Future Directions in the Evolution of the L4 Microkernel

Kevin Elphinstone

National ICT Australia Ltd.
Sydney, Australia
kevin.elphinstone@nicta.com.au

Abstract. L4 is a small microkernel that is used as a basis for several operating systems. L4 seems an ideal basis for embedded systems that possess and use memory protection. It could provide a reliable, robust, and secure embedded platform. This paper examines L4's suitability as a basis for trustworthy embedded systems. It motivates the use of a microkernel, introduces L4 in particular as an example microkernel, overviews selected embedded applications benefiting from memory protection (focusing mostly on security related applications), and then examines L4's applicability to those application domains and identifies issues with L4's abstractions and mechanisms.

1 Introduction

Microkernels have long been espoused as a basis for robust extensible operating systems. A small, efficient, flexible kernel that provides high assurance as to its correctness provides the foundation of the system. System services are provided by applications running on the microkernel, normal applications receive those services via interacting with the system applications via interprocess communication. Such a system is modular, robust as faults are isolated within applications, flexible and extensible via removing, replacing, or adding system service applications. The efficiency of such a system structure has been demonstrated to be sufficiently close to their monolithic counterparts [9], largely as result of improved efficiency of the microkernel's fundamental primitives [19, 26].

There are strong arguments for applying the microkernel approach to systems constructed in the embedded space. Embedded systems are becoming more powerful and feature the memory protection required to facilitate constructing protected systems, as exemplified by personal digital assistants, digital cameras, set-top boxes, home networking gateways and mobile phones. These platforms are no longer sufficiently resource constrained to warrant a built-from scratch, unprotected construction approach that forgoes the robustness and re-usability of basing development on an operating system.

An operating system for such embedded devices must be modular to ensure its applicability to a wide range of devices. It must the reliable as even in the absence of safety critical or mission critical requirements, embedded systems are expected to perform their function reliably, and usually do not have a skilled operator present to correct their malfunctions. It must be robust in the presence of external and local influences, including those of a malicious nature, given a device's potential presence on the Internet

or the ability to download and execute arbitrary code. It should provide strong integrity, confidentiality, and availability guarantees to applications on the embedded device both to protect data supplied by the user, and data and applications of the manufacturer, or content and service providers.

These requirements are strong motivation for a microkernel-like approach, as opposed to a monolithic approach to constructing an embedded operating system. A single monolithic operating system that contains all OS functionality is more difficult to assure as it is both larger and requires all OS functionality to be assured at the minimum level required for the most critical component. Fault-isolation is non-existent. Inevitable OS extensions make the situation worse, even to the point of allowing a well designed base system to be compromised or malfunction.

The L4 microkernel might provide a capable basis for an embedded operating system. It is both efficient, flexible, and small. It is currently undergoing formal verification [29] which would provide a high degree of assurance of correctness. One of its goals is to provide a basis for OS development for as many classes of systems as possible: "all things to all people". It has been successfully used in systems ranging from the desktop [9], to those with temporal requirements [8], virtual machine monitors [17], to high-performance network appliances [20]. Such broad success is strong motivation for exploring L4's application to the embedded space.

In the paper, we examine L4's applicability to the embedded space, and hence a potential direction in its future evolution. We first provide some background to L4 in Section 2. When the go on to examine selected application domains for embedded systems that would stand to benefit significantly from a protected operating system in Section 3, and summarized important properties required of an operating system in those domains. In Section 4, we critically examine L4's applicability to constructing systems with the identified properties by examining relevant conceptual model in both past and current versions of L4.

2 L4 Background

L4 is a small microkernel that aims to provide a minimal set of mechanisms suitable for supporting a large class of application domains. The basic abstractions provided are address spaces and threads. A classical process is the combination of the two. Interprocess communication (IPC) is the basic mechanism provided for processes to interact. The IPC mechanism is synchronous, threads themselves are the sources and destinations of IPC, not the process (address spaces) that encapsulates them. The IPC mechanism has a basic form and an extended form. The basic form simply transfers up to 64 words between source and destination in a combination of processor registers and memory dedicated to the purpose, with the exact combination being architecture specific. The extended form of IPC consists of *typed messages* sent via the basic mechanism which are interpreted by the kernel as requests to transfer memory buffers or establish virtual memory regions.

Address space manipulation is via the *map*, *grant*, and *unmap* model as illustrated in Figure 1. The figure consists rectangular boxes representing address spaces. σ_0 initially possesses all non-kernel physical memory; A is an operating system server; C and D are

two clients of A. L4 implements a recursive virtual address space model which permits virtual memory management to be performed entirely at user level. It is recursive in the sense that each address space is defined in term of other address space with initially all physical memory being mapped within the root address space σ_0, whose role is to make that physical memory available to new address spaces (in this case, the operating system server A and another concurrently support OS B). A's address space is constructed by mapping regions of accessible virtual memory from σ_0's address space to the next such that rights are either preserved or reduced.

Memory regions can either be *mapped* or *granted*. Mapping and granting is performed by sending typed objects in IPC messages. A *map* or *grant* makes the page specified in the sender's address space available in the receiver's address space. In the case of *map*, the sender retains control of the newly derived mapping and can later use another primitive (*unmap*) to revoke the mapping, including any further mappings derived from the new mapping. In the case of *grant*, the region is transferred to the receiver and disappears from the granter's address space (see Figure 1).

