Microkernels

Asif Iqbal, Nayeema Sadeque and Rafika Ida Mutia

Department of Electrical and Information Technology
Lund University
Sweden

Abstract—**The minimalistic approach that is essential to the foundation of a microkernel has attracted a plethora of significant implementations for it. This simple and modular kernel with minimum amount of code has translated to the microkernel offering increased reliability, security and great adaptability with the potential to implement a wide range of systems on top of it. Furthermore, its flexibility has made it possible for microkernels to become a foundation for embedded systems. In addition, the microkernel is now being used to host applications requiring different platforms. This paper provides an introduction to microkernels, their advantages over monolithic kernels, the minimum abstractions to realize them and the essential part that it plays in technological developments at present and in the future.**

Index Terms—**microkernel, Interprocess communications, security, embedded systems, virtualization, Mach, L4**

I. Introduction

The kernel is a mandatory part of an operating system and common to all other software. A computer system is designed as series of abstraction layers and each relies on the function beneath it. The kernel is the lowest level of abstraction that is implemented in software. It manages the system's resources, i.e. communication between hardware and software. The kernel acts as a bridge between applications and actual data processing done at the hardware level.

When the boot loader starts loading into the RAM, it starts executing the kernel in supervisor mode. Then the kernel initializes itself and starts the process, later the kernel does not typically execute directly, only in response to external events. In idle mode, the kernel provides a loop that is executed whenever no processes are available to run.

Early operating systems induced monolithic design by kernel mode/user mode architectural approach for protection purposes. In fact, every module needs protection which is preferably to be included into the kernel. The relation between monolithic design and privileged mode can be reconducted to the key issue of mechanism-policy separation. Furthermore, the privileged mode architectural approach combines with the mechanism of security policies leading naturally to a microkernel design.

As to compare, the monolithic kernel is a kernel architecture where the entire operating system is working alone as the supervisor mode in the kernel space while microkernels are the minimized version of the operating system kernels that is structured to specific design principles. Microkernels implement the smallest set of abstractions and operations that require privileges, such as address spaces, threads with basic scheduling and message-based interprocess communication (IPC). Other features such as drivers, filesystems, paging and networking are implemented in user-level servers.

The microkernel design supports easier management of code, since the user space services are divided. This increases security and stability resulting from reduced amount of code running in kernel mode. The advantage of building an operating system on top of a microkernel compared to on top of monolithic kernel is that it offers better modularity, flexibility, reliability, trustworthiness and viability for multimedia and real-time applications. Microkernels mechanisms are paramount to achieving the goals of security. Insecure mechanism is the main problem, but there are other problems that arise; inflexible or inefficient mechanisms.

First generation microkernels were derived by scaling down monolithic kernels. It has drawbacks, such as poor performance of IPC and excessive footprint that thrashed CPU caches and Translation Look-aside Buffers (TLBs). Second generation microkernels, emphasizes more on the performance with a minimal and clean architecture. Second generation of microkernels are Exokernels, L4 and Nemesis. [1]

This paper is organized as follows. Section II briefly explains the characteristics of the first generation microkernels. Their drawbacks are highlighted including the reasons for them and the necessity for the development of second generation microkernels. Both first generation and second generation microkernels are compared, ending with a comparison of different implementations of a second generation microkernel. Section III introduces the basic abstractions of a microkernel and explanations of these concepts. The next section, Section IV, addresses the critical role that a microkernel can play when it comes the issue of security. The following section, Section V, looks at the future prospects of microkernels; which seems bright when it comes to using it as a foundation for implementing secured embedded systems and virtualization environments. The last section provides a conclusion of this paper.

TABLE I
COMPARISON OF COMPILER BENCHMARK

SunOS	Mach 2.5	Mach with Unix Server
49 secs	28.5 secs	28.4 sec

II. EVOLUTION OF μ-KERNELS

In [2] provides an overview of how microkernels have evolved. One of the first generation microkernels addressed was Mach which was developed at Carnegie Mellon University. The most prominent current use of it is as a foundation for Apple's Mac OS X operating system. It hosts some notable features such as a new virtual memory design and capability-based IPC. It's basic abstraction are tasks, threads, ports and messages. It provided a small number of simple operations and consistency when used for UNIX, but this weakened as more and more mechanisms like System V IPC and BSD sockets were added.

