kernel only makes the situation worse since it inhibits the emulators from implementing smarter or more suitable primitives. As it is, all they can do is to act as an interface to the Mach ports. The reason for this emulator strategy is, I think, to support as wide a range of software as possible.

I have found no evidence of the kernel itself being modularized, and distribution seems not to have been the main issue when developing Mach although it does support inter-machine communication using the Network Message Server [7].

CORBA

CORBA [11], or Common Object Request Broker Architecture, is a standard defined by OMG, the Object Management Group. CORBA is not in itself an operating system, but it has become somewhat of a standard if one wishes to communicate with objects over a network.

The CORBA design is usually described as if the CORBA ORB, object request broker, replaces the network. Clients are hooked on to the ORB and perform their requests to an object, running somewhere else. The actual communication work is usually done by stubs on the client side and skeletons on the server side that in turn calls the actual object. The stub can be replaced by a Dynamic Invocation Interface, allowing you to access the object anyway using an Interface Definition Language defined by OMG.

Many of the research operating systems [17] [18] use stubs to access the services in the object-oriented kernel. CORBA has the possibility to use stubs, but can also call objects dynamically. This introduces a great flexibility. As we will see with agents further on a language for invoking objects must be defined for agents as well. In the case of CORBA they use their own IDL. The trouble is that it is only possible to send data as if it were a normal function call. You must also be aware of which functions that you can call, whereas agents usually have a common language in which they can talk about what to talk about. The calls are also synchronous, meaning that the caller is suspended until an answer is returned. Furthermore a function call reeks somewhat of client-server which, as we will see, is not compatible with the agent paradigm.

CORBA is usually implemented on top of the operating system, as a part of the applications that wishes to use it and as a separate ORB. Having to rely on kernel-level primitives to actually send data inflicts performance. As yet, I have not seen any operating systems that support CORBA in the kernel.

1.3 Ongoing Research

Having presented some of the popular operating systems and object enabling techniques I will now move on to the research community with focus on what is being said and done in the field of operating systems. In particular, two research projects shine more than the others. These are Spring [5] from Sun and Aegis [6] from M.I.T.

Generally, the research in operating systems tends to be much aimed at object-orientation. There is much discussion regarding persistent objects, migrating objects and processes [3], and even omnipresent objects [4]. One of the main issues seems to be to invent a global naming scheme to be able to find objects over the network. Persistent objects discussions seems to be aimed at how to store objects without loosing too much in performance. All in all, there is much emphasis on *how* and not so much on *what* to do. The research on objects are on varying levels of detail. Some suggest kernel support [2] for persistent objects and others deal with more abstract reasoning on how to name objects in a global network [19].

Spring

Spring is an experimental distributed environment developed at Sun Microsystems. It consists of a distributed operating system and a support framework for distributed applications. Briefly one can describe the structure of Spring as being a set of interfaces rather than actual implementations. This is a decision taken to support the creation of many differing implementations of a given interface. Spring has a specified interface language [17] in which all interfaces should be written. From these definitions stubs are generated for the programming language of choice.

Spring has a somewhat different object model compared to other approaches. A standard approach is to have an object reference locally that points to a remote object. Spring distributes the actual object, making sure that it can only exist in one place at a time. Before passing the object on one can make a copy of it. These two objects will then point to the same underlying state.

The kernel in Spring is object-oriented, and all access to objects in the kernel is done through stubs defined in the IDL. At a quick glance this looks as an agent system except that the IDL and calling mechanisms is as limited as that of CORBA in that it works like ordinary function calls.

Nevertheless, Spring is a serious effort to make a flexible and modularized system but I have not found evidence that the objects in the kernel can be replaced during run-

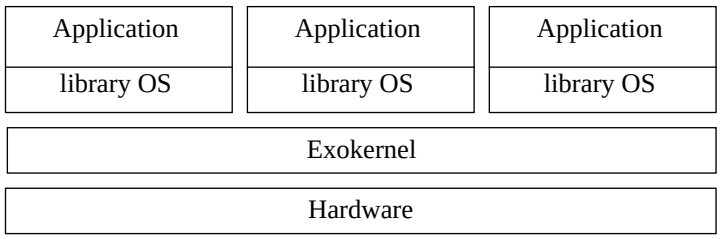


Figure 2. Aegis design

time, which is a requirement to make the system run-time flexible. Spring is stuck in the old client-server paradigm and assumes that a server should only reply with an answer and not 'talk back' to the client.

Aegis

Aegis is developed at the M.I.T laboratory for Computer Science. The goal of the project is to prove the point that operating systems need not act as an abstraction layer for applications. The underlying idea behind Aegis is that the only task that the operating system should perform is to allocate resources securely to client applications. This is because the authors feel that operating systems have become increasingly large in their pursuit to support every available device in every possible desired way. As they put it: "Applications know better than operating systems what the goal of their resource management decisions should be." [6]. This implies that as long as security can be guaranteed everything from caching to actual file system layout should be handed over to the client applications. These can then, by using library operating systems, access the hardware.

The functions that the exokernel performs can be summarized as to securely *expose hardware*, *expose allocation*, *expose names* and *expose revocation*. This means that it should expose as much of the hardware, its DMA capabilities, etc. as possible and that all allocation should be done in consensus with the library operating system. Physical names should be exported, and the kernel should visibly reclaim resources according to a defined protocol.

The layout of Aegis is more of a standard operating system than a distributed ditto. No emphasis is put on distributing objects or similar topics of interest in distributed operating systems. Flexibility is achieved by minimizing the kernel functions.

Aegis may not have much to do with the construction of a distributed operating system but it's design proves a very important point; user-mode software can equally well manage resources as the kernel. This theory is used in many other more interesting operating systems like Amoeba and Mach. It also implies that the kernel can be minimal and still provide the functionality required.

1.4 Summary

A general trend in the newer operating systems seems to be to move as much as possible away from the kernel. It is interesting to notice that the operating system that increases most in usage [16], Windows NT, goes in the exact opposite direction by adding more and more functionality to the kernel.

Distribution seems in most cases (except Amoeba) to be to show the user that there exists more than one computer in the network, sometimes giving the user a possibility to execute software remotely. File systems are centralized and shared across the network. Inter-process communication is many times cumbersome for the applications and the possibility for it is often added to the system almost as an afterthought.

The only development company that has truly spent time trying to make a modularized, distributed, and flexible system is Sun with their Spring system. However, the object invocation in Spring suffers from the same disadvantages as CORBA in that you must know the names of the functions to call.

All of these problems can be addressed if one uses the abilities in a microkernel architecture to load and lose modules at run-time. Another aid needed is to give ample support for inter-process communication and inter-machine communication.

File systems, finally, is traditionally viewed as part of the kernel and in some extent it has to be so that at least the file system server can be loaded but there is no need to view the entire file system, its ability to be shared, or its caching mechanisms as part of the kernel. By utilizing the microkernel design once again, file systems of arbitrary complexity may be built.

2. Agents

In the previous section I presented some of the operating systems around, and the research being performed in the area of operating systems. The main focus of this survey was what could be regarded as agents or modular design in operating systems of today. Agents were mentioned in the previous section, but to get a better understanding of the concept they require more explanation. This chapter tries to present such an explanation. The explanation is followed by some examples of agents and their uses in connection to operating systems.

2.1 Introduction

The software industry was revolutionized when object orientation was introduced in the early 1970:s [20]. The critics claimed that object orientation did not contribute anything new; that in theory you could not do things that you had not managed before, and that the Church-Turing thesis about computability [21] still held. Indeed, you could not do anything new with object orientation; it did not make non-computable problems computable all of a sudden. What object orientation did, however, was to bring structure into chaos. Development times decreased, the amount of re-use increased, and the training time of new staff decreased since they did not really have to understand the entire system before starting to work on some part of it [22].

Agents provide the same paradigm change as object orientation once did, but this time on the level of processes. Just as object orientation introduced a new level of abstraction, agents introduce another, even higher abstraction level. Instead of having one enormous piece of software that encapsulates all functionality, agents are usually small modules with very well-specified purpose that interacts with other agents to achieve some specific goal contributing to the desired total functionality. Naturally, the benefits from such a design is best found in distributed computing. By having programs that are socially aware of other programs on the network, you can distribute tasks and control over the network. It is also very easy to exchange parts of the functionality by adding new agents that either influence the existing agents to achieve the goal differently or to replace some of the earlier agents altogether.

When designing an agent solution, there is a major difference compared to an object oriented design. In object orientation you design according to the objects that constitute the actors in the solution. Agent designing is task-oriented. Instead of looking at what actors are involved in an operation, you look at what tasks and subtasks the operation consists of. Agents are then created to solve these tasks. Whereas object orientation does not say anything about the actual tasks but rather expects the objects to solve them implicitly, agent orientation concentrates on the tasks at hand and creates actors that can help in solving these tasks. An agent has thus a clearly outspoken goal with its existence, and this goal is part of the design, decreasing the need for a design rationale. Despite these differences, it is not my intention to say that object orientation cannot coexist with agent orientation. An agent can very well be composed of a set of objects, just as an object can be a complete agent and an agent can be considered to be a complex object.

Agents bring enormous advantages in conceptual grasp of what is done, but you can still not do more than before. Still the Church-Turing thesis hold. Calculations of algorithmic complexity are still valid, and NP-complete problems still take exponential time to solve. However, you can distribute the algorithm to more than one computer, thus engaging the CPU speed in all of the machines. This is also something that has been done for a long time, but with severe implications on the readability of the code and lots of tricky protocols and communications overhead.

Despite all of the benefits one can gain by using agents, there is no clear definition of what an agent is. People tend to claim that they have developed an agent solution because it is state-of-the-art technology, even when it is nothing more than an ordinary program. To create an understanding of my view of what an agent is, a definition of what I see as an agent will be presented. This definition is vital in order to understand the intentions of the rest of this paper.

I will explain what is needed to be called an agent according to the general opinion, and top this off with my view of what an agent should do. I will explain how agents work together, what communication protocols they use, and what a multiagent system is. An investigation of what has been done about agents in relation to operating systems is rounded off by motivating why agents are so suitable for this particular use.

2.2 Agent characteristics

So far, the agent community experiences troubles in determining what should be called an agent, and there are as many definitions as people trying to define agents. This diversity of agent definitions is both an advantage and a disadvantage. The advantage is that one can easily fit in much new and interesting behaviour into the agent paradigm, whereas the disadvantage is that you can never tell that "this is an 100% agent". The only thing one can say is that a program is more or less of an agent, with respect of how many agent qualities it possess and to what extent. A definition of the agent qualities that I like, because it is fairly simple, is summarized by Hyacinth S. Nwana [24].

According to Nwana's definition, an agent can be *static* or *mobile*. A static agent does all its work from one single computer and has no wish nor mechanisms to move to another host, as the mobile agents does. An agent can furthermore be classified as *deliberative* or *reactive*, where a deliberative agent has the ability to reason and can plan its own actions to coordinate and negotiate with other agents. A deliberative agent is also known as *pro-active*, because it can initiate a chain of events without external influence. A reactive agent responds to changes in its surrounding environment according to a preset pattern. A reactive agent is idle until it receives some signal, which it then processes.

In addition to this, an agent must possess at least two of the qualities *autonomous*, *learning* or *cooperative*. *Autonomous* means that the agents should be able to operate without guidance from human operators. This also implies that they should be proactive, to take the initiative in causing changes rather than simply reacting in response to the environment. *Cooperation* refers to the social ability to interact with other agents and humans using some communication language. To achieve the sense of smartness, an agent's actions must be based on previous events, so it needs to be able to *learn* what has happened, and use this learning when making decisions.

Figure 3 illustrates these qualities. If an agent is autonomous and cooperative it is said to be collaborative. If it is cooperative and learning it is a collaborative learning agent. An autonomous learning agent is called an interface agent because they are generally interfacing towards the user, serving him in some way. Typical examples of interface agents are personal assistants and customizable search agents.

The areas in Figure 3 are not definitive, it is just a statement of what areas a given agent focus more on. The cooperativeness can, for example, have many degrees, ranging from simple signals to complex messages in a communications protocol like KQML [30]. I am also not entirely certain about the classification of autonomous learning agents as interface agents. I can think of many examples where an autono-

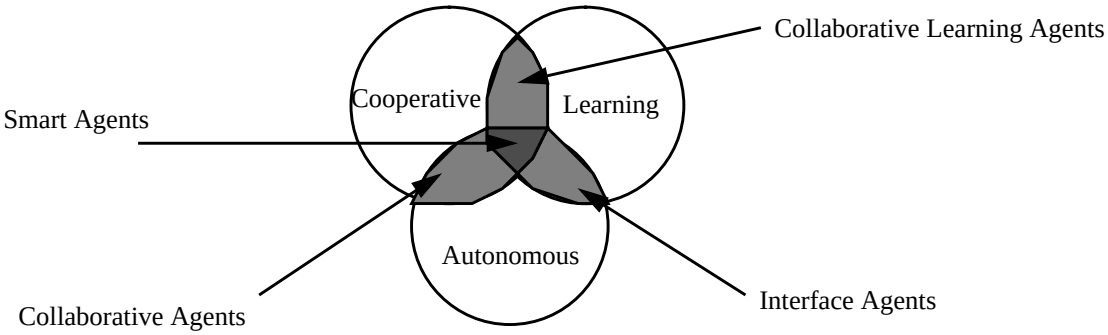


Figure 3. Agent Typology

mous learning agent does not interface to anything, but rather observes the environment to adapt its own behaviour.

Pro-activeness is a concept stressed by the agent community. The fact that a program can take the initiative to start a chain of events is something that appeal to the agent researchers as it emphasizes the fact that agents are autonomous. I claim that no program can ever be pro-active. No matter how you see it, it is some external event that starts the chain. The first event is always that the program is started. After this, many other events of varying kinds can be the cause for the agent to react, but even if the agent uses polling to check the state of something, it is bound to the timer events. If it decides to enter a tight loop instead of falling asleep between the pollings, there is still the start-up-event. Consequently all actions, even if they are preceded by two weeks of computations, are the result of an external event, which means that the agent is reactive and not pro-active. All of this makes it hard to say that an agent is pro-active. The common definition of pro-activeness is that the agent reacts to some event and foresees future results from this event and takes action against or towards these results. A pro-active agent is thus a reactive agent that predicts future events and how pro-active it is just a definition of how far into the future the agent can predict. If one views the agent as a physical entity, for example some embedded equipment, the reasoning about timer events above is falsified, because in such cases the timer event is generated by the agent itself. My reasoning still holds though, because the agent is still started at some point and this is of course also an event.

Agents originated from the artificial intelligence (AI) community. AI researchers being as they are tend to want agents to have all sorts of AI concepts like planners, theorem provers and such [34]. This strategy makes agents more complex and actually more unfit for some tasks, being forced to carry around extra functionality. I claim that an agent need not be smart as defined by the AI community to be called intelligent. Indeed, Ekdahl argues against the use of terminology like intelligent, reflective, or even learning software [35] since this is inherently not possible within a formal system, and a computer can not perform anything which is not describable in a formal system. My view is more relaxed. I have no objections to using the word smart or intelligent about an agent, but my interpretation is that the agent is programmed in a smart way or that it seems more intelligent in its behaviour than other software. As for learning, I agree with Ekdahl that the only learning that a computer program can do is to obtain information already present. This is also enough for the tasks that I see fit for agent use, and is certainly enough to achieve a higher service degree by not repeating mistakes and keeping faulty assumptions.

There is a common assumption that to be called agent, the software needs to be mobile. Following the above definition, this is obviously not needed. The communication primitives ensures that you can reach any agent, no matter where it resides.

Mobility does, however, give certain advantages in some situations, but requires the agents to be small enough to be able to migrate. A task that require much communication can be done by sending an agent to negotiate for you, thus reducing the network load. Another example is when your machine can be disconnected from the network. In such cases, you can send off the agent onto the network first, then disconnect your computer and connect it again somewhere else. The agent will find you in your new location and "dock" with your computer again, providing you with information you have requested [33].