Page faults are handled by the kernel transforming them into messages delivered via IPC. Every thread has a pager thread associated with it. The pager is responsible for managing a thread's address space. Whenever a thread takes a page fault, the kernel catches the fault, blocks the thread and synthesizes a page-fault IPC message to the pager on the thread's behalf. The pager can then respond with a mapping and thus unblock the thread.

This model has been successfully used to construct several very different systems as user-level applications, including real-time systems and single-address-space systems [5, 9, 10, 21].

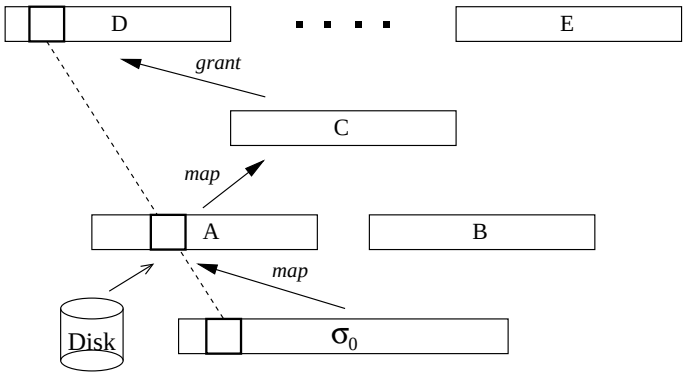

Fig. 1. Virtual Memory Primitives

Device drivers are outside of the kernel. The kernel enable drivers to run as normal applications by allowing the registers (or ports) required for device access to be mapped into the address space (or port space) of applications. Device interrupts are transformed into messages from apparent kernel threads, they are acknowledged by sending a reply IPC to the identity of the sending kernel thread.

We can see that the basic concepts and mechanisms L4 provides are few, while at the same time, are quite powerful enablers of higher-level systems constructed on the kernel. The few concepts and mechanisms (including the lack of device drivers in the kernel) means L4 is relatively small kernel (10,000 lines of code) that could be a highly assured basis of an embedded system.

3 Future Embedded Applications

In this section we examine three application domains of embedded systems: dependable systems, secure systems, and digital rights management. These application domains stand to benefit significantly when the embedded operating system can provide protection between components in the system.

3.1 Dependable Systems

Dependable systems are systems where there is justifiable grounds for having faith in the service the system provides [15]. Dependability is a desirable property of many past, existing and future embedded systems. Methods for obtaining dependable systems can be broadly classified [16] into (or combinations of) the following:

fault prevention where fault occurrence or introduction is avoided,
fault tolerance where expected service is maintained in the presence of faults,
fault removal where the number or impact of faults is reduced,
fault forecasting where the number of present and future faults, and their consequences, is estimated.

As dependable embedded systems are scaled up in terms of overall complexity, the above methods become increasingly difficult to apply. It is essential that it be possible to construct and validate subsystems independently to make the problem more tractable, while at the same time ensure that the validated properties of subsystems remain when they are composed as a whole. This approach is especially applicable to independent subsystems and leads to the idea of *partitioning* or a partitioning kernel [23].

A partition is an execution environment in which an application is isolated from all other activities on the system. One can consider a partition a virtual machine that provides exactly the same level of service to its application independent of other activities on the system. Partitions provide impenetrable barriers between subsystems in order to guarantee fault containment within a partition. If faults can propagate between independent subsystems, the problem of assuring dependability becomes significantly more difficult. One study reported that the length of fault chains (the sequence of faults leading to a failure) was two or more 80% of the time, and three or more 20% of the time [6]. When fault chains can cross subsystem boundaries it creates extremely complex failure modes that should be avoided if possible.

Partitioning can be divided into *spatial* and *temporal* partitioning. Rushby [23] defines them as follows.

Spatial partitioning must ensure that software in one partition cannot change the software or private data of another partition (either in memory or in transit) nor command the private devices or actuators of other partitions.

Temporal Partitioning must ensure that the service received from shared resources by the software in one partition cannot be affected by the software in another partition. This includes the performance of the resource concerned, as well as the rate, latency, jitter, and duration of scheduled access to it.

One method for achieving spatial partitioning is to use hardware-based memory protection available on a processor complete with memory management unit (MMU). The MMU can be used to control access to physical memory to ensure partitions boundaries are enforced. Note that this assumes that the hardware itself is dependable which in some applications (or hardware arrangements) may not be warranted. A partitioning kernel needs to ensure its own memory is inaccessible, and also enforce a partitioning access control policy between partitioned subsystems. This is analogous to secure systems enforcing a mandatory access control policy which is a mature, well understood field of research.