Unix was implemented as a user task on top of the Mach microkernel. The Unix acts as a server for facilities like files and sockets while the low level hardware-dependent part is handled by Mach. Advantages include portability since Unix server is not a platform-specific code, network transparency through the use of Mach IPC and extensibility since implementation and testing of a new Unix server is possible alongside an old Unix server. The Unix server consists of two parts; the server itself and an emulation library. The library transparently converts Unix system calls of user tasks into requests to the server. Application programs uses the Mach IPC facilities to communicate with the Unix server. Since this solely was the case for communications, optimizations was carried out and thread scheduling was improved. In addition, the number of messages to the Unix server was reduced by handling some of the Unix system calls directly in the emulation library. Furthermore, previously each C thread was directly mapped into each Mach kernel thread. Optimizations had to be made here as well as high number of kernel threads limited the performance. The thread library was modified so that a number of C threads shared one kernel thread. In [2], certain benchmarks for SunOS, Mach 2.5 and Mach based Unix server were compared, tabulated in Table I.

Both Mach 2.5 and Mach based Unix server outperforms the SunOS in a compiler benchmark; SunOS takes the longest time followed by Mach 2.5 and Mach with Unix server respectively.

However, there are a number of performance issues with first generation microkernels including Mach. One of the most significant drawbacks include limited performance due to the inefficiency of the IPC. Furthermore, a regular Unix system call ($\sim$20μs) has 10 times less overhead than a Mach RPC ($\sim$230μs) on a 486-DX50 microprocessor. In computer science, a remote procedure call (RPC) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. An RPC consists of two IPC messages. Mach also has a peak performance degradation of 66% for certain Unix applications when compared to Ultrix which is a monolithic kernel. Another weakness in performance is the way the external pager was implemented. The pager only has limited control of the main memory and is not suitable for multimedia file servers, real-time applications and frame buffer management that requires complete control of the main memory. Lastly, although early microkernels aimed at reducing functionality, they were still large and implemented many concepts. For example, Mach 3 has over 300Kbytes of code and its API consists of nearly 140 system calls.

TABLE II
COMPARISON OF DURATION OF AN IPC MESSAGE

	Mach	L4
8Kbytes	115μsecs	5μsec
512Kbytes	172μsecs	18μsec

The second generation microkernels are built with the goal to further minimize the concepts to be implemented with a clean and simple architecture. One of the significant successors is the L4 microkernel. Developed by the German National Research Center for Information Technology in 1995, the only abstractions it provides are address spaces, IPC and threads. Its code size is minimal with 12Kbytes and its API consists of just 7 system calls. On a 486-DX50, the duration of an IPC message for L4 and Mach was compared in [2]. The values are tabulated in Table II.

Additionally, the external pager is made flexible since the L4 uses an interface to virtual address spaces which allows memory management policies to be implemented outside of the kernel. Furthermore, the address spaces are constructed recursively outside of the kernel using the three basic operations; grant, map and unmap, which are explained in detail in Section III-C.

Paper [2] also summarizes a paper that evaluates the performance of L4 by running Linux on different implementations of it. It compares certain benchmarks to evaluate the performance of L4Linux with an unmodified Linux and also two versions of MkLinux; one runs Linux as a user process, just like in L4Linux and in the other version, the Linux server is co-located in the kernel. The idea is to investigate the performance of L4 as a foundation compared to the native Linux, whether the performance depends on the microkernel that the Linux is running on and to determine whether co-locating the Linux server inside the server improves the

TABLE III
COMPARISON OF SYSTEM CALL OVERHEAD AND PEFORMANCE IN RELATION TO THE NATIVE LINUX

	System Call Overhead	Performance Compared to Native Linux
Native Linux	1.68μsecs	
L4Linux	3.94μsecs	91.7%
MkLinux in-kernel	15.41μsecs	71%
MkLinux in-user	110.60μsecs	51%

performance. The performance of system call overhead for the native Linux, L4Linux and MkLinux in-kernel mode and MkLinux in-user mode were compared in [2]. The values obtained are tabulated in Table III.

The system call overhead for the native Linux was the least followed by MkLinux in kernel mode and MkLinux in user mode. Furthermore, the MkLinux in-user mode provided the worst performance when compared to the native Linux while L4Linux provided the best. It can be concluded that the underlying microkernel used has an effect on the overall system performance and the co-locating the server inside the kernel can not solve the problems of an inefficient microkernel.

III. SOME μ-KERNEL CONCEPTS

L4 aims at being highly flexible, maximizing performance while being highly secure. In order to be fast, L4 is designed to be small and provides only the least set of fundamental abstractions and mechanisms to control them. To fulfill the high-performance implementation desired, there is a contrast between maximizing the performance of a specific implementation and its portability to other implementations or across architectures. This problem can be solved depending on the specification of the microkernel. The specification is designed to compromise the apparently conflicting objectives. The first, is to guarantee full compatibility and portability of user-level software across a matrix of microkernel implementations and processor architectures. The second is to choose architecture-specific optimisations and trade-offs among performance, predictability, memory footprint, and power consumption.