Agent systems can use one of two approaches; a federated approach or a fully autonomous [27]. In a federated approach agents are not truly autonomous and relies on support from a facilitator, an agent host. All communication goes through this facilitator and it provides the agents with their view of the world. In the case where agents are fully automated they keep track of their reality themselves and contain support for communication. This can be viewed as a standard process running on an operating system, whereas the federated model looks more like threads within a process. If one should use agents in an operating system the federated approach is naturally not feasible unless one sees the operating system kernel as the facilitator. To have one and only one process running which manages the rest of the operating system as internal threads falls on its own ridiculousness, even if it can be argued that this is exactly the situation we are faced with in many operating systems today.

The 'three-quality-model', described above, says nothing about the size of an agent and it may be hard to actually say anything about this. I claim that an agent should have a limited but very well-defined behaviour. This enables modular programming in multiagent systems according to the same principles that underlies Sun's JavaBeans [25] and Microsoft's Active-X technology [26]. More complex behaviour is generated by combining a set of agents that each perform a single task very well. Keeping the agents small also facilitates the ability to migrate, which is often desired even if not always needed.

2.3 Multiagent systems

Agents rarely come alone. Usually agents are part of some larger system of interacting pieces of software. These systems are usually referred to as multiagent systems. In a multiagent system each of the agents take care of a small and well-defined task. A single agent need not be intelligent, but the total of the system achieves more than simple behaviors.

One important thing to note is that the agents are peers. No agent is worth more than others, and they have equal control. Even if some agent decides to solve a task by breaking it down into subtasks and let other agents help in solving the subtasks, the aides are not considered to be serving the one that had the initial task. The agent system solves the task as a whole with no definition of what is client and what is server. Hence, there are no rules as to who is allowed to send a message to whom, whereas in a client-server solution the server is usually silent until the client initiates a communication link.

As with everything else, there is no single way to design these multiagent systems. One way is to view the agent collaboration as that of a blackboard pattern [28], where the agents communicate via a server process, the blackboard. The blackboard is not a server in the true sense, but rather a coordinator. It can contain the data that the other agents put there and also decide who should get a chance to work with the data next. The blackboard may in turn be distributed using a hierarchical tree of connected blackboards. In a blackboard system all agents have equal status and control is handed out

by the actual blackboard to the most suitable. I claim that such a design violates the idea of autonomicity since the agents do not get the right to decide for themselves who is next to run. The idea to have certain agents that contain the shared state of many agents is however sound. These could act as repositories for information needed by more than one agent. They can also volunteer information to agents that they think should do something about the information. Co-ordinating agents like the blackboard agents are called intermediate agents.

Blackboard collaboration is just one of many ways to solve a task. One of the simplest collaboration forms to grasp is that of subtasking. In subtasking one agent commits itself to a certain task and divides it into smaller tasks that is handed out to other agents, that in turn can subtask this even further [29].

Even if the agents are peers, they can either be cloned from the same agent, or they can be specialized agents. Both of these strategies have their advantages. If the agents are cloned, they need to contain everything needed for the entire model, even if they do not exercise all of these skills at one given time. A multiagent system would, instead of having several agents each implementing the complete behavior, have agents that each comprise a part of the solution. These partial solutions, albeit cloned, can together see and solve the task at hand. The other form is to have a heterogeneous environment in which no agent has to be like any other. As with the homogenous environment these agents together see the full task and the entire environment, collaborating to solve their tasks. The difference here is that certain agents may have special and unique skills, whereas in the homogenous environment all agents possess the exact same skills.

2.4 Agent enablers

Having talked about agents in general and about collaboration models it may be interesting to look at some of the techniques used for achieving this modularity and social abilities. The agent community has more or less agreed upon a standard communications language, KQML [30], that supports the easy communication required. I will present this language in brief, showing the main concepts of it. I will also present a programming language designed to facilitate the building of agents and agent systems.

KQML

To enable agents to talk to each other, you need a standard way of communicating. The ideal language is called ACL, Agent Communications Language. This ACL is commonly used by theoreticians when they need a language, but there exists no implementations or even definitions of ACL. A number of approaches has been made to this ACL, of which KQML is one of the more well-known, together with FIPA's proposal [38]. A KQML message consists of a performative, and a set of arguments that states who is the caller, the recipient, a message id and a reply-message id, which ontology to use and finally the actual command or contents. The language of the contents is stated as another argument and can be in for example Prolog or KIF [37]. The performative states the type of the message, for example 'inform' or 'request', and the contents contains the data of the inform message or the wishes for a request.

With this structure a KQML message looks very much like an e-mail message. I believe that KQML is a sound approximation to the platonic ACL. The fact that the message body can be stated in any language allows you to select the language most suitable for the task at hand. On the other hand, you risk getting agents that are incapable of communicating with each other because they do not understand the language of the other. As long as you understand the language used, you can understand messages

of arbitrary complexity in almost any discipline because the ontology used is stated in the message. I am not saying that this is simple, only that it is possible. Implementing support for ontologies is quite complex and may not be needed for simple homogenous systems.

April

April [31] is a process oriented programming language developed at Imperial College, London by Keith Clark *et al*. It contains ample support for creating processes and communicate between the processes. Each process will have a unique identifier or a handle associated with it. The identifier can either be provided by the programmer or automatically generated by the system. An API is also provided that allows easy integration of April processes with external programmes. Making use of the API allows these external programs to exchange April messages with ordinary April processes. April gives the programmers an easy way to create agents. Communication primitives and process management is handled by the April abstract machine, and is not dependent on any underlying operating system. If the April abstract machine were to run directly on top of the hardware, one could expect gains in for example performance.

As a programming language April provides just the primitives one would want in order to develop agents. It is not visible to the user where process management is done and the communication primitives are easy to use. Finding another process or agent is a relatively easy task for the programmer. Code can be sent to execute in another agent, enabling agents to learn new behaviour over time. All of this makes April an example of a successful platform for implementing agents. As for communication, April does not lock you into a certain language, on the contrary it can be used to implement such languages. For example, you can use April to easily provide an implementation of the KQML communication language.

2.5 Operating system agents

With this definition of agents and how they interact in mind, it would be interesting to see what has already been done with agents in operating systems. However, it is very hard to find evidence of anyone attempting to support agents in operating systems. Looking through both what is done in the operating systems community and the agent community, few papers deals with agents and operating systems together. The TACOMA project [23] has developed an agent platform for mobile agents. Another example is the mail transfer system MOTIS [32], which is also an ISO-standard. The most well-known example of agents in operating systems are probably the various daemons in UNIX systems. They might not be known as agents but, as I will show, they certainly are.

As seen earlier, many attempts has been made to define naming schemes for objects, defining how to make objects persistent and so on. These objects usually lack the autonomous quality or some of the other qualities needed to be called an agent. Also, as said before, there is more emphasis on *how* to support objects rather than *what* the objects should do. In the agent community, the situation is quite the opposite. Here one assumes underlying communication primitives and support for agents exists and works and instead defines the behaviors of the agents. Nevertheless, the discussions are usually held on a high level, not giving any concrete examples of what needs to be done and not one even hints at using agents to perform operating system tasks.

TACOMA

The TACOMA project is an exception to this ‘what’-wave visible in agent research communities, since it is concerned with operating system support for mobile agents. The TACOMA project uses mobile agents and the metaphor that they should do as people; visit a place, use a service and move on. Granted, if the service to use involves lots of communication or transactions requiring security, the mobility of the software is a desired feature. For most tasks, though, the network can equally well relay the messages instead of the agent binary code. The TACOMA model also employs a model with folders, or briefcases, to carry an agent’s state around the network. Surprisingly, these folders are not part of the agent, but must be transferred to wherever the agent wishes to use them.

TACOMA is implemented in Tcl/Tk [36] on top of a UNIX kernel. This means that you need a Tcl interpreter on each host in the network, acting as a sort of facilitator for the agents. With the TACOMA agent support, a mechanism for exchanging electronic cash has been implemented to demonstrate its abilities. Agent-based schemes for scheduling and fault-tolerance has also been implemented using TACOMA.

The main disadvantage of the TACOMA project is that they focus on mobile agents, claiming that this is the only type of agents that the operating system needs to support. They have built an agent platform in Tcl/Tk on top of a UNIX kernel, but still claim that they have implemented operating system support for agents. Their main focus is that of electronic commerce, adding solutions to the operating system tasks that are needed for this such as fault-tolerance and security.

MOTIS and X.400

MOTIS is similar to the SMTP [32] in the TCP/IP suite. MOTIS, which is more a complete message transfer system than a mere transfer protocol, is based on X.400 [32], defined by ITU-T. As with many ISO-standards regarding network protocols and systems it is very well defined but rarely used. Most systems today use the standards defined in the TCP/IP protocol suite.

The X.400 system can be characterized as having a User Agent (UA) that communicates to a local Message Transfer Agent (MTA) using a Submit/Deliver Service Element (SDSE). The MTA communicates to the recipient’s MTA using a Message Transfer Service Element (MTSE). The recipient’s User Agent then acquires the message from this MTA and presents it to the user. All in all, four protocols are used; one between User Agents, one between the SDSEs and two between the MTSEs. MOTIS looks very similar to the X.400 model but one MTA manages an entire site and acts as a bridge to other sites via the X.400 pubic message handling system.

The X.400 system is illustrated in Figure 4 as presented by Halsall [32]. The users communicate via a user agent to the SDSE. The SDSE connects to the local post office and another SDSE. After going through some checks and systems the message exits the post office via an MTSE to the recipient’s post office. From here on the path is reversed on the recipient’s side. The UA polls an SDSE that in turn connects to the local post office and receives the message. Communication from UA to UA is done in protocol P2, SDSEs communicate via P3/P7 and MTSEs via P1.

MOTIS at least calls the different software components agents, and indeed they are. The message system is autonomous in that it acts without a user interference. It can take decisions on where to route a message, and a smart system might even learn certain routes over time. It is also collaborative consisting of several autonomous units that communicate via defined protocols.

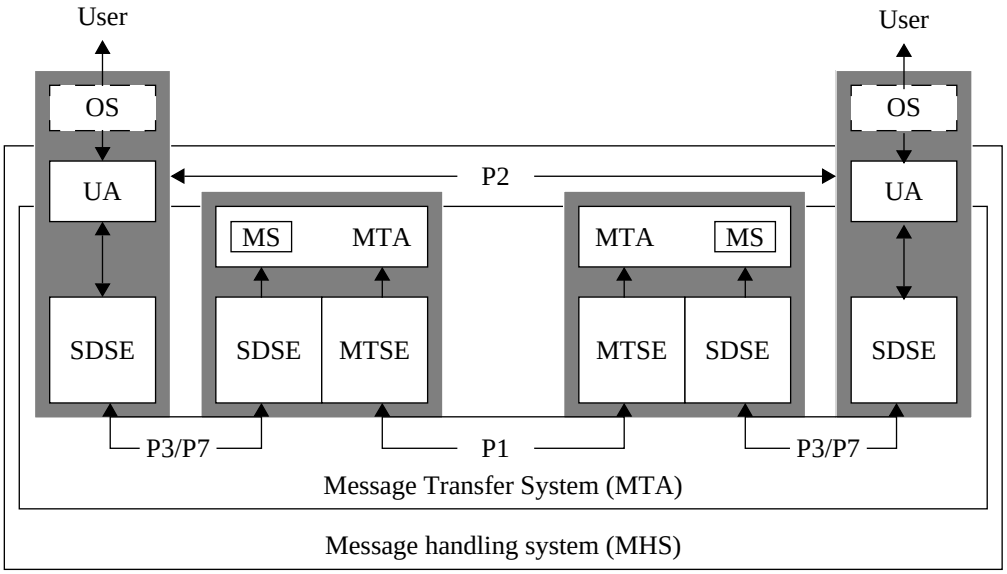


Figure 4. X.400 Functional model

There are two major guidelines in the network society. The first one is to use as many acronyms as possible, which is why I have included these above lest someone should miss them. The other guideline is not to use ISO-standards. ISO-standards are generally newer, meaning harder to integrate with existing standards, than the commonly used. They are also considered to be slower and causing more overhead because they are better structured with more layers and this usually decrease performance.

UNIX Daemons

To say that agents have never been used in operating systems before is not entirely true. In UNIX-systems much of the extra service that is provided and which is often viewed as part of the operating system is managed by so-called daemons. A daemon is a piece of software that takes care of some service, often listening to a certain communications channel. When requests come on this socket the daemon processes it accordingly. Examples of such daemons are the telnet daemon, the www daemon, and the ftp daemon. Other daemons react on the clock like the cron daemon, or on events in the system like the syslog daemon. Some daemons like the pageout daemon can be considered to be part of the kernel, but I argue that it is not. The pageout daemon exists merely to enhance performance by freeing memory for future use when the machine is more idle than otherwise, but the operating system can manage without this memory page cleanup by only evicting memory pages when needed.

These daemons have many qualities that would classify them as agents. They are reactive, since they idly listen to a port or device until a request comes along. They are autonomous, because they need no baby-sitting from the user, and the user is rarely even aware of their existence. Many of them are social, communicating with the caller according to some protocol to process the requests. In some cases like the sendmail daemon they are also collaborative, helping each other in delivering e-mails to the right user and the right place. Daemons like the telnet and ftp daemons does, however, have some qualities that makes it dubious whether they really are agents. A program that simply listens on a port and responds when spoken to belongs more to the client-server paradigm than the agent paradigm. To be called an agent, the program would need to be able to initiate communication using a peer-to-peer protocol. Nevertheless daemons are the closest thing to agents to be found in operating systems today. A common treat is that they provide additional functionality which is not needed by the

operating system but provided as an extra feature. Indeed, some daemons are not even provided together with the operating systems but have to be downloaded and installed as any other software.

2.6 Motivation to use agents

As we have seen in the case of UNIX-daemons agents are suitable for many of the service tasks in an operating system. Any task that requires monitoring of some device or network socket should manage without interference from the user. As I also have argued, such daemons are not quite agents but rather a standard client-server solution. Within the operating system kernel there are other tasks where the subsystems work more like peers. So does for example memory and process managers need to coordinate their work to suspend a process waiting for a memory page to be read from disk. I believe that by viewing the entire kernel of an operating system as a multiagent system, you can benefit from the increased support for communication and the possibilities to implement parts of the kernel with arbitrary levels of intelligence.

To have an operating system kernel that is composed of several stand-alone modules that interact in a structured and extensible way gives enormous possibilities to create a smart and extensible system. Instead of the standard interaction method of function calls you can really maintain a dialogue between the subsystems. Hopefully this, together with other techniques, will yield a more intelligent operating system. The fact that parts of the kernel can be replaced during run-time and that you can add more modules to help in a certain task makes such a system extremely flexible and extensible.

An operating system kernel that is composed of a set of autonomous modules, or agents, can seamlessly be made into a distributed operating system by adding a global naming scheme for the agents. You can, in fact, have a single memory and process manager for an entire network. Or you can have the memory and process managers communicate with each other to decide on scheduling policies as you would in a multiagent system.

3. Operating System Tasks

This section takes a deeper look at what tasks an operating system should perform, presenting the problems and pitfalls connected to each of the tasks. No attempt is made to solve the tasks, focusing instead on giving an unbiased view of the problems. This chapter should be read as an introduction to what I will attempt to solve with the agent-based operating system in the next chapter.

3.1 Introduction

A. Tanenbaum regards an operating system as either an extended machine providing a better interface to and an abstraction from the actual hardware, or as a resource manager providing support and control for hardware access [7]. The kernel should thus hide the complexities of for example memory and process management from the applications. It should also be able to allocate hardware to processes in a consistent and coherent way. This extended machine should basically hide memory layout, disk layout and I/O intrinsics. It should also hide the fact that more than one process is running on the same CPU from the applications. The last condition also implies that applications should see resources as their own all the time, even if they are shared by many other clients.

Using the ideas of Amoeba, Mach, and Aegis, the file system can be managed by a process running outside the kernel. There is also research suggesting that the kernel need not bother about process management either, but that this can be managed via CPU inheritance scheduling [39]. Aegis goes a step further, saying that the applications should themselves manage memory and I/O as well so that the operating system can live the easy life just pointing at who should get to do the work next. The four tasks identified are, however, not adequate for modern distributed operating systems. They were barely sufficient for ancient single-user, single-computer operating systems and distributed applications require more support for their distribution if they are ever going to be developed at a large scale.

During the rest of this chapter I will present tasks needed for a distributed operating system that, even if they may not be part of the kernel, needs to be addressed. Most of the information below is fetched from Tanenbaum [7] and Stallings [8].