Temporal partitioning is a much more challenging property to enforce. While resource sharing can be minimized by design, some resource sharing is unavoidable, such as processor time, cache memory, the translation look-aside buffer (TLB), etc. Temporal partitioning is related both to scheduling and security. The scheduling discipline has a direct role in processor time sharing, and indirectly to cache and TLB sharing, and potentially on other shared resources (disk, network bandwidth, etc.). From the field of security, the existence of covert timing channels implies the existence of temporal partition violations. Hence, various techniques for identifying covert timing channels (such as shared resource matrix methodology [14]) are applicable for detecting potential violations of temporal partitioning. However, unlike covert timing channels whose utility can be reduced or practically removed via adding noise to the channel [11], temporal partitioning is violated by any external partition-induced variance in temporal observation of a service. It is clear for the security literature that non-trivial covert-timing-channel-free systems have proved elusive, which implies complete temporal partitioning, while extremely desirable, will prove at least equally elusive.

L4 has been examined previously in the context of dependable systems [3]. It was observed that it has shortcomings in the areas of real-time predictability and in communication control. Solutions that address these issues are proposed that improve the situation, but do not completely solve the partitioning problem in the context of L4. The most notable omission from the work is any proposals for kernel resource management. Rather than introduce and analyze L4's partitioning issues here, we postpone the discussion to Section 4.

3.2 Secure Systems

A secure system is a system that can ensure some specific security policy is adhered to, usually expressed in terms of confidentiality, integrity, and availability. Security in embedded systems differs from tradition secure systems in several ways as described by Ravi et. al. [22]. Secure embedded systems are constrained in the processing power

available, and hence there is a trade-off between the strength of cryptographic algorithms employed by a device and the bandwidth of communications. Battery life is limited and thus security-related processing also limits a devices availability. Embedded systems are potentially deployed in hostile environments, which requires tamper resistance to defend against potentially sophisticated and invasive attacks. An embedded operating system is deployed in a wide variety of application domains, resulting in the need to support a wide variety of hardware and software configurations, which in turn make assurance of security properties more difficult.

Security in embedded systems has received renewed interest with the proliferation of personal computing devices such as PDAs, mobile phones, and the like. Embedded systems vendors have to balance the "apparent" power, features and flexibility the software platform provides to device users against the likelihood of devices being compromised by the users themselves or third-parties. The recent Symbian "cabir" worm demonstrates that embedded environments are not immune to the current mayhem that exists in the desktop personal computer market [1]. A substantial change is required from the large, monolithic, feature-rich, OS and application development cycle. An approach that considers security systematically and holistically is required to avoid history repeating itself.

One approach to address some of the issues described above is the small kernel, small components and small interfaces approach. This is another way of stating the *principle of least privilege* and the *principle of economy of mechanism* [24]. The operating system kernel typically has full privilege on the system it supervises. Applying the above principles strongly argues for a small kernel, with a small, well-understood interface, with careful management of the resources it arbitrates over. Such a kernel is also more conducive to high levels of assurance compared to larger, more complex kernels. A small kernel with appropriate mechanisms enabling security-policy enforcement can provide a system basis that warrants a high degree of confidence in operation.

A small kernel does not necessarily provide a secure system. The same principles that motivated the adoption of a small kernel must be applied holistically to the entire system. Such a system would consist of small components implementing well-understood functionality with the minimum privilege required to do so. The components would provide their services through well understood small interfaces. Small components may also provide the flexibility required of embedded system to be deployed in widely varying application domains via component composition, substitution, and subtraction.

3.3 Digital Rights Management

The Internet is enabling new methods of content distribution for the entertainment industry beyond traditional methods such as physical media (DVD, CD), or broadcast or cable TV. However, the Internet is also dramatically reducing the barrier to wide-scale copyright infringement. For content providers to embrace the Internet as a distribution medium, they require confidence that the users of their content adhere to the conditions of use of the content. Conditions of use can be represented by a set of rights the end user receives with respect to the content delivered to him. Ideally, content providers would like guarantees that the set of rights granted for content are enforced, and restricted to

only those authorized. The concept of specifying, enforcing, and limiting rights associated with digital content is encompassed by the term *digital rights management* (DRM). Note that we have used the entertainment industry as the motivator for DRM, however businesses requiring access-right enforcement for their own internal documents also stand to benefit from digital rights management.

To elaborate on the future role L4 might play in the DRM space, an introduction to a typical generic DRM architecture is warranted. Note that many methods can be used to directly or indirectly perform digital rights management (e.g. watermarking), however we will focus on the architecture depicted in Figure 2.

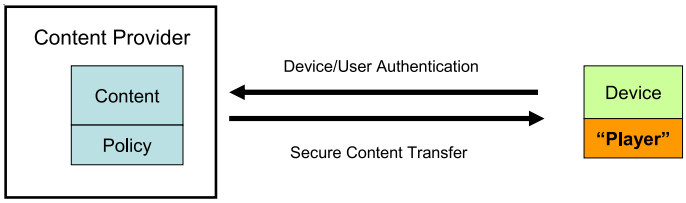

Fig. 2. General DRM architecture

Figure 2 depicts a content provider complete with content and a policy with respect to that content which he wishes respected. The user possesses a device upon which he wishes to view the content. The are many facets to this picture which require solutions prior to the user viewing the content.