The L4 microkernel provides four basic mechanisms, i.e., address spaces, threads, scheduling and synchronous inter-process communication. The basic mechanisms are not only used to transfer messages between user-level threads, but also to deliver interrupts, memory mappings, asynchronous notifications, thread startups and preemptions, exceptions and page faults. An L4-based operating system has to provide services as servers in user space that monolithic kernels like Linux or older generation microkernels include internally.

A. Interprocess Communication (IPC) and Threads

In μ-kernel based systems, operating system services such as file systems, device drivers, network protocol stacks, graphics, etc, are not included in the kernel but are supported by servers running in user mode. Thus when an application wants to use any of these services, it has to communicate with the server first. So, all the communication between system services and applications involves IPC to and from servers implementing those services.

Message passing via IPC can be synchronous or asynchronous. In asynchronous IPC, the sender sends a message and keeps on executing, the receiver checks for a message by attempting a receive or by some other notification mechanism. During asynchronous IPC, the kernel has to maintain buffers and message queues and deal with the buffer overflows. It also requires the message to be copied twice i.e., from the sender to the kernel and from the kernel to receiver. This kind of message passing is analogous to network communications. During synchronous IPC, the sender is blocked until the receiver is ready to perform the IPC and vice versa. It does not require any buffering or multiple copies by the kernel.

The first generation μ-kernels, Mach, supported both synchronous and asynchronous IPC and experienced poor IPC performance. Jochen Liedtke, a German Computer Scientist, found that the design and implementation of IPC mechanisms were the reason for the poor performance. He designed a new μ-kernel named L4, a second generation μ-kernel, while keeping high performance in mind, such can be found in [3][4]. IPC mechanisms used in L4 will be discussed later.

1) Threads: A process is the representation of a program in memory; it consists of smaller, programmer-defined parts, called threads. Threads, the smallest unit of an executable code, allows execution of a process in parallel. A thread running inside an address space has its own instruction pointer, a stack pointer and state information containing address space information.

A thread is a basic unit of execution in L4 μ-kernel. L4 threads are light-weight and easy to manage. Fast inter-process communication, along with the concept of light-weight thread are the main reasons behind the efficiency of L4.

A thread is always linked with exactly one address space and an address space can have many threads associated with it. Each thread has a complete set of virtual registers also called Thread Control Registers (TCRs). These registers are static i.e., they keep their values unless changed by the thread. They contain a thread's private data e.g., scheduling information, IPC parameters, thread IDs, etc. The part of the address space where TCRs are stored is called Thread Control Block (TCB). Each TCB is divided into a user TCB and a kernel TCB, accessible by user threads and the kernel respectively.

Each thread is linked with a Page-Fault Handler thread and an Exception Handler thread. They are used to handle page-faults and exceptions caused by the

thread. Each thread also has its own stack but a shared heap with the other threads of the same address space. Each thread has a local identifier and a global identifier i.e., a local identifier is used by the threads sharing the same address space whereas the global identifier can be used by any thread.

2) Communication between Threads: Certain applications or servers rely on other applications or servers for certain tasks. One of the main activities threads engage in is communication with other threads (e.g., request services, sharing info etc). Threads communicate with each others in two ways:

- Using shared memory (Address Space Sharing)
- Using L4's Interprocess Communications

Shared memory is used to exchange larger amounts of data, whereas, IPC is used to exchange smaller messages and synchronization.

Communication using shared memory will be explain such as followed. When two threads, belonging to the same address space, want to communicate with each other, the easiest and most efficient way is to use shared memory. As threads within the same address space already share memory, they do not have to use L4s memory mapping facilities. Provided that both threads agree on the shared memory location or variables to use, they are allowed to communicate in this way. When threads communicate in this way, it is important to provide mechanisms to avoid race conditions, i.e., both threads should never write into a shared location. L4 does not provide mutual exclusion primitives and these should be provided at the user level.

If the threads from different address spaces want to communicate with each other, then both the threads should have access to the same memory region. This is done by having one thread map a region of its own address space into the address space of the other thread. Once this is done, the threads communicate just by reading from or writing to the shared region of the address space.

Another way of thread's communication is by using L4 IPC. L4's basic inter-process communication mechanism is implemented via message passing, allowing L4 threads from separate address spaces to exchange messages with each other. IPC messages are always addressed using the threads unique (local or global) identifiers. Message passing IPC can transfer data between two threads in two ways; either by value i.e., copying of data between the address spaces, or by reference, i.e., via mapping or granting of the address space. L4 IPC is normally synchronous and is also used for synchronization purposes. L4's IPC is used for exception handling i.e., L4 μ-kernel converts an exception fault to an IPC to a user-level exception handler, memory management and interrupt handling by converting the page-fault or interrupt into an IPC to a user-level pager or interrupt handler.