3.2 Process Management

Process management ranges from CPU-time sheduling to process creation. You first need a way to create a process, either by creating a new process control block, or by cloning an existing. While the process is running, you need to be able to suspend it and let it wait for some device without consuming CPU time. The CPU should be shared among a set of processes with different priorities. The user should preferably not notice that he actually does not have one CPU for each program he runs, so priorities needs to be shuffled in accordance to the user activity. Processes that are waiting for something should not be kept running, since this is a waste of resources (CPU-time). The user should be able to address the different processes in some way, at least to be able to kill or suspend them.

A common way to manage processes is to keep information about them in a process table. The entry for a certain process typically contains its process ID, its priority, whether it is running or blocked and for scheduling purposes the amount of time it has been run. There is usually also information about the program counter, stack pointer, open files, and memory blocks that the process controls. Scheduling differs between various operating systems mostly in the way processes' time quanta are handled.

One might think that process management must reside in the kernel, but UNIX System V shows that this is not true. In System V, the processes manage most of the tasks described above using library functions within the process itself. The only concession, if you can call it that, is that the library functions are run in kernel mode. For standard, non-mobile processes this is a fine and well-working model, but if processes have the ability to move to another machine, it causes overhead if this code should be moved as well. The operating system code may not be able to run on the receiving machine either, since this host can be running another operating system in which the system code will not work.

3.3 Memory Management

Memory management is simply the concern to provide each process with memory mapped to its address space. This can be done simply by letting the process address all the physical memory, or by providing a virtual address space to the process and mapping the memory contents to a certain place in the physical memory. Of course, in a modern multi-process operating system you need the latter model since one of the main ideas is that processes should not be aware of other processes unless they really want to. Hence, it would not do if processes had to share memory space with someone else. This would also cause great problems with security issues. The usual way to solve the mapping between virtual memory to physical memory is by dividing the memory into pages and to swap in pages as they are needed. This paging is performed totally transparent for the processes, that does not know when it happens or how it works.

Processes can be presented either with a full view of all the memory, or with a set of segments, each comprising a part of the full address space. For some purposes this can be practical but, since segmentation is usually achieved by utilizing some of the bits in the address space to select the segment, I find segmentation rather pointless except as a conceptual notion. Segmentation made more sense before hardware supporting paging was invented and implemented into computers.

Another thing to take into consideration is whether processes should be allowed to share memory pages or not and, in such cases, how much and how to do it. Common approaches to this is to share full pages or segments from a pool of memory pages. This pool is often of a fixed size, thus limiting the amount of shared memory that the system can manage. Lots of research have also been put into how to share memory in distributed applications [40], letting processes on different hosts share memory as if they were working with the same physical memory.

With memory management there is not much that can differ between different operating systems since this is mostly controlled by the hardware. Paging, being a practical solution, is used by all modern operating systems and what varies is how selection of pages to evict is performed. Shared memory, on the other hand, can be managed in many different ways. It may for example be done blockwise or per segment basis. Another thing that can vary from system to system is process migration and replication. Since these tasks do not specifically involve hardware, it can be done in any of a number of ways.

As far as I can see, memory management can be handled in the same way as process management, by library functions in each process. Some parts can also be handled as stand-alone processes, e.g. swapping. In UNIX systems process 0 is the swap daemon, managing all swap activity and also the start-up of the first “real” process, the init daemon. Paging may also be done in a separate process but the cost of context-switches makes this solution impractical. A common approach is to have a separate pageout

process, responsible for freeing up memory to a certain level by paging out non-used memory pages.

3.4 I/O Management

I/O management is something that is consuming an increasing part of the code in operating systems. This is due to a generally held conception that the operating system should act as an abstraction layer between software and hardware, creating a uniform interface to all types of devices. Especially needed is this on Intel-based systems, where a plethora of possible devices exists.

The tasks involved in I/O and device management are numerous. Interrupts thrown by the device must be caught somewhere and distributed to the right process. Processes that have been blocked waiting for reply must be awaked (Which, in fact, is the process manager's task.). If the device communicates by direct memory access (DMA) the operating system must allocate buffers and make sure that these are not paged out, implying communication with the memory manager. Applications require intuitive names on the devices, names that must be maintained and kept uniform. The ever so important error handling must be managed and the operating system must throw errors that make sense to the applications if they expect error messages at all. The operating system must also decide whether the device can be shared by many users simultaneously, or whether applications must gain exclusive rights for the device.

I/O is usually handled by adding a device driver for each device to the operating system kernel. These drivers are commonly linked into the kernel on start-up, so that one needs to restart the computer for them to take effect or to delete one. The operating system will then provide interface to these device drivers in a uniform way so that for example disk drives are accessed in the same way as hard disks. Applications can either communicate directly to the device driver or through yet another abstraction layer handled by the operating system.

In some cases one wishes yet another layer, residing in user space mode. This layer acts as a buffer, so that no application can lock the device indefinitely, blocking access from others. Applications will instead talk to the buffer layer that can handle concurrent requests and queue them according to some criteria (e.g. 'First In First Out' or 'Shortest Job First'). For devices visible to the network, someone needs to manage queues and network communication to the device. This can be done either in the actual device driver or in a layer resting on top of the device drivers.

What differs between operating systems regarding I/O management is mainly how the devices are presented to the user and applications. At one extreme is UNIX, in which all devices are presented as either character devices or block devices, and at the other extreme is Windows NT, where you must basically know exactly what you want to do before you do it. In UNIX you need no special knowledge of a certain device, just its name under the '/dev'-directory, whereas in Windows NT you have to know that it is for example an audio device you are going to communicate with using a certain audio-API.

Usually the presentation scheme is deeply coded into the operating system kernel, whereas management of different devices is done by pluggable device drivers. It would thus be possible to make the presentation scheme into a specific module as well. This would yield a flexible operating system that can be tailored for specific needs or use a general naming scheme if desired. Processes need some place to communicate to register interest in certain events that devices may throw. This can, consequently, also be done in a certain replaceable module.

3.5 File system Management

Naturally, it would not do to present hard disks as simple block devices and ask of the user and applications to try and find their way around this. Some more abstraction is needed. Firstly something is needed to find out what block belong to what file, and whether the files are stored sequentially or blockwise. These files need some handle that can be presented to the user, and you will typically need some way to structure these handles into groups, or directories. The operating system will also need some strategy on how to cache files.

File systems have come to a standstill with a structure of directories in which one puts files. The files are commonly spread blockwise over the harddisk. How to find the blocks vary; some use linked lists, others use i-nodes. There is some variation how caching is performed and whether to use write-through caching or lazy writes. In some cases, for example in database systems, you wish to access the raw disk device since a standard file system simply do not have the structure and hence not the performance required. In such cases the operating system should preferably be able to present applications with a raw block device.

Having a local file system is only half the truth. In order to facilitate for system administrators, much of the software and home directories rest on a server, accessed via some network file system. These network file systems usually try to maintain the same protocols and access techniques as if they were local file systems. Sun's network file system, NFS [52], uses a technique where you have a virtual file system layer that decides whether a call should go to a local disk or an NFS disk. The client applications need thus not worry whether a call goes to a local disk or a remote.

In the past 20-odd years, not much has happened to file systems. File systems from different vendors use the same conceptual artifacts with files and directories. What differs is how files are stored on the physical disk, and how they are accessed. How disks and other file system media are presented to the user and system is another area where different operating systems differ vastly, but the concepts are the same. Files are dormant phenomenons, viewed as a storage dump for other applications. Fredriksson shows that documents can very well manage their own activities, being more than an passive occurrence [43]. I claim that this also holds for files at large. The time is ripe for a paradigm change regarding files and how they are viewed. In Chapter 5 I will explain further what such a file system would look like, and what benefits one can gain by applying Fredriksson's theories.

3.6 Communication support

The above presented topics are the ones that are traditionally viewed as the core functionality of an operating system kernel. Most operating systems also provide mechanisms for process communication. UNIX, for example, provides *pipes* which is a basic way to connect one thread or process with another. Amoeba, being a distributed system provide a mechanism similar to pipes, but with support for inter-machine communication as well.

In a modern distributed operating system I think that communication, and especially inter-machine communication, is highly important. Preferably, applications should not have to define their own protocol for data either unless they really want to, and the communication primitives should be intuitive and easy to use.

The problems regarding interprocess communication, IPC, are manifold. First of all there is the question of binding an application to a communication layer so that the application can receive data. Incoming data must somehow be transferred to the appli-

cation and its memory for processing, and outgoing messages needs to be transferred from the applications memory to outgoing buffers or the recipients memory space. Once this has been handled, there is the question of keeping connections alive so that a two-way communication can be upheld and at the same time ensure that the messages reach the right recipient. To get a connection, you need to have a name, or an address, to the process you wish to communicate to. This name should be visible over the network and it should also be possible to transparently send messages between machines. Support must be included for both synchronous and asynchronous message transfer, meaning that clients should be able to select whether they wish to block while the transfer is done or continue immediately.

Once the basic primitives have been implemented, there is the question of providing the programmers with an easy-to-use interface to these routines. RPC [41], CORBA [11], and RMI [42] are examples of such APIs. These APIs generally tries to give the illusion that you are calling a local function or a method in a local object, which puts some demands on the underlying communication primitives. In some cases a higher abstraction might use another API. It would, for example, be possible to implement a CORBA API using RPC.

A good support for communication should also include group communication primitives, with all the new problems that this brings. Group messages should be received simultaneously, or be perceived to be received simultaneously. Messages should arrive in the same order to all the recipients and you need some way to ensure that the message actually reached all the recipients or whether you should do a rollback on the operation.

Even if modern operating systems provide support for interprocess communication, it is often only supported for processes on the same machine. Commonly, there exists no easy-to-use low-level communication primitives to access other machines, and you need to know exactly where you are going before you can start communicating. What is needed is an API that makes it transparent whether the receiving process resides on the same machine or not, and a global naming service to find it, no matter where it may reside. This API could be implemented fully in higher-level layers, but for performance reasons, and because inter-process and inter-machine communication is becoming increasingly common, it is best housed by the operating system kernel.

3.7 Synchronization

Synchronization is related to communication, and especially group communication. To determine the ordering of messages some sort of timestamp or uniquely ordered ID is needed for each message. The easiest way would be if all the processes shared the exact same time down to the nanosecond and to include the send-time in messages. Combined with a host-based running ID this would make it an easy task for the recipient to figure out in which order to put messages and whether there are any missed messages.

However, due to imperfect crystals no computer clock ticks at exactly the same rate as another. Hence computers need to update their clock at regular intervals with some central time. This synchronization can either be done against a centralized clock or by some distributed algorithm. It can furthermore work using a polling method or a broadcast method. The troubles with having a distributed algorithm is that the synchronization is not accurate to the nanosecond, or even microsecond, depending on network traffic and machine load. For each step away from the original server this variance will increase and at the same time decreasing the accuracy of the time measurement. Other distributed algorithms exist in which each machine regularly broad-

casts its time, letting the others set their time as a mean value of all the times broadcast.

A time server usually gets its values from some global place, for example the nearest atomic clock, and the clients in a local network gets their time from the time server. This design is used to reduce the load on the global time servers and because communication within the local network is probably faster than to the atomic clock.

Why, then, is time synchronization such a cumbersome task in networked computer systems? In the 'real world', there exists a short wave radio station with call sign WWV that broadcasts a signal at the beginning of each UTC (Universal Coordinated Time) second. The accuracy of this signal is +/- 1 msec, but atmospheric disturbance makes it accurate only to +/- 10 msec. The reason that this signal can be so accurate when it cannot be so over a network cable is simply the fact that the WWV signal need not compete with other traffic. Thus, if one could halt all network traffic when the time broadcast is due, one could reach equal accuracy in this medium as well. Furthermore, knowing how many metres of cable there is between client and server makes it easy to calculate the propagation time.

For some strange reason time synchronization is not traditionally part of the operating system. The system must manually be set up to synchronize its time with a server. On UNIX systems a common entry in root's crontab[1]-file is to set the clock against some server. Berkeley UNIX, on the other hand, has a broadcasting method with a central time daemon that generates a mean time of all the clients connected to it. To set the time against an atomic clock, one often have to rely on third-party software like xntp [44].

Time synchronization is only one of the problems, but it is certainly the most difficult to solve. Other synchronization problems that need support by the operating system are semaphores and monitors. Monitors can be implemented using semaphores, so you in fact only need support for semaphores. To create a working semaphore you also need some hardware support, the test-and-set assembly call that tests if a memory bit is set and if not sets it. If it is already set, a message about this is returned and the process can go and add itself to a queue for the semaphore. Semaphores are commonly provided at a more abstract level as operating system calls. This call takes care of both the flag-checking and the queuing. When the semaphore is released, the operating system wakes up a suitable candidate from the waiting queue.

3.8 Security

One final topic that often is neglected is the issue of security. Security is related to all of the above topics. File systems needs some access control, I/O devices should only be accessed by authorized users, you should not be able to read data in another application's memory unless it explicitly gives you the right to do so, and unauthorized processes should not be able to snoop network packages not designated to them.

Another aspect of security is fault tolerance. *Force Majeur* events like earthquakes or power failures should preferably not result in loss of data, or the loss should at least be minimized. The human factor, which is sometimes even more disastrous and definitely more unpredictable than earthquakes or floods, should also be considered. Thus the system should not crash or cause irreparable damage because of a simple human error. A common cure against user errors is to require a certain access level to be able to do

[1]The cron-daemon runs tasks listed in the crontab at the specified time.

stupid things. Regarding *force majeur*, there is not much one can do, since most fault tolerance schemes result in lower performance. This is not to say that there is nothing one *can* do, it is just hard to make something fully fault tolerant. Database vendors have had this problem since the very start, and there exists sound principles describing how to achieve fault tolerance. In recent years the operating systems research community has begun to realize this as well and papers concerning for example log file systems [45] are beginning to emerge. Modern operating systems are also fairly fault tolerant without any modifications or special schemes.

The area in which breaches occur most often and where the most improvement can be made is in the area of protection against mischievous users and processes. In principle one can say that for every feature you provide the user you create at least one security problem. UNIX has long had the policy to trust its users to some extent. Hence, passwords are often sent as unencrypted text when connecting via telnet or ftp. The file in which passwords are stored is also commonly available by default. To acquire access to such a system is a mere matter of listening to the network traffic until someone starts up a telnet session. Using publicly available software it is just a question of time until one has decoded a number of passwords in the password file.

The reason for this naive approach in UNIX is the TCP/IP protocol, that does not include coding. TCP/IP only spans level 1 to 4 in the OSI-model [32], whereas coding and encryption is handled at level 6, the presentation layer. The implementors of the applications (like telnet and ftp) have not considered security as an issue, either ignoring the problem or assuming that a lower layer has taken care of it.

3.9 Summary

As you might have guessed, one of my main goals is to deprive the actual kernel of as much power as possible. The underlying theory behind this is the same as with aegis; it is not the operating system's task to manage resources, all it should to is to allocate them to processes in a secure way. I argue that almost everything can be handled by processes not part of the kernel. The kernel can accordingly be simplified enormously, thus reducing its size and increasing flexibility.

As the situation is today, most of the tasks above are handled within the kernel. A change in memory management or process management would result in a recompilation of the kernel, and most certainly a restart of the system. Adding and removing devices is a bit more flexible, but still often requires a restart. To add support for new file systems is equally cumbersome, even if some operating systems like Linux handles this fairly well [46]. As for security, the different subsystems often need to implement their own security check, or at least contact some global security manager before executing certain commands. This model is a paradise for security breaches, all you need to do is to make a service fault in such a way that the security check is skipped. A better solution would be to go through a security layer before reaching the service required.

In the previous section I explained the concept of agents. An agent is a small, standalone piece of software that can easily interact with other software. I claim that such agents are highly suitable for these kernel tasks, chiefly because of their flexibility and ability to interact. In the next section I will describe a microkernel operating system that supports these kernel agents.

4. Top-Level design of ABOS

In the previous section I described what tasks a modern operating system should perform. I made no attempt at solving the problems I described. In this section I will design ABOS, an Agent-Based Operating System. ABOS is an example of an operating system that uses agents at the kernel level. As in the previous chapter, I will make no explicit attempt at solving the problems, but rather concentrate on the design and functionality.

