The LINK Operating System Architecture and Security Model

Anders Franz Terkelsen

27th July 2007

Department of Computer Science
Aalborg University

Title:
The LINK Operating System Architecture and Security Model

Topic:
Distributed systems and semantics, operating system architectures, formal security models

Group Members:
Anders Franz Terkelsen

Project Group:
d602a (room B2-201)

Supervisor:
Josva Kleist

Semester:
Dat6

Project Period:
Feb 1st 2007 to Jul 27th 2007

Copies:
5

Pages:
Thesis: 73
Appendices: 8
Total: 82

Synopsis:
LINK Is Not a Kernel (LINK) is a new operating system architecture developed for IA-32 (x86) computers. In LINK there is no kernel, but instead a set of *system services* which cooperate to perform the duties of an OS. All these system services, except one, run at privilege level 3. The only privilege level 0 system service is the *task switcher* which has the responsibility of performing context switches between tasks.

A new security model has been developed for LINK that use hierachically named capabilities. This security model is formally analysed and it is proved that it can be used to reason about access control and information flow. It is also proved that the LINK security model can simulate the Unix user-group security model.

Acknowledgements

I would like to thank Arild Haugstad for spending many hours with me discussing various security model ideas, and molding the Security Model section with me until we both found it satisfactory.

Thanks also goes to Willard Rafnsson for the many hours I spend at his place drawing like a madman on his many whiteboards and discussing with him my architectural ideas. Often also in the company of Arild as well.

I would also like to thank Simon Konghøj and Robert Olesen for taking the time to discuss my work with me and give feedback on a drafts of this thesis, and an extra thanks goes to Simon for putting the L in LINK.

As a computer scientest at Aalborg University (AAU) I also feel obliged to thank our social computer scientist club called F-klub. Thanks to the many great social events put together every semester by F-klub it will never become boring to be a computer scientist at AAU.

Contents

1. Introduction **9**

2. Contributions **14**

3. Related Work **14**
3.1. Monolithic kernels . . . 15
3.2. Micro-kernels . . . 16
3.3. Object-Oriented and Component-Based Operating Systems . . . 17
3.4. Exokernels . . . 18

4. LINK Architecture **20**
4.1. Original LINK Architecture . . . 21
4.2. New LINK Architecture . . . 24

5. Implementation **26**
5.1. The Benchmark Tool . . . 26
5.2. Benchmark Results . . . 28
5.3. Proof of Concept LINK (POCLINK) . . . 29
5.4. Summary . . . 34

6. Security Model **35**
6.1. Hierarchical Protection Graphs . . . 38
6.2. De Jure Rules . . . 40
6.3. De Facto Rules . . . 46
6.4. Combined Transfers . . . 50
6.5. Modelling User-Group Using HNCs . . . 54
6.6. Summary . . . 63

7. Future Work **63**
7.1. PXELINK . . . 63
7.2. Security Model . . . 68

8. Conclusion **68**

A. Benchmark Results **75**
A.1. Paging Disabled . . . 75
A.2. Paging Enabled . . . 76

B. Setting Up a Test Environment **77**
B.1. DHCP Server . . . 78
B.2. TFTP Server . . . 79
B.3. Using Real Hardware . . . 79
B.4. Using QEMU . . . 79

List of Figures

1. Monolithic kernel overview. 16
2. Micro-kernel overview. 17
3. Exokernel overview. 18
4. The old LINK architecture using hardware task switching. 21
5. LINK memory organisation 23
6. The new LINK architecture 25
7. POCLINK source code file tree. 30
8. Hierarchically named capability structure, as proposed by Mazieres. 36
9. Hierarchically named capability structure, as used in LINK. 37
10. Example of a combined transfer. 51

List of Tables

1. Benchmark comparison of different mechanisms for performing context switches. 28
2. Hardware task switcher benchmarks with paging disabled. 75
3. Software task switcher benchmarks with paging disabled. The entire task state is saved. 75
4. Software task switcher benchmarks with paging disabled. No task state is saved. 75
5. SYSENTER/SYSEXIT benchmarks with paging disabled. 76
6. Hardware task switcher benchmarks with paging enabled. 76
7. Software task switcher benchmarks with paging enabled. The entire task state is saved. 76
8. Software task switcher benchmarks with paging enabled. No task state is saved. 77
9. SYSENTER/SYSEXIT benchmarks with paging enabled. 77

1. Introduction

LINK Is Not a Kernel (LINK) is a new operating system (OS) architecture developed for IA-32 (x86) computers. As the name implies there is no kernel in the LINK architecture, instead the OS is designed as a set cooperating system services. LINK is an attempt of rethinking the OS, so let us define what we actual mean by an OS:

Definition 1.1 (Operating System) An *operating system* is a software system designed to create and maintain suitable and safe environments for applications to run in. □

Now what is a suitable and safe environment for applications ment to run on the OS? If the application can trust that its data will not be corrupted or manipulated with by other applications then the environment is safe. If the application has all the resources available which it needs to function properly then the environment is suitable.

Our definition of an OS is very broad but that is on purpose. An OS is from our point of view the entire *Trusted Computing Base* but exactly what this TCB should consist of our definition does not say. In operating systems like various BSD and Linux systems the TCB consists of the kernel and a set of trusted applications and libraries (shells, c compiler, standard c library, etc.). The TCB, as defined by Lampson[LABW92] is:

> A small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security.

In LINK the TCB has become a set of system services, libraries, and applications. The system services are actually also just applications running in user space (privilige level 3) as any other application would do.

The only system service that run at privilege level 0 is the *task scheduler* as it needs access to privileged machine code instructions in order to perform context switches between tasks. We talk about *tasks* in LINK instead of processes, as LINK is developed for the IA-32 computer architecture, and this architecture has the concept of tasks which are a kind minimalistic processes, if compared to for instance Linux processes.

But why create yet another OS? Can we not be content with systems like the various flavours of BSD[ope, fre, net] and Linux[lin]? There are still many issues in the world of operating systems that needs to be resolved, so our answer is no. We have identified some problems with the current available operating systems which we now look into:

Performance One of the greatest virtues of any OS is to get the most out of the hardware it is running on. Performance is in no way trivial as the *central processing unit* (CPU) must multitask a bunch of applications while constantly being interrupted by I/O devices that needs work done as well, and preferably do all this without wasting any CPU cycles. The art of wasting as little CPU cycles as possible is very dependant on what the system in mind is going to be used for. This has lead to a lot of research in process scheduling and how to dynamically create scheduling policies that satisfy every process on the system as best as possible.

Today we also have the concept of *real-time* applications to further complicate scheduling, as some processes now have strict deadlines which must be met and it is the job of the OS to ensure this. In general purpose OSes the problem only concerns *soft real-time* where some deadlines may be missed, as there are no way to guarantee *hard real-time* on a system that can have an arbitrary amount of processes running which all need CPU cycles to do their job.

The problem as we see it is that given some OS it is not possible to create a proper scheduling algorithm unless it is known what applications will be run on that system. Furthermore, the OS has no way to know the actually needs of an application. In every OS different application programming interfaces (API) are available to application developers allowing them to interact with the OS and through these interfaces specify the application's needs. The problem is however that there is no way to create an interface that allows all kinds of applications to sufficiently express their needs. This results in applications not getting exactly what they need and thus we get sub-optimal performance.

Dawson R. Engler [Eng98] found a possible solution to this problem. The OS kernel, which is where the scheduling mechanisms normally are located, should not try to intelligently schedule processes. It should simply allow the applications to say what quantums (time slices) they want. In fact, the kernel should not try to manage resources at all, but only safely multiplex them to applications. This idea has been proven to be very efficient[KEG+97] and we will look closer at the ideas of Engler et. al in Section 3.4.

Portability With all the different kinds of computer architectures that exists today it is important that an OS can be ported to another hardware without to much trouble. NetBSD[net] is an excellent example of an OS that is very portable. It runs on most workstation, server, and PC architectures, as well as several game systems. To make OSes portable various techniques are used, such as adding a *hardware abstraction layer* (HAL) to conceal the actually hardware and present the rest of the system with a generic computer no matter what hardware it in reality is running on. The problem with creating generic OSes in this way is that the HAL might prevent the hardware from being used optimally. As described in the above point about performance, only the applications knows their needs, and perhaps some applications could use a hardware specific feature but since only some hardware architectures have this feature, the HAL abstracts it away and the application might not even know that the feature is available. One could of course just make hardware specific additions to the HAL but this would in turn force application developers to create different versions for different hardware platforms; something a HAL is ment to prevent. The real problem causing this is actually the commonly accepted philosophy behind operating systems: An operating system has a kernel which abstracts away the hardware from applications. But why abstract these things in the kernel where it is impossible to change it? Instead let shared libraries provide the abstractions, and let the kernel present the hardware for what

it is. This way application developers can either work directly with the primitives provided by the kernel or use a library which has an appropriate level of abstraction.

This does not mean that the kernel has to be programmed from scratch for each new operating system, as many hardware architectures has similar features, but these things can be appropriately handled using proper code management techniques. As another solution to this problem, one could eliminate the kernel and create a set of system services that together provide the same functionality as the kernel. Then some of the system services might be usable more or less as-is on various hardware platforms, and others might have to be created from scratch to make proper use of hardware specific features.

The problems with supporting different hardware platforms also arise when support for new hardware devices is needed. Again, the typical way to handle this is to let the kernel have generic interfaces that allow applications to use the different hardware components. But then the problem with interfaces described above arise. So instead of letting the kernel manage this, let there be a system service for each hardware device. Then some library or other system service can serve as a generic interface to a set of similar devices (e.g. ethernet adapters) and applications can use the generic approach if they do not have any special requirements.

Flexibility Much of what has been discussed above are actual flexibility issues. In Definition 1.1 we stated that the OS should create a suitable environment for applications. An environment that is suitable for one application might be close to inhabitable by another, so how should an OS ever be able to make application developers content? By not forcing any high-level abstraction upon them. Hardware should be abstracted as little as possible, and as much information as possible should be available about the system and its hardware – without it compromising security.

This approach gives applications the freedom to use the available resources as they please, and shared libraries could supply the generic functionality to all the applications without special needs.

Extensibility An OS should also be extensible. That is, if new hardware is added, new network protocols invented, etc., then it should be possible to add these new features to the OS without having to rebuild a kernel, or something similar. What's more important is that extending the system should not only be something OS developers can do. Extensibility is a special case of flexibility but a case more about what the operating system can do, and less about what the applications can do. A problem with extensibility is how to control what new software can become part of the OS and what cannot. If there is a kernel, then how do we grantee that the code we are about to load into kernel space does not crash the entire system? Instead of trying to solve this problem directly there is a way to avoid it altogether. Do not have a kernel. Have a set of system services that cooperate in order to function as an operating system. Then the problem of adding a new feature to a kernel has been transformed into to the problem of creating a new

system service. System services should of course work in such a way that if one service crashes it cannot bring the entire system down. So system services should have their own address space, and this in turn means that for x86 architecture that they should run at privilege level 3 (in user space). Some system services as for instance a task scheduler or memory manager cannot afford to crash as this would kill the entire system therefore have to be marked as trusted and special care must be taken when handling these. However, the question of extensibility could probably never end up concerning a trusted system service as these services are part of the hardware that normally do not change.

Parallelism Today multi-core processors are becoming more and more common, and multi-processor systems have been common for many years. On top of this distributed system continue to evolve and mature. The problem of parallelism exists in all of these cases. When we go from single-core CPU multitasking to true parallel computation a whole new set of problems comes along as well. A modern OS should support true parallelism from day one. An OS with a kernel needs to take protective measures such that the kernels code can be safely run in parallel. It is however often hard to fathom the intricacies of intra-kernel communication and thus implementing efficient parallelism is hard. The problem is how to properly implement parallelism using only non-blocking mechanisms. The more places during execution of a parallel process it has to wait for another process to finish, the greater the inefficiency of the overall system becomes. Implementing parallelism using blocking constructions (e.g. spinlocks) is the same as directly implementing inefficiency into it, but sometimes it might of course be needed to ensure proper program behaviour. It should however only be done as an absolute last resort as it forces parallelism into sequential bottlenecks.

If a kernel were to be split up into a set of system services then each system service must live up to the requirements of parallelism. The good thing about system services versus a kernel is that they are much smaller and less complex. Less complexity greatly improves the possibility of implementing proper parallelism as their small size makes it much more probable to for instance create formal concurrency models of them and verify these in a model checker to ensure correct behaviour. The importance of the ability to easily create formal models cannot be emphasized enough. It is humanly impossible to foresee all the possible interactions between parallel process and we need formally proved methods of verification to ensure that our programs indeed behaves correctly.

Using a set of system services, all able to run in parallel with each other also simplifies the task of creating distributed software for such a system. By ensuring parallelism in every system service, a system can be thought of as nothing more than a set of resources which can be shared among applications, whether these are running on the same machine or somewhere else connected via some network.

Safety In Definition 1.1 we stated that the OS should create and maintain *safe* environments for applications. The OS must ensure that an application's data is

private, and that applications safely can share their data with other applications. This can easily be done using virtual memory (paging) as is the method used by most OSes today. Virtual memory allows each application to have its own address space and if an application wishes to share some memory with another application then the OS simply creates an alias to the memory page in the other applications address space. This does of course means that data shared will at least be the size of a page. There is however one part of the system which normally lacks memory protection and that is the kernel. The kernel has access to all the systems memory since the memory manager, which is part of the kernel, is responsible for managing it all. This is however a very bad thing as anything in the kernel can cause total and utter havoc to the system. If a new device driver is loaded into kernel space it can crash the system in an instant if it does something it should not. In reality, on the x86 architecture, only the memory manager needs to have access to the entire memory, and only the part of the kernel which does context switching needs to have access to the privileged machine code instructions. So again, if the kernel is split into a set of system services, then the memory manager system service can as the only service have access to all memory, and is of course considered a trusted system service. This will make OS development a lot easier as the causes of errors suddenly become easy to locate to a single system service and a failure in one system service cannot cause failures in other system services. If a trusted system service crashes it can however still crash the entire system, but as a system service is very small and simpel compared to an entire kernel they are likely to be less error-prone.

Security There is an important difference between *safety* and *security*: Safety refers to the protection of resources using protection mechanisms, and security refers to the policies used to control the protection mechanisms.

An OS must have some way of allowing the system's administrator to control what each user and application has the right to do on the system. That is, the OS must have some way to specify and enforce *access control*. But access control is not always enough, as some systems might need the insurance that a user cannot give a certain piece of information along to others. In other words, an OS with such requirements must have some way to specify and enforce *information flow* policies as well.

System security is an entire research field in itself and will be looked upon in detail in Section 6.

The above problems were the motivation for the invention of a new OS architecture, and the LINK architecture is the result of this.

LINK as described in this thesis is already mentioned designed for the IA-32 computer architecture and it is thus assumed throughout this thesis that the reader has some basic knowledge about this specific architecture. Otherwise, this thesis is self-contained. The full documentation for the IA-32 architecture can be found in [inta, intb, intc, intd, inte].

In the following section we will present the contributions of this thesis. In Section 3 we look into other operating system architectures. Especially one particular architecture

which has been a great inspiration when designing LINK. In Section 4 we present the LINK architecture and in Section 5 we present two systems which have been implemented as part of this project: A benchmark tool for benchmarking different task switching mechanisms, and a proof of concept implementation of the LINK architecture. A new security model has been developed for the LINK architecture and is discussed in detail in Section 6. We discuss future work in Section 7 and conclude in Section 8.

2. Contributions

This thesis contributes with the following to the field of computer science:

- A new operating system architecture called LINK (Section 4).
- A proof of concept implementation of LINK (Section 5.3).
- Performance benchmarks of the hardware task switching mechanism that is available on the IA-32 architectures as well as benchmarks of a simple software task switching mechanism, and a comparison of the two (Section 5.1).
- A new security model based on hierarchically named capabilities, along with a formalism for reasoning about it (Section 6).
- Analysis of how the developed formalism can be used to reason about access control and information flow in an operating system applying the model (Section 6).
- Proof that the developed security model is both a discretionary access control model and a mandatory access control model (Section 6).
- A formal model of the Unix user-group security model expressed using the formalism developed for hierarchically named capabilities (Section 6.5),

3. Related Work

Ever since the invention of the stored program computer, operating systems have played an important part of the evolution of both software and hardware. New hardware is invented and the OS must support it or sometimes new ideas arise in the field of software which results in new hardware being invented.

Thus many different operating systems have been invented over the years. In this section we will look at some of them. We start by looking at different general OS architectures and then go into detail with a couple of specific architectures which have inspired some of the ideas behind LINK.

3.1. Monolithic kernels

A *monolithic kernel* operating system is what we today think of as the classical OS architecture. Linux and most Unix systems all have monolithic kernels[Lov05]. The kernel is the heart of such an OS. It manages the hardware and abstracts it to a level where it is deemed suitable for applications to use. Monolithic kernels have always been known to be big and hard to maintain due to the very high coupling between the components of the kernel. They are however known to be fast and provide a single point of entry for applications wishing to interact with the underlying hardware. The main reason for monolithic kernels still being so widely used and popular is that they are fast. The IA-32 architecture have different privilege levels (or protection rings as called in IA-32 terminology[inta]) built into the CPU . The kernel normally runs at privilege level 0 in a single all-encompassing address space, which means it has full control over the entire system; and applications run at privilege level 3, each having their own address space, meaning that they for instance do not have access to privileged machine code instruction for manipulating the virtual memory, and cannot manipulate the memory of other applications. Crossing a privilege level boundary, for instance when an application calls into the kernel, consumes considerable more CPU cycles than for instance a normal function call within the same privilege level and address space. This is due to the time it takes for the CPU to enable the privileged instructions and load the new set of page tables etc.

Loading and running an application typically means creating a new process, loading the executable into that process, allocating the needed memory and so on. This means that a lot of different components of the kernel become active during this: the process scheduler, memory manager, disk driver, etc. Since the kernel resides in a single address space and has full privilege all these intra-kernel calls are just normal function calls. The performance gained by this does however not come for free: A change in the hardware means that a new kernel must be build and loaded. This problem has been partially solved in some systems by using *modules*. For instance, a lot of the functionality in the Linux kernel can be build as modules, which are blocks of binary code that can be loaded into and unloaded out of the kernel while the system is running. However, if an erroneous module is loaded into kernel-space it can easily crash the entire system. Furthermore, not everything can be build as a module.

On Figure 1 an overview of an OS architecture with a monolithic kernel can be seen.

On the figure there is a box labelled *System calls*. This represents the interface the kernel supplies to the rest of the system (normally supplied as an API), and which is the only point of entry into the kernel. All the unlabelled boxes shown in kernel space are various parts of the kernel, such as a process scheduler, memory manager, disk driver, etc. If the kernel supports the loading of modules then a module would be shown as such a box as well. The kernel may interact directly with the hardware and some system call might even just be wrappers for calling some hardware device directly.

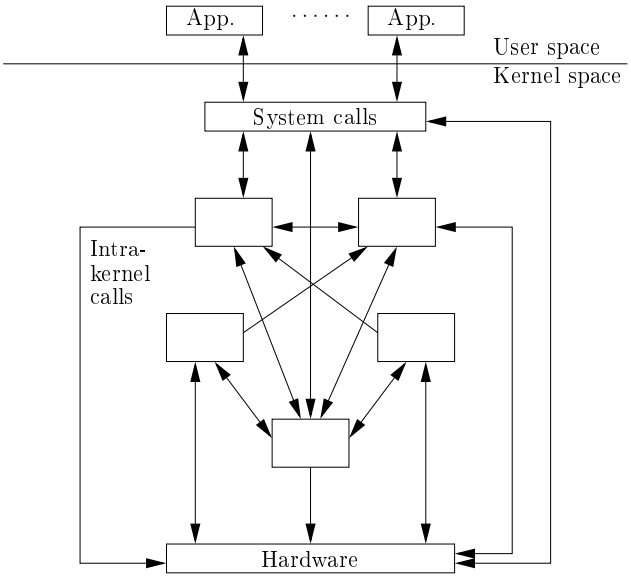


Figure 1: Monolithic kernel overview.