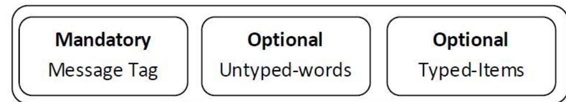

Fig. 1. L4Message Syntax

An IPC message is shown in Figure 1, which consists of one mandatory and two optional sections. [3]

The Message Tag contains the control information (like the size of the message and the kind of data contained within) and a message label (helps in the identification of the message and can be used to encode a request key or to identify the function of the thread to be invoked upon reception of the message). The untyped-words section contains the data which is copied directly to the receiver. The typed-items section contains typed data such as map and grant items.

L4 transfers these messages using Message Registers (MRs). These MRs can be implemented as either special purpose or general purpose hardware registers or general memory locations. The sender writes the message into its own MR and the receiver reads the message from its own MR, and μ-kernel copies these messages into the MRs. Each thread can have 32/64 MRs (based on the architecture). A thread can use some or all of the MRs to transfer a message but the first MR i.e., MR_0 should contain the message tag. Furthermore, MRs are transient read-once virtual registers, i.e., once read, their value becomes undefined until written again. The send phase of an IPC completely reads all the MRs and the receive phase writes into MRs.

A thread only receives those types of messages which it has agreed to. The Acceptor Thread Control Register (TCR) is used to state which typed items will be accepted when a message is received. If the acceptor says that the map or grant items are accepted, then it also tells where the memory should be mapped into the receiver's address space.

Sending and reception of messages is done via IPC system call. IPC is used for inter and intra address space communication. L4's IPC communication is synchronous and unbuffered, i.e., a message is transferred from sender to the receiver only when the receiver has invoked a corresponding IPC operation and it accepts the type of items present in the MRs. Due to synchronous communication without buffering, the amount of data copied is reduced, leading to a high-performance IPC.

A single IPC call involves an optional send and optional receive phase, implicitly specified by the parameters to the IPC call. It is also possible to specify blocking/non-blocking IPCs. Different kinds of IPCs are implemented by different combinations of blocking

and send and receive phases; e.g., IPC with parameters including both a send and a receive phase with no-blocking implements a synchronous IPC that blocks until a reply is received, whereas, an IPC with only a receive phase and no blocking implements a blocking call that waits for a message to arrive.

3) Performance Optimizations for L4's IPC: As we have already mentioned, in L4 μ-kernel, the IPC is not just used to exchange messages between the user-level threads, but also to deliver asynchronous notifications, interrupts, thread startups, thread preemptions, memory mapping, exceptions and page-faults. So due to the high frequency of their usage, any kernel change that increases IPC costs will increase the over-head and thus degrades the performance.

In the following sections, we will discuss two IPC optimization techniques named Direct Process Switching, that avoids running the scheduler during kernels critical paths, and Lazy Queuing, that postpones the updating of its ready queue. Both these optimizations decrease the IPC costs but affect the real-time scheduling behavior of the system in major and minor ways. In the following sections, to demonstrate the two optimizations, we will consider the case of an IPC when one process calls another process or server requesting a service, resulting in caller becoming blocked and the called becoming runnable; the reverse happens as a result of the response. Both these schemes were proposed by Jochen Liedtke in 1993 to improve μ-kernel IPC performance and can be found in [5][6].

4) Direct Process Switch: Generally speaking, whenever a process in an OS becomes blocked, OS invokes the scheduler to select the next process to run based on its scheduling policy. However, if a process blocks during an IPC path, which happens a lot in μ-kernel based systems, invoking the scheduler can be a costly operation impacting IPC performance. So in order to avoid this, L4 switches directly to the newly runnable IPC destination, disregarding the relevant scheduling criteria, such as priority of the threads in the system. There are three main advantages of this approach.

- The overhead involved in the calling of the scheduler, in performance-critical IPC path is avoided. The advantage is obvious.
- The delay of the reaction to the events delivered via IPC is reduced. It allows the service of the interrupt earlier thus improving I/O utilization.
- The cache working set may be reduced (avoiding pollution of cache by the scheduler). A client and server interacting closely can share the cache.

Direct switching makes the real-time scheduling analysis for a specific scheduling policy difficult as the scheduler is not involved in the majority of the scheduling decisions. Moreover, scheduling decisions due to IPCs occur a few thousand times per second, whereas, the scheduling by the scheduler happens a few hundred times per second.[5]

5) Lazy Queuing: When performing a remote procedure call (RPC) with a synchronous IPC, the sender threads becomes blocked after sending the message, and the waiting receiver thread becomes unblocked as soon as it receives the message. This blocking and unblocking of threads result in the manipulation of the ready queue of the system; i.e., the blocked thread must be removed from the ready queue while inserting the unblocked thread into the ready queue. L4 μ-kernel provides two techniques for ready queue management.