4.1 Introduction

The trade-off between flexibility and performance often result in large monolithic kernel structures where changes are done by recompiling the kernel. The kernels commonly only have one memory space because the cost for an in-process function call is considered significantly lower than an IPC-call. As I have indicated earlier, the kernel can be deprived of much control and still be functional provided that these tasks are handled elsewhere. This ‘elsewhere’ could be small, autonomous server processes, so called agents. Naturally, this also implies that IPC is today fast enough even for time-critical tasks.

Today, agent platforms are usually built on top of an existing operating system. If we are going to use agents in the kernel this is naturally not sufficient. I have earlier described two agent models, the fully autonomous and the federated agent platform. Fully autonomous usually means that each agent is run as a separate process, whereas in a federated system all the agents are run within a certain process, commonly called facilitator. In the system I propose, the agents will be run as separate processes, but they will still run on a core designed especially for the agents. In this sense the system will host fully autonomous agents but still be a federated agent system.

There exists similar operating system platforms that support dynamic objects and components like Shag/OS [47], GLOBE [49], and Paramecium [48]. I claim that the step from dynamic or persistent objects to agents is not that far, and my work will rely somewhat on the theories and conclusions of these and similar projects, but the ABOS kernel is not quite like any of these.

In this chapter I will draw an outline of a flexible and extensible agent-based operating system. Starting with the core I will work my way outwards to the end-user applications. I will not describe these in detail, but rather stop at the service level, where ordinary operating system boundaries go. Exactly how the tasks in the operating system are performed is not explained either, unless they are of particular interest for the agent aspects.

4.2 General layout

Flexibility is commonly achieved by modularity. Modularity alone is however not enough to achieve full flexibility. A kernel can be one large binary and still be built with separate modules. In such cases you need to recompile the kernel and restart the machine for every kernel level change. It will not help to store the modules separately either. They are still loaded into a shared memory space and are highly dependent on each other. By instead making the kernel extremely small and let everything in the operating system be run by separate processes, the same flexibility and more can be gained without recompilation and restart of the entire operating system.

ABOS consists of such a set of small and autonomous modules, or agents. They are autonomous in the sense that they require very little support from other modules, and even less support from the human users. The system is structured as a set of layers

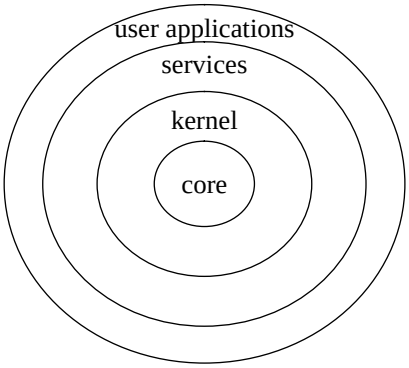


Figure 5. General layout of ABOS

around the core in the middle. The level of the services provided increase for each layer. The hardware is accessed wherever it is feasible, not forcing calls to propagate through more layers than is necessary. The layer structure is retained in spite of its decreased significance because it provide a conceptual structuring of the privileges that processes within a certain layer should possess.

Figure 5 illustrates the various layers in ABOS. The bottom layer is the *core* system. This system provides very basic support for process' and memory management. Around this layer is the actual *kernel* wrapped, providing more advanced process and memory management. File systems are also defined in the kernel, as are more advanced communication primitives and I/O management. Outside of the kernel layer is a *service* layer, providing things that are not part of an operating system kernel, but are still part of the operating system like user management and resource allocation. *User applications*, finally, are run on top of the service layer.

4.3 Core

At the very core of the system is the bootloader that in turn loads the core modules. The core modules take care of process management, memory management and communication primitives. The modules support very little intelligence, just a basic scheduler and primitive memory management. In fact, they should only support the creation and running of the advanced services in the kernel. It would also be convenient if the core modules were the only modules that interfaced the platform-specific hardware like CPU and memory, letting the kernel bother with the more strategic decisions.

The reason for this division between core and advanced service levels is to gain flexibility. By keeping the services at this level simple, more interesting functionality can replace the default without disturbing the low-level hardware interface. It is, however, not advisable to remove the core layer altogether since you need some support to get the rest of the system up and running.

Process management is required at this deep level to be able to run the rest of the kernel as separate processes. You need to have at least support for creating processes at this level. More advanced process management can very well be handled at the kernel level. The same reasoning holds for memory management. Once a process is created, its code needs to be loaded into memory. Everything else can be done by library functions within the process.

A process is created by a request to the process manager. The process manager initiates a process control block and assigns priorities etc. to it. A unique ID is also generated and associated with the process. This ID is used by all other processes to find the newly created one. At this level, no automation is present, so you need an extra request to the memory manager requesting a page allocation and the binary code to fill

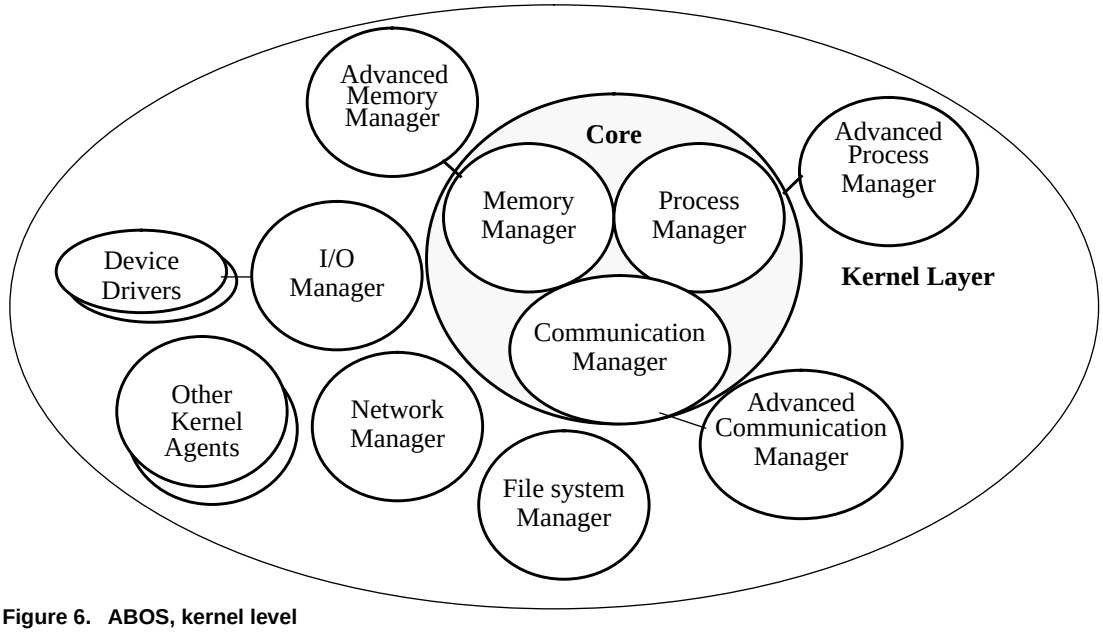


Figure 6. ABOS, kernel level

the pages with. With this allocation scheme it is easy to load the rest of the kernel since the binaries are specified on creation. The bootloader can load the core and then start to load the rest of the kernel using the core. As soon as enough of the kernel is loaded for it to carry on by itself the bootloader can terminate.

The support for process communication is also placed at this low level, since the kernel agents uses the available IPC mechanisms to communicate with each other and with the core system. At this level the communication is restricted to simple pipes, described in more detail below. Additional primitives to for example use a particular communication language is handled by library functions within the other processes. The core modules can of course also communicate with each other so that incoming messages for a process results in a wakeup of the process and memory is freed upon process termination.

4.4 Kernel

At the kernel layer the functionality is found that is more commonly part of the operating system. The following tasks are handled in the kernel layer:

- Final process management
- Additional memory management
- File systems management
- Communication naming schemes
- I/O management
- Network management

As seen in Figure 6, the core takes care of the basic memory management and process management, whereas advanced services resides in the kernel layer outside of the core. All managers should be considered agents, even those in the core. The core agents have given up part of their independence, but they are still autonomous and social. They all communicate using a predefined agent communication language, ACL.

The final scheduling algorithms for processes etc. are handled at this level, using the core process manager to do the actual switching of processes. Swapping and paging

algorithms are categorized as advanced services, using the core functions when needed to do the actual work. Listings of shared memory pools are also managed outside of the core, together with access information and authentication. The kernel agents use the core communication management to access the advanced communication services. The advanced communications manager provide intelligible naming and network-wide communication.

Letting the core manage all processor and memory access means that only the core functions and perhaps the device managers need to be run in kernel mode. The rest can very well be taken care of in user mode, even if it may be desirable for security reasons to let other tasks run in kernel mode. Removing the need for other agents to be aware of platform-specific details also means that there is a great flexibility to exchange parts without affecting others and without needing to rewrite these agents for every new platform.

Division between kernel and core

In the cases where you have a manager in the core and an advanced manager in the kernel it is vital that there is a clear definition of what should be done in the kernel and what should be done in the core. In my view the core should only manage the very basic things, leaving all policy decisions and more complicated services to the advanced managers in the kernel layer. Consequently, the core process manager should only manage the creation of new process control blocks and run a process until it yields or falls asleep on some operation. When a process is removed from the run-queue, a new process will be selected to run. The most basic run-queues (running, blocked, and ready) should be managed, but it should also be possible to create other queues and move processes between the queues on command from the advanced services. The advanced process manager should maintain these extra queues and command movement of processes according to its algorithms. Creating a process at the kernel level can also include more than one operation in the core. Memory may be allocated and communications channels set up, according to the advanced process manager's strategies.

The core memory manager should be able to allocate and deallocate virtual memory pages to physical memory. It should be possible to fill an allocated page or a set of pages with a binary string. The advanced memory management performs paging and swapping to and from disk. Lists of which memory pages that are shared are maintained by the advanced memory manager. Page faults are handled by the core memory manager, but the advanced memory manager has to be asked for what to do when a page fault occur.

Communication at the core layer is merely a matter of receiving an address and a string and deliver this to the right process, notifying the process manager that this process should be put in the ready queue if it is blocked. At this level, communication is done using pipes, similar to those used in UNIX. At the kernel layer, a process can create more than one communication channel, or service, binding it to a certain name. The kernel communication manager also has the ability to make a name visible over the entire network, and forwards messages via the network manager to the right host.

4.5 Services

The service layer is wrapped around the kernel layer. In an ordinary operating system there are tasks that belong to the operating system, but not to the kernel. These are traditionally run in user-mode with the help of kernel-mode programs and functions when needed. It is my opinion that by adding a certain layer in the operating system

structure for such services, they may get the acknowledgment they deserve without giving them too much power.

At this service layer things like resource allocation and time synchronization should reside. It is also here that the users are managed. This is not to say that there doesn't exist a user notion in the kernel and core, but at the service level this user ID is mapped against login names, authorities and restrictions, home directory paths and environment variables, etc. Process migration should be provided as a service at this level. Since this feature is not a part of the primary operating system functions it should not be included at kernel level. Agents for encryption and authentication is another thing that this layer provides. The reason is the same as for user management and process migration. The operating system can manage without it, but it is a convenience to provide support for it.

Since network management and communication is provided at a lower layer, the service layer can be global for the entire network, creating a foundation for a distributed environment and hiding the peculiarities of the individual systems. Many of the services are global by their nature, like user management, but it might be practical to duplicate the services to every machine for performance reasons. Since the services are agents it is no problem for them to cooperate and propagate changes to other machines in the network.

4.6 User Applications

The final layer is the user applications. These typically involve user interfaces and are more interactive than the previous layers. A smart process manager could take advantage of this information and manage processes' time quanta and priorities accordingly, giving larger quanta for services than for user applications. Reducing time quanta for interactive processes is something which is utilized in Windows NT [50] where time quanta differ between the server version and the workstation version of the operating system. In both UNIX and Windows NT, every process that receives user input from keyboard or mouse gets a boost in priority, which is another way of ensuring that user processes gets the CPU time needed. By identifying processes as user applications or system services, one can make such priority boosts even more fine-grained.

4.7 Summary

Comparing this operating system architecture with the commonly available kernel architectures, it differs mostly in that the kernel functionality is run as a number of stand-alone processes. The number of context switches increase, and hence the performance decrease. However, the flexibility is increased manifold by allowing new services to be added or replace old ones at runtime. For instance, you can install a new release of the operating system without even having to restart the computer! Since the functionality is all located in autonomous modules, all dependencies and open connections will still be open and valid when a new service is installed to replace an older one. Unlike UNIX where almost the entire operating system is duplicated as library functions into every process [8], the functionality in ABOS is divided into stand-alone processes.

The division into the four layers core, kernel, service, and user gives a possibility to consider the type of process when scheduling and setting time quanta. Server processes usually gain by running longer periods, user processes work better with smaller time quanta. In a traditional operating system, there is usually only one rule when scheduling is considered, namely to spend as little time as possible in kernel mode. As

there only exists two modes, user and kernel, you have no possibility to affect the scheduling policies further.

Since the different parts of the kernel are no longer compiled together, one can think that there are high demands on well-specified protocols and interfaces for the various agents. This is true to some extent but not fully. You need a minimal set of commands that are standard for each service, but each agent can provide any number of extra functions as well. Agents can also reduce the original set of calls by simply ignoring non-supported messages or reply with an 'invalid command' message. The point of having at least a minimal set of commands is that the semantics of these method calls can be coded into calling agents. On top of these minimal protocols, you can agree upon a certain ontology in which you will talk to the specific agent.

To achieve the flexibility strived for, the calling protocols must define mechanisms to take over control for specific tasks. So can, for example, a file-encryption agent take control over the 'write' and 'read' commands from the file system agent, and in turn call the file system agent to store the file once it has been encrypted. Another way to solve this would be to tell the file system agent to ask the file-encryption agent before storing or loading. Which strategy to use is just a matter of design, and is not handled in this paper.

The solution with advanced managers in the kernel layer makes it possible to have more than one manager for processes and memory. In this way an application can provide a special process manager that takes care of a subset of the processes to schedule these in another way. The paging algorithms can in the same way differ depending on the application. More forms of communication can be added in a similar way by adding communication managers. Having several process or memory manager requires that these are implemented to be aware of this situation, communicating with each other to decide who should take care of what process.

It should be noted that I have taken the agent strategy to an extreme. The point I am trying to make is that you *can* perform all the tasks of an ordinary operating system with a multi-agent system. This does not mean that it is the best solution to do so. It might be better to have certain parts as a standard module-based operating system and employ agent-based techniques in other places of the operating systems.

Assignment of functionality

The core modules, acting as an interface to hardware, are inherently harder to exchange than the kernel agents outside of the core. This is not because they are so deeply needed by the system, but rather because they involve machine specific code and optimizations. Still, I have strived to keep them as small as possible, putting the more algorithmic behaviour into agents in kernel-space. This approach enables functionality to be replaced and added at runtime without disrupting the system. Behaviour can also be altered without extensive knowledge about the target platform.

The operating system can very well work without the notion of users, and processes should be secure from each other by default, unless they state otherwise by sharing memory or communicating to untrusted agents. Accordingly, I do not introduce the notion of users until at the service layer. Users should be seen as a convenience to achieve security but the lower layers function without them. Nevertheless it helps to have some user awareness like the owners of processes and files. The kernel agents map a user ID as ownership and ask the user manager agent in the service layer for information about the access rights when needed. If no user manager is present, the kernel agents fall back to some default behaviour. The reasoning considering process migration is similar. It is a desirable feature, but the operating system can manage very

well without it. Strictly speaking, the support given at the kernel level is with some exceptions that of a single-user, single-computer system and the service layer extends this to be a multi-user distributed system.

Naturally, this division can be discussed from the perspective of performance, but I believe that the cost for context switches and IPC communication is small enough to be negligible, considering the vast improvements gained in flexibility and structure. There is virtually an equal amount of data to be stored when performing an in-process function call as when doing a context switch. The data to be loaded for the new process is however larger, but still small enough not to make this a real problem. With modern processors it is a matter of mere microseconds to conduct a context switch. As for IPC, one can by using shared memory reduce this task to the same level as if you were putting something on the heap before a function call.