3.2. Micro-kernels

The micro-kernel (sometimes written μ-kernel) was invented to solve some of the problems found with monolithic kernels. The micro-kernel, is as the name implies, a small kernel. The idea is that only the most essential functionality should be located in the kernel, and the rest of the functionality should be handled by user space *servers*. The user space servers, like any other application in user space, have their own address space but have more privileges than the average user application. The idea is that a file server has the necessary privileges needed to supply applications with file system services but it should not be allowed to do more than that. In a micro-kernel system the applications use inter-process communication (IPC) to communicate with servers and each other. Every time IPC is performed it means a protection boundary is crossed. The kernel must save the message, load the address space of the process which the message is for and then give the control over to that process. The first generation of micro-kernels suffered a serious performance overhead caused by IPC. However, today's second generation micro-kernels have highly optimized IPC and can now be compared in speed to monolithic kernels[HHL$^+$] though still slower, the performance gap between the two is getting smaller and smaller.

An overview of a micro-kernel system is shown in Figure 2.

As can been seen on the figure applications and servers all execute in user space.

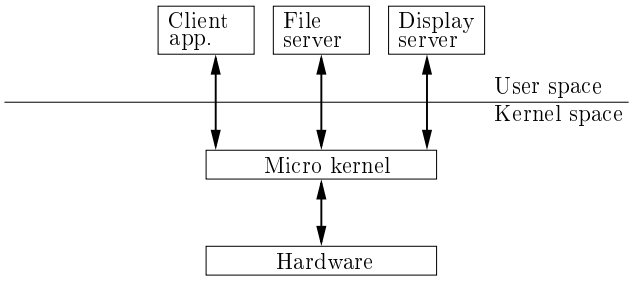


Figure 2: Micro-kernel overview.

3.3. Object-Oriented and Component-Based Operating Systems

The huge success of objected oriented design and programming languages of course also led to the invention of *objected oriented operating systems* (OOOSes). There have been different kinds of OOOSes, each supporting objects in different ways. But common for all of them is the wish to incorporate concepts of object orientation into an operating system in one way or the other. One OOOS worth mentioning is Spring[MGH+94] which was created by Sun Microsystems. Objects were specified using an Interface Definition Language (IDL) and techniques like inheritance could be used throughout the system.

Component-Based operating systems are sometimes hard to differentiate from OOOSes as it is not official defined anywhere what makes an *object* and what makes a *component*. Greg Law has, however, given a good way to mentally categorise the two[Law01]:

> A good differentiator is that objects are fundamentally a programmer's tool while components are more concrete entities. That is, traditional objects exist in the program's source code only, and are pertinent mainly to type-theory. For example, once a C++ program is compiled, the boundaries between objects disappear; indeed, it is not possible to state with 100% confidence whether a binary were produced using C++ or C as its source (or even assembly). On the other hand, the boundaries between components are concrete and are present in the running system — it should be trivial to produce a tool to allow the user to examine what components exists at any time. In this regard, a traditional file is closer to a component than is an object. In fact, a process is a better analogy still since a process includes behaviour as well as state.

Like with OOOSes there are different kinds of Component-Based Operating Systems. One of the recent and quite interesting Component-Based OSes is Greg Law's OS, called Go![Law01]. In Go! there is no longer the ordinary notion of a kernel. Instead it has an *Object Request Broker* (ORB) that is responsible for managing the system's available objects. It can be argued that an ORB is just an extremely small kernel with a very limited service, and thus some people refer to component-based OSes like Go! as *nano kernel* operating systems. What really makes Go! a different kind of OS is that everything

runs in kernel space and the OS scans the instruction stream before executing it to ensure no disallowed instructions are contained within it. This does however not have anything to do with it being component-based.

3.4. Exokernels

In the *exokernel* architecture[Eng98] the kernel is even more minimalistic than a micro-kernel. This is mainly due to two principles which the exokernels abide to: "Separate protection from management" and "expose hardware". An exokernel does nothing more than safely multiplex the hardware and all the abstractions over the hardware is up to user-space programs. On an exokernel system one can use user-space servers like with micro-kernels but with exokernels it is preferred to use *library operating systems* (libOSes). Library operating systems are shared libraries together with some controlling processes which applications can communicate with via IPC. The difference between a libOS and a user-space server is that a libOS is not a "gate keeper"; meaning that applications do not need to use a specific libOS, they can if they want to, communicate directly with the exokernel. This has the great benefit of increasing the flexibility and extensibility of the OS. Applications can use abstraction provided by libOSes or they can build their own; or a mixture of the two. A libOS is actually just another application residing in user space.

Exokernels also apply fine-grained protection due to their low-level abstractions over hardware. For instance, access control mechanisms on the disk block level instead of the file level. The obvious reason for this is that in order for the kernel to perform access control on a file it must understand the file system on the disk, and that would in turn mean that the file system was part of the kernel. But a file system is a high-level abstraction and is more about management than protection and thus do not belong inside the exokernel.

An overview of an exokernel OS is shown in Figure 3.

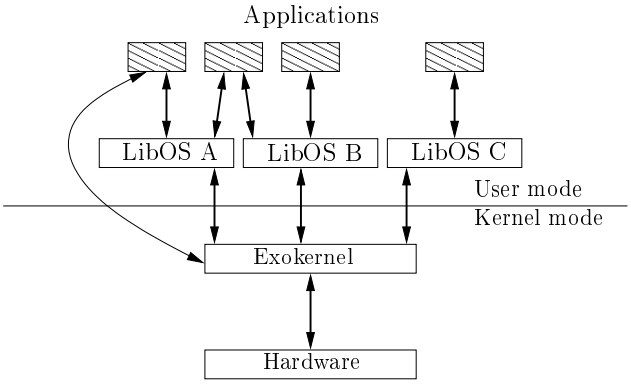


Figure 3: Exokernel overview.

The exokernel, like any other kernel, serves as interface to the hardware, though with

a much lower level of abstraction. As can be seen on the figure, applications typically make use of libOSes but the leftmost of applications also directly use the kernel's interface, which is shown by the arrow directly from the application to the exokernel.

3.4.1. The MIT Exokernels

At MIT, a couple of exokernels named Aegis and Xok, were developed during the mid-nineteen-nineties and early two-thousand. These exokernels were the first of their kind and have later inspired other system architectures such as for instance Nemesis[RF, LMB+] and Xen[BDF+03b, BDF+03a].

In this section we will look into some of the general principles and concepts that were developed for Aegis and Xok, which became the principles for exokernels in general.

Design Principles As already mentioned, the exokernel is build around the principle of "seperate protection from management". The principle of "expose hardware" follows from this, and as explained so does the principle of "protect fine-grained units".

Furthermore, the exokernel principles require the exposure of allocation, revocation, names, and information. Exposing allocation means that applications explicitly state which resource they want to allocate. In practice it will of course be common to implement the possibility for applications to just request a certain amount of a type of resource and not have to state that it for instance wants to allocate exactly that and that memory page for reading and writing. Going hand-in-hand with exposing allocation comes exposing revocation. When resources are going sparse the exokernel decides on an application and tells it to release a certain amount of resources. The application decides what instances of the resource to release and thus has the possibility to make intelligent decisions on which instance to release. The kernel must of course implement some kind of protocol for handling applications which do not release resources when asked to. This is what Dawson R. Engler refers to as an *abort protocol*[Eng98]. A simple abort protocol would be to simply kill applications which do not release the required resources within a certain time limit.

Exposing names means exposing the physical names of resources (i.e. hardware) whenever possible and exposing information means exposing as much system information to applications as possible without compromising security. The exposure of names and information gives applications a lot useful data which they can use to take intelligent decisions on the use and allocation of resources.

All the design principles can be linked back to the seperation of protection from management. This is the fundamental principle of the exokernel and it is this which gives the exokernel its unique capabilities as a fast, flexible, extensible operating system. One could say that the exokernel applications lives in freedom under responsibility. They can allocate resources as freely and madly as they want as long as they release these resources when told to. If they do not, the abort protocol defines the consequences.

Protection vs. Security Separating protection from management is possible since protection does not imply the submission to any policy. As soon as policies for how to

protect resources are added we are no longer talking about protection but security. Security is a question of how one wishes to manage the rights to certain resources and does not fall inside the domain of the exokernel. Every resource must be protected at the finest possible granularity and if this is done any security policy can be applied on top of it in user-space. A security policy can never be ensured if the protection mechanism cannot enforce it. Thus the exokernel concerns itself with protection mechanisms and not security.

Secure Bindings Even though the name might imply it, *secure bindings* have nothing to do with security. A secure binding is a protection mechanism.

Definition 3.1 (Secure Binding) A *secure binding* is a protection mechanism that decouples the authorisation from the actual use of the resource. □

The definition says that when an application requests a resource, protection checks are performed, and if all qualify then the resource is bound to that application. All further access to the resource from the application no longer needs to be checked.

Secure bindings is an efficient way to implement fine-grained protection of resources since it ensures that protection checks only need to be done at *bind time*. If the checks were done at *access time* it would create an enormous performance overhead.

A simple example of a secure binding is when an applications requests to allocate a certain page in memory. The kernel performs protection checks which means it checks if the page is free; if it is, then it maps the page into applications address space. From that point on the application can access the resource without any further protection checks.

Hierarchically Named Capabilities Keeping management entirely out of the kernel is impossible if there is to be any hope of having a secure operating system. Some security model is needed such that it is possible to specify which applications, processes, or users have access to what resources. The problem with a security model is that it can easily impose too many restrictions.

Mazières[MK97] proposed the use of *hierarchically named capabilities* in the exokernel as this is a simple and elegant security model which allows other models to be implemented on top of it. Hierachically named capabilities will be examined in detail in Section 6.

4. LINK Architecture

The basic LINK architecture was developed in the autumn of 2006[Ter07] as part of the preliminary research for this thesis. In last semester's technical report it was noted that the use of the IA-32 architecture's hardware task switching mechanism might not be as fast using a software task switcher. We have since then benchmarked the two against each other and found that the software task switcher we wrote indeed is faster. We will discuss the benchmarks that prove this in Section 5.1.

The original LINK architecture was built around the hardware task switching mechanism and has thus has modified to use a software task switcher instead. This section describes the LINK architecture as it was originally developed followed by a description of the new architecture and discuss why the changes made are insignificant to the general architectural idea and principles behind LINK.

4.1. Original LINK Architecture

An overview of the old LINK architecture is shown in Figure 4. Everything seems to be running directly "on the bare metal", and indeed, that is the case. To understand how this is possible two things must be remembered: One, switching between tasks is handled in hardware, and two, protection of memory is done using virtual memory (paging). Instead of having a kernel providing the most essential system services, LINK has a set of system services which each has a specific job to perform. For instance, all memory management is handled by a memory manager system service, tasks are scheduled by the task scheduler system service, and so on. Each of these services has their own protected address space and communicate with other services via remote procedure calls. The procedure calls are remote in the sense that they are inter-task procedure calls.

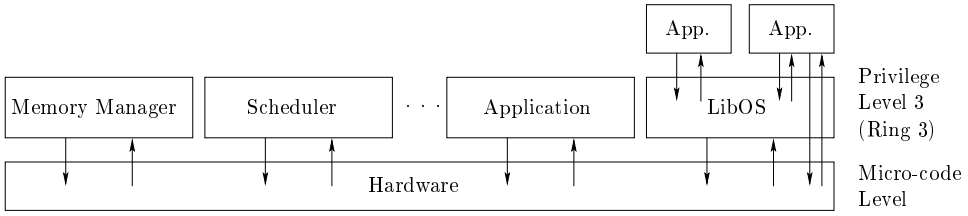


Figure 4: The old LINK architecture using hardware task switching.

The virtual memory is the glue that binds everything together. The overview in Figure 4 shows a running LINK OS. Booting the OS is done using a bootstrap task which sets up the virtual memory, loads the system services, then unloads itself and gives control to the task scheduler which selects the first task for execution. Applications do not need to make use of libOS functionality if they do not want to as can be seen on the figure. Applications can also combine the functionality of libOSes just as it is the case in exokernel systems (the right-most application).

The architecture has been developed following the same principles that the exokernel was built around. The system services must therefore only protect resources, never manage them. Even the memory manager only protects resources. So why call it a manager? Because LINK uses virtual memory, so in reality the memory is being managed; not by the memory manager system service but by the virtual memory hardware. The memory manager therefore simply provides support for functionality that exists in hardware.

Memory Organisation The CPU associates a *Task State Segment* (TSS) with each running task which contains the entire state of the task. The state consists of the following

[intd][1]:

- General purpose registers: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP.
- Segment registers: CS, DS, SS, ES, FS, GS.
- EFLAGS register.
- Page Table Base Register (PTBR): CR3
- Instruction pointer: EIP
- Additional three stack segment selectors and stack pointers for privilege levels 0, 1, and 2.
- Address of the I/O bitmap.
- Value of the Local Descriptor Table (LDT) register.
- Pointer to the previously executed task.

Basically, a TSS contains all information about a task except the Global Descriptor Table (GDT) selector and the floating point registers, and all this information can be handled directly in hardware. By constructing TSSs for every task on the system the hardware task switching mechanism can be used to switch between them. To support this the memory has been organised as shown in Figure 5.

As can be seen on the figure, all applications share the same TSS. This is because the GDT only has 8192 entries[intd], where two of them are reserved (one for the null segment selector and one for the LDT selector, not taking into account various segment selectors), leaving 8190 entries left for the OS to use. It is not possible to use multiple GDTs as loading the GDT register requires the current privilege level to be 0 and LINK always runs in privilege level 3. Since there might be more than 8190 applications running on a single system the applications share a single TSS. It is now the job of the task scheduler to keep the TSS data of all tasks and use this data when switching between tasks. Before a task switch, the task scheduler saves the TSS data of the currently running task, loads the next task's TSS data, and performs the switch. This might seem like a lot of data to save and load on each task switch but in practice only a single value has to be read and stored. If the task scheduler has write access to the part of memory which holds the GDT then it can simply keep TSSs for each task, structured as the CPU wants them, and then it just changes the applications TSS selector in the GDT, and performs the task switch.

The *service calls* have a TSS each, and every system service may implement one or more service calls. A service call is what in other OSes is called a *system call*, and more than 8000 possible service calls should be more than enough for any OS. Especially an OS like LINK which uses low-level abstraction over hardware and thus wont need service

[1] It is assumed throughout this thesis that the CPU is 32 bit and running in what Intel calls *legacy mode*.

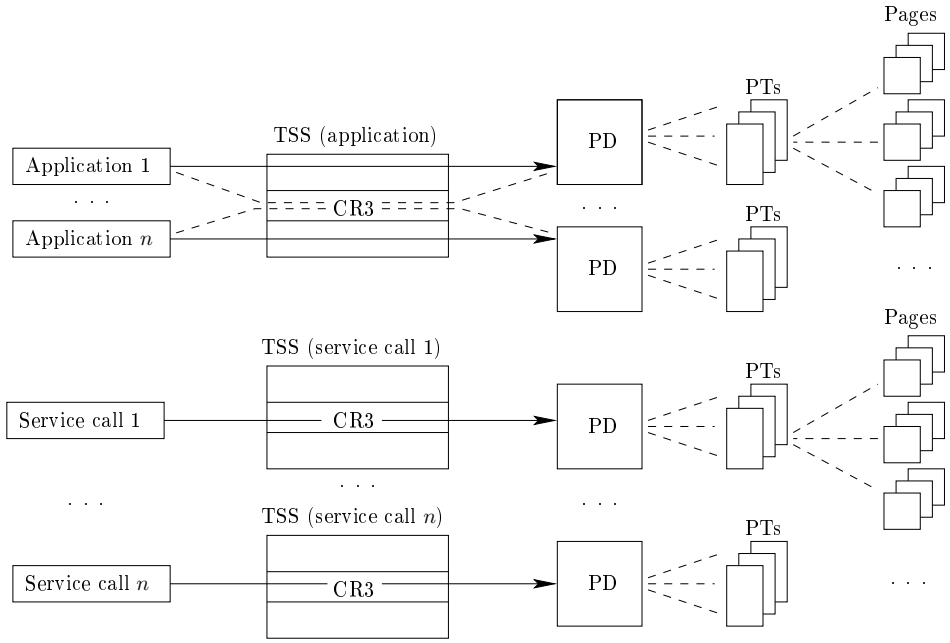


Figure 5: LINK memory organisation

calls for interfaces to higher level abstractions. In fact, there will probably be very few service calls in total, and as an optimisation, heavily used applications (e.g. libOSes) could get their own TSS to make switching to them faster. The entries in the GDT function as nothing more than entry points into code somewhere in memory, and by calling such an entry point the hardware makes sure to switch to the appropriate context before continuing.

What makes the LINK architecture possible is the fact that everything is just data structures in memory. The CPU may require some specific privilege level before a task can use a certain machine code instruction but most of the work that needs to be done by system services is no more than the manipulation of data structures in memory. The key to making it all work is simply to set up the memory in such a manner that each system service has access to the appropriate data structures in memory, and without giving it access to any more than it needs.

Since every task is running with privilege level 3, and every entry in the GDT requires privilege level 3 to call, every task can perform a hardware task switch to any other task. This means that if for instance an application calls a system service, and that system service needs to call another service as the last thing it does, it could just ask that second system service to return directly to the application. Applications which communicate a lot with each other could also return a reply by doing the task switch themselves to the previously running task, without having to switch to the task scheduler. This trick would however require one more entry in the GDT since only one TSS is used for all applications. Using two entries would allow an application to switch to another

application using the task scheduler and then the called application could switch directly back to the first application. This saves the applications one call to the task scheduler and could give a significant performance gain if two applications communicate a lot. The amount of communication would however have to be enough to outweigh the extra cycles used to manage the extra application GDT entry.

The architecture we have just described is built on the assumption that hardware task switching is fast. If hardware task switching is slow, then it would be senseless to use it. This is in fact the case as benchmarks have shown which will be described in Section 5.1. The result means that the architecture has to be modified, and a software task switcher needs to be added.

4.2. New LINK Architecture

An overview of the new LINK architecture can be seen on Figure 6. Any application (including system services) can call the Software Task Switcher (STS) by using the SYSENTER instruction. SYSENTER ("Fast System Call") does a fast switch of the CPU state into privilege level 0, and its companion SYSEXIT ("Fast Return from Fast System Call") returns from privilege level 0 back into the context from where SYSENTER was called[2]. Using SYSENTER/SYSEXIT is a faster way to perform context switches than forcing a context switch using a software generated interrupt — which were the original method used by IA-32 OSes to implement system calls; and SYSENTER/SYSEXIT is also faster than using hardware task switching. One reason for SYSENTER/SYSEXIT being so fast is that the instructions do not save or restore any context state (not even the return EIP is saved). This is a huge advantage over hardware task switching (HTS) as the HTS might have saved some unneeded state, and thus wasted CPU cycles. The task switcher should do as little work as possible and leave the saving of task state up to applications, or more likely, libOSes.

As can be seen on Figure 6, events generated by hardware call directly up to system services that enforce protection upon it, so this is the same as in the old architecture. Since HTS is no longer used we wont need a TSS for every application and system service and we wont be needing to keep entries for all the TSSs in the GDT. So we are no longer restricted to only having a total of approximate 8000 TSSs. The STS has its own data structures and more and more applications can be added to the system as long as there are memory available. The memory organisation is however still nearly the same as previously explained and showed on Figure 5, except that the TSS data structures are defined by the task scheduler system services, and applications has a data structure each just like service calls.

The STS can do all sorts of clever optimisations since the actions performed by the SYSENTER/SYSEXIT duo are fully dependant on values in some of the CPUs registers (see [intd] for details). So when SYSENTER is called and STS is invoked, the STS loads values of the new task to run into the specific registers and calls SYSEXIT. Now the CPU switches to the new task and its context. More details about the intricacies of the

[2] SYSEXIT can actualy return to another context instead if setup to do so.

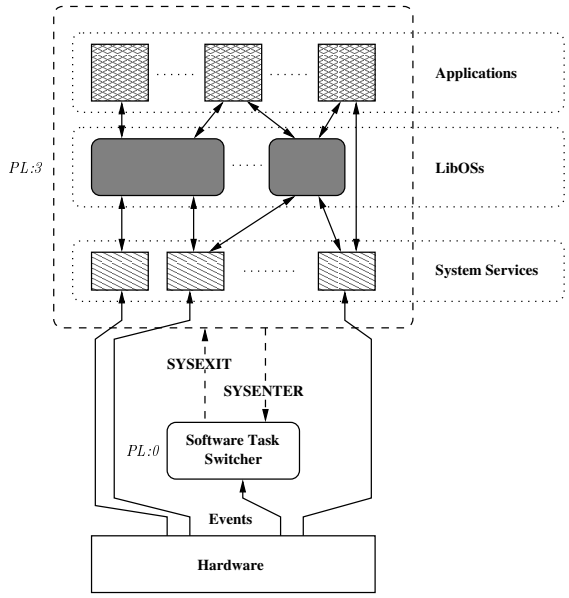


Figure 6: The new LINK architecture

STS will be given in Section 5.3 where the implementation of a proof of concept LINK OS is explained.