- A blocking thread is not removed from the ready queue right away, but its removal is postponed until the scheduler is called. The scheduler then searches for the next thread to run and removes any blocked threads it comes across in the ready queue.
- The μ-kernel preserves the invariant that all the ready threads not currently running remain in the ready queue, while the currently running thread does not need to be in the ready queue. If the running thread changes the state from running to ready, it is added to the ready queue.

Now in L4 μ-kernel, both of these techniques ensure that when IPC results in one thread becoming blocked and another running, short-lived updates are avoided and the unavoidable ready queue maintenance is postponed as much as possible. Finally the ready queue is updated when the scheduler runs because of the time-slice exhaustion or a blocking IPC to a busy thread. The advantages of Lazy Queueing are the saving of direct cost of the queue management, avoiding of pollution of Translation Look-aside Buffer (TLB) entries, and the indirect cost of cache lines polluted by the queue management [5].

B. Scheduling

The L4 provides kernel-scheduled threads with specification of 256-level. The scheduler of L4 kernel resembles monolithic operating systems where all runnable threads are enqueued into a processor-local ready queue. Invocation of L4 kernel scheduler is triggered by timer interrupts, blocking kernel operations and user-initiated thread switches. However, for efficiency purpose, L4 does not invoke scheduler during IPC calls. Instead, it employs a time donation model where the blocking thread passes its time slice to the partner. The thread is scheduled preemptively using the method of Round-robin scheduling. Round-robin scheduling assigns fix and equal portions of time slice in circular order and handle the process without priority.

The length of time slice could be varied between shortest possible timeslice, 'e', and 'infinity'. But the standard length of timeslice is 10 ms. If the timeslice is different from 'infinity', it is rounded to minimum level granularity that is allowed by the implementation. The minimum level depends on the precision of the algorithm used to update it and the verification of exhaustion. Once a thread exhausts its timeslice, the thread is enqueued at the end of the list of the running threads with same priority, in order to give other threads a chance to run. With this algorithm, Round-robin gives some advantages, such as fairness, progress guaranteed, fast average response of time and no starvation occurs since no priority is given [1].

As a comparison, monolithic systems attain a service by a single system call which requires two mode switches, which are the changes of processor's privilege level, whereas in microkernel-based system, the service is attained by sending IPC message to a server and results are obtained in another IPC message from the server. For this, a context switch is required if the drivers are implemented as processes. If the drivers are implemented as procedures, a function call is needed instead.

L4 provides an interface that the parameters can be adjusted in user level programs. L4 arranges threads into a hierarchy by associating a scheduler thread with each thread. In the current implementation, the thread may act as its own scheduler where L4 permits the scheduler to set and adjust the parameters of their subordinate threads, with the restriction that priorities may not exceed the own priority of the scheduler.

Since microkernel-based systems are currently widely-spread, demand for extensible resource management raises and leads to demand for flexible thread scheduling. In [7], the investigation of exporting the scheduling task from kernel level to user level is elaborated with a novel microkernel architecture, which involve the user level whenever the microkernel encounters a situation that is ambiguous with respect to scheduling and allows the kernel to resolve the ambiguity based on user decision.

A satisfactory and qualified modern microkernel scheduler should provide several things such as recursive, user-controlled, priority-driven, efficient, generic and flexible scheduling. In a real-time situation, the scheduler must fulfill some or all the requirements at the same time. FIFO (First In First Out) is closely related to the Round-robin algorithm and in real-time applications it is the most appropriate scheduling algorithm to be used since it simply queues processes in the order that they arrive. FIFO-scheduled threads run until they release the control by yielding to another thread or by blocking the kernel. The L4 microkernel could emulate FIFO with Round-robin by setting the thread's priorities to same level and having the time slice of 'infinity'.

C. Low Level Address Space Management

L4 microkernel's approach to memory management includes placing all the kernel memory logically within a kernel's virtual address space. The address space model is recursive and allows memory management to be performed entirely at user level. All physical memory is mapped within a root address space σ_0. At system start time, all addresses except σ_0 are empty. New address spaces can be created by mapping regions of accessible virtual memory from one address space to another. The μ-kernel provides three operations for constructing and maintaining further address spaces.