Service layer

Something completely new that I have introduced is the service layer. I have not found evidence of anything similar in the research papers and operating system descriptions that I have read. The reason for this layer is that there are things that are part of the operating system, but not needed for the actual kernel. These services should be allowed to run in kernel mode if they need to, and they should be able to communicate with the kernel in a leisurely fashion. In some cases the kernel agents are even aware of the existence of services in the service layer to for example authenticate requests from users with the user manager.

In traditional operating systems there is a very strict division. Either you are in the kernel, or you are outside it running in user mode and forced to rely on the functionality of the kernel. I believe that by adding a service layer one makes this border more flexible since you can have non-essential functionality running in kernel mode and the kernel agents can also rely on the presence of the service.

The chicken and the egg problem

Removing the file system from the bootloader causes an unwanted problem; how to load the file system manager from a non-existent file system. Before you can load the file system, you also need to load the I/O manager and the disk device driver. I have no obvious solution to this problem. To simply move the file system manager into the core inflicts the flexibility goals since this means that the I/O manager and device driver also would need to be put into the core.

In traditional operating systems this is not so much of a problem since there you know what file system you are using and can support at least read-operations in the bootloader, but with the flexible model used in ABOS you have no way of knowing how the files are stored and can consequently not know how to load them.

One could make a trade-off like the one with process and memory management, having a basic I/O manager and file system manager in the core and an advanced manager outside, but this would still, in my view, make the core system too large and too much a central component in the system.

Another solution would be to have a separate file system for the boot loader, loading the core system and the file systems manager using some default disk access device driver and then, when the system has started both file systems manager and I/O manager together with disk device drivers, discard these initial device drivers and loader systems. The real file system manager could then mount this initial boot file system to a suitable place in its directory tree, providing access to it for changes. This solution

may be the easiest to work with, but I believe that having a special file format for the boot disk is not desirable. It forces the file system manager to keep track of more than one type of file system, making it larger than what should be needed.

The best solution would be to have a bootloader that contains a dummy device driver and also at least a subset of the file systems manager so that files can be read and fed to the memory manager or wherever they are supposed to go. I believe that the solution with a separate file system is, unfortunately, the best way to solve the boot procedure, but this file system should be managed by a separate agent that works like any other file directory agent with the exception that it manages the entire boot disk directory tree.

4.8 Achieved goals

The flexibility goal has been achieved with the ABOS kernel described above. Replacing a kernel agent is simply a question of starting the new service parallel to the old one, and then gradually take over the functions from it, by querying about it's status. The final cut is done by moving the communications channels to the new agent. Adding functionality is even easier, since it merely involves starting up a new agent.

Having extensive support for communication and network communication in the kernel to rely on should increase productivity when developing distributed applications. If these services also define a flexible communication protocol like KQML [30] that can be used as an all-purpose communication language, it should at least be easier to program in a distributed fashion. Supporting process migration and including location independence in the communications protocol facilitates such programming even further.

Security should always be a goal when designing systems like this. The goals regarding security involves fault-tolerance and user privacy. Fault-tolerance and such are handled by the autonomous qualities of the various subsystems but is ultimately controlled by their respective implementation. Security for malignant users is achieved by adding security agents in the service layer. The communications manager can call an encryption agent in the service layer before sending messages over network, as can the file systems manager if file encryption should be enabled. Agents knowing that their services are of a delicate nature can require authentication from the service layer before executing requests.

Performance, which is also always expected from operating systems, is something that I have not taken into consideration when designing the system. A quick glance at the design shows that the number of context switches will increase significantly compared to an ordinary system, as will interprocess communication. The division of the operating system kernel into several layers increases the possibilities for smart scheduling and varying time quanta depending on in which layer the process can be found. Running kernel modules as separate processes means that parts of the operating system can be swapped out at times, decreasing memory demands. If the scheduling algorithms are similar to UNIX, processes that have been idle a long time will be swapped out and removed from the primary run queue [7], implying that the entire kernel need not be in the primary run queue.

5. Examples

The previous section presented the design of ABOS, an agent-based operating system, showing the core and kernel layout. To further test the design, I will in this chapter take a few tasks and examine further, making a more thorough design and discussion of these.

5.1 Introduction

In the previous section I made a rough design of ABOS. I outlined the different layers and gave examples of tasks that resides in the different layers. I showed that the functions provided by a traditional operating system can be handled by agents running as separate processes. I also introduced the notion of a service layer that can hide things like process distribution and enable process migration, among other things.

As mentioned earlier, there is not much one can do about process management and memory management since these are ultimately controlled by the hardware. What differs between operating systems are the actual scheduling algorithms and paging algorithms used. The same goes in all that is essential for the device drivers. They should act as a mere abstraction to the varying hardware devices, not differing much in ABOS from what is common practice.

Communication management can be done in many ways but I assume a simple model that for the programmer looks like streams or pipes on which data is sent using a protocol like KQML. Naming services and network routing is done by a separate agent, the advanced communication manager.

This brings us to the parts that *can* differ widely from current praxis. I have stated earlier that the area of file systems can be greatly improved upon compared to the *de facto* standard that is used today. I will examine this further by designing an agent-based file system.

Resource allocation, for example to print a document, is often requested by the user. Naturally the user have certain preferences regarding where he wants his document to be printed, the minimum quality acceptable, and a number of other details. In a traditional operating system, it is up to the user to decide exactly where he want to have his document printed, forcing him to know whether the printer is working, has paper, its printing resolution, and where it is located. This goes for most other network devices as well, even if the printer example is very significative. Users being as they are will probably find a favorite machine/device and always use this regardless of its current load, which will create an unbalanced resource usage. If you instead assign a potentially mobile agent with the resource request or job to be performed this agent can find the most suitable resource for performing the request, optionally moving to a host closer to the device for the more communication-intense bits. This is the second task that I will examine further and create a design for.

As I discussed in the section about operating system tasks, synchronization, and in particular time synchronization, is very hard to achieve. Since you have no way of measuring the time a packet is *en route* on the network, you have no exact way of synchronizing clocks. I will investigate whether mobile agents can perform this task better. As for other synchronizations like ordering of events there exists several working theories so I will not go deeper into these areas.

These three topics (file systems, resource allocations, and synchronization) will be examined in further detail below. A design is presented that relies on the agent operating system described in the previous section, and this design is evaluated.

5.2 Agent File system

Over the past 20 years, not much has happened with the file systems we use. The same conceptual notions of files and folders are still used. Files are dead objects, a dumping place for applications to ensure persistence. Directories were added to be able to group logically connected files and handle them as a single entity. In recent years an ability to encrypt files has been added [51]. This was done because users requested a security level not supported by the applications in use.

Networked file systems are commonly added as an afterthought, working on top of the traditional file system. An explicit command is needed to distribute a certain directory tree, causing troubles when one is working from another machine. This could be viewed as an advantage when it comes to security, but security is handled with access rights for the individual files, so this safety effort can be considered redundant.

Caching is in most systems rudimentary at best. There is always a problem to ensure that the file read is the most recent one, and that no cached versions exist somewhere in the network. One common way to solve this is to have write-through caches, which decreases performance but ensures that all data is directly stored to disk. If the network should go down for some reason you cannot save any files since the remote disk and the network file server is no longer accessible.

Active Documents

Fredriksson has recognized the fact that documents are passive objects [43]. He writes that you have to rely on external software like word processors and database systems for the documents to get any behaviour. The attempts at making documents more active, like compound documents [26] and OLE [26] only go so far. The autonomous behaviour one can achieve in compound documents is a patch solution, and cannot be modified to fit ones own needs. OLE is primarily concerned with visualization and linking of data between a set of documents. The documents are still inanimate and rely on the word processor to support the OLE or compound document framework. Fredriksson also outlines a solution to this application dependence, namely by embedding each document in an agent. This agent can have arbitrary behaviour, sending updates to concerned parties or simply update itself according to some rule.

I claim that Fredriksson's theories are just as valid on files at large, not only documents, and that by making each file an agent we can gain many benefits. Things like file dependencies, for example between source code files and their compiled equivalence, can be managed by the files themselves, thus removing the need for a specific 'make'-program. Caching over the network can also be handled in a smart way by actually moving the file to the computer where it is used most, storing an image on the server when the computer and network load permits, or when the server requests the file for backup.

Solution

The file system manager resides in kernel mode. It defines standard file operations like open, read, write, and close. In addition, it also defines interfaces for setting up association tables for newly created files, to make sure that files of a certain type gets an agent of corresponding type wrapped around the data. A word document should for example get a word-document agent by default. File types that do not have an agent associated with them are provided with a default agent from the file system manager. This default agent implements the traditional operations and provide basic protection mechanisms.

A request for reading or writing to a file is forwarded to the specific file, and this file may then decide whether to accept the request or not. Files can thus decide on their own who should be allowed to operate on it, and also which programs that should be allowed to access the file. The role of the file system manager is by this reduced to a simple interface to load the file agents from the disk device driver. A default caching mechanism can be added, but the files should have an option to manage their own caching strategy. This will enable databases to have their own caching mechanisms, whereas other files may decide that they are temporary enough never to ask to be written to disk.

A distributed file system is accessed through a certain mount point, which is a separate file agent. This mount point agent will act as a bridge to the remote system, showing the contents of the remote directory as if it existed on the local disk. All file operations come through the communications manager, so the file system manager need not know whether it is a request from another computer or a local request that it is processing. The actual directory agent that is accessed via the mount point agent decides whether to accept or to refuse access from a certain host or mount point. The causes for a refusal can be anything from not being on a trusted network to a temporary lockout to perform system management or backups.

Evaluation

The agent layout described above gives enormous flexibility, since each file may have an individual behaviour. You also reduce the need for specific programs traditionally needed to manage the files. Since the files carry their own behaviour they can easily move to a new host within the network or even a completely different host system on another network and still execute in their intended way.

The reason for the file system manager to implement the read and write operations is that you do not know whether the file system is agent-based or a standard file system. The manager should also have a possibility to load the file agent into memory if it is not already loaded. Furthermore, the communication manager needs to set up a communications channel to the object if this has been closed down.

Having each file handling their own security and behaviour, and also the decision rights on who should be allowed to do what, gives an extensible security model. The file agent will not allow the file to be transported to an insecure host, nor will it give data to an unauthorized client. The file agent can also choose to show different views of the file to different clients in accordance to their specific security status. The files can also decide for themselves whether to encrypt the data or not. In the overview of ABOS I stated that the file system manager could request this from an encryption agent in the service layer. Letting the files manage this by themselves removes the need for the file system agent to know specifics about every file. Different encryption algorithms can be used for different files, improving performance on files with low demands on encryption compared to files with high demands.

By letting each file agent manage its own caching mechanisms, file performance can to some extent be tuned for the individual systems. The main disadvantage regarding performance is that for each file request where the file agent is not already active, a new process will have to be created and the request forwarded via IPC. Removing some of the autonomicity of the files, reducing them to mere threads within the file system manager, can reduce this bottleneck. To achieve better fault tolerance, one can have a set of file-agent nurseries that each takes care of a number of agent threads. These nursery agents can quickly optimize their numbers with regards to the average

number of file requests so that there are no more nursery agents than are needed at any given time.

The mount point agent provides possibilities for redundant systems, letting the agent determine which file system to provide access to from a set of choices. If one server is down, the agent points to the next one. It can also dynamically determine the fastest of a set of servers, and let the request go to this one. If the mounted file system is not just a read-only file system but is updated as well, the servers should of course work as a cluster, cooperating to store the data in a fault-tolerant way and update servers that have been down with the changes made during their absence. The mount point agent can also merge more than one directory to a single mount point, which might be useful if you need to install something that is larger than the disks you have available. If you in the future let each file creation request in this directory tree use the full path, the mount point agent can dynamically put the files on the least used disk, or work in accordance to the file's own preferences on storage space. A database log file agent may wish to have a disk with lots of space to grow in, but the actual database storage space knows its exact size on creation and is not expected to grow uncontrollably which means that it can be placed on the disk with the tightest fit.

There is a data storage overhead involved for each file, since its state and perhaps binary executable should be stored together with the actual data. Doing a quick check on all my java bytecode files gives that a standard class-file is approximately 5 kB in size. Assuming that a file agent requires several classes, the overhead might be as large as 50 kB. It should be noted that these figures may not be significative at all, they are just based on estimations from my side. The actual *state* of a file is however just a matter of a few bytes, so one can avoid this overhead by storing the binary code for the agent as a separate file, letting the file system manager launch this binary when a file is requested. This would mean that an agent is needed that can ensure that the required binaries are present at the future host if a file wishes to migrate. This is best done by the process migration agent. After all, a migrating file agent is no different than any other migrating agent, and the mechanisms for migrating are still the same. References needs to be updated and forwarders left so that it can be found in the new location.

Naturally there is a risk that the users get overloaded with extra parameters to keep in mind when creating a new file. It is not my intention to create more overhead. A file agent is automatically selected and created with respect to the file type similar to how files are displayed with a specific icon in Windows NT. The users can write file agents of their own, but it is more the task of the software suppliers to provide agents for the different file types.

5.3 Resource Allocation

Basically, one can reduce the problem of resource allocation to 'finding the right resource for the right price'. This may involve a number of parameters like the user's preferences, or a company policy of which resources should preferably be used because they are cheaper, or a wish to use the device closest to the target. The resources can be everything from a printer to a fax-machine or simply a certain hard disk or sound card. This multitude makes it hard to create a general resource allocation policy. In the case of a fax machine the one closest to the recipient should be used, and in the case of a printer the one closest to the sender should be used. In other cases the choice is easier, like in the case of the sound card where you only have the choice to use a local card or none at all.

A common way to solve resource allocation is simply to have different programs for all different resources, forcing the user to keep track of the various commands for the

resources. The user thus have to know that an audio device is found in '/dev/audio', but the printer should only be accessed via the 'lp' command (Examples are taken from UNIX, System V). The user also needs to know where the device is situated, since this affects the cost of the operation. Clearly, this is not a desirable situation since it requires both the user and facilitating programs to keep track of the different allocation schemes, commands and locations of devices. It would be preferable if the system provided a common interface language for all devices, and a common interface for allocating the device. Instead of having this plethora of different programs and access ways, it would be convenient to have a single point to which all resource allocation requests goes, a resource manager agent. This agent can in turn create any number of allocation agents that can negotiate the specific allocation by communicating with the device drivers.

A resource allocation can be preceded by much communication, trying to decide which device is the most suitable to allocate, how to allocate it, and for how long it should be occupied. If the device or its device driver is situated on a remote machine, perhaps with a slow connection, this communication may take a while. In such cases the allocation agent can benefit from transferring itself to some place closer to the device driver so it does not have to communicate over these bottleneck networks.

Solution

In the service layer we have a resource manager, from which you request what device to use. The resource manager agent creates an allocation agent for the desired type of device. This device allocation agent knows the specifics of the resource it is supposed to allocate, and can thus initiate a dialogue with the user or calling software requesting values for the parameters needed to set up the device. By hooking a policy agent to the resource manager the allocation agent can also acquire guidelines of how the company views the allocation. Another policy agent provides the allocation agent with the user's preferences.

In a distributed environment where the devices and resources may be spread over the network the resource allocation agent gathers the information needed from policy agents and interaction with the calling party and sends a lackey with the allocation criteria to the driver's host with the power to negotiate a deal. After a deal is struck the lackey returns to the allocation agent with this deal so that the allocation agent and the calling party can proceed.

As with file agents the idea of a nursery can be utilized, supposing that there are many allocation requests. The allocation agents will in such cases not exit when the allocation is done, but instead stay on to take up a new job. The allocation agents can also utilize encryption agents and other security policies according to the security model for the device in question. The device driver agent will consequently have certain access levels that it checks each request against. It can also monitor the number and size of requests from a specific user and perhaps deny access when the user has exceeded his quota. It can also contact a bailiff agent to make a transfer of capital for the service performed.

The device allocation agent can also monitor the job as it is being processed and return any error messages to the user. If you for example print a document on the company printer from your home you will want to know whether you can expect it to be lying at the printer when you arrive the following morning. The agent can monitor the progress of the print job, notifying you if a paper jams. Furthermore, the agent could find someone that is present in the building and ask of him to fix the printer before giving up and notifying you of the failure. Sticking to the printer example, if you have a very large

document to print and the company has an array of printers to take care of print jobs during peak hours, the agent can split the job and print it in parallel on all or some of the available printers if you launch the print job during the idle hours.