Input/Output The IA-32 architecture has 256 available I/O ports. Whether or not a task can access an I/O port depends on either its I/O Privilege Level (IOPL) or if it has an I/O Bitmap loaded, and whether the bit for the specific port in the bitmap is set or cleared. The use I/O Bitmaps is a simple way to control I/O on a per task basis. If every task gets an IOPL equal to 3 it will be the I/O Bitmap supplied for each task which decides whether or not access to a specific port is granted. Creation of I/O Bitmaps could be done as part of a standard task creation procedure.

In the case of memory mapped I/O, access control simply becomes a memory management issue. One elegant solution to this would be to have a system service for handling all memory mapped I/O and when a task requests access to memory mapped I/O then the system service would decide whether or not access should be granted, and if granted it could tell the memory manager to map the specific page into the tasks address space. That way protection is done on time of authorisation and not on time of access (i.e. using secure bindings).

Interrupt and Exception Handling All interrupts and exceptions are handled via the Interrupt Vector Table (IVT). In short, the table entries specify the address of Interrupt Service Routines (ISRs). The Advanced Programmable Interrupts Controller (APIC) also generates interrupts when either the Local or External APIC devices needs the CPU to handle their events. An obvious way to manage all these kinds of events would be

to create an event manager system service. This system service would have access to the IVT and APIC data such as the Local Vector Table (LVT) and could then manage ISRs in a similar way to how memory mapped I/O could be handled. As an ISR is automatically used from the moment it is inserted into a vector table, controlling access like this again means using secure bindings.

Portability With the new LINK architecture there is no reason why the architecture cannot be applied to other hardware platforms than the IA-32 architure. The only requirement LINK has is that the hardware platform has some sort of flexible memory protection mechanism like the virtual memory mechanisms the IA-32 architecture has. Since virtual memory is the common way to do memory protection today, LINK versions can be build for many different hardware platforms.

5. Implementation

Two systems have been implemented as part of the work for this thesis. First a benchmark tool was developed that benchmarks the amount of cycles needed to perform context switches using different mechanisms. Later a proof-of-concept version of LINK was implemented, called POCLINK (Proof-Of-Concept LINK). We will start out in this section by explaining the benchmark tool and then look at the results it gave. Then we will look at POCLINK in detail. Both systems are mainly written in assembler for the Net-wide Assembler (NASM) and expects to be run in an preboot execution environment (PXE) [Cor99] which is available on most PCs with modern ethernet adapters. Instructions on how to set up a test environment can be found in Appendix B. Source code for both the benchmark tool and POCLINK can downloaded at `http://www.cs.aau.dk/~zion1459`. The source code is available in a single gzipped tar-file. The file contains two directories, `microbenchmark` and `poclink`, where `microbenchmark` contains the benchmark tool source code and `poclink` obviously contains the POCLINK source code. To compile the code the NASM assembler, GCC compiler, and utility Make must all be available. Compiling is then a simple matter of entering either the `microbenchmark` or `poclink` directory and running the command `make`. For further information about compilation refer to the Makefiles themselves which are purposely overly simplistic. The compiled binaries are so-called Network Boot Programs (NPBs), and should function in any PXE 2.0 environment[Cor99].

5.1. The Benchmark Tool

The performance of an operating system is highly dependant on the time it takes to perform a context switch. Thus, in the case of LINK, the performance of the task switcher is very important. Originally LINK was designed to use the hardware task switcher (HTS) as it was an elegant way to perform context switches, but before definitively choosing to use a HTS, a software task switchter (STS) were build and benchmarks of both were made. We wrote a benchmark "tool" in assembler code for the Net-wide Assembler (NASM). The benchmarks the tool can perform are so-called *micro-benchmarks*

as they benchmark low-level functionality. Even though micro-benchmarks of an OS are good, the overall performance can still be bad due to other factors. Nevertheless, micro-benchmarks are useful as they illustrate the highest possible performance of a certain mechanism and in our case where there is a choice to make between using a HTS or STS, the benchmarks are essential.

The tool we wrote is nothing more than a single source file named `microbenchmark.asm` where macros define the kind of benchmark the assembled binary will perform. The tool can make four types of benchmarks:

- Hardware task switching.
- SYSENTER/SYSEXIT.
- Software task switching with no task state saved.
- Software task switching with with state saved.

Each of these types of benchmarks can be performed with and without paging enabled and with various combinations of cache settings.

When the binary runs of the target machine it performs 100 context switches and prints the number of CPU cycles it took to perform each switch to the screen. After the 100 numbers it prints an empty line and after that it prints the average number of cycles used (this saves the tester from the trouble of reading the hundred numbers of the screen and performing that calculation himself). To do more than a 100 benchmarks can easily be done by making a small modification to the assembler code.

Note that it is the number of cycles used to switch from one user space task to another that is benchmarked, which means that in the case of the STS it is the total amount of cycles used to enter privilege level 0, save the old task state, load the new task state, and switch back to privilege level 3 into the new tasks context. The SYSENTER/SYSEXIT benchmarks gives the number of cycles used to enter and exit privilege level 0 using those instructions, but without performing any work while in privilege level 0. This simply benchmarks the pure SYSENTER/SYSEXIT mechanism and does not perform a context switch from one user space task to another.

The file `microbenchmark.asm` only contains 527 lines of assembler code, including comments. The file can however still seem confusing at first glance due to the use of preprocessor macros. The use of macros did however make it easy to develop the different types of benchmarks in a single source file and also makes it easy to quickly compile different kinds of benchmark programs. To make the assembler easier to read, set the macro values to your liking and then invoke NASM with the "-e" parameter which causes it to preprocess the file but not assemble it (refer to the NASM manual for details). The result will be the assembler code that is actually assembled, which should be self-explanatory even without comments (which is also removed by the preprocessor).

The benchmarks are performed in protected mode at privilige level 3. When paging is enable a single page directory is used, and a single page table containing the needed pages. To keep things simple, all pages are identity-mapped, meaning their physical

and virtual addresses are the same. Both tasks use the same page directory, but `CR3` is nonetheless reloaded when the STS switches tasks since an actual OS most likely would load a new page directory for each task.

5.2. Benchmark Results

We assembled `microbenchmark.asm` for each of the possible valid macro value variations and ran all the binaries on an old AMD Athlon 700MHz PC with 384MB RAM. A complete set of benchmark results can be found in Appendix A.

We will in this section only compare the average number of cycles used for each of the four types of benchmarks with paging enabled and with the parameters set to the values that they would use in a typical running system. These results shows us if it is faster or slower to use a STS instead of a HTS. The results are shown in Table 1.

Mechanism	CR0.CD	CR0.NW	PWT	PCD	Cycles
HTS	0	0	0	0	483
STS, w. state	0	0	0	0	490
STS, no state	0	0	0	0	280
S.ENTER/EXIT	0	0	0	0	134

Table 1: Benchmark comparison of different mechanisms for performing context switches.

The four parameters mean the following:

CR0.CD and CR0.NW: These two parameters represent a bit each in the CR0 control register. When both the Cache Disable (CD) and Not Write-through (NW) are clear (i.e. 0), caching of memory locations for the whole of physical memory in the CPUs internal and external caches is enabled.

PWT and PCD: The two parameters represent control bits in page directory and page table entries of the currently active pages. When the Page-level Write-Through (PWT) bit is clear write-back caching is enabled for the associated page or page table.

When the Page-level Cache Disable (PCD) bit is clear the associated page or page table can be cached.

These parameter settings are most like the settings that a typical system would use as caching improves performance greatly. However, on multi-CPU systems caching might be disabled at some points to ensure cache-coherence, but even with other parameter settings the relative difference between the four types of benchmarks remain the same, as can be seen on the results in Appendix A.

The results in Table 1 show us that our own STS, which saves the same amount of task state as HTS, only use 7 cycles more to do so. This is a negligible difference. But when the STS does not save any task state except the state needed to perform the switch it only use 280 cycles. This is something the HTS has no possibility of doing, and this

makes the STS the better choice. Sometimes we might want to switch to another task, but we might not need to save all the previous tasks state. Maybe we know it has not changed, or maybe the changes are unimportant. Using a STS we can greatly optimise task switching in some cases. Even if we never use the STS without saving the entire task state, the STS has the advantage that it can be used on other platforms which may not have a task switching mechanism implemented in hardware. Also, in Section 5.3 it is shown that the POCLINK STS in fact never needs to save as much task state as the HTS.

5.3. Proof of Concept LINK (POCLINK)

To prove that the LINK architecture is possible to use in practice we implemented a small proof of concept OS called POCLINK (Proof of Concept LINK). When compiled POCLINK is a single binary which like the benchmark tool can be sent to a client computer via the network. The original idea was that the binary sent via the network would initialise the system and put it in a state where it could use the PXE API to download the rest of the operating system as a set of ELF binaries. Unfortunately, there was a bug in either QEMU or Etherboot which prevented us from using the PXE API (refer to Appendix B for an explanation of the test environment used). The problem we encountered was that the API calls never returned, but the system did not crash either. We could call into the API but then the machine would just hang. The QEMU monitor allowed us to see the value of the EIP (Instruction Pointer), and thus verify that we had entered into the code supplied as part of the PXE environment. After double-checking several times with the PXE specification and trying various setups, we concluded that it must be a bug, and continued development without the PXE API. We will further test QEMU and Etherboot in the near future to locate the bug and file a bug-report. Without the API we would have to implement the network driver and protocols ourselves, which we did not have the time to do, so instead we "cheated". The PXE environment allows the network boot program (NBP) to have a size up to 64Kb. This is enough for the entire POCLINK system so we simply include everything in the NBP. NASM allows us to do this easily with the instruction `incbin`. This means that running POCLINK is done in exactly the same way as running the benchmark tool.

POCLINK consists of the following:

- A bootstrapper
- A software task switcher
- A task scheduler
- Three small applications
- A tiny shared library

The bootstrapper is the actual NBP and the task switcher, scheduler, applications, and shared library are included in that binary as described above. The set of pages

and page tables are static, meaning that all memory needed must be allocated during bootstrapping. The bootstrapper sets up a page directory along with page tables for the entire systems memory, with all pages identity-mapped. Had there been a memory manager this would be the page directory it would use, which would allow it to access all memory, and thus manage it, even when running at privilege level 3.

The software task switcher is the only task after bootstrapping is complete that runs at privilege level 0, as it should be. POCLINK has been developed in QEMU and runs perfectly, it has also been successfully tested on a Fujitsu Siemens laptop with a Core Duo 1,6 GHz CPU and 2GB of RAM. Only one of the CPU's cores are used by POCLINK. The reason why POCLINK has been tested on a different machine that the micro benchmarks were performed on is because our original machine for testing died some time between the development of the benchmark tool and POCLINK.

A tree list of all the source code files of POCLINK, excluding the Makefiles, can be seen in Figure 7. The files listed in Figure 7 consists of just 2498 lines of code total, including comments.

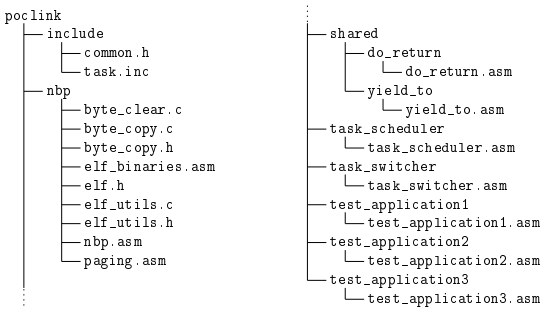


Figure 7: POCLINK source code file tree.

Here follows a short description of each file's purpose:

`include/common.h`: C header file containing a few type definitions.

`include/task.inc`: Assembler "header" file, therefore the `.inc` postfix on the name instead of the typical `.asm` postfix. The file contains the definition of the structure for containing a task's state.

`nbp/byte_clear.c`: Contains the funtion `byte_clear` which given a 32 bit memory address and `size` in bytes clears `size` bytes starting for the specified address.

`nbp/byte_copy.c`: Contains the function `byte_copy` which given a source address, destination address, and `size` in bytes, copies `size` bytes from the source address to the destination address.

`nbp/byte_copy.h`: Contains a few preprocessor definitions.

`nbp/elf_binaries.asm`: Includes all the ELF binaries using the `incbin` NASM instruction.

`nbp/elf_utils.c`: Contains functions used for relocating the text and data sections of ELF binaries.

`nbp/elf_utils.h`: Contains various type and structure definitions and preprocessor constants all obtained from the Tool Interface Standard (TIS) Portable Formats [Com93] specification of the Executable Linkable Format.

`nbp/nbp.asm`: Contains the bootstrapping code.

`nbp/paging.asm`: Contains the code setting up page directories and page tables for the task switcher and the three test applications.

`shared/do_return.asm`: Contains the routine `do_return` which is called by the task switcher on exit and loads the newly scheduled task's unprivileged state before returning control to the task itself.

`shared/yield_to.asm`: Contains the routine `yield_to` which given an index into the Task Table yields the quantum to the task which has its entry at the specified location in the Task Table. This function is only used once in `nbp.asm` for switching to the first task and thus starting the system. The Task Table is the data structure in memory containing the state of all tasks in the system.

`task_scheduler/task_scheduler.asm`: Contains the scheduling algorithm which is a simply round-robin algorithm that traverses through all tasks in the Task Table.

`task_switcher/task_switcher.asm`: Contains the task switching mechanism which get run when the SYSENTER instruction is executed.

`test_application1/test_application1.asm`: The three test applications are identical except for their names. They print to the screen the APIC Timer counting down and their own Task Table index.

The entry point is found in `nbp.asm` at the label `_start`. This is where execution starts in the compiled binary `poclink.bin`. The first thing that happens when execution starts is that the PXE environment is detected. If PXE 2.0 or later is found execution continues and switches the processor into protected mode. Bootstrapping the processor is done by setting up the following data structures:

- Global Descriptor Table (GDT)
- Local Descriptor Table (LDT)
- Task State Segment (TSS)
- Interrupt Vector Table (IVT)

The GDT is set up with six descriptors:

code32dsc_dpl0: Code segment descriptor with Descriptor Privilege Level (DPL) 0. It spans the entire 4GB address space. This is only use by the task switcher.

data32dsc_dpl0: Data segment descriptor with DPL 0. It spans the entire 4GB address space. This is only use by the task switcher.

code32dsc: Code segment descriptor with DPL 3. It spans the entire 4GB address space.

data32dsc: Data segment descriptor with DPL 3. It spans the entire 4GB address space.

ldtdsc: LDT descriptor. The LDT is not used but it must nonetheless be specified and loaded. It is however just loaded with a base address and size of 0, since it never is used. This descriptor has DPL 3.

tssdsc: Even though hardware task switching will not be used a single TSS must be specified, this is the descriptor for that TSS. This descriptor has DPL 3.

The six descriptors are located in the order they are listed, with `code32dsc_dpl0` having GDT index 1. Index 0 is referred to as the *null descriptor* and cannot be used when referencing a descriptor. Entries 1 through 4 must be located in the relative order shown above, as the SYSENTER/SYSEXIT instructions calculate the addresses of the rest of the descriptors from the address of the `code32dsc_dpl0`.

Before interrupts can be enabled the IVT must be set up. The first 20 interrupt vectors are specified by Intel (with the exception of vector 1 and 15 which are marked reserved). These vectors are all set up with a Interrupt Gate pointing to the location of the `dummy_isr` routine. `dummy_isr` is a small routine which simply re-enables interrupts and returns from the interrupt. That is, none of these interrupts are used for anything in POCLINK. The only interrupt that is used is the one sent to interrupt vector 42. This is where the Advanced Programmable Interrupt Controller (APIC) Timer is set up to deliver its interrupts. The Interrupt Gate for vector 42 points to the entry point of the task scheduler, meaning that every time the APIC timer generates an interrupt the task scheduler is invoked.

The Task Table structure is not part of the Intel specification but a structure we created to keep the data of the tasks in the system. Each entry in the Task Table is 44 bytes long and contains the following:

- Instruction Pointer (EIP).
- Stack Pointer and Stack Base Pointer (ESP and EBP)
- Page Directory Base Address (CR3)
- EFLAGS
- General purpose registers: ESI, EDI, EAX, EBX, ECX, and EDX.

As all tasks use the same code and data segments there is no need to save the segment registers. The following events take place during a task switch:

1. The APIC Timer generates an interrupt and the task scheduler is invoked.

2. The task scheduler saves the interrupted tasks state in a special memory page and invokes the task switcher using the SYSENTER instruction. Before invoking the task switcher it locates the next task that should have a quantum and passes its Task Table index along as a paramenter to the task switcher.

3. The task switcher saves the task info found in the special memory page into the task's entry in the Task Table, loads the next task's state, and then executes the SYSEXIT command.

4. The SYSEXIT command does not return directly to next task but to a routine called `do_return` which loads the unprivileged parts of the new task's state (general purpose registers, etc.) and then returns to the actual new task.

The reason why the task's state is temporarily saved in a predefined memory location is due to the task scheduler running at the same privilege level as the tasks. This means that when an interrupt occurs it has access to the same memory pages as the interrupt task. It could just push the values onto the stack but that would mean the task switcher would have to know the address of each task's stack. This value can of course be passed along as a parameter, but before this address can be used by the task switcher it would need to get that page mapped into its address space. A task cannot manipulate the task switching routine in any way by polluting that special memory location, the only thing that can occur if it did that is that it most likely would crash itself next time it got its quantum. To prevent a memory leak the temporary data should be cleared by the task switcher before returning to the next task. This is not done since security is no issue in POCLINK and we prefer only to include the most essential code.

We cheated a bit when it came to reading ELF binaries. Since implementing full support for ELF binaries would take more time than we had available for implementation we have only implemented the very basics of ELF support. The bootstrapping code in POCLINK can detect whether a block of data in memory is an ELF binary or not, and if so it can relocate its data and text section to specified locations. However, since no dynamic linking has been implemented the final address of the text and data sections must be specified when the ELF is originally compiled. For instance, if one looks in the Makefile in the `test_application1` directory one will see the following line:

```
ld -o test_app1 -Ttext $(TEXT_OFFSET) -Tdata $(DATA_OFFSET) $(OBJECTS)
```

Which sets the text and data offset in the file to values specified above in that file. Also, in order for the relocated text section to be referred to as the entry point for a binary, we made sure the label `_start` is located at the very beginning of the text section. Had any of the ELF binaries contained more than one entry point, like for instance a shared ELF object might, this hack would not work, but in our small implementation this was not needed so no problem arose. These two small hacks made it possible to use ELF binaries for POCLINK, as a proper LINK OS would do, but with minimal effort and time spent on implementing something which does not have anything directly to do with the architecture.

The task switcher and each of the three test applications have their own page directory. They only have access to the most essential. There is, however, one page that the applications have access to which they should not have access to: The page containing the Task Table. Each task should have read access to its own task state contained in the Task Table but by giving them read access to the page containing the Task Table they obviously can also read the state of all other tasks on the system. This situation does, however, have a simple fix: Each task should have a read-only memory page containing its own task state (and perhaps other useful read-only data as well) and the Task Table should then contain a list of addresses instead of the actual task state data structures. Basically, this is the same thing which is done in the GDT with the TSS descriptors. The reason why this has not been done in POCLINK is that it would require a substantial change in the code (one of the troubles when programming in assembler) and due to lack of time this change was not made. However, it is obvious that this issue can be resolved used very simple measures.

5.4. Summary

Task switching has been benchmarked and we found that the software task switcher and hardware task switcher have nearly the same performances when all task state is saved. But when only some of the task state needs to be saved the software task switcher is the better choice. As our proof of concept implementation POCLINK has shown we do not need to save all the task state. Hardware task switching takes into account that different tasks use different segments but the LINK architecture uses the same segments for all tasks (with the exception of the task switcher) and thus no segment information needs to be saved when switching tasks. As this will improve the software task switchers performance it is clearly a better choice than the hardware task switcher.

Our implementation of POCLINK has shown that the LINK architecture can be applied in practice and that it can be done using very little code and with very little complexity. POCLINK is of course only a proof of concept implementation but the essentials of a LINK operating system are there. With POCLINK as a reference an actual LINK OS can be build one system service at a time: Memory manager, device manager, security manager, and so on. Implementing an OS in such a maner removes a lot of complexity. Even though POCLINK is mainly implemented in assembler a lot of it could be implemented in C, with an occasional couple of lines of inline assembler code. The bootstrapping code is however needed to be implemented in assembler. Each component can be unit tested and their small size makes it easy to create formal models of them for making sure that concurrent execution does not cause errors.