- The content provider must be able to specify the policy he wishes respected. XrML [2] is one emerging standard for expression of digital rights, however for the purposes of this paper we assume a policy exists, is expressible, and interpretable by software on the end-user device.
- The user (or user's device) must be authenticated to the content provider. Again, we do not focus on the issue of authentication and simply assume it can be achieved.
- The content (and policy) must be securely transfered to the device. To prevent the content from being stolen by a third party (or even the user itself), the content is usually encrypted to ensure it remains confidential outside the player. Again, we assume this can be achieved.
- The player decrypts the content when viewing is required by the user. The player is expected to honor the content-use policy, not leak the unencrypted content, nor the key to decrypt the content.

We can see that successful enforcement of the content use policy is contingent on the player (where the content is in plain text form) respecting the policy. Content providers have in the past placed their trust in hardware solutions such as satellite TV set-top boxes where their single purpose nature, trusted manufacture, and tamper resistance of the device has mostly proved sufficient to justify the content provider's faith in their ability to honor the content provider's use policy.

One can see that in a general-purpose computing environment, where the user has complete control of the device, the content-use policy can be violated by the end user in

many ways, ranging from reverse engineering the player, modifying the player, or running the player on a modified operating system such that is renders the plain text content to a file, hence the reluctance of content providers to widely embrace the Internet as a distribution medium.

One approach to tackling this problem (as exemplified by Microsoft's recently renamed NGSCB [4]), is to provide the content provider assurance that a trusted player on a trusted operating system is the only software that has access to the plain text content. The fundamental idea is to have tamper-resistant hardware provide direct or indirect *attestation* of the software stack required to view the content. The hardware attests that the software running is what it claims to be (alternatives include hardware only exposing decryption keys to trusted software). If the content provider has faith in the identity of the software stack, it is in a position to determine if is trusts the particular software stack to honor the provider's content-use policy.

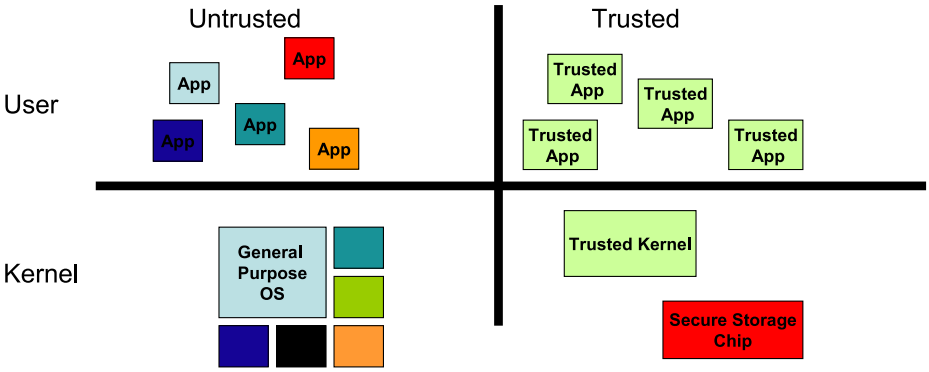

Fig. 3. An example digital-rights-management operating system.

Figure 3 depicts an exemplary OS architecture for DRM that support both a legacy OS with its applications and legacy kernel extensions like device drivers, and a new trusted mode of operation expected to enforce DRM policy. Very briefly, the system functions by introducing a new trusted processor mode that is more privileged than *kernel* mode. Hardware enforces a boundary between the trusted mode and all other modes (including precluding DMA from untrusted mode devices and their drivers). As expected, only the trusted mode kernel can influence what is in trusted mode or not. The secure storage chip provides attestation for the trusted kernel, which can in turn attest to the trusted nature of the applications it support, forming a chain of trust back to the tamper resistant hardware. Therefore, content providers can obtain assurance that the player of their content has a chain of trust rooted in hardware, and hence can expect their content policy to be honored.

A major issue with this approach is that trust is really a label applied to software running in trusted mode, it is not a guarantee that it will always behave in a trusted manner. One would expect that as more and more software acquires trusted status, eventually the trusted partition will approach the size and complexity of the existing legacy system,

unless an alternative construction paradigm is employed. The original motivation for kernel mode was to enable the execution of untrusted applications in a controlled way, trusted mode is little different.

A promising approach to building a secure *trusted* DRM OS is similar to the approach for building a secure system in general — small kernel, small components, and small interfaces, as outlined in Section 3.2. While only those components authorized would be permitted to execute in trusted mode, trusted mode itself should be a secure system in its own right, capable of defending itself against compromised trusted applications. In addition to security, the trusted-mode kernel needs to participate in the attestation process. Given an attested secure kernel, content providers can have a high level of confidence in their content-use policies being honored.

3.4 Summary

We have examined three important application domains for embedded systems possessing hardware memory management functionality: dependable systems, secure systems, and digital rights management capable systems. All three application domains require very similar properties from an operating system kernel for that application domain. A secure kernel capable of enforcing confidentiality, integrity, and availability policy for the kernel services itself might be a capable basis for all three application domains. A small secure and assured kernel when combined with a set of application domain specific operating system components running as applications on the secure kernel is a promising direction to explore in developing a new embedded operating system for future embedded applications.