All of the above suggestions are hypothetical ideas of what a device allocation agent can do. This is naturally depending on the implementation of the agent. I am just hinting at ideas of usage that distinguishes themselves from an ordinary inflexible and static allocation scheme.

Evaluation

By embedding each resource request in an agent you achieve device independence since you leave it to the agent to find out the specific parameters of a certain device by interacting with both the device and the user. A parallel to UNIX would be if you had a single command, ‘allocate’, to allocate a device and had to answer a set of questions to setup everything.

Usage of agents is highly suitable for batch jobs since you can provide the allocation agent with the tasks to perform as well and not only under which parameters it should work. For interactive resources, using agents to allocate the resource may result in unnecessary overhead. On the other hand many tasks can be batched even if they normally are not. In fact, anything that does not require input *and* output in one operation, where the output is dependent on the input, can be batched. Even tasks with both input and output can be batched, but this requires more intelligence from the agent performing the job. In the case of an interactive device, the resource allocation agent can also provide an advantage by letting the agent watch the device until it becomes available, signalling to the appropriate program that the resource is allocated and available for interaction. The user or program will hence not need to manually retry if the device is allocated at the time of the initial request.

The mobility of the device allocation agent ensures that you can allocate any resource from anywhere in the network and the device need not even know that the allocation is performed by someone from another host. For security reasons it might be advisable for the device agent to do some check anyway, to see whether the request comes from a trusted host, allocation agent, and user. This could also be done if the agent used the advanced communications manager and the network to communicate the request, but this way credentials could easier be faked and could also result in much network traffic over a possibly slow network connection.

5.4 Synchronization

In many situations, for example in database applications, the order in which messages arrive is important. In a distributed environment it is also important in which order they are sent since part of the calculations may have been done on another machine and messages are sent to integrate the results. The traditional way to solve this is to use Lamport’s algorithm [7] in which messages are timestamped according to a logical clock, and where this clock is updated according to the messages received. This is a working solution and I see no reason to modify this.

There is also the question of time synchronization, which is a somewhat different problem than event synchronization. Time synchronization is in many cases of grave importance. A traditional ‘make’ program, for example, is extremely dependent on that the clock in the file server is synchronized with the clock in the workstation. If the clocks are not synchronized the ‘make’ program will not work, recompiling more than is needed or perhaps nothing at all even if there are updated files.

As described in the chapter about operating system tasks, the problem with time synchronization is that there is no guaranteed time in which a network packet can be delivered. This time is dependent on many factors like network load and the load of the sending and receiving machines. To remove at least the network factor, global time should preferably be broadcasted under the protecting umbrella of a jam-signal. However, this would slow down the rest of the network, if it is at all possible. Using the jam-signal for anything other than a network collision would also be viewed as highly unorthodox, and it would be better to introduce an entirely new signal to use for broadcasting the time.

Solution

Even if the network was quiet during the time signal, so that a certain delivery time could be guaranteed, there is still the possibility that a computer is so loaded with work that it cannot process the time synchronization at once. There are two solutions to this. The most obvious solution is to let the network interface manage time synchronization messages, noting them with the local time. The network interface commonly have a very high priority since it works with interrupts, so this local timestamp can be considered accurate enough. Since you now have both the desired time and the local time in the message it is an easy task to calculate the actual time at a later stage when the machine finds time to do so. This solution relies on that the network device have a high priority, and that it works on an interrupt-driven basis. Supposing that it is not interrupt-based or that we do not know whether it is, we cannot assume that the local timestamp will be the correct one. In such cases we need another strategy for synchronizing time. Fortunately time synchronizations need typically not be performed very often and can furthermore be scheduled to be executed during idle times like the night shift. Using this assumption we can halt the entire network for the duration of the time synchronization according to the solution below.

Once again I propose the use of a lackey agent, sent out by the machine responsible for the time synchronization. This lackey can be sent at any time and it is the agent's task to agree with the machines which time would be suitable for a halt. This time may be significantly different between different machines if their respective clocks are very much off the actual time, but should still be relatively close together in their distribution around the real world time that the agent aims at for the synchronization event. Once the time of synchronization has been agreed upon it is the job of the lackey to make sure that all processes are suspended from the agreed time and forward, being resumed either by a time-out or by a completion of the time synchronization. As soon as the lackey has achieved total idleness, it sends a message to the time server stating this. When it is safe for the time server to assume that all machines have ceased all other activity it broadcasts the time on a by now silent network. Since nothing else is running on the receiving machines they can process the time message at once, adjusting their internal clock accordingly. The machines can then resume their previous activities and the time lackey terminates. On such a silent network with idle machines, it is also very easy to find out the propagation time to each machine by letting the timeserver send a unicast message to each of the machines, which will in turn reply immediately. This can be done once when a new machine is added or on command from the system administrator and need not be done every time a synchronization is conducted. Knowing this propagation time ensures even better accuracy in the time adjustment if you take the delay into calculation when adjusting the clock.

Evaluation

The main and most obvious disadvantage with this solution is that the network and the machines needs to be idle for an arbitrary period of time. This time may vary on differ-

ent machines, depending on the state of their internal clock. A machine that falls asleep early has to wait for the slowest machine to achieve stillness. The algorithm thus have a tendency to penalize some machines more than others. On the other hand, the machines with the least load will most likely be the ones succeeding to fall asleep first which will minimize the effects of this penalty.

There is, of course, also the question of whether one *can* suspend all activities on a network. In effect, the network needs to be stand-alone and not connected to any other network like the internet to eliminate the risk that an external message enters the network looking for a web-server or something similar. The router can of course play along and block all external messages for the duration of the time synchronization, but this would require that the router is intelligent enough to support such a policy.

What, then, do I gain by using this model? I gain the accuracy of a VVW-signal, assuming that the network can be silenced. I also get a basis for a distributed time management, since there is nothing saying that there needs to be a central time server. Any machine can initiate a time synchronization, so I can have a cluster of machines that decides which of them should get the time from the nearest atomic clock and distribute it to the rest of the network. I still have the benefits of a 'push'-solution, meaning that the server broadcasts the time instead of having each client asking the server. Furthermore I get an informal guarantee that all machines have received and set their clocks according to some timestamp, since they should have been silenced by the lackey and the lackey should also have sent a message to the time synchronizer saying that it is ready to receive. As soon as the synchronization is complete, the lackeys can start sending acknowledgments to the server as well, thus giving a more formal proof of that the synchronization succeeded.

There is, of course, no real need for the network to be silent or the machines to be idle when performing the time synchronization unless you want an extremely accurate time. If a moderate time synchronization is enough it might be sufficient to just silence the network, or perhaps to suspend all processes. It might even be enough to do nothing at all and trust that the network is fairly stable and the network agent has high enough priority for the synchronization to achieve the accuracy you wish.

5.5 Summary

The examples above certainly do not constitute an entire operating system. They are simply tasks that I have found more interesting and more suitable for applying an agent based approach to than others. Process management, memory management, and I/O management would make the operating system complete, but these parts are not that different from other operating systems so there is no point in explaining and designing these further. The presentation of these in the previous chapter should give enough information on how these differ from traditional approaches.

The area in which you gain the most benefits is, in my view, with the file agents. These provide a vast advantage in flexibility and understandability compared to a standard file system. To let the files be able to modify their own state gives a very clear conceptual notion of how the system works. It is no longer a program that modifies the file, it is the file that modify its own state. In many cases the need for an external software can be removed altogether by letting the file take care of all of the work itself.

In the other two examples, resource allocation and time synchronization, the benefits are not as clearly visible. Resource allocation can be done with the same possibility to introduce policies even if it is not done by agents. In this case the main contribution with my solution is that I identify these programs as being agents, I suggest that the

allocation agent can be mobile, and I also suggest that the agent can supervise the job in a more involved way than current allocation strategies. As for time synchronization, this is today done with daemons that in effect are agents which is why I developed a more complex solution using mobile agents. The time synchronization example is a good example in many ways, showing a non-polled version of time synchronization and also showing how to eliminate many sources of errors, but it is also an example of making a relatively simple task too complicated.

One problem that all of the above solutions may suffer from is lack of performance. Having so many autonomous agents causes a large number of extra threads or processes, which in turn gives more context switches than in an ordinary operating system. This is naturally a major Achilles' heel. Even if computers are becoming faster and faster the operating system should still not be time-consuming enough to be noticed. In the case of file system agents, the gains in flexibility can in many ways defend the many context switches but with other tasks, like synchronization, it is probably overkill to use agents.

Still, the three examples presented above are areas where one can gain at least some benefits from agents. Another example springs to mind where agents would not be very suitable, namely network packages. Embedding every network package with an agent would give each message the power to find the fastest or the cheapest way to its destination, and the sender would not have to worry about the package encountering different policies somewhere else since the policies are embedded in the network package agent. Whereas this solution gives much power and control to the packages, the overhead cost in transferring the agent binary and start up on each routing host on the way is simply too expensive. This is just one example where agents are absolutely not recommendable. As always there is a trade-off between 'cool techniques' and practical solutions. It is just a question of using agents where one can benefit from them, but one should not use agents in places where they clearly are unsuitable.

6. Evaluation

In the previous chapters I have described agents and operating system tasks. I have also designed an operating system consisting of agents and furthermore given examples of more detailed tasks in this agent-based operating system. Remaining is to evaluate the operating system designed against the criteria set out for a modern operating system in Chapter 3. This chapter will evaluate the operating system designed in Chapter 4 and Chapter 5 against these criteria. I will verify that the tasks required are fulfilled and hold a general discussion about the agent operating system solution.

6.1 Introduction

As I described in Chapter 3, an operating system should minimally manage processes, memory, I/O, and file systems. I also claimed that you need communication support and security primitives in the operating system to make it complete and able to meet the demands on a modern operating system. Going through these different areas I will investigate whether the agent operating system described in the previous chapters meets these requirements.

6.2 Process management

Naturally ABOS manages processes, otherwise it would not be an operating system. The core system takes care of the creation of processes like creating process control blocks and such. It also manages the various run-queues. By putting all other logic into stand-alone processes that run in the kernel layer the agent operating system ensures that scheduling algorithms can be added and customized while the operating system is running. Since you can have more than one advanced process manager, you can also have more than one way to create a process. The easiest is to ask a process creation of your own manager, but you can also ask another manager for the creation. A process can also move to another process manager if it needs to change scheduling algorithm while running. In this way a process is not limited to a certain scheduling policy during its entire lifetime. As the application proceeds into different stages it can change policy by transferring to another process manager.

Having the process managers as separate processes creates some interesting dilemmas. The most obvious is how the process managers should be selected to run without having any scheduling policies for themselves. The answer is that at least one process manager is always present in the run-queue. This process manager should act as a nanny to the other process managers and manage their process control blocks when they are unable to do so themselves. Another dilemma is *when* a process manager should be run. According to the standard scheduling model it should wait for its turn like any other process, but for performance reasons it might be better if the process manager is run as soon as someone sends requests to it. This question is more related to communication management, and I will discuss the problem further in this section.

The division between what the core process manager and what the advanced managers should do is not clear in the current design. On one hand I claim that all scheduling policies should be handled by the advanced managers, and on the other hand I argue that the core process manager should choose new processes to run. According to the design described, the core process manager does not even use preemption, but instead waits for the processes to yield. This results in a situation where one process can hold the CPU forever, never yielding to let the advanced process managers run. To solve this, we need to add preemption to the core process manager. Any process should be able to block for a specified time, and when this time has expired the core process manager should be able to preempt the running process to run the blocked thread. To

avoid abuse, we can further limit this by only letting kernel agents be guaranteed execution within real-time boundaries.

So far, the process management does not supply support for multiple threads within a process. Threads share the same memory space, but should otherwise be treated as separate processes. Many approaches to developing agents, like that suggested when using April [31], use a method where every thread is a stand-alone process. I believe that an agent should not consist of more than one thread, or it will be too complex. If an agent needs more than one thread, it can equally well fork off a child-agent to solve the extra task. However, it is an easy task to write a new process manager that supports multi-threading, should need arise. One can also solve many tasks with the use of shared memory between full processes.

As with any preemptive multi-tasking operating system, ABOS has difficulties to meet any hard real-time boundaries. A process does not know when or for how long it will be run, so you can not guarantee any execution times. However, by communicating with the process manager an agent can set up conditions that it for example “needs to have exclusive access to a CPU for xx clock cycles starting yy cycles from now”. The process manager can then make sure that all other activity is suspended or rescheduled to another CPU before this time. It can furthermore influence the timer not to generate any interrupts to this CPU during the duration of the exclusive access. This means that a process need not have a fixed priority. The agent can negotiate with the process manager to set an adequate priority at any given time. This is very different compared to traditional operating systems where the user commonly has to set the priority manually, and is in some cases unable to change the priority once the program has been launched.

There are no troubles involved in supporting multiple CPUs. The advanced memory managers might need to adapt their caching policies, but apart from this no extra modifications are needed. The core will need to be aware of the other CPUs and be able to move a process to a new processor, which implies that also the core process manager needs to be replaced with a multi-processor version. The advanced process managers in the kernel can be aware of the extra CPUs, but it is not necessary. The core process manager can keep the single run-queue and act as if it only had one processor even if the system actually has more. Multiple processors might however affect scheduling decisions, so the advanced managers will probably benefit from being aware of the extra processors.

6.3 Memory Management

As with process management, at least some control is needed in the core to be able to run the rest of the system. The core module only manages the assignment of virtual memory pages to physical memory. The core can decide to move pages in memory and copy or move the contents of a page, but eviction mechanisms and communication with the disk device for swapping and paging is handled by the advanced memory manager in the kernel layer. Again, as with process management, you can have more than one advanced memory manager, each having different functions. One manager can handle local memory and another shared memory while a third participates in the management of a distributed, network-wide pool of memory.

Bereaving the core memory manager of the page eviction algorithms causes overhead when a page fault occurs. In such cases the core memory manager needs to send a message to a memory manager informing of the problem, after which the advanced memory manager should communicate to the disk device manager and acquire the requested page and then send this back to the core memory manager. Optionally

another page needs to be evicted, meaning communication to the core manager to first get available pages together with all the extra info needed about the pages and then more communication to receive the selected page and send this to the disk device driver. Supposing that it is one of the programs in this chain that had the original page-fault, you will enter an infinite loop trying to load the memory page, which will cause a page fault, and so on. This is a problem that does not occur in a traditional operating system since the entire kernel is always loaded into memory. Fortunately, there is a solution to this. Since it is the job of the memory manager to keep track of which memory pages that should be evicted to the swap disk, it can also choose not to evict pages that are part of this essential chain. This forces the memory manager to be aware of what kernel agents are involved when loading and storing a page to disk.

It is not at all clear how shared memory should work. A process who wishes to share a memory area should request this from the advanced memory manager. The manager finds the appropriate pages and marks them as shared. When another process wishes to share the pages he requests this from the advanced memory manager. As a reply he gets the size of the shared area, after which he sends another request to the manager with a memory address to map the pages onto. Interesting is that a shared memory area can have a name of arbitrary complexity. I prefer a text string describing the purpose of the memory, but a unique number is also possible.

The fact that the memory manager is an agent just as the processes are gives opportunities to communicate and negotiate with the memory manager. By knowing what an application is planning to do, the memory manager can customize the page swapping algorithms to best suit the task at hand and thus increase performance. This is of course assuming that the application informs the memory manager of its intentions and negotiate to get a suitable paging algorithm. An application should be able to store a certain paging scheme under a certain name, so that it only needs to send a message to the memory manager stating that “in the next xx clock cycles I intend to do the operation nn” when it wishes to use a certain algorithm. The memory manager can then adjust the paging algorithms accordingly, trying to join the demands of this algorithm with all the paging schemes that other applications are using.