POCLINK is only a proof of concept implementation and as such it does not try to stay true to all the LINK principles only enough to prove the architectur's feasibility. For instance, POCLINK does not use an explicit revocation policy, but a complete LINK OS should.

6. Security Model

LINK has mechanisms for supporting fine-grained protection of resources but these mechanisms are no good without security policies defining when to use those mechanisms. That is, we need to define what secure means in an operating system context. The problems with security policies is that they need to be tailored to the specific needs of each system. For instance, an average computer user is often both administrator and user on his own computer systems and does not care that much about high local security as he is the sole user of the system. On the flip side there are computers used at military facilities where strict security policies has to be followed[Lan81]. Therefore, creating a fixed and unchangeable security policy for an OS is out of the question. Instead OSes support one ore more security models which allows administrators to define policies using those models.

The problem in case of LINK has now become creating a security model that is flexible enough to enforce fine-grained protection of resources and at the same time be simple and elegant enough to be useful in practice. But why think so much about what model to use? Why not just use the user-group model from the Unix world? And perhaps extend it with access control lists (ACLs) on files, which modern file systems already support... Because the protection model is too coarse-grained,

Even when the user-group model is extended with ACLs on files, it still only has users, groups, and files, not to mention it works on a to high level of abstraction. Also, there are a lot of problems with the user-group model which have been known for a long time[MK97], and many new security models have been proposed since then. One attempt to make an OS with tight security is the Security-Enhanced Linux (SELinux) project[selb]. With SELinux the administrators can choose from various kinds of security models and use the one they find appropriate, or create their own. SELinux thus adds a finer granularity to Linux security but has been criticized for being to complex to use[sela, NSRL06]. However, SELinux is a step in the right direction. There is a security model in SELinux which apparently is flexible enough to simulate other security models on top of it. The different models SELinux supports are often called policies, but that is to diminish the true power of them. SELinux supports, amongst others, role-based access control [SCFY96, TJ98] policies, but the role-based access control is in fact an entire security model in itself.

When talking about security models there is often talk about either *Discretionary Access Control* (DAC) models or *Mandatory Access Control* (MAC) models. Both terms have been defined by the US Department of Defense as follows[oD85]:

Discretionary Access Control: A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control).

Mandatory Access Control: A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects

and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity.

It is quite common to combine both DAC and MAC in one security model. Security models have in many decades been categorised as either MAC or DAC models, or a combination of the two. However, role-based access control models are actually neither MAC nor DAC models, but an independent component of access control coexisting with MAC and DAC[SCFY96]. So security models is not just a question of using DAC, MAC or both.

The LINK security model is build around *hierarchically named capabilities* (HNCs), which were also used for the MIT exokernel Xok as proposed by Mazières[MK97]. This model uses both *capability lists* (CLs) and *access control lists* (ACLs). Each *principal* (e.g. users, or processes) has a CL containing its capabilities and each *resource* (e.g. files or a block of memory) has an ACL containing a list of capabilities which has access to it and which specifies what kind of access the capabilities have. Using HNCs gives us a discretionary access control model but we actually have the ability to perform mandatory access control as well. This will become apparent later when we do a formal analysis of the security model in Sections 6.1 to 6.4.

A HNC has a name and a set of access permissions. If a HNC A's name is the prefix of another HNC B's name, then A is said to *dominate* B, meaning that A has all the rights B has (through B). This is the key advantage of HNCs. The hierarchy enables us to ensure the *principle of least privilege*. A principal should never have more privilege than it needs and when a principal dominates another and uses the dominated principal's capabilities, it is those privileges that are used, and not its own (which dominates the others). That is, the least privileged capability is always used.

The HNC structure on Figure 8, which is the structure as proposed by Mazières, has a name up to 7 bytes long and 1 byte is used for access/control settings.

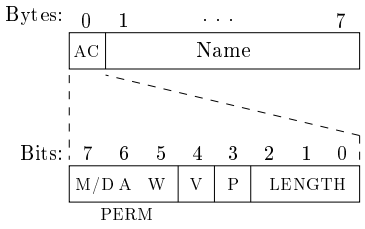


Figure 8: Hierarchically named capability structure, as proposed by Mazieres.

The access/control bits do the following:

`M/D`: Modify/duplicate permission, depending on whether the HNC is in a ACL or a CL. If in an ACL it means the current HNC gives *modify* capabilities to the ACL; if the HNC is in a CL it means that owner of CL are permitted to *duplicate* this HNC (meaning the capability can be copied along to others).

`A`: Allocate resource.

`W`: Write permission.

`V`: Valid bit. Makes it easy to temporarily enable/disable a capability in a CL or ACL.

`P`: Pointer bit for extended ACLs[MK97].

`LENGTH`: The length of HNC's name.

Mazières explains the usage of HNCs in the exokernel as follows[MK97]:

> The kernel maintains a list of capabilities owned by each process, and an access control list (ACL) of capabilities allowed access to each object in the system. When a process requests access to a particular resource, it must explicitly specify which of its capabilities it intends to gain access with. The kernel then traverses the resource's ACL. If it finds an ACL entry that either matches or is dominated by the designated capability, and if the appropriate permission bits are set both in the process' capability and in the access control list entry, then the request is granted. Otherwise, it is rejected.
>
> Applications can create or forge new capabilities at will, but a new capability must be dominated by an existing one...

One thing to notice, and keep in mind, about HNCs is that they function differently whether they are in a CL or ACL. Another thing about Mazières' HNC structure is that it has no permission bit for read rights. This means that when you have a capability for something you automatically have read access to it. This is a problem if one wishes to model other security models on top of HNCs; models such as the Unix user-group security model. In Unix systems it is possible to have *write-only* access to files, something which Mazières capabilities cannot express. Specifying the length of the name is needed as the value 0 in this context is the empty string. Thus if the name is more than 7 bytes long, `LENGTH` can be set to 0 to indicate that, and the system can look up the next 8 bytes of the name.

To remedy the lack of write-only access, LINK uses a modified version of Mazières' HNCs. The pointer bit has been dropped and is used to indicate read access instead, as we could not find any good reason for having the pointer bit in the HNCs name. The LINK HNC structure can be seen on Figure 9.

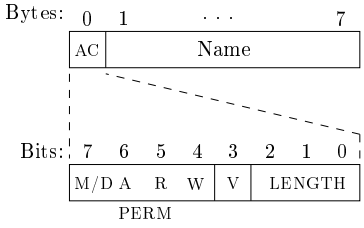


Figure 9: Hierarchically named capability structure, as used in LINK.

Having defined the HNC structure is not enough. Building a security model around HNCs seems like a good choice for an OS applying fine-grained protection to resources, but is it really? To get a better understanding of HNCs we have created a formal model to express the semantics of access control rules within a HNC system. This model also provided insight into information flow channels in a system using HNCs; insight that allows us to create security policies concerning information flow as well.

Finding a formalism for describing security models was not as trivial as first expected as many different models have been proposed during the last three decades[Lan81, McL94]. The main reason for even wanting to make such a formal model is that when working with formalisms all minor details about the model will come to light. The model will allow us to ask questions such as: "If we have a LINK system with these LibOSes on them, and these and these processes running, can application A then ever get direct write access to resource R?". We need to be able to give concrete answers to such questions as it time and time again has been shown that a small misconfiguration in for instance access permissions to files can cause unexpected security holes. Also, having a formalism we might use that formalism to model other security models and for instance compare their expressive power.

The first idea that came to mind was to use formal language theory[Sip06], due to the obvious approach of using pattern matching on capability names to reason about capability hierarchies (if one name is a prefix of another then that name dominates the other, thus capturing the hierarchy of the capabilities). However, it ended up being the only part of that idea which trivially made sense under such an abstraction; everything else simply became too cumbersome. After that we turned towards transition systems, more specifically different variations of the π-calculus[MR00, Lho04], in hopes of finding an elegant solution in an already existing formalism. But to actually express what we wanted we had to create a type system for the calculus (many already exists but the type system is what actually describes the security model in such a system so one for HNCs would be needed). So it was not the calculus we wanted at all but a type system to reason about. We therefore turned away from the calculus and looked directly at proof theory, since type systems are just proof systems in a certain context. This however brought us a bit away from the actual reality of the questions we wanted to ask. We want know what can happen in a system given the use of a certain type system (our HNC based security model). So this led us in the direction graph theory and we suddenly ended up at some articles from the late seventies about the *Take-Grant model* and how it can be formally analysed using graph theory[LS77, BS79]. The idea is to model the security model in a graph, called a *protection graph*, with a set of different types of vertices, and a set of labels to be added to the oriented edges between vertices. The security model would essentially be expressed as a set of graph rewriting rules (which in reality also is a form of proof system).

6.1. Hierarchical Protection Graphs

Our idea of using protection graphs comes mainly from the article by Matt Bishop and Lawrence Snyder about transfer of information and authority in the Take-Grant

model[BS79]. We have, however, extended their definition of a protection graph with capabilities, in order to capture the hierarchy between HNCs.

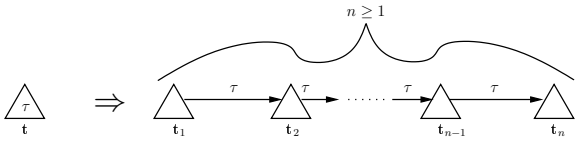
Definition 6.1 (Hierachical Protection Graph (HPG)) A *hierarchical protection graph* is a finite directed loop-free graph with three types of vertices: Principals ($\circ$), resources ($\bullet$), and capabilities ($\triangle$).

Edges from one capability to another are labelled with the symbol τ. All other edges are labelled with a set of symbols α, where $\alpha \subseteq \{m, r, w, a\}, \alpha \neq \emptyset$. □

The labels correspond to the permission bits in the HNC structure. `M/D` we label using m. Two shorthand notations will be used throughout our models.

Definition 6.2 (Principal or resource) The symbol $\otimes$ represents either a principal or a resource. □

Definition 6.3 (τ-path) A *τ-path* is a chain of one ore more capabilities. Graphically,



The left-most (or first) capability in the τ-path is called the *initial* capability of the τ-path and the right-most (or last) capability in the τ-path is called the *terminal* capability of the τ-path. □

Our formalism has principals which are active entities (e.g. users or applications), and resources which are passive (e.g. files or a block of memory). Capabilities are the middle-men between principals and resources. If a principal has a $\{r\}$-labelled edge to a capability, and there are other $\{r\}$-labelled edges from that capability to a set of resources, then that principal has *read access* to those resource. The $\{\tau\}$-labelled edges between capabilities shows the hierarchy amongst capabilities.

With our formalism we wish to reason about things like when a principal can aquire a certain access permission to a certian resource. There are in general two ways a principal can aquire some "right" to a resource:

De jure aquisition which means that the access right is transferred to the principal.

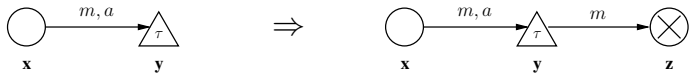
De facto aquisition which means that the principal gets the information without getting direct authority to access it.

De jure acquisitions can be seen as questions about access control, and de facto aquisitions can be seen as questions about information flow. A security model generally describes de jure rules, that is, how access rights can propagate through the system. We will now define a set of de jure rewriting rules which represents the functionality of hierarchically named capabilities as used in LINK. In Section 6.3 we define a set of de facto rewriting rules which allow us to reason about information flow as well.

6.2. De Jure Rules

We take a couple of liberties in the following graphical representations of the rewriting rules. Instead of labeling an edge $\{m, a\}$ we simply write m, a over the edge and when we write m, α where α is a subset of a set of labels, the label really should say $\{m\} \cup \alpha$. We now define the de jure rewriting rules.

CreateR/P: Let $\mathbf{x}$ be a distinct principal and $\mathbf{y}$ a distinct τ-path in a HPG G. Let there be an edge from $\mathbf{x}$ to the initial capability of $\mathbf{y}$ labelled with $\{m, a\}$. *CreateR/P* defines a new graph G' by adding a new principal or resource $\mathbf{z}$ and an edge from the terminal capability of $\mathbf{y}$ to $\mathbf{z}$ labelled $\{m\}$. Graphically,

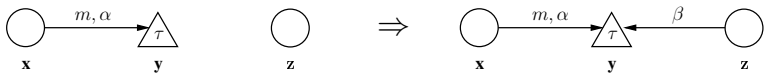


Informally: Any principal which has a capability granting it allocate rights can use that right to allocate either a new resource or principal. The newly created resource/principal will be managed by the capability which was used to create it.

CreateCap: Let $\mathbf{x}$ be a principal and $\mathbf{y}$ a τ-path in a HPG G and let there be an edge from $\mathbf{x}$ to the initial capability of $\mathbf{y}$ labelled α, where $\alpha \subseteq \{m, a, r, w\}, \alpha \neq \emptyset$. *CreateCap* defines a new graph G' by adding a new capability $\mathbf{z}$ and an edge from the terminal capability of $\mathbf{y}$ to $\mathbf{z}$ labelled $\{\tau\}$. Graphically,

Informally: Any principal can create a new capability which is dominated by another capability in their CL.

Duplicate: Let $\mathbf{x}$ and $\mathbf{z}$ be distinct principals, and $\mathbf{y}$ a distinct τ-path in a HPG G. Let there be an edge from $\mathbf{x}$ to the initial capability of $\mathbf{y}$ labelled $m \cup \alpha$, where $\alpha \subseteq \{r, w, a\}, \alpha \neq \emptyset$. *Duplicate* defines a new graph G' by adding an edge from $\mathbf{z}$ to the terminal capability of $\mathbf{y}$ labelled with β, where $\beta \subseteq \{m\} \cup \alpha, \beta \not\subseteq \{m\}$. Graphically,

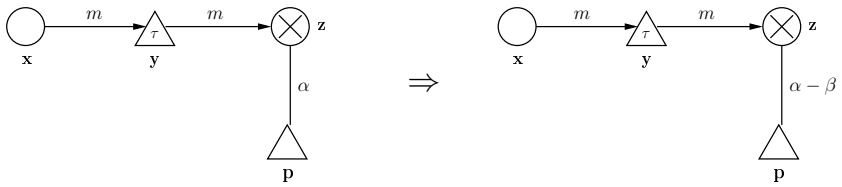


Informally: A principal which has duplicate rights to a capability can give that capability or a subset of it to other principals.

Add: Let $\mathbf{x}$ be a principal, $\mathbf{z}$ a principal or resource, $\mathbf{n}$ a capability, and $\mathbf{y}$ a τ-path in a HPG G. Let there be an edge from $\mathbf{x}$ to the initial capability of $\mathbf{y}$ and another from the terminal capability $\mathbf{y}$ to $\mathbf{z}$, both labelled $\{m\}$. *Add* defines a new graph G' by adding an edge from $\mathbf{n}$ to $\mathbf{z}$ labelled α, where $\alpha \subseteq \{m, r, w, a\}, \alpha \neq \emptyset$. Graphically,

Informally: A principal which has "modify ACL" rights to another principal or resource can add new capabilities to that principal's or resource's ACL.

RemoveOther: Let **x** be a principal, **z** a principal or resource, **p** a capability, and **y** a τ-path in a HPG G. Let there be $\{m\}$-labelled edges from **x** to the initial capability of **y** and from the terminal capability of **y** to **z** and let there be an edge labelled α from either **z** to **p**, or **p** to **z**, where $\alpha \subseteq \{m, r, w, a\}, \alpha \neq \emptyset$. *RemoveOther* defines a new graph G' by removing the set of labels β from α where $\beta \subseteq \alpha, \beta \neq \emptyset$. If $\alpha = \beta$ the edge between **z** and **p** is removed. Graphically,



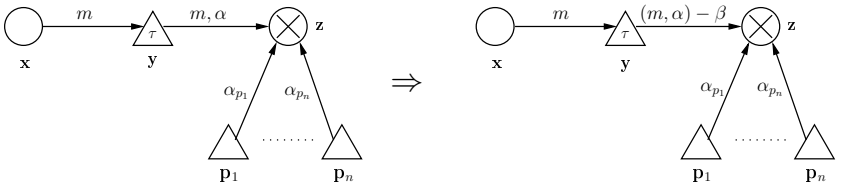
Note that the edge between **z** and **p** can be directed in either direction, and therefore the edge is drawn without direction.

Informally: A principal that has modify ACL rights to another principal or resource can remove any other capability from that principal's or resource's ACL.

RemoveSelf: Let **x** be a principal, **z** a principal or resource, $\mathbf{p}_1, \ldots, \mathbf{p}_n$ capabilities, and **y** a τ-path in a HPG G. Let there be an edge from **x** to the initial capability of **y** labelled $\{m\}$, an edge from the terminal capability of **y** to **z** labelled $\{m\} \cup \alpha$ where $\alpha \subseteq \{r, w, a\}, \alpha \neq \emptyset$, and edges from $\mathbf{p}_1, \ldots, \mathbf{p}_n$ to **z** labelled $\alpha_{p_1}, \ldots, \alpha_{p_n}$, respectively, where $\alpha_{p_1}, \ldots, \alpha_{p_n} \subseteq \{m, r, w, a\}, \alpha_{p_1}, \ldots, \alpha_{p_n} \neq \emptyset$. *RemoveSelf* defines a new graph G' by removing the set of labels $\beta \subseteq \{m\} \cup \alpha, \beta \neq \emptyset$ from the edge between the terminal capability of **y** and **z**, if one of the following three cases hold:

1. $m \notin \beta$.
2. $m \in \beta$ and $m \in \alpha_{p_1} \vee \ldots \vee m \in \alpha_{p_n}$.
3. $m \in \beta$ and the edge from the terminal capability of **y** to **z** is the only $\{m\}$-labelled incomming edge to **z**.

If **z** no longer has any incomming or outgoing edges the vertice is removed.



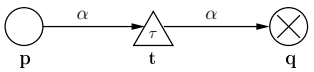
Informally: If a principal has the right to modify another principal's or resource's ACL then it can remove its own capabilities from the ACL. It can however only remove its right to modify the ACL if its capability is the only one in the ACL or if there is another capability in the ACL which has modify rights as well. If this side condition is left out then it would be possible to create a resource or principal that no one controls (and never again can control) which obviously would be a problem.

By studying the above rewrite rules one will notice that the symbol a (allocate) only has a function in labels on edges going to capabilities. If a principal has an edge labelled a going to a capability it means it can use that capability to allocate new resource and/or principals. Allocating a resource or principal in practice may mean many different things depending on the specific resource or principal. From an access control point of view we do not wish to concern ourselves with such implementation details and simply state that if a principal has allocate access with a certain capability then it can create new resources or principals which will be managed by it.

Also notice that given the above rules it holds that in any HPG the τ-paths form a tree. We call this tree the system's *capability tree*. That this holds is obviously true as capabilities can only be created using another capability and in doing so an $\{\tau\}$-labelled edge is added from the old capability to the new one. This is the only way a capability can be created, and thus results in capabilities being structured as a tree with the edges oriented from the root node towards the leafs of the tree.

Having the de jure rules we can now define a predicate for when two principals can share a resource. We define the predicate *can-share(α, **p**, **q**, G_0)* where $\alpha \in \{m, a, r, w\}$, **p** is a principal, **q** is a resource or principal, and G_0 is a protection graph containing **p** and **q**.

Definition 6.4 *can-share(α, **p**, **q**, G_0)* $\Leftrightarrow$ There exists a sequence of HPGs $G_1, \ldots, G_n$, $n \leq 0$ such that G_{i+1} follows from $G_i, 0 \leq i < n$ (henceforth written $G_0 \vdash^* G_n$) by one of the de jure rules, and in G_n there exists a τ-path **t** such that there is an α-labelled edge from **p** to the initial capability of **t** and another from the terminal capability of **t** to **q**. Graphically,



□

We now state when the predicate is true.