- **Grant**, Upon an acknowledgement from the recipient, the owner of an address space can grant any of its pages to another address space. This page is removed from the granter's address space and placed in the grantee's address space. The granter can only grant pages that are accessible to itself, not the physical page frames.
- **Map**, Like the granting, an address space can map any of its pages to another address space provided that the recipient address spaces agrees. Unlike the granting case, the pages are not removed from the mapper's address space and can be accessed in both address spaces. Here also, the mapper can only map pages that are accessible to itself.
- **Flush**, Pages from an address space can be flushed by the owner of the address space. The flushed page remains accessible to the flusher's address space. However, the page is removed from all other address spaces that received the page directly or indirectly from the flusher. Hence, it is important to note that when address spaces receive pages via map or grant it includes an agreement to potential flushing of these pages.

To illustrate the flow of these operations, a granting example is given below:

In Figure 2, pager F combines two file systems implemented as A and C on top of the root address space σ_0 implemented as the standard pager. In the figure, pager A maps a page to pager F which then grants this page to user C. Since the page was granted by F to C, it does not remain in F. The page is available in A and user C and the effective mapping is denoted by a single line on the figure above. Now, if the standard pager flushes the page, it will also be removed from A and user C. If it is flushed by A, the page will only be removed from user C. In either case, F is not affected as it used the page only transiently employing the grant operation avoiding redundant booking and address space overflow.

The kernel also manages device ports. Device drivers are outside of the kernel, but the drivers run as normal applications inside the kernel by allowing ports for device access to be mapped into the address space of the

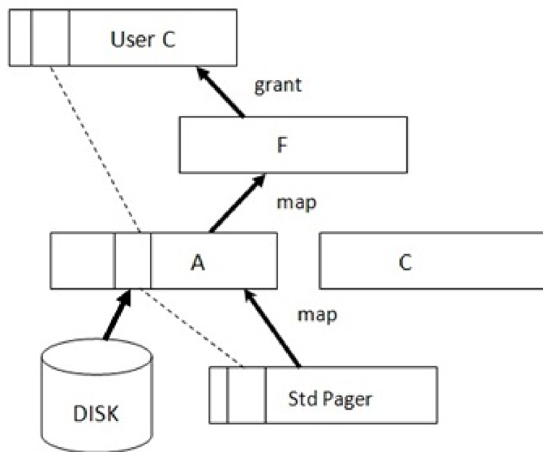

Fig. 2. A Granting Example

applications. Hence, memory managers and pagers are responsible for controlling I/O rights and device drivers on top of the μ-kernel. [4]

IV. Security Aspects of μ-kernels

A system suffers from the lack of the capability to resolve viruses due to the inability of underlying base system to apply and enforce the principle of least authority, which undermines the ability of the higher-level layers to provide security. Some attempts to secure platforms by using scanners, fire walls and integrity checkers are not sufficient to assure the security of a system.

Furthermore, increased dependency on embedded systems has made reliability and security of it to be an issue. Strong security is a fundamental requirement for some existing and emerging embedded application domains and devices, such as personal digital assistants, mobile phones and set-top boxes. To build a secure system, security must be considered in all layers of the system, from hardware to software applications. And security has become a critical component of new embedded devices such as mobile phones, personal digital assistants, etc, since users and providers usually store sensitive data in it.

In addition, extensive use of wireless communication makes the embedded system vulnerable. The movement from low-level communication from hardware to software made the infrastructure of wireless communication potentially vulnerable.

Trustworthiness of a system somewhat depends on its size. A large system with thousands or more Lines of Code (LOC) have inherently more bugs than a small system. It is particularly relevant to the kernel as it manages hardware at the lowest level and it is not subject to protection or fault isolation mechanisms provided by hardware. Bugs in the kernel could lead to fatal error of the system since the kernel is always part of the Trusted Computing Base (TCB). Hence minimizing the exposure to faults means minimizing the TCB, i.e., a small kernel. A trustworthy TCB has to be guaranteed. Microkernel is the solution to this issue. TCB provides a small, secure and highly assured microkernel as fundamental building block. And upon this building block, different system components of security and assurance can be started with strong separation guarantees from other components. A minimal kernel that only contains the minimal code must be privileged, any functionality that can be performed by unprivileged code should remain unprivileged. [8]

The interactions between L4 microkernel applications via IPC are another aspect to be pointed out. Secure communication control should enable both integrity, i.e., unforgeable identity, and confidentiality, i.e., control of information flow. As to deal with the complexity of L4, modularity is implemented to separate the problem into more tractable segments. In contrast to monolithic systems, microkernel-based operating systems are usually realized as a set of cooperating servers where each server is dedicated to its certain task. The separation of task is an advantage for security and robustness of the operating system. Microkernel enforces interfaces between components, in the form of address spaces and the mechanism of Inter-Process Communication to control the communication across address spaces.