6.4 I/O Management

I/O management is the question of making devices visible for communication in a uniform way. The different applications should not need to worry about the specifics of a certain device, but rather concentrate on what they wish to do with the device. ABOS is similar to ordinary operating systems in that each device is handled by a certain module, but unlike traditional approaches all communication with the device is done through the same communication channels as if they were any other process. Whereas this approach enables a transparent allocation and utilization policy, it decreases performance. Many applications like games require direct and immediate response from devices, for example the audio device, and if each new sound effect should be sent and treated as a standard IPC message, this real-time aspect might not be achieved. On the other hand, I have argued that IPC performance is improving, and that the use of IPC instead of in-process calls is no longer any restriction to performance. The loss of performance must, accordingly, be caused by something else. If you use an in-process call, the execution is performed immediately after the call to the responsible method, but with an IPC call, there is no guarantee that the receiving process is run immediately after the call. Indeed, if the call was sent asynchronously, it may not even be desired to break the running process to start executing the receiving process. As with process management, this problem will be discussed further in the section about communication support.

Due to limitations in the hardware, the computer generally needs to be shut down when installing new hardware devices. This makes the flexibility and extensibility of ABOS somewhat malplaced. A feature perhaps more desired is that of dynamically being able to replace device drivers with new or more specific versions. Supposing that a system upon start-up detects a new device, it can start up a generic driver to manage the device which can later be replaced by one from the hardware vendor without having to restart the system again. The fact that the device drivers are agents enables interesting solutions to distribute work load as well. If a task involves much pre-processing before the interfacing to the actual device, the device driver agent can spawn peers to help with this. The newly created agents will then communicate with each other to gain exclusive access to the device once the processing has been done.

One of the main tasks of the operating system is to present the devices to the user in a uniform and intuitive way. ABOS does not specify any particular way to categorize the devices. It is the task of the I/O manager to keep track of the available devices, and to start up appropriate device drivers when new hardware is detected. Once a device driver is started, it will bind itself to a name in the communications manager. All access to the device will henceforward be made through this communication channel. In this way, a device is presented as any other agent or resource, and communication to it is done in the same intuitive way. The I/O manager can furthermore act as a skill server, tagging each device with its capabilities and usage statistics. To further enable users to find and communicate with the devices, a special directory agent can be added to the file system that presents the devices much like the ‘/dev’ directory in UNIX. If the file system does not support agents a special file system manager can be installed that presents the device drivers as block or character devices, again in the fashion of UNIX.

6.5 File system Management

In the basic kernel design nothing is said about how the actual file system should be managed. All that is said is that you have one or a number of file system managers that each manage a certain type of file system. So does for example one manager handle UNIX file systems while another handles NTFS file systems. This division of file systems into different agents enables an interesting feature, namely that one partition on a hard disk can contain more than one file system. Suppose that a system has been running FAT and now wishes to change to NTFS, no files needs to be converted; you simply let both file systems coexist on the same disk. New files that are created will get the NTFS structure, and old files will be read from FAT and perhaps converted as well. In this example all files will eventually have been converted to NTFS, but the agents can be set up to coexist so that no file system is favored.

Being able to extend and replace these low-level interfaces as the file system managers are gives possibilities to extend the set of operations allowed on a file system. So can for example a FAT file system be extended with the ‘defrag’ operation that would move around blocks on the hard disk to enhance access times. All file systems can be extended with a backup operation, and in some file systems one can force a garbage collection. In traditional operating systems, distribution of a file system is done by a certain part of the operating system, and not by the separate directories and files as in the agent file system. This means that to, in ABOS, add file system access to a host running a traditional operating system you would need to write either a separate SMB mount point agent or a separate SMB file system manager.

The agent file system described in Chapter 5 brings enormous flexibility to add and modify the default behaviour of the file system. As stated, many applications can be removed altogether by replacing them with a special file agent. The common problem

with applications polling a certain directory for new files to process can be avoided by writing a new directory agent that takes care of the file as soon as it is added. Such an opening can work as an intuitive access point for other software that do not necessarily know the interfaces of the program responsible for the directory agent.

An agent-based file system gives better robustness than ordinary distributed file systems, because files can be transferred to another host if need arises. If the server is shut down for maintenance, the open documents can move themselves to the host where they are edited and wait for the server to come on-line again. This means that the user will not notice that the server is gone. The intelligent mount-point agent that switches the mount to another file system on another host also guarantees that a server shutdown and restart can occur virtually unnoticed by the users.

Every type of file does not benefit from the same caching techniques or even storage techniques. The fact that every file in the agent file system is an autonomous agent enables every file to be cached and stored in a way that suits the file best. A database file must not be cached by the operating system, whereas a temporary file should maybe not be stored at all. These are strategies that can be implemented with the agent file system. As for storage, the database file may wish to be stored sequentially for fast searches and a log-file may wish to be stored close to the centre of the hard disk to minimize disk arm movement when storing the data. These preferences will be negotiated with the file system managers to achieve an optimal situation.

The agent file system also enables a truly distributed file system by letting all hard disks on the entire network participate in the global file system. This would mean that all the disk space that today is left unused after installing the local operating system can be utilized. It would, however, also mean that the global file system is dependent on a multitude of disks to be fully operable all the time. It would only require someone to trip over a wire to bring the entire network to a standstill. Accordingly, I do not argue for the completely distributed solution, instead I recommend the use of one or more central disks and file servers.

The problem that every file will grow to include the code for the agent as well has been discussed previously. This is of course not good, but measures can be taken against this by storing the agent code separately and merely store the state together with the file. Having two or more physical files per logical file will cause overhead and hence reduce performance. In fact, having to let every file request go through the file agent to do some processing will also inflict on the performance. Fortunately, hard disks are magnitudes slower than processors, so some extra milliseconds will probably not be noticeable. Greater possibilities for smart caching and customized behaviors of the files will hopefully also reduce the performance penalty, provided that the possibilities are used.

Today, when many applications are object-oriented, it is sometimes convenient to be able to store each object as a separate file. A particular example is the EPIDEMIC compiler [53], where each node in the parse tree is represented as a separate object. When you change the source code, the parse tree should update accordingly. For large files it would be convenient if parts of the parse tree were dumped to disk to save memory. With the agent file system this can be done. The compiler objects would still be notified when there is a change even if they are currently lying dormant on the hard disk, and this functionality need not even be implemented into the compiler.

6.6 Communication support

Because of the extremely modularized structure of ABOS, the demands on communication support are very high. First of all, the agents should be able to communicate locally as smoothly as if they were parts of the same process. An IPC call should preferably not be much slower than a traditional function call to make the ideas of an agent-based operating system acceptable. The solution described is message based as compared to the stream-based solutions that is common in traditional operating systems. The communication that goes through the core communication manager is however stream-based, but is not fitted for lengthy transmissions like those of a UNIX socket or pipe. The communication channels are furthermore one-way pipes, forcing replies to be sent as separate messages craving separate communication channels. This solution makes it possible and even easy to use asynchronous message transfers.

I have already mentioned the scheduling problem that arises from communication. In many cases like when trying to communicate with a process manager it is not acceptable to wait for the recipient to run according to the preset scheduling algorithm. The recipient should preferably be run immediately after the caller has sent the message or when it has finished its time quantum. This can be done by letting the core communications manager talk to the core process manager and re-schedule the recipient to some high-priority message-receive-queue, so that it is run immediately after the sending process' time quantum has run out. If the sender wants a synchronous message transfer, it will most likely wait for a reply immediately after it has send the original message, so this policy would guarantee response times not very different from a standard function call. The other agents that are not part of this particular exchange of messages will however suffer from such a strategy. If two agents engage in a lengthy discussion no other process will get a chance to run; they will starve.

To avoid such problems, the process managers will mark each process with a flag in its process control block stating whether it is a fast-response, custom-response, or a standard-response agent. A fast-response agent will reply immediately, and is re-scheduled entirely by the core. A standard-response agent will not be re-scheduled in any way, and a custom-response agent will cause a trigger from the core process manager to the advanced process manager which can then place the agent into a queue of arbitrary choice. Kernel agents are always fast-response agents, assuming that kernel agents are more considerate than other agents and are not likely to engage themselves in any lasting discussions.

Without having stated exactly how the communication should work, down to the level of how the bytes are transferred to the recipient, it is very hard to say anything about the performance. I believe that one can make primitives that fulfills the goal of not costing more than a function call. Supposing this, it is hard to find anything negative about the communication strategies. The advanced communication manager will take care of all network messages, as well as being able to set up more than one channel to a given agent. These extra channels are provided to give logical names to services, but does in effect not provide more functionality than a single mailbox. Being able to distribute names over the network helps in creating distributed applications since the clients need not explicitly know the location of a service as long as it is on the same network. If it is on another network, the domain server can take care of requests and route them to the right host, whereas the clients only need to know what network the service is available on.

To find a service, it is necessary to know the name of the communication port, which makes it necessary to be able to query the advanced communications manager about what names are available together with some description of the skills of the agent

responsible for a certain channel. This could also be handled by a separate skill server agent, which will be referred to by the advanced communication manager when queried about the tasks of some agent.

6.7 Synchronization

There are two different categories of synchronization in an operating system. The first and most important is that of synchronizing messages so that they arrive or appear to arrive in the same order as they were sent. If a distributed system should appear as if it were a single system it is also important that all participating hosts share the same time.

Whereas message synchronization and ordering is handled by the communications manager in the kernel layer and by library functions within each process, time synchronization is not part of the kernel functions. This is also visible in ABOS, where time synchronization is done by agents in the service layer. The synchronization is initiated and performed on the command of a single machine so one can guarantee that all machines actually gets synchronized. If one follows the algorithm fully, silencing both the network and the machines before the synchronization, one is also guaranteed a highly accurate synchronization compared to the traditional approaches.

I have already argued against the solution in Chapter 5, claiming that it is unnecessarily complex and that the need for the participating machines to be idle during the length of the synchronization procedure makes it waste valuable computation time. One can also question the necessity for a mobile agent. The whole operation can equally well be performed using simple messages. There are at least some advantages in having the agent mobile, though; you do not need to start up any special software on every client when you boot it up and you do not need a certain agent on each client spending most of its time in an idle state, waiting for the synchronization message. The mobile agent will instead transfer itself to each host and begin to execute when it is time for the synchronization, and can disappear when the synchronization is done.

Unlike the traditional ostrich algorithm, ABOS can implement deadlock detection and recovery. An agent can when it requests a resource learn who is holding the resource. Communicating with this other agent, you can find out which other resources it is holding and whether it is blocked waiting for another resource, and in such cases which resource. By introducing the requesting and the holding agent to each other, the holding agent can also store information as to what resources it indirectly blocks by holding a certain resource. It can then decide to give up one of the resources, abort the operation it is working on, or ask the requesting agent to give up one of its resources that is directly or indirectly needed. I am thus attacking the circular wait condition by way of the no preemption and the hold-and-wait conditions, assuming that it is safe to hold a resource as long as no other process needs it, and that I am not indirectly waiting for myself to release a resource. The detection algorithms can be implemented both in the agents themselves, or in the resource allocation agent. Where to put the intelligence is just a matter of taste. However, to make sure that the problem is not ignored by application developers, it may be best to implement it in the resource allocation agent.

6.8 Security

Security is not only the issue of protection against malignant users, it is also to protect against and recover from faults in the operating system and system crashes. It is often possible to distribute and replicate software over a set of machines to protect against such system crashes. The trouble in distributed systems is commonly that instead of

having one single source of error, you end up depending on several machines to be operable. The entire structure of ABOS is done to enable distribution. There are very few central components that can break down and cause delays for the users. Some agents, chiefly in the service layer, can be shared among a set of computers thus making them centralized but they can also be implemented on each host, communicating in a distributed manner. For the unavoidable central components like file servers there are schemes allowing redundant servers and the clients change to backup servers transparently for the user. The caching mechanism supported by the agent file systems that files actually move to the host where they are used the most can cause loss of data if the hard disk where the file currently resides should break down, but this loss can be minimized by sending images of the file to the server at regular intervals.

I have earlier argued that mobility is often unnecessary, and that most tasks can be handled by standard message-based communication over the network. To protect against unauthorized entry, one can think that mobility is even more undesirable. However, actually sending the code to execute on the receiving side brings some advantages to the security situation. If you, as an agent, get a message it may be hard to determine that this message originates from a trusted agent that is not up to some mischief. If the code is transferred to your host you can let it pass through a filter much like a virus check before starting to communicate with it. If the host from which the agent originates is on the local network this search can be done on the sending side and eliminate the need to transfer the agent code, but if the agent comes from another network you have no way to trust it without actually checking the code.

Even with such scans, ABOS is in its current state very open for virus attacks because of the flexibility that I have prized so much. Malevolent users can write their own kernel agents and install them, replacing the real ones. Obviously some sort of security check needs to be made to ensure that a certain user has the access rights necessary to install something into the kernel. This security check can thus allow some users to install kernel agents, whereas others may only run them. Other users again can be allowed to install and/or run agents in the service layer, and some users may only be allowed to run programs in the user layer. This gives a security model where practically everything of interest can be configured on a per-user-basis. Because files are active entities, and because anyone should be allowed to write their own file agent, the agent file system will be a major breeding ground for virus attacks. I fear that people will use the opportunity for flexibility just as they have used the macro-feature in programs like Microsoft Word. What is even worse is that the files can migrate by themselves to new networks and hosts, and the user has no opportunity to select whether a file should be allowed to run or not.

Because of the possibility to add more managers in the kernel that communicates with the default ones, you achieve high fault-tolerance. If one process or memory manager blocks or bugs on some operation, only the processes involved with this manager will notice this, and all other applications will continue as usual.

6.9 Performance

Performance was not one of the goals when I designed ABOS. Nevertheless I have tried to keep it in mind when designing, trying to avoid the most ineffective solutions. In many cases, like the agent file system, having to execute code in situations that today are done as a simple disk-to-memory operation will of course slow down the overall performance of the system. It is my hope that by executing these extra bits of code some smart behaviour can be implemented to negate this effect by speeding up the perceived performance for the user.

Scheduling decisions will also cause some troubles with performance if one is not careful. Operations that in traditional operating systems are performed as a function call should be done in a similar way in ABOS by ensuring that the receiving agent is executed immediately after the calling agent. However, care must be taken to avoid starvation of the other processes when a lengthy discussion is being held. One way to avoid this is to let the receiving agent share the same time quanta as the calling process. In this way no extra time is allotted for communicating agents.

The amount of context switches can be a liability compared to a traditional operating system. For every context switch CPU registers needs to be saved, memory references switched, open files noted and stored, and so on. This takes some time, and doing this often can result in reduced performance. A short survey yields that a common situation in a UNIX system is to have 40 to 80 processes in total, but in ABOS there can be this many processes even before users have logged in and started any applications, which should prove that there is an increase of context switches in an agent-based operating system.

By installing a network process manager you can distribute work over the hosts in a network. There is an overhead in finding a suitable host and perhaps transfer the code, but for processes running a longer period of time this overhead can be worthwhile. This is valid for agents without any user interface, but if the agent requires input from the user, the demands on the network bandwidth increase. This can be solved by just distributing computation-intensive tasks as separate agents to other machines. All tasks that involves user interaction will remain on the local host. This means that the agents themselves must decide whether they are suitable for load balancing, which is in accordance to the rest of ABOS that encourages such local decisions.

It is possible to achieve real-time execution by negotiating with the process manager. Having real-time execution means that ABOS can be used in embedded equipments, and that hard real-time deadlines can be met while at the same time have a multi-tasking environment. On a single-processor machine, the system is of course not multi-tasking while a process is running in real-time mode. This implies that real-time behaviour should only be allowed for shorter periods of time to let other processes run as well. When an agent is running in real-time mode all agents that it calls should be promoted to real-time mode as well, provided that the call is synchronous. If an asynchronous call is made this can cause some problems because the called agent has no way of running on a single-processor system until the real-time process has finished running.

The fact that the scheduling priorities can be customized to almost any extent and even support real-time execution implies that the performance can be very much tuned for the specific applications. By allowing the applications to themselves negotiate for a change in priority also means that better performance can be achieved when needed. To avoid abuse, there should be a cost involved in running a higher priority, so that no process runs indefinitely at an unnecessarily high priority. The system is scalable to any number of processors, provided that the core process manager is programmed in such a way, which implies that performance can always be improved by installing more processors. Whether the performance improvement is linear or not I will not try to evaluate.

6.10 Other

In the design of ABOS there is no mention as to what decides whether an agent is a kernel agent or a service agent or a standard user program. This is an implementation decision, but could have severe impact on the total of the model. The agents can them-

selves decide whether they are kernel agents, or they can be assigned a place in the system. If they are assigned a place, someone needs to supervise that they keep themselves to this layer and someone needs to maintain lists of what agents to start in a certain layer. If the agents decide themselves you do not need this supervisory function, but you still need some check to ensure that the agents do not exceed their authorities.