Theorem 6.1 *Let **p** be a distinct principal and **q** a distinct resource or principal in a HPG G and let $\alpha \in \{m, a, r, w\}$. The predicate can-share(α, **p**, **q**, G) is true if and only if one of the following hold:*

1. *There exists a τ-path $\boldsymbol{t} \in G$, where there is an α-labelled edge from $\boldsymbol{p}$ to the initial capability of $\boldsymbol{t}$ and another from the terminal capability of $\boldsymbol{t}$ to $\boldsymbol{q}$.*

2. *There exists a principal $\boldsymbol{s}$ and a τ-path $\boldsymbol{t} \in G$, where there is a $\{m, \alpha\}$-labelled edge from $\boldsymbol{s}$ to the initial capability of $\boldsymbol{t}$ and an α-labelled edge from the terminal capability of $\boldsymbol{t}$ to $\boldsymbol{q}$.*

3. *There exists a principal $\boldsymbol{s}$, and τ-paths $\boldsymbol{t}$, $\boldsymbol{c} \in G$, where there exists $\{m\}$-labelled edges from $\boldsymbol{s}$ to the initial capability of $\boldsymbol{c}$ and from the terminal capability of $\boldsymbol{c}$ to $\boldsymbol{q}$, and there is a α-labelled edge from $\boldsymbol{p}$ to the initial capability of $\boldsymbol{c}$.*

4. *There exists principals $\boldsymbol{s}$, $\boldsymbol{v}$, and τ-paths $\boldsymbol{t}$, $\boldsymbol{c} \in G$, where there is exists $\{m\}$-labelled edges from $\boldsymbol{v}$ to the initial capability of $\boldsymbol{t}$ and from the terminal capability of $\boldsymbol{t}$ to $\boldsymbol{q}$, and there is a $\{m, \alpha\}$-labelled edge from $\boldsymbol{s}$ to $\boldsymbol{c}$.* □

Proof We prove the two directions of the bi-implication in turn.

If one of the four statements in Theorem 6.1 is true then can-share(α, $\boldsymbol{p}$, $\boldsymbol{q}$, G) is true:

For the graph configurations defined in each of the four statements we show that using only de jure rewriting rules we can come to a new graph for which the *can-share* predicate holds.

1. *There exists a τ-path $\boldsymbol{t} \in G$, where there is an α-labelled edge from $\boldsymbol{p}$ to the initial capability of $\boldsymbol{t}$ and another from the terminal capability of $\boldsymbol{t}$ to $\boldsymbol{q}$.*

   This case makes the predicate trivially true as the principal **p** already has "α-rights" to **q** and no rewriting rules needs to be used. Formally,

   can-share(α, $\boldsymbol{p}$, $\boldsymbol{q}$, G) is true.

2. *There exists a principal $\boldsymbol{s}$ and a τ-path $\boldsymbol{t} \in G$, where there is a $\{m, \alpha\}$-labelled edge from $\boldsymbol{s}$ to the initial capability of $\boldsymbol{t}$ and an α-labelled edge from the terminal capability of $\boldsymbol{t}$ to $\boldsymbol{q}$.*

   The principal **s** has α-right to **q** and also has manage rights to the capability granting it α-right. Thus it can give **p** (or any other vertex) this same capability. Formally,

   $G \vdash^{Duplicate} G'$, where *can-share(α, $\boldsymbol{p}$, $\boldsymbol{q}$, G')* is true.

3. *There exists a principal $\boldsymbol{s}$, and τ-paths $\boldsymbol{t}$, $\boldsymbol{c} \in G$, where there exists $\{m\}$-labelled edges from $\boldsymbol{s}$ to the initial capability of $\boldsymbol{c}$ and from the terminal capability of $\boldsymbol{c}$ to $\boldsymbol{q}$, and there is a α-labelled edge from $\boldsymbol{p}$ to the initial capability of $\boldsymbol{c}$.*

   The principal **s** can give **p** an m-labelled edge to **q** and then **p** using this new edge can create an α-labelled edge from **c** to **q**, thus getting α-right to **q**. Formally,

   $G \vdash^{Duplicate} \vdash^{Add} G'$, where *can-share($\alpha$, $\boldsymbol{p}$, $\boldsymbol{q}$, G')* is true.

4. *There exists principals* s*,* v*, and* τ*-paths* t*,* $c \in G$*, where there is exists* $\{m\}$*-labelled edges from* v *to the initial capability of* t *and from the terminal capability of* t *to* q*, and there is a* $\{m, \alpha\}$*-labelled edge from* s *to* c*.*

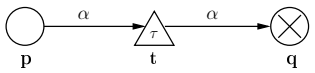
   The principal **v** can create an α-labelled edge from **p** to **c** and principal **s** can create an α-labelled edge from a capability in **t** to **q**. Formally,

$$G \vdash^{Duplicate} \vdash^{Add} G', \text{ where } \textit{can-share}(\alpha, p, q, G') \text{ is true.}$$

Now we show the other direction:
If can-share(α, p, q, G) *is true then one of the four statements from Theorem 6.1 holds.*

We must in other words prove that the four graph configurations in Theorem 6.1 are the only graph configurations that can can result in the *can-share* predicate becoming true. We prove this by back-tracking from the configurations where the predicate is true, and showing that for any possible series of traces we end at fixed point which is exactly one of the four configurations from the theorem.

As defined by Definition 6.4 the predicate is true if we by using only HNC de jure rules can come to a graph with the following configuration in it:

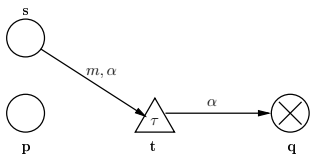


There are five general cases from which we the above configuration might be be possible to create using de jure rules if it did not exist in the graph already. We test each of these five cases in turn.

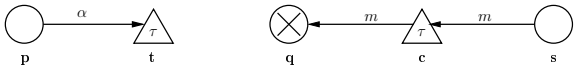
1. If there is an α-labelled edge from **p** to some capability **c**, and another α-labelled edge from some capability **d** to **q**, but there is no τ-path from **c** to **d**. Is it possible to create a τ-path that connects **c** and **d**? The answer is no. The only rule which can create a $\{\tau\}$-labelled edge is *CreateCap* but it can only do so by creating a new capability which this edge points to. Thus two distinct capabilities which are not in the same τ-path can never be in such a path.

2. If there is no outgoing α-labelled edge from **p** then the only rule which can create outgoing edge α-labelled edge from a principal is *Duplicate*. From *Duplicate* we get that in order to use the rule there must be some principal, say **s** which has an $\{m, \alpha\}$-labelled edge to the initial capability of some τ-path, say **c**. This means that there must be some previous graph configuration which using de jure rules has led to this configuration. So if there were no $\{m, \alpha\}$-labelled edge from **s** to the initial capability of **t** which rule could then have been applied to create it? The answer is again *Duplicate*, since no other rule can create outgoing edges from principals. Thus we have the situation where we can only create a $\{m, \alpha\}$-labelled edge from some principal **s** to the initial capability of some τ-path **c** if there exits some other principal **v**, which has an $\{m, \alpha\}$-labelled edge to the initial capability

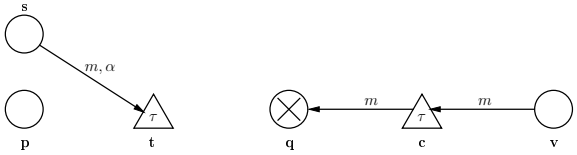
of some τ-path **d**. In other words we have a fixed point, since we have proved that we can only create an $\{m, \alpha\}$-labelled outgoing edge from a principal if such an edge already exists somewhere in the graph. Furthermore, from the first case we know that the τ-path **c** must in fact be the τ-path **t** as there is no way to connect capabilities which are not already connected. Thus the fixed point configuration looks as follows:



3. If **q** has no incoming α-labelled edge then the only rule which can create an incoming α-labelled edge to **q** is *Add*. For this rule to be usable there must be some principal, say **s**, that has an outgoing $\{m\}$-labelled edge to the initial capability of some τ-path **c**, and there must be another $\{m\}$-labelled edge from the terminal capability of **c** to **q**. So back-tracking from this configuration we get that the only way a capability can get an $\{m\}$-labelled edge to **q** if it does not already have one is if there exists another capability which has an $\{m\}$-labelled edge to **q** and a principal which has an $\{m\}$-labelled edge to that capability or another capability which dominates it. In other words, a $\{m\}$-labelled edge to **q** cannot be created unless there already exists one. We have thus found the fixed point. Graphically, the fixed point configuration looks as follows:



4. If both the α-labelled edges are missing, then what previous configuration could cause these to be created? We know when we can create the α-labelled edge from **p** to the initial capability of **t**, and when we can create the α-labelled edge from the terminal capability of **t** to **q**, so we now combine those two. The fixed point configuration looks as follows:



As can be seen on the graph the τ-path (which can be a single capability) does not need to be connected to **p** or **q**.

5. If only the two vertices **p** and **q** are present the configuration needed to create an α-labelled edge from **p** to the initial capability of some τ-path **t**, and another α-labelled edge from the terminal capability of **t** to **q** is the same configuration as in the previous case.

The first configuration (case 1) in Theorem 6.1 is the case where the path discribed in Definition 6.4 (*can-share*) exists already, and the remaining three configurations described in cases 2–4 in Theorem 6.1 are exactly the three distinct fixed point configurations shown above.

We have now proved both directions of the bi-implication in Theorem 6.1. ■

Having proved Theorem 6.1 we have also shown the decidability of the *can-share* predicate. The decidability is not guarantied for a model as ours as we given some graph can create infinitely many new graphs using our de jure rules. The system becomes infinite because of the rules *CreateCap* and *CreateR/P* which are essential to our model. Having proved the decidability we can go on to writing a *can-share* algorithm. This we will however not persue in this thesis, the reader should however notice that Theorem 6.1 give us the essentials of such an algorithm and an unoptimized algorithm would be very trivial to implement using the theorem as a guide.

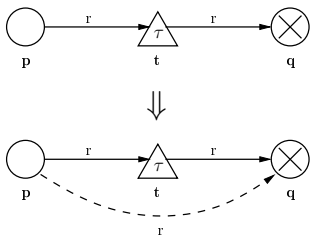
6.3. De Facto Rules

De jure rules allows us the reason about the potential accesses to principals and resources in a system using HNCs. Our model however also allows us to reason about information flow. We define information flow as a set of rewriting rules we call the *de facto* rewriting rules. These rules capture some information flow but not all. They do not capture information flow via *covert channels*[Lan81], which we will not get further into in this thesis. To ease the reasoning about information flow we extend the HPG model:

Definition 6.5 (Extended Hierachical Protection Graph (EHPG)) An EHPG is defined as a HPG extended with a new type of directed edges called *implicit edges* between principals and/or resources. Implicit egdes are graphically represented as dotted lines, and are always labelled $\{r\}$. □

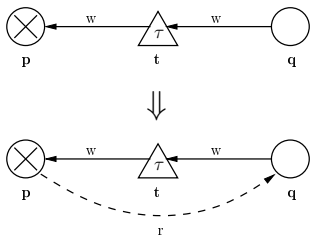
The implicit edges show where information flow may happen, but this information is already available in HPGs. The extension to EHPGs just gives us a more elegant way to represent it. Using EHPGs, we now present the de facto rewriting rules:

Read: Let **p**, **q**, **t** be vertices in a EHPG G, where **p** is a principal, **q** is a principal or resource, and **t** is a τ-path. Let there be $\{r\}$-labelled edges from **p** to the initial capability of **t**, and from the terminal capability of **t** to **q**. *Read* defines a new graph G' by adding an implicit edge from **p** to **q**. Graphically,



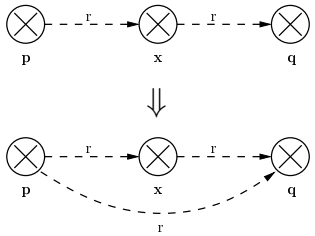
Informally: If a principal has read rights to a resource (or other principal) it might invoke that right.

Receive: Let **p**, **q**, **t** be vertices in a EHPG G, where **q** is a principal, **p** is a principal or resource, and **t** is a τ-path. Let there be $\{w\}$-labelled edges from **q** to the initial capability of **t**, and from the terminal capability of **t** to **p**. *Receive* defines a new graph G' by adding an implicit edge from **p** to **q**. Graphically,



Informally: If a principal has the right to write to another principal or resource then that principal or resource receives information.

Spy: Let **p**, **x**, **q** be principals or resources in a EHPG G and let there be implicit edges from **p** to **x**, and from **x** to **q**. *Spy* defines a new graph G' by adding an implicit edge from **p** to **q**.



Informally: If a principal (or resource) can read from another principal (or resource), which again can read from a third principal (or resource), then the first principal can "spy" on the second, thus indirectly receiving information from the third.

We now define a predicate *df-can-know(**p**, **q**, G_0)* where **p** is a principal, **q** is a resource (or principal) and G_0 is a HPG containing **p** and **q**.

Definition 6.6 *df-can-know(**p**,**q**,G_0)* $\Leftrightarrow$ G_0 is a HPG and there exists a sequence of EHPGs $G_0 \vdash^* G_n, n \geq 0$ using only de facto rules and in G_n there exists an implicit edge from **p** to **q**. □

To help us state when the predicate *df-can-know(**p**, **q**, G_0)* is true we need a few definitions.

Definition 6.7 (*rw*-τ-path) A *rw-τ-path* in a HPG G is a sequence of vertices $v_0, v_1, \ldots, v_{k-1}, v_k$, where $k \geq 2$, $v_1, \ldots, v_{k-1}$ is a τ-path from v_1 to v_{k-1}, v_0 is a principal, v_k is a principal or resource, and there is an edge from v_0 to v_1, and from v_{k-1} to v_k, both edges being labelled either $\{r\}$, $\{w\}$, or $\{r, w\}$.

v_0 is called the initial vertex, and v_k the terminal vertex, of the *rw*-τ-path. □

Definition 6.8 (*rw*-τ-chain) A *rw-τ-chain* in a HPG G is a sequence of *rw*-τ-paths $t_0, \ldots, t_k$, $k \geq 1$, such that the initial or terminal vertex of t_i is either initial or terminal vertex of t_{i+1}, $0 < i < k$. Such vertices are called *shared vertices*.

The initial vertex of t_0 is called the initial vertex of the *rw*-τ-chain, and the terminal vertex of t_k is called the terminal vertex of the *rw*-τ-chain. □

Definition 6.9 (Admissible *rw*-τ-chain) Let there be a sequence of *rw*-τ-paths t_0, $\ldots, t_k$, $k \geq 1$, that form a *rw*-τ-chain $\mathbf{c}$ in a HPG G. $\mathbf{c}$ is an *admissible rw-τ-chain* if and only if the following hold simultaniously for all i, where $0 \leq i < k$:

1. If the shared vertex bewteen t_i and t_{i+1} has an incomming edge from the τ-path in t_i then that edge is labelled α, where $r \in \alpha$.

2. If the shared vertex between t_i and t_{i+1} has an incomming edge from the τ-path in t_{i+1} then that edge is labelled β, where $w \in \beta$.

3. If the shared vertex between t_i and t_{i+1} has an outgoing edge then it is a principal. □

Now we state when the *df-can-know* predicate is true.

Theorem 6.2 *Let* $\boldsymbol{p}$ *and* $\boldsymbol{q}$ *be vertices in a HPG G. Then df-can-know(*$\boldsymbol{p}$*,* $\boldsymbol{q}$*, G) is true if and only if one of the following hold:*

1. *There exists a τ-path $\boldsymbol{t} \in G$ with an $\{r\}$-labelled explicit edge from* $\boldsymbol{p}$ *to the initial capability of* $\boldsymbol{t}$ *and another explicit $\{r\}$-labelled edge from the terminal capability of* $\boldsymbol{t}$ *to* $\boldsymbol{q}$*.*

2. *There exists a τ-path $\boldsymbol{t} \in G$ with an $\{w\}$-labelled explicit edge from* $\boldsymbol{q}$ *to the initial capability of* $\boldsymbol{t}$ *and another explicit $\{w\}$-labelled edge from the terminal capability of* $\boldsymbol{t}$ *to* $\boldsymbol{p}$*, and* $\boldsymbol{q}$ *is a principal.*

3. *There is an admissible rw-τ-chain from* $\boldsymbol{p}$ *to* $\boldsymbol{q}$*.* □

Proof We prove each direction of the bi-implication in turn.

If one of the cases in Theorem 6.2 are hold the df-can-know($\boldsymbol{p}$*,* $\boldsymbol{q}$*, G) is true.*

For the graph configurations in each case in Theorem 6.2 we now show using only de facto rewriting rules that we can create new graphs for which the *df-can-know* predicate holds.

1. *There exists a τ-path $\boldsymbol{t} \in G$ with an $\{r\}$-labelled explicit edge from $\boldsymbol{p}$ to the initial capability of $\boldsymbol{t}$ and another explicit $\{r\}$-labelled edge from the terminal capability of $\boldsymbol{t}$ to $\boldsymbol{q}$.*

   Applying the rule *Read* once gives us a new graph G' that satisfies the predicate trivially. Formally,

   $$G \vdash^{Read} G', \text{ where } \textit{df-can-know}(\boldsymbol{p}, \boldsymbol{q}, G') \text{ is true.}$$

2. *There exists a τ-path $\boldsymbol{t} \in G$ with an $\{w\}$-labelled explicit edge from $\boldsymbol{q}$ to the initial capability of $\boldsymbol{t}$ and another explicit $\{w\}$-labelled edge from the terminal capability of $\boldsymbol{t}$ to $\boldsymbol{p}$, and $\boldsymbol{q}$ is a principal.*

   Applying the rule *Receive* once gives us a new graph G' that satisfies the predicate trivially. Formally,

   $$G \vdash^{Receive} G', \text{ where } \textit{df-can-know}(\boldsymbol{p}, \boldsymbol{q}, G') \text{ is true.}$$

3. *There is an admissible rw-τ-chain from $\boldsymbol{p}$ to $\boldsymbol{q}$.*

   From the definition of a admissible rw-τ-chain we know that for each rw-τ-path in the chain one of the following two statements hold:

   a) The initial vertex has read-access to the terminal vertex of the rw-τ-path.

   b) The initial vertex has write-access to the terminal vertex of the rw-τ-path.

   In case (a) we apply the de facto rule *Read*, to create an implicit $\{r\}$-labelled edge from the initial to the terminal vertex.

   In case (b) we apply the rule *Receive* to create an implicit edge from terminal vertex to the initial vertex.

   We do this for every rw-τ-path in the rw-τ-chain. Due to the ordering of direction amongst the rw-τ-paths in a rw-τ-chain this gives us a chain of implicit edges going from vertex to vertex, starting from the initial vertex towards the terminal vertex, and now we simply use the *Spy* rule until we obtain a graph G' that trivially satisfies the *df-can-know* predicate. Formally,

   $$G \vdash^{Read^{\frac{m}{2}}} \vdash^{Receive^{\frac{n}{2}}} \vdash^{Spy^{k}} G',$$

   where $m =$ number of α_i-labelled edges in the rw-τ-chain and $r \in \alpha_i, n =$ number of β_i-labelled edges in the rw-τ-chain and $w \in \beta_i \wedge r \notin \beta_i, k = \frac{m}{2} + \frac{n}{2} - 1$, *df-can-know($\boldsymbol{p}$, $\boldsymbol{q}$, G')* is true.

We now prove the other direction of the bi-implication: *If df-can-know($\boldsymbol{p}$, $\boldsymbol{q}$, G) is true then one of the four statements from Theorem 6.2 holds.*

We prove the four cases in Theorem 6.2 by back-tracking from the graph configuration defined in Definition 6.6, where the *df-can-know* predicate is trivially true, and show that

the fix-point configurations exists and that they are exactly the configuration showed in the cases of the theorem. *df-can-know(**p**, **q**, G)* is trivially true if there is an implicit edge from **p** to **q**. So if that implicit edge does not exist the only way that it can be created is if one of the following three cases hold:

1. If there exists a τ-path **t** and $\{r\}$-labelled edges from **p** to the initial capability of **t**, and from the terminal capability of **t** to **q**. In other words, the configuration where the rule *Read* can be applied.

2. If there exists a τ-path **t** and $\{w\}$-labelled edges from **q** to the initial capability of **t**, and from the terminal capability of **t** to **p**. In other words, the configuration where the rule *Receive* can be applied.

3. If there exists an admissible rw-τ-chain between **p** and **q**. Each rw-τ-path in the chain satisfy one of the above two cases and we have already shown that if there is an admissible rw-τ-chain between **p** and **q** then we can use de facto rules to add an implicit edge from **p** to **q**. That the ordering defined in Definition 6.9 (admissible rw-τ-chain) is the only ordering allowing the eventual creation of an implicit edge from **p** to **q** is obvious by simply examining the de facto rewriting rules.

We have now proven both direction of the bi-implication. ■

The above proof is of course very trivial as none of the de facto rules can create new vertices. *df-can-know* has its limitations as it only can tell us about the currently possible information flows in a graph given that we do not change the explicit egdes. Therefore we will now move on to combining the sets of de jure and de facto rewriting rules. The *df-can-know* is however not wasted as it simplifies the proof given in the following section.