Good security properties of microkernels are also beneficial for multi server operating systems where all system servers are encapsulated in address space. Each server is protected physically by the Memory Management Unit (MMU) hardware. If a process illegally accesses another process' memory, it will be detected by MMU and an exception will be raised. A server can not corrupt other servers but dependent servers may be indirectly affected. If server A relies on server B to perform a task, server A may be affected by a malicious action or malfunctioning by server B.

Furthermore, the L4 kernel supports the construction of a small TCB. TCB is actually larger than just the kernel, e.g. in Linux system, every root daemon is part of the TCB. A minimal operating system called Iguana has been developed specially used for embedded systems. It provides essential services such as memory management, naming, and support for devices drivers. The complete TCB can be as small as 20,000 LOC (Lines of Code) whereas a normal LOC for kernel is 200,000 LOC.

In a system without memory protection, TCB is the complete system whereas in a dedicated system, TCB can be very small; a specialized application's TCB would only be the microkernel and underlying hardware. An important security issue is that the components with hardware control are capable to corrupt the entire system. For example, device drivers with Direct Memory

Access (DMA) have the ability to corrupt memory of arbitrary processes by providing an invalid address to DMA controller. Some PCI chip sets have an I/O MMU that protect the drivers by mapping a PCI address space onto a known area of physical memory. However, more general solution for this problem is not yet available. [9]

However, TCB and its size are highly dependent on the functionality that the system provides. Systems with non-trivial user interfaces tend to have larger TCBs, where trustworthiness of the user's input is consumed by the right program is guaranteed. Besides the large size of TCB, another reason of poor match for requirements of embedded system is the model of access control. In Linux or Windows, different users of system are protected from each other but there is no restriction on a particular user's access to their own data. On the other hand, embedded systems which are typically single-user system restrict different programs run by same user to have different access rights, determined by their function instead of the identity of the user. This issue is called least privilege. The operating system runs every program with full set of access rights to the user which violate least privilege. It could lead to severe damage of the system by viruses and worms attack, since full access permits a virus embedded in the application program to destroy the user's files or steal its content.

Microkernel-based systems correct this issue by encapsulating software into components with hardware enforced interfaces where all communication must employ the IPC mechanism. With this mechanism, the kernel has full control over all communications between components and monitors security between components. The program imported into the system is only allowed to access files that have been explicitly assigned to it by the user; hence stealing information could be prevented.

Moreover, a project called secure embedded L4 (seL4) aims to address the inflexible or inefficient control of communication and ensure a suitable platform for L4 to provide secure embedded systems development. The seL4 then provides a secure software base upon which further secure software layers (system and application services) can be composed to form a trustworthy embedded system. It provides minimal and efficient lowest software level and the only part of software that executes in the privileged mode of hardware.

This project explores the usage of neutral capability-based communication to provide L4 with a basic, flexible, efficient mechanism to ensure communication policy. Policy neutral means that the mechanism is customizable by the specific system to match the application domain. Systems could have no communication restriction or be strictly partitioned, depending on the need. The seL4 project also aims to reduce the size of the kernel by simplifying some kernel abstraction where features of the kernel are not generally used by embedded applications. It also has a minimal TCB of 10,000 - 15,000 LOC. [1] Furthermore, it aims for a single-stack kernel to reduce the kernel's dynamic memory footprint. [10]

V. Future of μ-kernels

The L4 microkernel is finding its way to a wide range of applications, one of the most significant of which are its prospects in the area of Embedded Systems. Since the microkernel can be used as a basis for several operating systems, it is ideal for embedded systems that employs memory protection. The L4 can provide a reliable, flexible, robust and secured embedded platform due to its minimalistic approach. Applications running on the microkernel provide system services to normal applications interacting via interprocess communications. This modular implementation makes a system robust as faults are isolated within applications. It also offers flexibility and extensibility since applications can be removed, replaced or added easily. Compared to a microkernel, a monolithic operating system is difficult to assure as it contains all the OS functionality at the minimal level making it larger. Fault isolation is not possible in a monolithic kernel.

On the other hand, the microkernel provides a basis for OS development for a wide range of classes of systems. [11] investigates the possibility to employ the L4 microkernel to the embedded space. It examines its applicability to three application domains in embedded systems; dependable systems, secure systems and digital rights management(DRM). Microkernels has the potential to be the foundation for such applications since for example a system can be constructed from subsystems by partitioning the kernel. This attribute is required for dependable systems which should be able to provide fault prevention, fault tolerance, removal and forecasting. The small core that the microkernel provides can be used to implement security policies that offers confidentiality and integrity. Furthermore, a trusted processor mode can be created which can be separated by hardware from the kernel mode, thereby creating a secure, trusted DRM OS. Software requiring trusted status and authorization can be implemented and managed here.