An agent should be aware of its environment and should be able to decide whether it wishes to reside in the kernel layer or the service layer, but this also implies that the agent is aware of it being a part of the operating system which makes it more difficult to incorporate agents from other vendors into the system unless they have been customized to fit the agent operating system. There is also the question of abuse. If the agent claims to be a kernel agent, someone must perform a check to ensure that the user who launched the agent is authorized to start up kernel agents. You can obviously not trust the judgement of the agent itself in such cases. Considering all these aspects, I believe that it is desirable to have a certain agent that installs and supervises the other agents.

In a multi-tasked operating system it is common to use an interrupt-based approach to communicate with hardware. To have the entire operating system halt and wait for some device I/O is naturally not possible. The interrupt signal, when it comes, should be directed to the process that is interested in the signal. This process may not be the one that is running when the signal comes so some mechanism is needed to halt the currently executing process, make a note that an interrupt has arrived in the receiving agents process control block, resume running the halted process and finally notify the process that an interrupt has occurred when it is time to run it again. It may even be needed to run the receiving process directly, allowing it to clear buffers before new data arrives. Be that as it may, what is more important is that the core needs to recognize interrupts and to perform some action when one arrives. The core can already handle page fault interrupts and timer interrupts, but it must also be able to manage all other interrupts. The best way to do this would be to extend the core with an extra module that takes care of incoming interrupts and directs them to the appropriate core module. The core process manager needs a table in each process' PCB stating which interrupts the process subscribes for and a memory address to call in the process' memory space.

6.11 Summary

In the evaluation above, I have tried to impartially both criticize and praise the design of ABOS. I have solved the problems encountered and have tried to come up with new ideas of how to put the good things to use. In total this amounts to a better, more thorough design of ABOS. I believe that the main structure is as robust as it can be, and that many of the problems one will encounter during development has been covered. What remains to be done is, of course, the tedious task of designing every core module and kernel agent in detail, and to implement the system. This is however out of the scope for this thesis, I believe that I have proven that agents can be used to construct an operating system, and that I can gain many benefits in so doing.

I have gone through great trouble to really use the flexibility provided by ABOS. I have created a situation where applications no longer need to adjust to a static operating system. Instead, ABOS adjusts to the applications by adding new file agents, process managers and so on. This amounts to applications being able to execute on raw hardware, but still having the possibility to coexist with other applications running concurrently. In this sense I have kept the ideas from Aegis [6], but crafted the solution in a completely different way.

In my solution applications from different vendors can use the agents provided by another vendor. The applications need not be aware of which vendor a certain agent comes from. Unlike Aegis, where you can at best share operating system components when developing, you can in ABOS share them at runtime as well. This means that software from different vendors can be made to collaborate in ways that were not known or expected when the applications were developed. The components can furthermore be replaced and upgraded at runtime without the applications knowing or even noticing that this is happening.

One of the main idea behind ABOS is openness. Nothing should be so hidden in the kernel that clients cannot modify it. I believe that I have achieved this open system. Applications can customize the behavior of the kernel down to the smallest component. Interesting to notice is that such a customization is only noticeable for a subset of the processes, those that wants to use the particular scheme. Other applications will not be disturbed by these tunings, and can continue to work as normal.

7. Conclusions

This section concludes the work done in the previous chapters. A discussion is held regarding what I have achieved, and where to go from here.

7.1 Summary

This thesis consists of several parts. A survey of how operating systems are built today is followed by a description the major trends in operating system research, showing that whereas the current operating systems are built as large monolithic kernels, there are a number of interesting research operating systems. These research systems commonly focus on object orientation and modularization.

Following a description of agents and some of the concepts commonly connected to agents, is an investigation of what has been done with agents in operating systems already after which, in the realization that this area of research is still unexplored, there is further motivation for the suitability of agents in operating system kernels.

In order to be able to design an operating system, it is vital to know what tasks an operating system should be able to perform. Chapter 3 is spent identifying and discussing what tasks an operating system should perform, giving examples of common situations and trying to find pitfalls.

The two chapters following are spent designing ABOS, an agent-based operating system supporting the tasks identified earlier. The rough design in Chapter 4 is further refined by designing some tasks in even more detail. Chapter 6 is spent evaluating the design to ensure that the requirements has been met.

7.2 Conclusions

This thesis has achieved a confirmation that agents are indeed suitable for use in operating system kernels, and that you can in fact base the entire operating system on agents that interact with each other. To use agents instead of traditional approaches ensures that you get greater flexibility, as well as interesting opportunities to make a distributed operating system.

The design of ABOS shows some interesting qualities which are until now, and to my best knowledge, unheard of. The service layer is an example of such a feature. This extra layer in the operating system does not provide any extra functionality in itself, but is a very good way to categorize applications that are not quite kernel programs and not quite user applications either. By having a special layer for such services you can generate more detailed priority, access, and scheduling policies.

The agent-based file system described in Chapter 5 shows that there is still much work to do regarding how we perceive file systems. For decades virtually no attempt has been made, and certainly none has succeeded to add to or change the concept of files and directories. I believe that the agent-based file system will introduce a change in the vision of file systems, even if some aspects like security speaks against it. Agents as files provides so much new in flexibility and usefulness that it is my firm belief that the concept will be used even in commercial operating systems.

ABOS can also be used in real-time environments. There is a possibility to affect the scheduling for a process in such a way that it can gain exclusive access to the CPU for limited periods of time. Compared to other real-time systems, ABOS is much more flexible in that it does not use static scheduling for a predefined number of threads.

Unlike the classical operating systems you do not have to change the priority of a process manually, something which often renders the user incapable of lowering the priority again.

Furthermore, in ABOS a situation is created where the operating system adjusts to the applications currently running. The traditional approach is otherwise to assume that the operating system is static, and instead write the application to circumvent the liabilities in the operating system. If you can modify the kernel, there is no way to modify it for a single application only. The changes you make is visible to all other applications as well. In ABOS the situation is different. Here, you can modify the kernel by adding new kernel agents and negotiate with the existing ones, and the changes you thus make to the kernel is only visible to your application.

7.3 Future work

The design and evaluation of ABOS in the previous chapters is so far only a paper product. The next step is of course to implement the core and at least some of the kernel agents. This, however, is a much greater task than the scope of a master's thesis.

After the core and kernel has been implemented, a further evaluation should be made to verify that it works as a real product as well as a mind experiment. I would not be surprised if many things need to be made differently, but I also believe that the basic layout will remain, as will the flexibility.

References

1. Bershad, B. *"The Increasing Irrelevance of IPC performance for Microkernel-Based Operating Systems"*, USENIX Microkernels Workshop, 1992.
2. Dearle, A., Rosenberg, J., Henskens F., Vaughan, F., Maciunas, K. *"An Examination of Operating System Support for Persistent Object Systems"*, Proceedings of the 25th Hawaii International Conference on System Sciences, vol 1, 1992.
3. Douglis, F., Ousterhout, J., *"Transparent Process Migration: Design Alternatives and the Sprite Implementation"*, Software-Practice & Experience, 21(8): 757-785, August 1991.
4. Homburg, P., van Steen, M., Tanenbaum, A. S. *"An Architecture for A Scalable Wide Area Distributed System"*, Proceedings of the Seventh ACM SIGOPS European Workshop, ACM, New York, pp. 75-82, 1996.
5. Hamilton, G., Kougiouris, P. *"The Spring nucleus: A microkernel for objects"*, Proceedings of the 1993 Summer Usenix Conference, June 1993.
6. Engler, D. R., Kaashoek, F., O'Toole Jr., J. *"Exokernel: An Operating System Architecture for Application-Level Resource Management"*, Proceedings of the Fifteenth Symposium on Operating Systems Principles, December 1995.
7. Tanenbaum, A. S. *"Modern Operating Systems"*, Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.
8. Stallings, W. *"Operating Systems, 2nd edition"*, Prentice Hall, Englewood Cliffs, New Jersey 07632, 1995.
9. Rashid, R., Baron, R., Forin, A., Golub. A., Jones, M., Julin, D., Orr, D., Sanzi, R., *"Mach: A Foundation for Open Systems."*, Proceedings of the Second Workshop on Workstation Operating Systems (WWOS2), 1989.
10. Rashid, R., Julin, D., Orr, D., Sanzi, R., Baron, R., Forin, A., Golub, D., Jones, M., *"Mach: A System Software Kernel."*, Proceedings of the 34th Computer Society International Conference COMPCON 89, 1989.
11. *"The Common Object Request Broker Architecture and Specification"*, BNR Europe Ltd., 1995.
12. *"NDS For NT"*, Novell Inc., 1997.
13. Walli, S. R. *"OPENNT: UNIX Application Portability to Windows NT via an Alternative Environment Subsystem"*, Proceedings of the USENIX Windows NT workshop, Seattle, Washington, 1997.
14. *"JavaStation NC. Slashing the Total Cost of Ownership for Fixed Function Applications"*, Site on the Web: http://www.sun.com/javasystems/krups/ , Sun Microsystems Inc., (Date Unknown).
15. *"Sun Announces New HPC Server Clustering Capabilities Supporting up to 256 Processors"*, SUN press release, Palo Alto nov 11., 1997.
16. Wallström, M. *"NT passerar Unix inom arbetsstationer"*, Computer Sweden, 98-03-16.
17. Kessler, P. B. *"A Client-Side Stub-Interpreter"*, Proceedings of ACM Workshop on Interface Definition Languages, January 1994.
18. van Steen, M., Homburg, P., van Doorn, L., Tanenbaum, A. S., de Jonge, W. *"Towards Object-based Wide-Area Distributed Systems"*, Proceedings of the Fourth International Workshop on Object Orientation in Operating Systems, IEEE, New York, pp. 224-227, 1995.
19. van Steen, M., Hauck, F. J., Tanenbaum, A. S., *"A Model for Worldwide Tracking of Distributed Objects"*, Proceedings of TINA 96, Eurescom, pp. 203-212, 1996.
20. Higgs, B. J. *"History of Object-Orientation in Programming Languages"*, Lecture slides available at: http://niagara.rivier.edu/staff/bhiggs/FrontPageWebs/OOPandCPP/history/history.htm , Rivier College, 1998.
21. Brookshear, J. G. *"Theory of Computation. Formal Languages, Automata, and Complexity"*, Benjamin-Cummings, 1989.

22. Higgs, B. J. *"What Problems are We Trying to Solve? The Motivating Forces behind Object-Oriented Programming and Design"*, Lecture slides available at: http://niagara.rivier.edu/staff/bhiggs/FrontPageWebs/OOPandCPP/why/why.htm, Rivier College, 1998.

23. Johansen, D., van Renesse, R., Schneider, F. B. *"Operating System Support for Mobile Agents"*, Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems, pp. 42-45, 1995.

24. Nwana, H. S. *"Software Agents: An Overview"*, Knowledge Engineering Review, Vol 11, No 3, pp. 205-244, October/November 1996.

25. Morrison, M. *"Presenting JavaBeans"*, Sams.net publishing, 1997.

26. Persson, E. *"Component Technology. Infrastructures and Enabling Technologies - a Short Survey"*, LU-CS-TR:97-197, LUTEDX/(TECS-3078)/1-71/(1997), Department of Computer Science, Lund University, 1997.

27. Fredriksson, M. *"Agent Oriented Programming"*, Unpublished paper, University of Karlskrona/Ronneby, 1997.

28. Cohen, P. R., Cheyer, A., Wang, M., Baeg, S. C. *"An Open Agent Architecture"*, Proceedings of the AAAI Spring Symposium on Software Agents, pp. 1-8, 1994.

29. Liu, J-S., Sycara, K. P. *"Multiagent Coordination in Tightly Coupled Task Scheduling"*, Proceedings of the First International Conference on Multiagent Systems, pp. 181-188, 1996.

30. Labrou, Y., Finin, T. *"Semantics and Conversations for an Agent Communication Language"*, Proceedings of the Fifteenth International Conference on Artificial Intelligence, pp. 584-591, 1997.

31. McCabe, F. G., Clark, K. L. *"April - Agent PRocess Interaction Language"*, Department of Computing, Imperial College, London, 1994.

32. Halsall, F. *"Data Communications, Computer Networks and Open Systems, Fourth Edition"*, Addison-Wesley Publishing Company Inc., 1996.

33. Rus, D., Gray, R., Kotz, D. *"Transportable Information Agents"*, Proceedings of the International Conference on Autonomous Agents, pp. 228-236, 1997.

34. Woolridge, M., Jennings, N. R., *"Pitfalls of Agent-Oriented Development"*, Proceedings of 2nd International Conference on Autonomous Agents (Agents-98), Minneapolis, USA. (to appear)

35. Ekdahl, B. *"Computerized Agents from a Linguistic Perspective"*, Ph.D. thesis, Department of Computer Science, Lund University, 1997.

36. Ousterhout, J. K. *"Tcl and the Tk Toolkit"*, Addison-Wesley, Reading, Massachusetts, 1994.

37. Genesereth, M. R., Singh, N. R. *"A Knowledge Sharing Approach to Software Interoperation"*, Computer Science Department, Stanford University, 1994.

38. *"FIPA 97 Specification. Part 2: Agent Communication Language"*, Foundation for Intelligent Physical Agents, Geneva, 1997.

39. Ford, B., Susarla, S. *"CPU Inheritance Scheduling"*, Proceedings of OSDI'96, October 1996.

40. Ballesteros, F. J., Fernández, L. L. *"Advice: An Adaptable and Extensible Distributed Virtual Memory Architecture"*, Proceedings of the IASTED PDCS'96, Chicago IL, 1997.

41. *"Middleware -- The Essential Component for Enterprise Client/Server Applications"*, International Systems Group, Inc., New York, 1997.

42. *"Java Remote Method Invocation - Distributed Computing for Java"*, Sun Microsystems, Site on the Web: http://java.sun.com/marketing/collateral/javarmi.html, 1998.

43. Fredriksson, M. *"Active Documents and their Applicability in Distributed Environments"*, Unpublished Master's Thesis, University of Karlskrona/Ronneby, Pending release 1998.

44. Mills, D. L. *"Internet Time Synchronization: the Network Time Protocol"*, IEEE Trans. Communications 39, 10 (October 1991), pp. 1482-1493, 1991.

45. Matthews, J. N., Roselli, D., Costello, A. M., Wang, R. Y., Anderson, T. E. *"Improving the Performance of Log-Structured File Systems with Adaptive methods"*, Proceedings of the Sixteenth ACM Symposium on Operating System Principles, 1997.

46. Ward, B. *"The Linux Kernel HOWTO"*, Site on the Web: ftp://ftp.funet.fi/pub/Linux/doc/HOWTO/Kernel-HOWTO , 1997.

47. Barrus, F., *"Shag/OS: A Small, Dynamic, Object-Oriented, MicroKernel-based Operating System"*, Unpublished study-project proposal, Rochester Institute of Technology, 1995.

48. van Doorn, L., Homburg, P., Tanenbaum, A. S. *"Paramecium: An extensible object-based kernel"*, Proceedings of Hot Topics in Operating Systems V, ACM, New York, pp. 86-89, 1995.

49. Homburg, P., van Steen, M., Tanenbaum, A. S., *"The Architectural Design of GLOBE: A Wide-Area Distributed System"*, Technical Report IR-422, Vrije Universiteit Amsterdam, 1997.

50. Russinovich, M. *"Inside the Windows NT Scheduler, Part 1 & 2"*, Windows NT Magazine, July & August 1997.

51. "*Norton Your Eyes Only 4.1 for Windows NT/Windows 95"*, Symantec Corporation, Site on the Web: http://www.symantec.com/yeo/fs_yeo41-95nt.html , (Date unknown).

52. *"NFS Software from Sun: Bringing the Enterprise to your Desktop"*, Sun Microsystems, Site on the Web: http://www.sun.com/netclient/wp-nfs.sw/ , (Date unknown).

53. Hall, F., Hellspong, M. *"EPIDEMIC: A Framework for Object Oriented Compiler Construction"*, Unpublished Master's Thesis, University of Karlskrona/Ronneby, 1998.