6.4. Combined Transfers

We call the combination of the de jure and de facto rules for *combined transfers*. The reason for creating combined transfers becomes clear by looking at the example in Figure 10.

On the figure three graphs can be seen, G, G', and G''. In G there is no sequence of de jure rules which can create a $\{r\}$-labelled edge from **p** to **q** and there is no sequence of de facto rules that can do it either (i.e. create an implicit edge from **p** to **q**). So our predicates *can-share* and *df-can-know* are both false for the graph G, meaning that **p** cannot get information from **q**. However, we can apply a sequence of de jure rules and get the graph G', and then applying a sequence of de facto rules we get G'', where there is an implicit edge from **p** to **q**. So we need a new predicate for reasoning about combined transfers. In practice when we want to reason about information flow it is combined transfers we are interested in, i.e. any sequence of actions which can cause the information flow to happen.

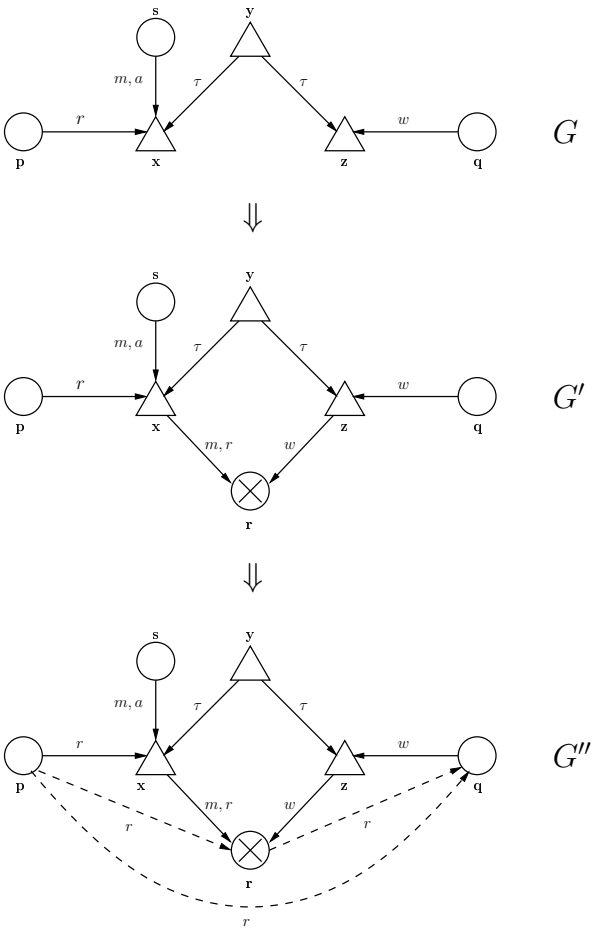


Figure 10: Example of a combined transfer.

Definition 6.10 *can-know(**p**,**q**,G_0)* $\Leftrightarrow$ G_0 is a HPG and there exists a sequence of HPGs and EHPGs such that $G_0 \vdash^* G_n$ using combined transfers and in G_n there exists an implicit edge from **p** to **q**. □

Theorem 6.3 *Let **p** and **q** be vertices in a HPG G. Then can-know(**p**, **q**, G) is true if and only if one of the following hold:*

1. *can-share(r, **p**, **q**, G) is true.*
2. *can-share(w, **q**, **p**, G) is true.*
3. *df-can-know(**p**, **q**, G) is true.*
4. *There exists a principal or resource **s** such that can-know(**p**, **s**, G) and can-know(**s**, **q**, G) are true.* □

PROOF Note that de jure rules are not defined on EHPGs. This means that any sequence of combined transfers must be ordered as first a sequence of de jure rewriting rules and then a sequence of de facto rewriting rules.

We now prove the first direction of the bi-implicitation: *If one of the cases in Theorem 6.3 hold then can-know(**p**, **q**, G) is true.*

For the graph configurations in each case in Theorem 6.2 we show that using de jure and de facto rewriting rule we can create new graphs for which the *df-can-know* predicate holds.

1. *can-share(r, **p**, **q**, G) is true.*

   We know from the definition of *can-share* (Definition 6.4) that the above means that there exists a sequence of de jure rewriting rules resulting in a graph with a τ-path **t** such that there are $\{r\}$-labelled edges from **p** to the initial capability of **t** and from the terminal capability of **t** to **q**. We can apply the de facto rule *Read* to create an implicit edge from **p** to **q**, and thus trivially satisfy the predicate:

   $$G \vdash^{Read} G', \text{ where } \textit{can-know(}\mathbf{p}, \mathbf{q}, G'\textit{)} \text{ is true.}$$

2. *can-share(w, **q**, **p**, G) is true.*

   We know from the definition of *can-share* (Definition 6.4) that the above means that there exists a sequence of de jure rewriting rules resulting in a graph with a τ-path **t** such that there are $\{w\}$-labelled edges from **q** to the initial capability of **t** and from the terminal capability of **t** to **p**. We can apply the de facto rule *Receive* to create an implicit edge from **p** to **q**, and thus trivially satisfy the predicate:

   $$G \vdash^{Receive} G', \text{ where } \textit{can-know(}\mathbf{p}, \mathbf{q}, G'\textit{)} \text{ is true.}$$

3. *df-can-know(**p**, **q**, G)* is true.

   We know from the definition of *df-can-know* that the above means that there we can use a sequence of de facto rules to create an implicit edge from **p** to **q**, and thus trivially satisfy the *can-know* predicate.

4. *There exists a principal or resource **s** such that can-know(**p**, **s**, G) and can-know(**s**, **q**, G) are true.*

   We know that it is possible, by using combined transfers, to get a EHPG G' where there is an implicit edge from **p** to **s**, and another implicit edge from **s** to **q**. This means that we can apply the de facto rule *Spy* and get a new graph G'' that trivially satisfy the *can-know* predicate.

We now prove the other direction of the bi-implication: *If can-know(**p**, **q**, G) is true then one of the five statements from Theorem 6.3 hold.*

As with the other two theorems we use back-tracking to prove that the cases in Theorem 6.3 are the only possible graph configurations which via combined transfers can lead to a graph that trivially satisfies the *can-know* predicate.

We know that the use of de jure rules on a graph G may lead to a new graph G' where de facto rules can be used that could not be used in G. That is, access control affects information flow. We also know that the use of de facto rules only can lead to the further use of de facto rules, as de jure rules does not work with implicit edges. That is, information flow does not affect access control. From Theorem 6.2 (*df-can-know*) we know the configurations that allow us to use de facto rules to create an implicit edge from **p** to **q**. Only de facto rules can create an implicit edge so the configurations identified in Theorem 6.2 are the configurations we must be able to create using de jure rules if the *can-know* predicate is to hold true.

The three cases are:

1. *There exists a τ-path $\mathbf{t} \in G$ with an $\{r\}$-labelled explicit edge from **p** to the initial capability of **t** and another explicit $\{r\}$-labelled edge from the terminal capability of **t** to **q**.*

   This is exactly when *can-share(r, **p**, **q**, G)* is true.

2. *There exists a τ-path $\mathbf{t} \in G$ with an $\{w\}$-labelled explicit edge from **q** to the initial capability of **t** and another explicit $\{w\}$-labelled edge from the terminal capability of **t** to **p**, and **q** is a principal.*

   This is exactly when *can-share(w, **q**, **p**, G)* is true.

3. *There is an admissible rw-τ-chain from **p** to **q**.*

   We prove this case by induction on the number of τ-paths in the *rw*-τ-chain. We know from Definition 6.9 (admissible *rw*-τ-chain) that a *rw*-τ-path with initial vertice **x** and terminal vertice **y**, found in an admissible *rw*-τ-chain, satisfy either *can-share(r, **x**, **y**, G)* or *can-share(w, **x**, **y**, G)*.

Base Case: For an admissible rw-τ-chain consisting of two τ-paths (the smallest possible rw-τ-chain) we trivially satisfy statement four of theorem: Each τ-path satisfy either statement one or two of the theorem, and the vertex **s** is the single shared vertex in the rw-τ-chain.

Inductive Step: Assuming that any admissible rw-τ-chain containing n τ-paths satisfy statement four of the theorem. We show that this also holds for admissible rw-τ-chains containing $n+1$ τ-paths.

The admissible rw-τ-chain starting from the first shared vertex satisfy our inductive hypthesis. The rw-τ-path from the first vertex of the original admissible rw-τ-chain to the first shared vertex satisfies either statement one or two of the theorem.

Thus for any admissible rw-τ-chain the theorem is satisfied.

The above three cases are exactly the cases identified in Theorem 6.3.
We have now proven both direction of the bi-implication. ■

Our proof of Theorem 6.3 is also proof of the decibility of information flow questions asked about a system using HNCs. So a HNC system actually has the abilitiy to support information flow control as well as access control, without having to extend the model. We thus get Corollary 6.4.

Corollary 6.4 *The LINK security model is both a DAC and a MAC model.* □

In practice one might allow certain users to create information flow policies, and maybe allow the grouping of resources into different security levels, or something even more exotic. All of this can simply be implemented in libOSes, i.e. applications, as it is nothing more than further abstractions put on top of HNCs.

6.5. Modelling User-Group Using HNCs

The purpose of using hierarchically named capabilities is that it should be possible to implement other security models on top of it. To show this we will now model the Unix user-group model using HNCs. Also, even though this should be obvious, show that the HNC model has more expressive power than the user-group model.

We start out by modeling the user-group model on top of HNCs: In Unix everything is a file and every file has a three-entry ACL. The first entry is the *owner*, the second is for the *group* the file belongs to, and the third is for *other* which means everybody else on the system. Each ACL entry specify three rights: read, write, and execute.

Users and groups on Unix systems both have 16 bit distinct names, a name used for a user can however also be used for a group. It is in fact common on many Linux systems (which use the same user-group model) that there for every user is a group with the same name that that user is the sole member of. So first off, there are two distinct concepts that HNCs must be able to simulate: The access control rights and the names (of users and groups).

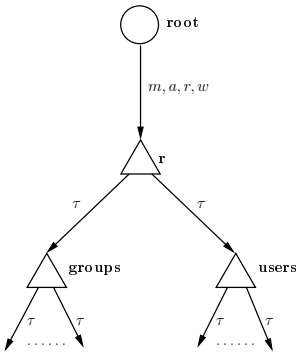
The access rights we have already touched upon in Section 6 when we discussed why Mazières's HNC structure could not capture write-only rights. The read and write rights

in the user-group model is obviously mapped directly to the read and write rights of our HNC structure. The execute right of the user-group model is mapped to the allocate right in HNCs since it basically is the same thing. The ability to execute a file in a Unix system, means to allocate a process, read the file into that process's address space and start executing the instruction stream. This is also why that in order to execute a file in Unix it is not enough to have execute access to it, one must also have read access, in order to read the files data. In a Unix system any process can spawn more processes, so it is not necessary to have execute access to a file in order to run it. Read access is enough. A user can spawn a new process, read the file into that process's address space manually and then manually jump to the newly loaded instructions. So the key to executing anything is the ability to allocate the needed resources to do so. Also, we are interested in access control flow, and thus do not concern ourselves with the hierarchy amongst files - i.e. files and directories. Directories which in turn just are special files in the Unix world anyway. Thus we simply model the execute right in HNCs as the allocate right.

Regarding the names in the user-group model, Mazières showed, through a small example, how one could model names using HNCs. One problem with HNCs is that they use 8 bit hierarchically ordered names, and the user-group model uses 16 bit for its names. This problem is simply solved as showed by Mazières[MK97], by using two levels of the hierarchy to represent one name. This is an implementation specific detail and is therefore ignored throughout the rest of this section. We ignore it as Mazières has already showed it not to be a problem. To solve the problem that users and groups can have the same names, we simply split the capability tree into two branches, right after the root node. One branch for groups, and one for users.

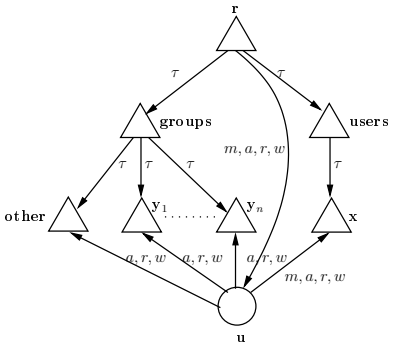
We now define the Unix concepts of *root*, *user*, *group*,*other*, and *files* using HNCs.

Definition 6.11 (root) *Root* is a principal with an $\{m, a, r, w\}$ labelled edge to the initial capability of the systems capability tree. Graphically,



The graph also shows how the capability tree is split into two branches right after root's all-dominating capability r, one branch for groups starting from the capability called **groups** and one for users starting from the capability called **users**. □

Definition 6.12 (user) A *user* **u** is a principal which has an edge labelled $\{m, a, r, w\}$ to exactly one capability **x** directly dominated[3]by **users**, has an edge labelled $\{a, r, w\}$ to the capability **other** which is directly dominated by **groups** and has an edge to capabilities $\mathbf{y}_1, \ldots, \mathbf{y}_n, n \geq 1$ labelled $\{a, r, w\}$ which are also directly dominated by **groups**, and has an incomming edge from the capability the root capability **r** labelled $\{m, a, r, w\}$. Graphically,

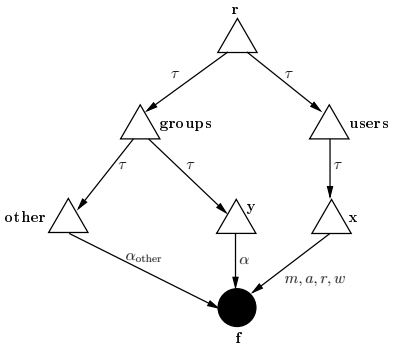


□

Definition 6.13 (group) A *group* is a capability **g** which is directly dominated by **groups** and where all incomming edges are labelled $\{a, r, w\}$ and all outgoing edges are labelled with nonempty subsets of $\{a, r, w\}$. □

Definition 6.14 (other) *Other* is a capability **other** which is directly dominated by **groups** and has incomming edges labelled $\{a, r, w\}$ from all principals, and outgoing edges to all resources labelled with nonempty subsets of $\{a, r, w\}$. □

Definition 6.15 (file) A *file* is a resource **f** with an incoming edge labelled $\{m, a, r, w\}$ from a capability **x** directly dominated by **users**, an incomming edge labelled α_{other} from the capability **other** directly dominated by **groups**, an incomming edge from a capability **y** with label α, where $\alpha_{\text{other}}, \alpha$ are nonempty subsets of $\{a, r, w\}$, and an incomming edge from the root capability **r** labelled $\{m, a, r, w\}$. Graphically,



[3]Directly dominated means that the there is only one $\{\tau\}$-labelled edge in the τ-path from **users** to the principal.

The reason why the edge from x to f is labelled $\{m, a, r, w\}$ is that the owner of a file can change its own access rights to it. So since a user in reality has the possibility to gain full access to any file it owns we simply define it to always have full access. This is a restriction we make in order to make our graphical representations simpler. In practice this restriction can be trivially removed. □

Having showed how to capture the main concepts of the user-group model using HNCs we now proceed to define de jure rules for the user-group model and show that these can be simulated using HNCs de jure rules.

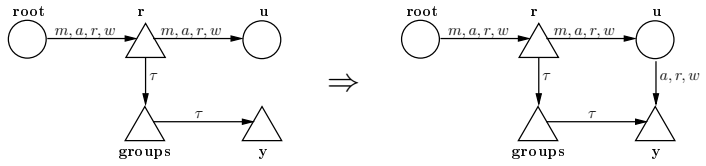
CreateU/G: Let **x**,**r**, and **users**/**groups** be distinct vertices in a HPG G where **r** is the root of the capability tree and **users**/**groups** is either **users** or **groups** as defined in Definition 6.11. Let there be an edge from **x** to **r** labelled $\{m, a, r, w\}$ and an edge from **r** to **users**/**groups** labelled τ. *CreateU/G* defines a new graph G' by adding a new vertice **y** and an edge from **users**/**groups** to **y** labelled τ. Graphically,

Informally: Root can create new users and groups at will. Root is also the only user that can do this.

Mapping to HNC: To come from graph G to G' by using HNC rewriting rules we only use the rewriting rule *CreateCap* once:

$$G \vdash^{CreateCap} G'$$

AddGroup: Let **root**, **r**, **groups**, **u**, **y** be distinct vertices in a HPG G, where **r**, **groups**, **y** are capabilities and **root**, **u** are principals, and let there be edges from **root** to **r** and **r** to **u** labelled $\{m, a, r, w\}$, and let there be $\{\tau\}$ labelled edges from **r** to **groups**, and from **groups** to **y**. *AddGroup* defines a new graph G' by adding an edge from u to y labelled $\{a, r, w\}$. Graphically,

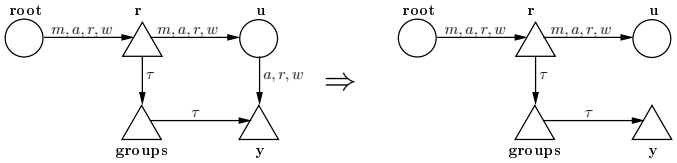


Informally: Root can make any user member of any group at will.

Mapping to HNC: To come from graph G to G' by using HNC rewriting rules we use the rewriting rule *Duplicate* once:

$$G \vdash^{Duplicate} G'$$

RemoveGroup: Let **root**, **r**, **groups**, **u**, **y** be distinct vertices in a HPG G, where **r**, **groups**, **y** are capabilities and **root**, **u** are principals, and let there be edges from **root** to **r** and **r** to **u** labelled $\{m, a, r, w\}$, from **u** to **y** labelled $\{a, r, w\}$ and let there be $\{\tau\}$ labelled edges from **r** to **groups**, and from **groups** to **y**. *RemoveGroup* defines a new graph G' by removing the $\{a, r, w\}$ labelled edge from **u** to **y**. Graphically,

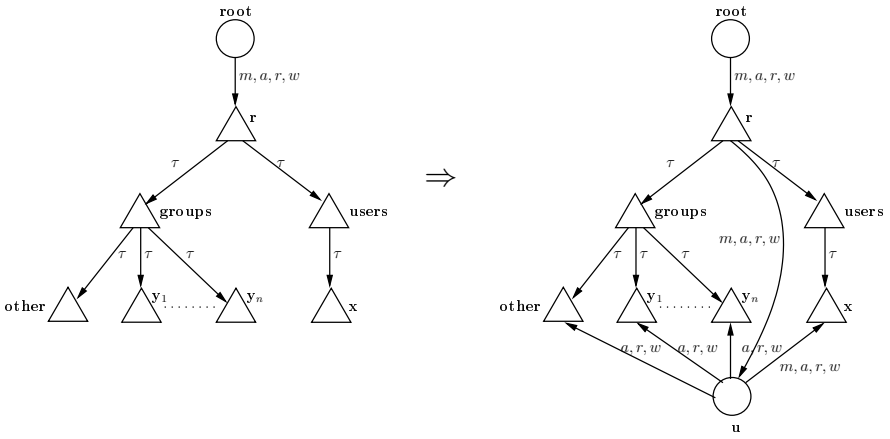


Informally: Root can remove group membership from any user at will.

Mapping to HNC: To come from graph G to G' by using HNC rewriting rules we use the rewriting rule *RemoveOther* once:

$$G \vdash^{RemoveOther} G'$$

Login: Let **r**, **groups**, **users**, **other**, **x**, $\mathbf{y}_1$, ..., $\mathbf{y}_n$ be capabilities, where $n \geq 0$, and let **root** be a principal in a HPG G. Let there be an edge from **root** to **r** labelled $\{m, a, r, w\}$ and let there be $\{\tau\}$-labelled egdes from r to **groups**, **r** to **users**, **users** to **x**, **groups** to **other**, and from **groups** to $\mathbf{y}_1$,...,$\mathbf{y}_n$. *Login* defines a new graph G' by adding a principal **u**, and adding edges from **u** to **other**, $\mathbf{y}_1$, ..., $\mathbf{y}_n$ labelled $\{a, r, w\}$, from **u** to **users** labelled $\{m, a, r, w\}$ and from **r** to **u** labelled $\{m, a, r, w\}$. Graphically,



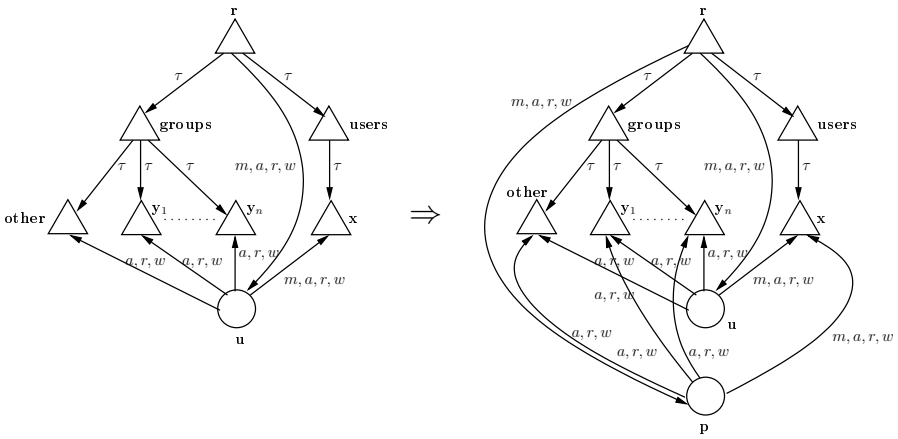
Informally: Root can take a user capability and a set of groups and create a process which have these capabilities. This is what happens when a user logs in to a Unix OS.