4 Impediments to L4's Adoption in Secure Embedded Systems

While L4 has been successfully employed as a research vehicle, and as the basis of systems in a variety of application domains, it has not been targeted specifically for secure systems with strong confidentially and availability requirements. Broadly speaking, the current L4 version has serious issues in the areas of communication control and kernel resource management, for which mature solutions are yet to emerge.

4.1 Communications Control

Interprocess communication forms the basis of all explicit interaction between processes running on L4. Note that shared memory regions are established via IPC, hence includes such interaction. To provide a basis for secure communication requires at least:

- control of the set of potential destinations that a process can send to, which implies control of the set of senders a process can receive from. Ideally, knowledge of the existence of other processes is limited to those processes to or from which communication is explicitly authorized.
- An unforgeable identifier must be delivered with the message to enable authorization to be performed in the recipient.

There have been at least five models investigated resulting in implementations in at least 3 cases. The models are *clans and chiefs* [18], *redirection* [13], *redirectors* [28], *virtual threads* [27], and *pclans* [3]. We will briefly examine each in turn and raise issues with them.

Clans & Chiefs The basic system (ignoring clans & chiefs for ease of introduction) consists of threads with in processes. Each thread within the system has a unique system-wide identifier which is used to specify the destination thread for IPC, can be used for authorization of requests in the recipient by receiving the sender's identify. Such a system provides integrity via the unforgeable thread identifiers, however confidentiality policy is unenforceable as any threads can communicate if they can guess the destination thread identity, a small, easily scannable name space.

To control communication flow, clans & chiefs introduces the idea of a *chief* of a *clan*. Every process has a chief statically assigned to it on process creation. The set of processes assigned to a chief is referred to as its clan.

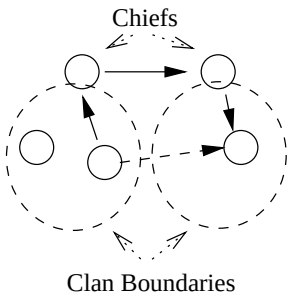

Fig. 4. Clans & Chiefs

Communication within a clan is unrestricted as before. Communication across a clan boundary is redirected to the chief for inspection. The chief can act as a reference monitor and enforce a communications policy between clans. In order to monitor transparently, chiefs are permitted to forge the sender identifier received by the recipient in a controlled way. A chief can forward a message that is redirected to it by impersonating the sender if and only if the apparent source and intended destination lie on different sides of a clan boundary.

In the most general case, each process has its own chief which mediates all communication sent and received by the process. Chiefs in such a scenario can enforce confidentiality policies[1], integrity is based on the integrity of the chiefs a message traverses which is determinable by the eventual message recipient (though normally the monitoring chiefs are within the trusted computing base).

The major issue with clans & chiefs is that of performance. In the general case, at least three IPCs are required between a source and destination: source → source's chief

[1] We acknowledge and ignore for now that thread identifiers are allocated within global name space which could form a covert channel.

→ destination's chief → destination. IPC can be avoided by placing processes with the exact same security classification within the same clan, however, as soon as the classification differs, the processes must be in distinct clans and suffer the penalty of extra IPCs via the chief. A smaller issue with clans & chiefs is that they are assigned statically at process creation, and hence cannot be changed if the security policy modified (and thus modifying the security classification of processes).

Having a chief on the IPC path between source and destination also changes the semantics of IPC, as IPC completion at the sender no longer implies delivery with a chief interposed on the path between source and destination [12]. Proposals to address this issue include postponing sender completion until delivery at the eventual destination which results in an unduely complex IPC model that seems prone to denial-of-service attacks. An alternative is to assume intermediaries are in place for all IPC, which implies delivery acknowledgment IPC in cases where the sender requires notification of successful message delivery. Neither solution is entirely satisfactory.

Redirection The redirection model was proposed and implemented to address the shortcomings of the clans & chiefs model. The model provided a mechanism that enabled for each potential source-destination pair of threads in the system, that IPC between the two could be disallowed, allowed, or redirected to an intermediary which could perform monitoring in a similar fashion to chiefs. One strong advantage of redirection is that it can enforce basic communications control without requiring an intermediary to be in place to forward or discard messages, thus the issues raised previously regarding preserving IPC semantics with intermediaries can be avoided. Another advantage was that redirection is dynamically configurable.

The major issue with redirection is that thread identifiers are still allocated in a global name space which will be prone to covert channels. Another issue is that if intermediaries are required for monitoring, a method for transparently forwarding messages is required. The restriction on impersonation required for forwarding is that an intermediary can impersonate a source to a destination if the intermediary is on the path of intermediaries between source and destination. This check is no longer as simple as the trivial clan & chiefs check, as it requires a search (hopefully short) for membership of a node within a path of a graph. However, in the worst case the length of the path is only bounded by the number of threads in the system. The issue of preserving IPC semantics in the presence of intermediaries, as described with clans and chiefs, also remains.

Redirectors The redirector model has been implemented in *L4Ka::Pistachio*. The basic model is that each process (termed *address space* in Pistachio) has a redirector which can be *nil* or an intermediary. If the redirector is nil, IPC is uncontrolled. If an intermediary is specified, all cross-address-space IPC is redirected to the intermediary independent of the destination. The intermediary can perform monitoring, auditing, debugging, etc.