While the future may look bright for implementing embedded systems on top of the L4 microkernel, there are some limitations that needs to be overcome in order for such systems to be successful. The investigations in [11] have highlighted that the L4's adoption in secure embedded systems is hindered by the lack of an efficient communication control mechanism and poor resource management. The cumbersome co-ordination of name spaces, the denial of service attacks due to lack of resources and the presence of covert channels due to too much resources being available are some of the issues that needs to be improved.

The L4's foundation to embedded devices can be taken one step further and virtualization technology can be

implemented in it as a platform for enhancing security while hosting different kinds of operating systems. An application generating a number of benefits such as load balancing, server consolidation, cross platform interoperability and much more, virtualization is already a prevalent application in the enterprise and in desktops. It provides a software environment in which several guest operating systems can run as if each owned a complete hardware platform.

System virtualization is implemented by a software layer called a Virtual Machine Monitor (VMM), also known as the hypervisor. The technology is now being implemented in embedded systems since it allows for co-existence of different platforms and architectural abstraction and security. By employing virtualization, developers of embedded applications can use it as a security tool by isolating application OSs from the Real Time Operating System (RTOS) which is inherently critical as well as relatively defenseless. The segregation of the guest OSs into individual virtual machines caused by CPU operating modes and memory management units, turns the communication between VMs into network-like mechanisms with explicit channels. [12] further emphasizes that security benefits can only be obtained if the guest OS and its program execution are placed entirely in user mode (non-privileged) and not in system mode (privileged). Hence running the guest OS in system mode, also known as 'OS co-location' increases the chance of the guest OS from going astray and rendering the system unsecured.

In this case, the L4 microkernel can provide significant advantages as a hypervisor, as it is the only code executing in system mode, while other services runs as user-mode servers with unprevileged address spaces. It provides compartmentalization which is a key aspect to tight security and also supports selective and secure sharing of I/O among the guests OSs. Furthermore, a microkernel based hypervisor platform allows for security evaluation and certification by reducing the scope of the Trusted Computing Base (TCB) to easily testable, manageable and verifiable code tests. Further security is achieved due to the parititions and security policies can be implemeted at user-level when communicating across the partitions. The security policies can be changed without changing or re-certifying the kernel. Of all the benefits that virtualization can offer to embedded systems, the greatest advantage lies in enhancing security.

VI. Conclusion

Microkernels only contains the privileged parts of a kernel, thereby reducing complexity of the code making it easier to detect bugs and faults. There are only three abstractions in the microkernel, the IPC, scheduling and memory handling, while device drivers, file systems and such are implemented in user space. Hence the mechanisms provided by the microkernel are basic providing higher security for systems implemented on it. It also allows for modularity of the entire system while supporting a small TCB. With all the amenities that microkernels offers till now, significant advancements have been made with applications in embedded space including virtualization in embedded systems. However, with the improvements in communication control and resource management expected in the near future, microkernels will evidently become the foundation for more innovative systems and applications.

Acknowledgements

We would like to thank our supervisor Martin Hell for his guidance, support and suggestions and also to Christopher Jämthagen for overseeing our work.

References

[1] S. Ruocco, "Real-time programming and l4 microkernels," *In Proceedings of 2006 Workshop on Operating System Platforms for Embedded Real-Time Applications*, 2006.
[2] T. Scheuermann, "Evolution in microkernel design," 2002.
[3] A. Au and G. Heiser, *L4 User Manual*, 2004.
[4] J. Liedtke, "On μ-kernel construction," *In Proceedings of the 15th ACM Symposium on OS Principles*, pp. 237–250, 1995.
[5] D. G. Kevin Elphinstone and S. Ruocco, "Lazy scheduling and direct process switch merit or myths?" *In Proceedings of the 3rd Workshop on Operating System Platforms for Embedded Real-Time Applications*, 2007.
[6] J. Liedtke, "Improving ipc by kernel design," *In Proceedings of the 14th ACM Symposium on OS Principles*, pp. 175–188, 1993.
[7] J. Stoess, "Towards effective user-controlled scheduling for microkernel based systems," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 4, 2007.
[8] G. Heiser, "Secure embedded systems need microkernel," *Journal of System and Software*, vol. 80, 2007.
[9] J. Herder, "Towards a true microkernel operating system," Master's thesis, Vrije Universiteit Amsterdam, 2005.
[10] K. Elphinstone, "L4 cars," *Embedded Security in Cars (escar 2005) Workshop*, 2005.
[11] Elphinstone, "Future directions in the evolution of the L4 microkernel," *National ICT Australia Ltd.*, 2004.
[12] G. Heiser, "Microkernel-based virtualization meets embedded security challenges," Tech. Rep., 2008.