Mapping to HNC: To come from graph G to G' by using HNC rewriting rule we first apply the rule *CreateR/P* to a new principal **p** along with an $\{m\}$-labelled edge from **r** to **p**. Then we use *Add* to add an edge from **root** to **p** and following that we use the rule *Duplicate* n times to add outgoing edges from **p** to **other**,**x**,$\mathbf{y}_1$,...,$\mathbf{y}_n$:

$$G \vdash^{CreateR/P} \vdash^{Add} \vdash^{Duplicate^n} G'$$

Where $\vdash^{Duplicate^n}$ means the rule *Duplicate* applied n times.

Fork: Let **r**, **groups**, **users**, **other**, **x**, $\mathbf{y}_1, \ldots, \mathbf{y}_n$ be capabilities, and let **u** be a principal in a HPG G. Let there be $\{\tau\}$-labelled egdes from r to **groups**, **r** to **users**, **users** to **x**, **groups** to **other**, and from **groups** to $\mathbf{y}_1$,...,$\mathbf{y}_n$, and let there be $\{a, r, w\}$-labelled edges from **u** to **other**,$\mathbf{y}_1$,...,$\mathbf{y}_n$, and $\{m, a, r, w\}$-labelled edges from **r** to **u** and from **u** to **x**. *Fork* defines a new graph G' by adding a new principal **p** with the exact same set of of incomming and outgoing edges as **u**. Graphically,

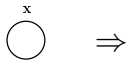


Informally: Any process can create a copy of itself. The copy having the same capabilities as the original process.

Mapping to HNC: To come from graph G to G' by using HNC rewriting rule we first use the rule *CreateR/P* to add new principal **p** with an $\{m\}$-labelled edge from **x** to **p**. Then using *Add* we add an incomming edge to **p** from **root** and using *Duplicate* n times we add edges from **p** to **other**, **x**, $\mathbf{y}_1$, ..., $\mathbf{y}_n$. Finally, we use *RemoveSelf* to remove the $\{m\}$-labelled edge from **x** to **p**:

$$G \vdash^{CreateR/P} \vdash^{Add} \vdash^{Add} \vdash^{Duplicate^n} \vdash^{RemoveSelf} G'$$

Kill: Let **x** be a principal in HPG G. *Kill* defines a new graph G' by removing **x** along with all its incomming and outgoing edges. Graphically,

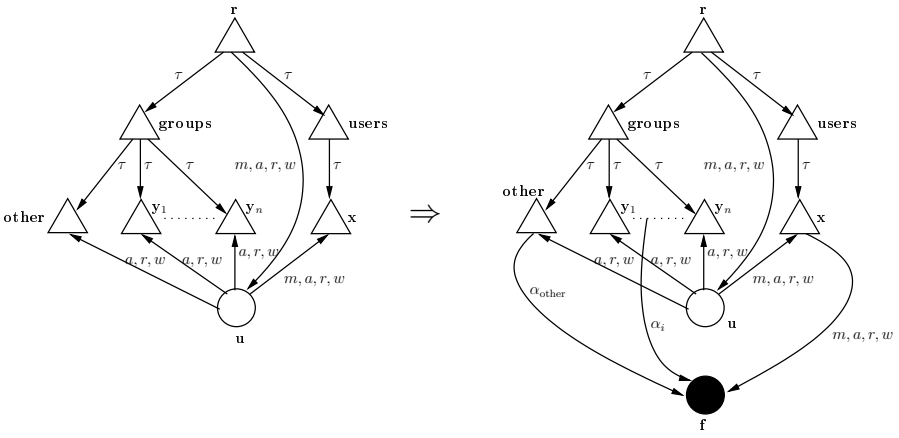


Informally: Any process can terminate itself at will. A process can also kill other processes if they all have the same user capability, and root can kill any process it wishes. However, these last two scenarious will not generate any graphs that are not already generated by the first scenario were any process can terminate itself at will.

Mapping to HNC: To come from graph G to G' by using HNC rewriting rules we first use the rule *RemoveOther* n times. Where n is the number of incomming edges to **x** minus one. That is, we first remove all other capabilities from the principal **x**'s ACL. Then we use the rule *RemoveSelf* to remove the last incomming edge to **x**:

$$G \vdash^{RemoveOther^n} \vdash^{RemoveSelf} G'$$

CreateFile: Let **r**, **groups**, **users**, **other**, **x**, $\mathbf{y}_1$, ..., $\mathbf{y}_n$ be capabilities, and let **u** be a principal in a HPG G. Let there be $\{\tau\}$-labelled egdes from r to **groups**, **r** to **users**, **users** to **x**, **groups** to **other**, and from **groups** to $\mathbf{y}_1$,...,$\mathbf{y}_n$, where $n > 0$, and let there be $\{a, r, w\}$-labelled edges from **u** to **other**,$\mathbf{y}_1$,...,$\mathbf{y}_n$, and $\{m, a, r, w\}$-labelled edges from **r** to **u** and from **u** to **x**. *CreateFile* defines a new graph G' by adding a new resource **f** and adding incomming edges to **f** from **r**, **x**, **other**, $\mathbf{y}_i$, labelled $\{m, a, r, w\}$, $\{m, a, r, w\}$, α_{other}, α_i, respectively; where $0 < i \leq n, n > 0$. Graphically,

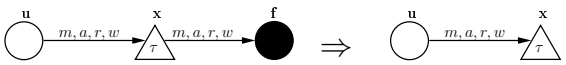


Informally: Any user can create a new file which it becomes the owner off, and which belongs to one of its groups (as well as the group **other**).

Mapping to HNC: To come from graph G to G' by using HNC rewriting rules we first use the rule *CreateR/P* to create the resource **f** along with an $\{m\}$-labelled edge from **x** to **f**. Then we use the rule *Add* four times to create edges from **root**, **other**, and $\mathbf{y}_i$.

$$G \vdash^{CreateR/P} \vdash^{Add^4} G'$$

RemoveFile: Let **u** be a principal, **x** a capability, and **f** a resource in a HPG G, and let there be a $\{m, a, r, w\}$ from **u** to **x**, and from **x** to **f**. *RemoveFile* defines a new graph G' by removing the resource **f** along with all its incomming and outgoing edges. Graphically,

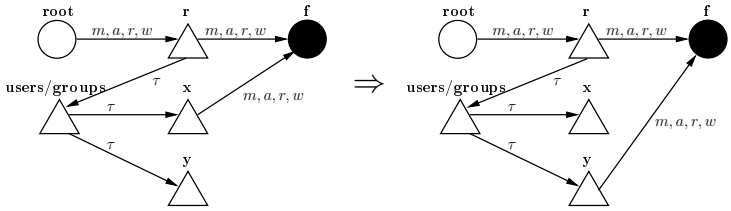


Informally: A user can delete any file which it is the owner of.

Mapping to HNC: To come from graph G to G' by using HNC rewriting rules we first use the rule *RemoveOther* n times. Where n is the number of incomming edges to **f** minus one. That is, we first remove all other capabilities from the resource **f**'s ACL. Then we use the rule *RemoveSelf* to remove the last incomming edge to **f**:

$$G \vdash^{RemoveOther^n} \vdash^{RemoveSelf} G'$$

ChangeO/G: Let **root** be a principal, **f** a resource, and **r**, **x**, **y**, **users/groups** be capabilities in a HPG G where **users/groups** is either the capability **users** or **groups** as defined in Definition 6.11. Let $\{m, a, r, w\}$-labelled edges from **root** to **r**, **r** to **f**, **y** to **f**, and let there be $\{\tau\}$-labelled egdes from **r** to **users/groups**, **users/groups** to **x**, and from **users/groups** to **y**. *ChangeO/G* defines a new graph G' by removing the $\{m, a, r, w\}$-labelled edge from **x** to **f** and adding a $\{m, a, r, w\}$-labelled edge from **y** to **f**. Graphically,

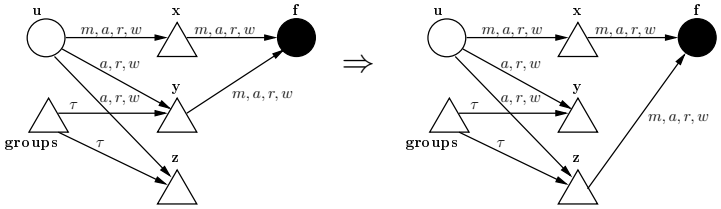


Informally: Root can change the ownership of a file. Choosing any user in the system as the new owner. The same goes for groups.

Mapping to HNC: To come from graph G to G' by using HNC rewriting rules we first use rule *Add* to add an edge from **y** to **f** and then we use the *RemoveSelf* to remove the edge from **x** to **f**:

$$G \vdash^{Add} \vdash^{RemoveSelf} G'$$

ChangeGroup: Let **u** be a principal, **f** a file, and **groups**, **x**, **y**, **z** capabilities in a HPG G. Let there be $\{m, a, r, w\}$-labelled edges from **u** to **x**, from **x** to **f**, from **y** to **f**, $\{a, r, w\}$-labelled edges from **u** to **y** and from **u** to **z**, and let there $\{\tau\}$-labelled edges from **groups** to **y** and **groups** to **z**. *ChangeGroup* defines a new graph G' by removing the $\{m, a, r, w\}$-labelled edge from **y** to **f** and adding a $\{m, a, r, w\}$-labelled edge from **z** to **f**. Graphically,

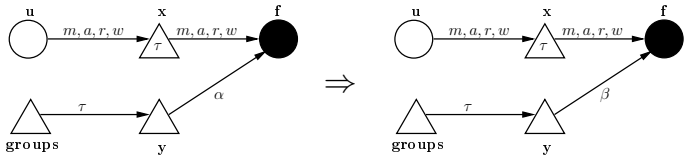


Informally: A user can change the group associated with a file as long as it is owner of the file and a member of the group it changes it to.

Mapping to HNC: To come from graph G to G' by using HNC rewriting rules we first use the rule *Add* to add an edge from **z** to **f** and then we use the *RemoveSelf* to remove the edge from **y** to **f**:

$$G \vdash^{Add} \vdash^{RemoveSelf} G'$$

ChangePermissions: Let **u** be a principal, **f** a resource, and **groups**, **x**, **y** capabilities in a HPG G, where **groups** is defined as in Definition 6.11. Let there be $\{m, a, r, w\}$-labelled edges from **u** to **x**, **x** to **f**, a $\{\tau\}$-labelled edge from **groups** to **y** and an edge from **y** to **f** labelled α where α is a nonempty subset of $\{a, r, w\}$. *ChangePermissions* defines a new graph G' by removing the α-labelled edge and adding a new edge in its sted labelled β where β is a nonempty subset of $\{a, r, w\}$. Graphically,



Informally: A user can change the access permission for owner, group, and other on any file it is the owner of.

Mapping to HNC: To come from graph G to G' by using HNC rewriting rules we first use the rule *RemoveOther* to remove the α-labelled edge from **y** to **f** and then use *Add* to add the β-labelled edge from **y** to **f**:

$$G \vdash^{RemoveOther} \vdash^{Add} G'$$

Note that there are no rules for how the root user can create files. This is not needed as we simply can add a user to the system which represents root and then the real root user can create files using that. This also reflects reality nicely as for instance on Linux systems there is a user and group called root. Also note that the τ-paths in the rules ensures that root can manage all principals and resources in the system and that the τ-paths only effects the root user.

We have defined a set of de jure rules for the user-group model and showed that they all can be mapped to HNC de jure rules. Now we need to show that HNC de jure rules

has higher expressiveness. We do this by showing that there exist an HNC de jure rule which cannot be mapped to the user-group de jure rules. Finding such a rule does not take long. The rewriting rule *Add* permits the addition of an $\{m\}$-labelled edge to a resource. In the Unix world this would mean that a file could have more than one owner. There is no user-group de jure rewriting rule, or sequence of rules, which can simulate the HNC *Add* rule. One can easily see this as the only user-group rule which can add an edge to an existing resource is the *AddGroup* rule and this rule can only add an edge labelled $\{a, r, w\}$. In our user-group model, resources have two incomming edges with m in the label, one for the owner of the file, and one for root.

Having showed that HNCs can simulate the user-group model serves as a good example of the models expressiveness. We have only showed how to simulate the de jure rules as there are no information control available in the classical user-group model. In fact, as soon as a model per default uses a construct like the user-group models *other* group, information flow becomes impossible to control.

There are many other security models which could be interesting to see simulated by HNCs. For instance, newer models based on role-based access control. We expect it to be possible to simulate such a model using HNCs but it is not certain, and will have to be further looked in to.

6.6. Summary

With LINK's HNC model we can reason about both access control and information flow, and we can implement other security models on top of it. What really makes a HNC model powerful is the hierarchy. Thanks to it a libOS can implement a security model of its own, but it still has to submit to the security policies forced it from higher levels of the hierarchy. The security model will in practice be part of a LINK OS in the form of a system service. This system service will then be queried by the rest of the system when access permissions needs to be checked.

7. Future Work

Obviously, we would have loved to develop a proper LINK OS, and not just a proof of concept implementation. But as such a project goes beyond the scope of my master thesis we did not persue it. However, a lot of ideas came to mind while developing POCLINK and those have ended up on the "future work" list. There is also a lot more that could be formally looked into about our security model. The research field of operating systems is a wast one and there will always be new things to research. We will in this section touch upon some of the future development plans we have for the LINK architecture and its security model.

7.1. PXELINK

While implementing POCLINK we got many ideas on how to design a disk-less LINK OS. The main idea was that it should initialise itself via the network using the PXE

environment (which POCLINK also would have done if the QEMU bug had not stopped us from doing this), but unlike POCLINK it would stay true to all the LINK principles. We named the OS we had in mind PXELINK and we will explain the general idea of the design now, one system service at a time.

Bootstrap The bootstrap program is the initial program sent from the TFTP server to the client machine, once the DHCP server has given the client an IP address. The bootstrap program should perform the following once loaded and executed at the client:

1. Verify that it has a sane environment to operate in. This means that is should verify that system has PXE 2.0 or greater, and then continue to check the sanity of the !PXE structure[4] using the check-sums located within it[Cor99]. Older PXE versions could be supported but another solution could be that if someone wanted to run PXELINK on such hardware they must boot the machine with some bootable media, like for instance an Etherboot CD as explained in Appendix B. This would make a proper environment available for the bootstrap program run in.

2. Dump all hardware information into a file and send it using the PXE API to the TFTPs IP address but to another port. The port should be defined in advance and at the machine hosting the TFTP server some server should be waiting for incoming connections. After the file has been successfully transferred to the server the client starts waiting for incoming data. The server in the meanwhile takes the data from the client, analyses it, and readies an ordered list of system services and starts transmitting the system service ELF binaries to the client one by one.

3. As the client receives system services from the server it relocates them to different memory addresses, each system service address starting at a page boundary, making it easy to turn on paging later.

4. Once all system services are received and relocated the bootstrap program sets up the basic system structures such a the Global Descriptor Table, Task-State Segment, etc.

5. The bootstrap program switches the system into protected mode.

6. As each of the received system services are contained in one or more ELF binaries, all their dependencies are specified in their ELF header. The bootstrap program expects that all dependencies are met and links all the system services together. That the dependencies are met must be guaranteed by the server.

7. Each system service has a procedure called *initialise* which the bootstrap program can call and then the system service initialises itself. This is why the server sent the services in a special order, as this is the order they must be initialised in. The first system service might be the memory manager, then the task switcher, and so

[4] "!PXE" is the name of the data structure containing information such as the PXE API entry point address, and is setup on boot on any system that supports PXE 2.0 or greater.

on. This ensures that when the next service in the list initialises itself, its *critical* dependencies are met. A critical dependency for a system service is one that it needs to initialise itself. If some cyclic critical dependencies are impossible to avoid the problem can be resolved by sending a special "initialiser" system services which knows how to handle the situation. This service would then remove itself once its job was done. The specific details are very hardware dependant and actual conventions and protocols will of course need to be created.

8. When all system services are initialised the bootstrap program gives control over to the task scheduler, which as this point, as part of the systems initialization, is ready to go to work.

Having a server side service allows us to move a lot of functionality from the client to server. As the server obviously can be considered trusted since the system is being booted from it. There should no problem in allowing the server to decide which system services the client needs. The client has no way of knowing what system services are available anyway, as PXELINK is disk-less, and therefore if any state must be saved from boot-up to boot-up it must be saved on a server on the network.

One of the trickier parts of the bootstrap process is how to handle access rights of the different system services. Even though the initial system services are considered trusted it would be nice, if not only from a software engineering point of view, to know what they can and cannot do. Without this protection guarantee debugging possible errors during initialization can become as nightmarish as it is known to be the case with many monolithic kernels. An idea for how to solve the problem is to make the security manager system service have two initialization phases. The first phase initialises the security manager but puts it in a special "system initialization phase" mode and should be initializid as the first thing on the system. In this phase all other system services should be able to use the security manager but all it really does is keep a history of the systems initialization and if two system services for instance try to write to the same memory area it can spot this, send the history to some server on the network for debugging, and halt the system. If no errors are detected it will be initialised again and enter the second phase, it now starts behaving like a security manager in a complete system. An even more "LINKish" way would perhaps be to create an extra system service which only acts as an initialization monitor and which removes itself once initialization is complete and no errors were detected. All in all, its an issue that has to be resolved but it is clearly solvable and only a question about finding a fitting solution which remains simplistic and elegant.

Memory Manager The memory manager is quite typical. It presents the memory to each task as if it was the only task on the system. All OS data is located at the same addresses and each task's entry point address is the same. Unlike the typical virtual memory manager it shows the global memory state to the entire system and allows tasks to request specific physical memory pages as well. When a task requests a physical page and an address in its own address that it wants it the physical page mapped to, the

memory manager checks that both address are free. If they are the page is mapped into the tasks address space. This functionality can for instance come in handy when working with memory mapped I/O.

Software Task Switcher The software task switcher does very little. It is invoked with an identifier to the target task and when invoked it saves the state of the current task, then loads the state of the next task, and gives control to it. The task switcher in PXELINK will be an optimized and brushed up version of the task switcher used in POCLINK. The task switcher might be extended such that it is possible to decide if not all task state should be saved. The POCLINK task switcher only loads the privileged parts of the next task as this does not compromise security and then leaves it to the task to load the rest of its own data (functionality which normally would be supplied by a shared library). The POCLINK task switcher does however save all task state each time (though still less state than if hardware task switching was used). If a selective task switching mechanism can be implemented without any significant performance downgrade of the average case full state saving task switch then it would be a great feature indeed. The selection mechanism will of course use some CPU cycles, the question is how many.

Task Scheduler The PXELINK task scheduler uses a scheduling algorithm like the one used for the MIT Exokernel called Xok [Eng98]. The CPU will be represented as a vector, each element representing one quantum. The length of the vector depends on the speed of the CPU and the wished quantum length. Each task can then allocate all the quantums it wants and the task scheduler uses a simple round-robin algorithm for scheduling. The idea of representing the CPU as a vector of quantums is a flexible way to allow applications to use their own scheduling algorithms on top of it. It might seem odd that any task can allocate quantums as insanely as it wants to but in practice this is not a problem and not really any different than what is possible in other operating systems. Many OSes allow processes to fork copies of themselves, and this is in fact the same as allocating more quantums. Many OSes also have a maximum number of processes it can handle. Both of these things are for instance the case with an OS like FreeBSD. To put some restraint on the tasks allocation of quantums, the task scheduler takes quantums away from the most greedy task when new quantums are requested but no more are available. Also, PXELINK must use an explicit allocation policy, so when one quantum has finished the currently executing task is notified and then has a certain amount of time to save its state a return control to the task scheduler. Any task which does not meet the time limit could for instance be punished by loosing some of its quantums or simply be killed.