Redirectors is a simplified model of redirection that avoids the bookkeeping and lookup required to redirect on a source-destination basis. However, doing so requires the single intermediary to handle all monitoring functionality required on any path from a source, as opposed to having potentially separate monitors that enforce security policy,

audit, debug, etc. Typically, the intermediary was a single central OS personality, and requiring a single intermediary was not problematic. Control of communication without an intermediary in place is not possible, unlike with redirection.

Redirectors suffer from most of the issues described for clan & chiefs and redirection. If any communication control is required on a source, an intermediary must be used for all communication from the source. Hence in the general case (assuming a single central intermediary), at least two IPCs are required per message for delivery. Having intermediaries in place changes the semantics of IPC. The check of permitting impersonation to enable forwarding has similar problems to the check for impersonation in redirection, it can result in searching a chain. The name space of thread identifiers is a likely covert channel.

Pclans Pclans is a hybrid between clans& chiefs and redirection. Each pclan has at least one process within it, and each process is a member of exactly one pclan. Within a pclan, communication is uncontrolled. Communication across a clan boundary is dealt with by a model similar to redirection, where an IPC can be blocked or forwarded to an intermediary. The motivation for pclans is based on the assumption that there will be significantly fewer pclans than processes, and hence the redirection table will be significantly smaller. Even if this assumption is true in general, some of the issues associated with redirection remain: covert channels over thread identifiers, semantics of IPC with intermediaries, and the requirement of an intermediary even if communication is permitted to processes external to the clan.

Virtual Threads Virtual threads is a model in which threads are named by virtual identifiers in a processes' local *thread space*, not by global system-wide thread identifiers. Note that processes are not identified explicitly, all communication is between threads whether it is intra- or inter-address-space. Each process's thread space is managed using mechanisms similar to the mechanisms provided to manage a process's virtual memory address space. Access to a thread is given by *mapping* or *granting* a reference to the real thread. The reference to the real thread is placed in and referred to by a location in thread space, its virtual thread identifier. The references to real threads in thread space can be considered IPC capabilities to threads that are indistinguishable in the recipient. Access to the thread can be removed via *unmapping* it.

To distinguish between senders, the IPC call delivers the index of the sender's virtual identifier in the recipient's thread space, if it exists, otherwise the IPC is denied. To speed up the search for the appropriate index and to distinguish between potential aliases of the sender in the recipient, the sender is expected to specify the thread index of itself in the recipients thread space.

Given that virtual threads implements a many-to-one mapping between virtual thread identifiers in thread space and actual instances of threads, one can use it to permit, block, or redirect IPC by controlling the mapping from thread space to threads. By having a local name space, a potential covert channel via global allocation of thread identifiers is avoided.

The main issue with the virtual thread model is requirement for the sender to provide its virtual identifier in the recipient. The coordination of name spaces required is

cumbersome, precludes name space re-arrangement in the recipient, makes transparent insertion of intermediaries problematic, and creates a shared name space between all potential senders to a destination (a potential covert channel). In general, sender provided identifiers violate the principle of encapsulation of implementation of the recipient.

Summary It is clear that existing and proposed communications control mechanisms are unsatisfactory for secure communications control. The proposal closest to being satisfactory is virtual threads, however, requiring sender-provided identifiers required to be valid in the recipient's thread space violates encapsulation of implementation of the recipient.

If the virtual thread model was modified such that the virtual identifiers themselves were distinguishable (not the sender itself), then the distinguished virtual identifiers could be used for authorization. In such a system, we are not actually dealing with virtual identifiers, but distinguished capabilities conveying the right to IPC to a particular thread. Such a capability-based IPC authorization model appears to be the most promising direction to explore in providing L4 with a secure communication model.

4.2 Resource Management

Precisely controlled resource allocation for kernel operations is a requirement for secure system construction. Poor resource management within the kernel can lead to denial of service when resources are exhausted, or covert channels when resource availability is widely visible.

The default resource allocation for L4 is a central allocator that allocates from a fixed memory pool allocated at boot time. The default kernel makes no claims to being suitable for an environment with strong confidentiality or availability requirements. In fact, it is trivial to mount a denial-of-service attack on the kernel-memory allocator.

Given the well-known limitations of the existing allocator, alternative strategies have been proposed. The initial proposal [19] was motivated with the goal of preventing denial of service attacks when executing downloaded web content. Its approach is to introduce a mechanism to provide physical memory (frames) to the kernel for a specific process in the event of resource exhaustion, which I will term the *lend to kernel* model. Examining *map* as an illustrative example, let's assume we have pagers P_1 and P_2, a client C, and σ_0 as illustrated in Figure 5. If C requires a virtual memory mapping established, but has insufficient resources to allocate a page table. P_2's map operation will fail and it then can choose to either deny service to C due to C's insufficient resources, or P_2 can choose to allocate and account one of its own pages (and corresponding frame) to C to supply service to C based on some resource management policy P_2 applies to C. P_2 calls to P_1 with a "lend the chosen page to C" message. Given that the chosen page in P_2 was original supplied by P_1, P_1 can apply a resource management policy to P_2, and if the page donation to the kernel is permitted, P_1 sends a similar request to σ_0 which lends the underlying frame to the kernel. When C is eventually deleted, the frames lent to the kernel on its behalf are freed and are available to the resource pools in σ_0, P_1, and P_2 on demand.

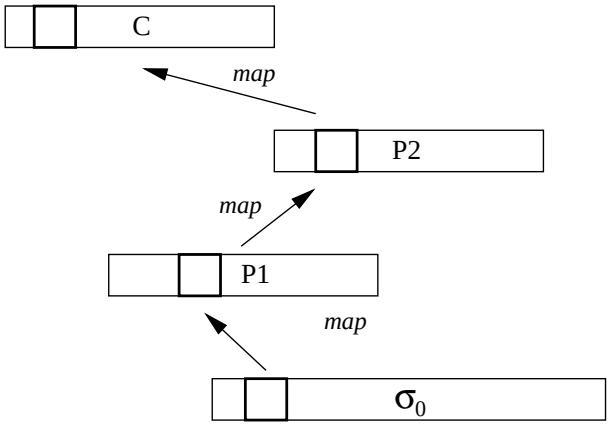

Fig. 5. Example pager hierarchy

The above lend-to-kernel resource management design has several problems. The first one is that revocation of frames allocated to a process is only possible upon deletion of that process, even though the frame contents might be discarded and later reconstructed via redundant data elsewhere. A page table node is the obvious example of such a frame that could be revoked and re-established depending on memory demands. As a result, a long running process will consume its peak kernel memory requirements, not its current requirements.

Another issue is that it is not clear how one could modify the design to limit the actual kernel memory consumption of a process such that its available kernel memory acts a cache of a larger memory space that is paged to disk, i.e. using part of physical memory as a fixed size cache of a process's kernel objects.

Great care is also required in a system based on the above as (i) the resources of a client consumed by a server performing a service on its behalf is only indirectly controlled by the client; (ii) there are no kernel enforced restrictions on who one can donate kernel memory to, it's only limited by resource management policy; and (iii) one does not necessarily have the right to revoke memory given to a client as it requires *delete rights* to the client.

An example of where issue (i) manifests itself is when a client receives a mapping from a server, the address of the mapping indirectly determines the page table requirements. Issue (ii) manifests itself with simplistic resource management policies such as "the OS personality can have as much as it needs, all normal processes are limited to X frames", upon which a denial-of-service attack can be mounted by donating all available memory to the OS personality. Issue (iii) occurs when a server supplies memory to a client (a peer) and does not posses the right to delete the client. This can be avoided by requesting a resource provider with delete rights to provide the memory, but this requires careful resource management co-ordination and accounting which may in turn result in more issues.

The lend-to-kernel model might be workable in theory, and the previous three issues are pragmatic ease-of-use issues, not necessarily flaws, however it remains to be seen how to build a practical system with precise kernel resource management based on the loan-to-kernel model.

A more recent proposal is user-level management of kernel memory [7]. In this proposal, kernel memory is managed on a per-process basis. Each process starts with zero memory and consequently must obtain all memory in a controlled way via the mechanisms provided. The mechanism associates a *kpager* with each thread. The kpager receives faults generated on behalf of a thread by the kernel. Faults (like page faults) are used to signal resource exhaustion of the thread's process and suspend the thread until the fault is resolved. The kpager can chose to supply a frame to the kernel based on a user-level resource policy or deny the request. If the frame is mapped to the kernel, it becomes opaque to the kpager, but still revocable via *unmap*.

Unlike the lend-to-kernel model, the kpager can revoke memory from the kernel. This is achieved via the kernel either zeroing the content if it is redundant, or exporting it back to user-level in a form that can be validated upon return. As an illustrative example of the utility of revocation, a kpager can implement a cache policy for kernel memory by preempting kernel memory (and potentially storing it to disk) and reassigning to another process. Each thread is assigned a kpager, which may be distinct if kernel memory should be managed differently for different concurrent processes, e.g. real-time versus best effort.

The issues with the user-level management of kernel memory proposal include the lack of precision of what the kernel uses the memory for, and the potential for all operations requiring an object to be allocated to block on a kpager fault. Devising a scheme to accurately reflect in the fault the subsequent use of the memory provided to the kernel was left as future work. Without accurately being able to determine the use of the memory, revocation has unknown consequences. Even without considering revocation, the kernel may use the memory for providing a kernel object for which the kpager might have delayed the fault handling (or denied it completely) had it known the eventual purpose of the memory required.

Having a thread block on faults when resources are unavailable creates a denial of service issue similar to that created when memory is copied from one task to another, where a page missing in the source or destination blocks both the source and destination. An example of the problem is when a server maps a page to a client who does not have memory for the needed for the page table node required. A kpager fault is generated blocking the server on a kpager related to the client. The kpager has a indeterminable trust relationship with the server, which leads to the server's reliance on timeouts to prevent the potential denial of service, but timeouts other than *zero* or *never* are problematic [25] and should be avoided as a concept fundamental to the design.