As it might be hard for application programmers to guarantee revocation deadlines are met, such functionality should be available from a shared library. Applications with special needs can then still implement their own. An idea came to mind while developing POCLINK that perhaps the APIC performance monitoring functionality could be used to invoke the task scheduler instead of the APIC timer. That way, every task would be guaranteed a certain amount of cycles each time it got its turn to execute and would make

calculations on reaching deadlines easier. This is merely an infant idea and experiments must be performed to find out if it is a good one.

ELF Loader The dynamical loading of shared libraries is handled by the ELF loader. The ELF loader keeps track of all libraries on the system and makes a list of all available libraries globally available. Shared libraries specify their own functions and if some functionality requires a context switch to be performed then a shared procedure can yield into that context. This way no extra system service is needed to keep track of the shared libraries and their functionality. As long as tasks know where to look up the list of available libraries, management is no different than in for instance a Linux system.

Event and I/O Managers One or more system services are needed to manage interrupts, exceptions, and input/output. The management of these resources can be done using the interrupt vector table, I/O Bitmaps, and other functionality made available by the IA-32 architecture. A driver for a hardware device will request to receive interrupts from it on a certain interrupt vector and if access is granted an interrupt gate for the drivers interrupt service routine is created in the interrupt vector table. If memory mapped I/O is supplied by the device the driver of course also needs to be granted access to those memory regions by the memory manager. Basically, these managers just handle functionality available in hardware.

Device Drivers A device driver for each device will be sent to the client from the server during bootstrapping. Device drivers should have a small shared library along with them defining their interfaces. It might be more practical to distinguish device drivers from other shared libraries, in that case a system service could handle this.

As PXELINK is a disk-less OS an ethernet driver is obviously needed, but more than a device driver is needed in order for the network to become usable. The different network protocols must be identifiable and understood by the system such that the packets can be directed to the right applications. This job is normally performed by what is known as a *packet filter*. The exokernels use a dynamic packet filter (DPF) [EK96] to allow applications to insert and remove new protocol definitions, and the DPF is clever enough, due to the language the protocol definitions must be written in, to detect over-lapping definitions. Even though dynamic package filtering is slower than static packet filtering, it has the clear advantage of giving more power to the application programmer. In Internet dominated times like these its clearly a valuable feature that applications can specify new protocols and use them without needing to have privileged access to the system.

Summary Some of the most essential system services for PXELINK have now been touched upon. More will of course be needed before PXELINK is done: Various kinds of device drivers, security manager, a shell, and so on. The LINK architecture has a lot in common with the exokernel architecture and the MIT exokernel systems can often be used as reference when designing a specific LINK system service. PXELINK is a good starting point for a proper implementation of the LINK architecture as it is disk-less.

Multiplexing the disk is a problematic affair, and even though a solution was made for the exokernels, more research is still needed before multiple libOS can truly coexists together on the same disk with their individual file systems, like is also noted by Dawson R. Engler in his PhD thesis [Eng98].

7.2. Security Model

There are many interesting things that still needs be looked into regarding the LINK security model. First of all the model needs to be implemented into a LINK OS and tested in practice. Secondly, algorithms must be created for efficiently looking for specific access control and information flow configurations in the system.

A language for specifying security policies could also be developed. This could be a convenient way for libOSes to enforce special restrictions on the applications using them.

It would also be nice to see other security models be implemented on top of the LINK security model; for instance, the different models that supported in Security Enhanced Linux.

8. Conclusion

We have shown the LINK architecture's feasibility, and that security can be handled in a proper and flexible manner. As mentioned, there is still research needed in order for the multiplexing of disks to be usable in practice, and there are also a lot of other multiplexing issues once we start bringing graphical user interface into the mix. The way to tackle all of the things that still needs to be done, if the LINK architecture is to have a future, is to implement a proper LINK OS. We believe PXELINK is an excellent starting point. Implementation of PXELINK will start as soon as possible and when a version is ready for public release it will most likely be available at `http://www.cs.aau.dk/~zion1459`. PXELINK will be open source and free.

We propose the LINK architecture as a solution to problems identified in Section 1. The LINK principles are basically the same as the exokernel principles except for different interpretations needed in some cases due to LINK being kernel-less. Since the exokernels also use hierarchically named capabilities, the formalism we have developed can also be used on for those systems. The main usage of the formalism and the proofs we have presented is that they show just how much security a HNC model can bring to a system. Algorithms must, of course, be devised which are efficient enough to be used in practice.

The LINK and exokernel architectures have many similarities, and an advance in one architecture can probably easily be ported to the other. What in our opinion makes the LINK architecture superior to the exokernel architecture is the improved reliability it gets from having protected address spaces for each of its system services, and not to forget the greater simplicity achieved by this. Simplicity is invaluable for an OS to have as it makes it easier for programmers to write correct code, and even more importantly, in the case of concurrency, it is actually possible to create formal concurrency models of the system and model check these. As LINK OSes only have a handful of system services, there will not be much extra memory usage in keeping them all in separate

address spaces, unlike for instance component-based OSes which might have thousands of components, the extra memory used for each of them matters.

A LINK OS might have a worse overall performance than an exokernel OS due to the extra context switches that need to be performed when system services communicate with each other. It is, however, quite possible that this will not be the case if the system services are developed properly (i.e. with performance in mind). One might even argue that in some cases a LINK OS can have better performance than an exokernel OS as applications can access each device more directly in a LINK OS. All this will, however, have to be proved by implementating a complete LINK OS like PXELINK, and performing macro-scale benchmarking.

References

[BDF+03a] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP 2003*, 2003.

[BDF+03b] Paul R. Barham, Boris Dragovic, Keir A. Fraser, Steven M. Hand, Timothy L. Harris, Alex C. Ho, Evangelos Kotsovinos, A.V.S. Madhavapeddy, R. Neugebauer, I.A. Pratt, and A.K. Warfield. Xen 2002. Technical report, University of Cambridge, Computer Laboratory, January 2003.

[BS79] Matt Bishop and Lawrence Snyder. The transfer of information and authority in a protection system. In *SOSP*, pages 45–54, 1979.

[Com93] TIS Committee. *Tool Interface Standard (TIS) Portable Formats,* version 1.1, October 1993.

[Cor99] Intel Corporation. *Preboot Execution Environment (PXE) Specification,* version 2.1, September 1999.

[EK96] Dawson R. Engler and M. Frans Kaashoek. DPF: fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of the ACM SIGCOMM '96 Conference: Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 53–59, Stanford, California, August 1996.

[Eng98] D. Engler. *The Exokernel operating system architecture.* PhD thesis, Massachusetts Institute of Technology, 1998.

[fre] The FreeBSD Project. `http://www.freebsd.org`.

[HHL+] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems.

[inta] *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture.*

[intb] *IA-32 Intel Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M.*

[intc] *IA-32 Intel Architecture Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z.*

[intd] *IA-32 Intel Architecture Software Developer's Manual, Volume 3A: System Programming Guide, Part 1.*

[inte] *IA-32 Intel Architecture Software Developer's Manual, Volume 3B: System Programming Guide, Part 2.*

[KEG+97] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russel Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth MacKenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th Symposium on Operating Systems Principles*, October 1997.

[LABW92] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

[Lan81] Carl E. Landwerh. Formal models for computer security. *Computer Surveys*, 13:247–275, 1981.

[Law01] Greg Law. *A New Protection Model for Component-Based Operating Systems*. PhD thesis, School of Informatics, City University, London, October 2001.

[Lho04] Cédric Lhoussaine. Type inference for a distributed π-calculus. *Sci. Comput. Program*, 50(1-3):225–251, 2004.

[lin] The Linux Homepage at Linux Online. `http://www.linux.org`.

[LMB+] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications.

[Lov05] Robert Love. *Linux Kernel Development*. Novell, second edition, 2005.

[LS77] Lipton and Snyder. A linear time algorithm for deciding subject security. *JACM: Journal of the ACM*, 24, 1977.

[McL94] John McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994.

[MGH+94] James G. Mitchell, Jonathan Gibbons, Graham Hamilton, Peter B. Kessler, Yousef Y. A. Khalidi, Panos Kougiouris, Peter Madany, Michael N. Nelson, Michael L. Powell, and Sanjay R. Radia. An overview of the spring system. In *COMPCON*, pages 122–131, 1994.

[MK97] David Mazières and M. Frans Kaashoek. Secure applications need flexible operating systems. In *Workshop on Hot Topics in Operating Systems*, pages 56–61, 1997.

[MR00] MatthewHennessy and James Riely. Information flow vs. resource access in the asynchronous pi-calculus. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2000.

[net] The NetBSD Project. `http://www.netbsd.org`.

[NSRL06] Prasad Naldurg, Stefan Schwoon, Sriram K. Rajamani, and John Lambert. NETRA: seeing through access control. In Marianne Winslett, Andrew D. Gordon, and David Sands, editors, *FMSE*, pages 55–66. ACM, 2006.

[oD85] US Department of Defense. Trusted computer system evaluation criteria. Technical Report 5200.28, US Department of Defense, 1985.

[ope] OpenBSD. `http://www.openbsd.org`.

[RF] Dickon Reed and Robin Fairbairns. Nemesis kernel overview. `http://www.cl.cam.ac.uk/Research/SRG/netos/old-projects/pegasus/publications/overview.ps.gz`.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 20(2):38–47, February 1996.

[sela] Lwn: An introduction to SELinux. `http://lwn.net/Articles/103230/`.

[selb] Security-enhanced linux. `http://www.nsa.gov/selinux/`.

[Sip06] Michael Sipser. *Introduction to the theory of computation.* Thomson Course Technology, second edition, 2006.

[Ter07] Anders Franz Terkelsen. Preliminary research for LINK: An operating system framework. Technical report, Aalborg University, January 2007.

[TJ98] Nayeem Islam Trent Jaeger, Jochen Liedtke. Operating system protection for fine-grained programs. *Proceedings of the 7th Security symposium (USENIX Association: Berkeley, CA)*, page 143, 1998.

A. Benchmark Results

While programming the small tool to perform the "HTS vs. STS" benchmarks it was straight-forward to make the tool support a few more parameters (e.g. cache settings) than was actually needed for the benchmarks we had in mind. They were included as they still yield interesting information about the CPU, and even though we did not need this extra data, others might, and if so, they hopefully stumble upon this thesis.

Four different general types of benchmarks were made:

- Hardware task switching.
- SYSENTER/SYSEXIT.
- Software task switching with no task state saved.
- Software task switching with with state saved.

Each of these were benchmarked with and without paging enabled and with various combinations of parameters (e.g. cache settings). Explainations of the four general types of benchmarks are given in Section 5.1.

We have divided the results into those with paging disabled, and those with it enabled. The following two sub-sections present both sets of results, respectively.

A.1. Paging Disabled

CR0.CD	CR0.NW	Cycles
0	0	495
1	0	6376
1	1	7861

Table 2: Hardware task switcher benchmarks with paging disabled.

CR0.CD	CR0.NW	Cycles
0	0	392
1	0	6878
1	1	6057

Table 3: Software task switcher benchmarks with paging disabled. The entire task state is saved.

CR0.CD	CR0.NW	Cycles
0	0	275
1	0	1368
1	1	1449

Table 4: Software task switcher benchmarks with paging disabled. No task state is saved.

CR0.CD	CR0.NW	Cycles
0	0	136
1	0	557
1	1	739

Table 5: SYSENTER/SYSEXIT benchmarks with paging disabled.

A.2. Paging Enabled

CR0.CD	CR0.NW	PWT	PCD	Cycles
0	0	0	0	483
0	0	0	1	1232
0	0	1	0	6425
0	0	1	1	6423
1	0	0	0	7782
1	0	0	1	7778
1	0	1	0	7781
1	0	1	1	7782
1	1	0	0	6431
1	1	0	1	6422
1	1	1	0	6425
1	1	1	1	6424

Table 6: Hardware task switcher benchmarks with paging enabled.

CR0.CD	CR0.NW	PWT	PCD	Cycles
0	0	0	0	490
0	0	0	1	1681
0	0	1	0	6500
0	0	1	1	6504
1	0	0	0	7132
1	0	0	1	7123
1	0	1	0	7130
1	0	1	1	7122
1	1	0	0	6502
1	1	0	1	6513
1	1	1	0	6514
1	1	1	1	6510

Table 7: Software task switcher benchmarks with paging enabled. The entire task state is saved.

CR0.CD	CR0.NW	PWT	PCD	Cycles
0	0	0	0	280
0	0	0	1	513
0	0	1	0	1577
0	0	1	1	1571
1	0	0	0	1363
1	0	0	1	1364
1	0	1	0	1361
1	0	1	1	1361
1	1	0	0	1571
1	1	0	1	1582
1	1	1	0	1573
1	1	1	1	1580

Table 8: Software task switcher benchmarks with paging enabled. No task state is saved.

CR0.CD	CR0.NW	PWT	PCD	Cycles
0	0	0	0	134
0	0	0	1	190
0	0	1	0	545
0	0	1	1	543
1	0	0	0	653
1	0	0	1	650
1	0	1	0	653
1	0	1	1	652
1	1	0	0	541
1	1	0	1	539
1	1	1	0	541
1	1	1	1	542

Table 9: SYSENTER/SYSEXIT benchmarks with paging enabled.

B. Setting Up a Test Environment

The benchmark tool and POCLINK both expect to be run in a Preboot Execution Environment (PXE). This means that we need two machines: A server running DHCP and TFTP services, and a client on the same Local Area Network (LAN) as the server. We will now show how to set up a Debian GNU/Linux system to act as a server, as well as how a QEMU virtual machine can be used as client. Using a virtual machine as a client allows one to have a complete testing environment on a single machine and tests can be performed much quicker than with real hardware. QEMU also has convenient interfaces for debugging. As Debian GNU/Linux is our current OS of choice we used such a system for development and testing. The instructions in this appendix should however

be trivial to use on any Linux distribution. We assume that no DHCP server or TFTP server is already available on the system, if however that is not the case you most likely know how to configure these services already and adding the extra configuration needed to support PXE clients should be easy. We assume the reader knows how to operate a GNU/Linux system and for those who do not use the Debian distribution, here is a small note:

Debian uses a package manager called `apt` and it basically works by writing the command `apt-get` followed by the action to perform and then a package name. For instance, the command `apt-get install zsh` would install the shell `zsh`, assuming that you are executing the command with proper permissions.

B.1. DHCP Server

First we need to install a DHCP server.

```
# apt-get install dhcp
```

The `dhcp` package contains a DHCP server, the version we used was version `2.0pl5-19.5`. This was the version available in the "testing" package repository at the time of development. Once the package is installed open the file `/etc/dhcpd.conf` for editing. The configuration used by us looks like this:

```
subnet  192.168.3.0  netmask 255.255.255.0 {
  option  subnet-mask           255.255.255.0;
  option  broadcast-address     192.168.3.255;
  option  domain-name           "fractal";

  host  pxeclient {
    hardware  ethernet  00:aa:bb:cc:dd:ee;
    fixed-address       192.168.3.20;
    next-server         192.168.3.1;
    filename            "poclink.bin";
  }

  host  vmpxeclient {
    hardware  ethernet  12:34:00:00:00:01;
    fixed-address       192.168.3.21;
    next-server         192.168.3.1;
    filename            "poclink.bin";
  }
}
```

This configuration makes the DHCP give out IP addresses on the subnet `192.168.3.*`. It also tells the DHCP server only to give out an IP address the to clients `pxeclient` and `vmpxeclient`, which have the burned in addressess (BIAs, also called MAC addresses), `00:aa:bb:cc:dd:ee` and `12:34:00:00:00:01`, respectively. Only giving addresses to

known clients makes it easier to avoid conflicts if there are other DHCP servers on the LAN.

The `filename` option supplied for each client is the relative path and filename for the Network Boot Program (NBP) that should be sent to the client. The `next-server` option is used to specify the IP address of the TFTP server as it might be different from that of the DHCP server. The above configuration is about as minimalistic as it can be and is not suitable for actually running a PXE server service on a LAN. It however suffices for our test environment.

The file `/etc/default/dhcp` contains a list of devices that the DHCP server should service. Once everything is setup to your liking restart the DHCP deamon by executing the command:

```
# /etc/init.d/dhcp restart
```

B.2. TFTP Server

Let us start by installing a TFTP server:

```
# apt-get install tftpd
```

The version at the time of development was `0.17-15`. When the TFTP package is installed continue to open the file `/etc/inetd.conf` for editing, and add a line for starting the TFTP service. We used the following:

```
tftp dgram udp wait nobody /usr/sbin/tcpd /usr/sbin/in.tftpd /srv/tftp
```

The last directory path given in the line above specifies the base path for the TFTP server to use. The file path and name specified in the DHCP configuration for each client are relative to this path. Restart the network services and the TFTP server will be running.

B.3. Using Real Hardware

Simply make sure that the client machine has its "boot via network" option enabled in its BIOS and everything should be ready. Now you can download the source code for the benchmark tool and POCLINK from `http://www.cs.aau.dk/~zion1459`, compile it using the included Makefiles, and copy the binaries to your TFTP servers base path.

B.4. Using QEMU

QEMU is a free and open source CPU emulator, and it is available through the Debian package repository, so simply install it with the command:

```
# apt-get install qemu
```

We used version `0.8.2-4etch1` of the `qemu` package as the newest version did not work with the configuration explained below.

There are special Linux kernel modules available for QEMU which makes it run a lot faster. If you have the need for speed then install the package called `kqemu-modules-?` where `?` is the version you need. Run the command `apt-cache search kqemu` to see the different available packages. If you have a custom built kernel you will most likely need to compile the `kqemu` modules yourself. Compiling it manually can be automated a bit my using the utility `m-a`, which stands for *module assistant*, which also is the name of the package containing the utility (without the white space of course).

QEMU does have options for simulating a boot via the network but we had no luck making it work and found another solution at `http://tomas.andago.com/cgi-bin/trac.cgi/wiki/QEMUPXE`. This website actually has complete instructions on how to set up a DHCP and TFTP server as well, which we actually used as a reference when setting up our test environment.

There is an open source project called *Etherboot* which among other things has a website at `http://www.rom-o-magic.org` which can be used to create bootable CD images that contains a PXE environment, thus allowing systems without PXE environments to boot via the network using such an image. We can use such a CD image with QEMU. Please refer to `http://tomas.andago.com/cgi-bin/trac.cgi/wiki/QEMUPXE` for instructions on how to create such an image using the rom-o-magic website.

Once you have the Etherboot CD image file you can continue with setting up bridging for your network. QEMU can use a virtual ethernet device (a so-called TAP device) as its network device. To make the DHCP work with this device another virtual device, called a *bridge* is needed. A bridge can have one or more network devices associated with it, making them behave like they are one their own LAN. Then the rest of the system can communicate with all the bridged devices at once by using the bridge device.

Two packages are needed to make QEMU and bridging work. The two packages are called `uml-utilities` and `bridge-utils`. Install them with the following command:

```
# apt-get install uml-utilities bridge-utils
```

The package `uml-utilities` gives us a utility called `tunctl` which we can use to create TAP devices, and the package `bridge-utils` contains the utility called `brctl` which we can use to create and manage bridge devices.

To setup your DHCP server to use the bridge device that we will show how to create shortly, add the device `br0` to the `device` line in the file `/etc/default/dhcp`.

To setup bridging for the DHCP setup described above, the following commands can be used:

```
# brctl addbr br0
# ifconfig br0 192.168.3.1
# tunctl -u username -t tap0
# ifconfig tap0 0.0.0.0 promisc
# brctl addif br0 tap0
# /etc/init.d/dhcp restart
```

The first command creates a new bridge device called `br0`. The second line gives the `br0` an IP address which becomes the IP address used by the DHCP server when communicating with the bridged devices. The third command creates a TAP device `tap0` which can be used by the user named "username". The line after initialises `tap0` with no IP, as it will get an IP from the DHCP server. Finally, line five adds the `tap0` device to the bridge and line six restarts the DHCP server.

Now QEMU can be run with the command:

```
# qemu -cdrom qemu-pxe-boot.iso -m 256 -boot d\
  -net nic,macaddr=12:34:00:00:00:01 -net tap,ifname=tap0
```

Where `qemu-pxe-boot.iso` is the CD image obtained from rom-o-magic.org and `12:34:00:00:00:01` is the BIA for the virtual ethernet device. See the QEMU manual for more information.