Summary Precise, controlled kernel-resource management is something that has eluded L4 to this point in time. Without a coherent, practical, precise mechanism for kernel memory allocation, L4 will remain unsuitable as a basis for systems requiring strong availability and confidentiality guarantees.

5 Conclusion

One potential direction for L4's future evolution is into the domain of trustworthy embedded systems, as exemplified by dependable systems, secure systems, and some classes of digital-rights-management-capable systems. We have examined L4 (both the current design, and previous designs) for its suitability for supporting trustworthy embedded systems. We have identified two general areas where L4 is lacking: communications control and kernel-resource management. We have examined all existing schemes proposed or implemented (to the best knowledge of the author) for communication control and kernel-resource management for L4 in particular, and no scheme is entirely satisfactory.

Given we have clearly identified what we believe are the major obstacles to L4's adoption in the domain of trustworthy embedded systems, and that we are confident we can resolve these issues, we expect L4 to become an ideal basis for the development of future trustworthy embedded systems.

References

1. `http://www.symbian.com/press-office/2004/pr040618.html`.
2. `http://www.xrml.org/`.
3. M. D. Bennett and N. C. Audsley. Partitioning support for the l4 kernel. techreport YCS-2003-366, Dept. of Computer Science, University of York, 2003.
4. P. England, J. D. DeTreville, and B. W. Lampson. Digital rights managements operating system. US Patent 6,330,670, December 2001.
5. A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J.E. Tidswell, L. Deller, and L. Reuther. The SawMill multiserver approach. In *9th SIGOPS European Workshop*, Kolding, Denmark, September 2000.
6. Jim Gray. A census of Tandem system availability between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4), October 1990.
7. Andreas Haeberlen and Kevin Elphinstone. User-level management of kernel memory. In *Advances in Computer System Architecture (Proc. ACSAC'03), Lecture Notes in Computer Science*, volume 2823. Springer-Verlag, October 2003.
8. H. Härtig, R. Baumgartl, M. Borriss, C. J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schnberg, and J. Wolter. DROPS - OS support for distributed multimedia applications. In *Proc. 8th SIGOPS European Workshop*, Sintra, Portugal, 1998.
9. Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proc. 16th Symp. on Operating Systems Principles*. ACM, 1997.
10. Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software Practice and Experience*, 28(9), July 1998.
11. Wei-Ming Hu. Reducing timing channels with fuzzy time. In *IEEE Symposium on Research in Security and Privacy*, May 1991.
12. T. Jaeger, J.E. Tidswell, A. Gefflaut, Y. Park, J. Liedtke, and K. Elphinstone. Synchronous IPC over transparent monitors. In *9th SIGOPS European Workshop*, Kolding, Denmark, September 2000.
13. Trent Jaeger, Kevin Elphinstone, Jochen Liedtke, Vsevolod Panteleenko, and Yoonho Park. Flexible access control using IPC redirection. In *7th Workshop on Hot Topics in Operating Systems*, Rio Rico, Arizona, March 1999.

14. Richard A. Kemmerer. Shared resource matrix methodology: an approach to identifying storage and timing channels. *IEEE Transactions on Computer Systems*, 1(3), August 1983.
15. Jean-Claude Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Proc. 15th Fault Tolerant Computing Symposium*, Ann Arbor, MI, June 1985.
16. Jean-Claude Laprie. Dependability of computer systems: concepts, limits, improvements. In *Proc. 6th Symposium on Software Reliability Engineering*, October 1995.
17. Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. Unpublished OSDI Submission, 2004.
18. Jochen Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, Kiel, March 1992. Springer Verlag.
19. Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Hermann Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance. In *6th Workshop on Hot Topics in Operating Systems (HotOS)*, Chatham, Massachusetts, May 1997.
20. Jochen Liedtke, Vsevolod Panteleenko, Trent Jaeger, and Nayeem Islam. High-performance caching with the lava hit-server. In *USENIX Annual Technical Conference*, New Orleans, June 1998.
21. Frank Mehnert, Michael Hohmuth, and Hermann Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *Proc. 23rd Real-Time Systems Symposium*, December 2002.
22. S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady. Security in embedded systems: Design challenges. *ACM Transactions on Embedded Computing Systems*, 3(3), August 2004.
23. John Rushby. Partitioning for safety and security: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Also to be issued by the FAA.
24. J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), September 1975.
25. Jonathan Shapiro. Vulnerabilities in synchronous IPC designs. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, 2003.
26. Jonathan S. Shapiro, David J. Farber, and Jonathan M. Smith. The measured performance of a fast local IPC. In *Proc. 5th Int'l Workshop on Object-Orientation in Operating Systems*, Seattle, WA, 1996.
27. Espen Skoglund. Confinement, virtualization and rights delegation using virtual threads. Personal Communication, 2004.
28. L4Ka Team. *L4 eXperimental Kernel Reference Manual*. System Architecture Group, Dept. of Computer Science, Universität Karlsruhe, revision 6 edition, October 2004.
29. Harvey Tuch and Gerwin Klein. Verifying the L4 virtual memory subsystem. In Gerwin Klein, editor, *Proc. NICTA workshop on OS verification 2004, Technical Report 0401005T-1*, Sydney, Australia, October 2004. National ICT Australia.