Management of Revocable Resources

Romain Treboux – `romain.treboux@centraliens.net`

February 26, 2010

Declaration
I declare to have written this work independently and without using unmentioned sources.

Erklärung
Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, 26. Februar 2010

Romain Treboux

Abstract

In microkernel based systems, applications use complex server hierarchies that implement modern operating system services. A server that consumes its own resources to provide its service breaches the isolation between its clients and is vulnerable to Denial-of-Service attacks. To avoid these problems, letting the clients provide the resources needed to satisfy their requests is a promising approach.

In this diploma thesis, I consider the case of memory. I discuss conditions under which a server can provide a service using memory from its clients. I present two mechanisms that allow tasks to securely trade memory for a service. In the first one, clients directly donate resources to servers. Donation revocations trigger faults that are exported to the server application logic. The second approach requires a third entity to intermediate between the client and the server. This entity also can consolidate memory exchanges to allow trade with a sub-page granularity, which helps optimizing memory usage.

Contents

Chapter 1

Introduction

Contents

1.1 The L4 Microkernel

Modern operating systems (OS) use hardware facilities to enforce isolation between concurrent applications. A part of the operating system —the kernel— runs in the most privileged mode (called kernel mode), while applications run in an unprivileged mode (called user mode). Some CPU instructions, registers and memory pages (depending on their access rights), are accessible only in kernel mode. An application that needs to use such resources issues a system call, so that the kernel acts on its behalf or denies the access.

Microkernel-based systems are built on top of a minimal kernel, that is, a kernel designed following the minimality principle defined in [Liedtke 95]. According to the minimality principle, the kernel should limit itself to functionality that cannot be equivalently implemented in user mode, because doing so would lead to a security breach. However, many microkernels do not fully comply to the minimality principle for performance reasons[1]. The fundamental abstractions implemented by the microkernel allow user-level tasks to provide usual OS services. These tasks are subject to kernel-enforced security restrictions.

The rationale of the microkernel approach is to make the kernel less error prone by keeping it small, and to make resource management more flexible by exporting the management policies to the applications. Aside from increased kernel reliability, the microkernel architecture fosters isolation between user-

[1]For example, the scheduler is part of the L4::Fiasco microkernel

level tasks: Tasks must not interfere with one another except by interprocess communication (IPC). This is the independence principle defined in [Liedtke 95].

L4 is a second generation microkernel initially designed by Jochen Liedtke. First-generation microkernels (such as Mach ([Mach Web page]) and Chorus ([Rozier *et al.*])) were deemed too complex and not efficient enough for real use ([Liedtke 93]). Liedtke initially wrote L4 in x86 assembler with performance as a primary design objective. The L4 kernel has later been reimplemented in other languages and for other platforms. Widely known implementations include L4/MIPS at University of New South Wales (Australia), L4Ka::Pistachio at the University of Karlsruhe, and the fully preemptive L4::Fiasco implementation at the TU-Dresden, which I used for this work.

The abstractions that L4 implements are address spaces, threads, and a message-passing inter-process communication (IPC) mechanism. The efficiency of IPC operations, which are used for both communication and synchronization, is crucial for the global system performance ([Liedtke *et al.*]). The L4 IPC primitives are synchronous and blocking: both the sender and the receiver may have to wait (with a blocking system call) for the other side to be ready to exchange a message. A thread may wait for a message from a given partner (closed wait, or receive) of from any partner (open wait). A thread waiting to send a message must specify the recipient thread with a thread identifier. When two threads wait for the same IPC operation (we say that the rendezvous takes place), the microkernel delivers the message and both partners continue execution. L4 IPC is rendezvous-based (that is, a rendezvous takes place before the message passing) and non-buffered (as there is no asynchronous delivery, there is no need to buffer messages).

L4 IPC allows threads to exchange memory access rights as well as data. IPC operations that transfer memory access rights are called flexpage IPC[2]. To initiate a flexpage IPC, the sender issues an IPC system call that specifies which memory pages of its own address space it wants to share with the receiver. Likewise, the receiver-issued system call specifies a destination region in its address space (the receive window). When the IPC rendezvous takes place, the kernel modifies the receiver's page table in such a way that the memory pages of the sender task appear in the receiver's window. We say that the sender maps its memory in the receiver's address space. The sender can later revoke this mapping with the unmap system call.

1.2 Microkernel-Based Operating Systems

In microkernel-based systems, user-level tasks implement traditional OS functionality (for instance, file systems or device drivers). Therefore, these systems often apply the client-server concept intensively. A task providing a service is called a server, the tasks using this service are called its clients. Client-server communication relies on inter-process communication (IPC).

An OS personality is a set of tasks that offers the functionality of some OS. L4Env ([TUD OS group]) is an OS personality that runs on top of the L4 Microkernel (see Section 1.1). L4Linux ([L4Linux Web page]) is another OS personality, which emulates Linux on top of L4Env. Several OS personalities

[2]Flexpage, a portmanteau from flexible and page, is the name of the data structure that describes virtual address space regions in L4.

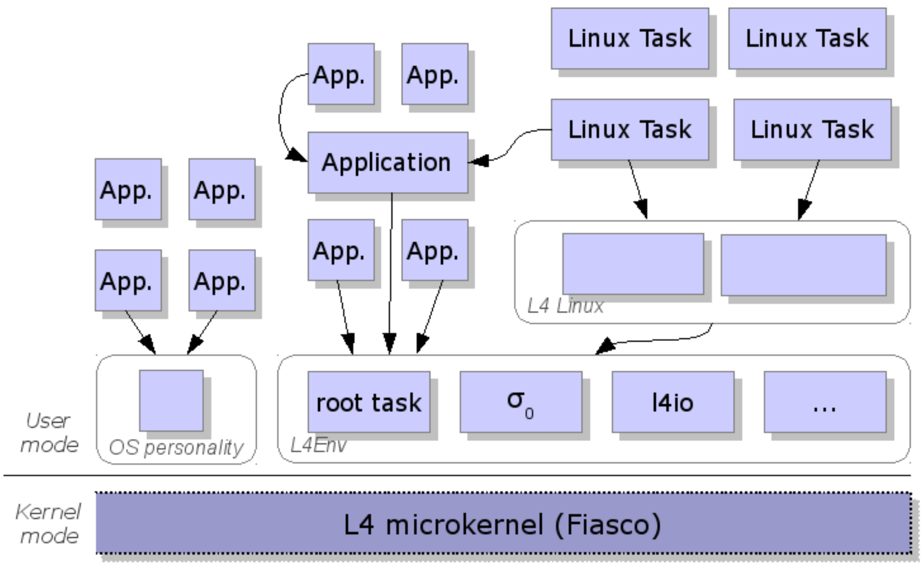

Figure 1.1: A Microkernel System with Three OS Personalities.

User tasks form a hierarchy. Tasks that use different personalities can communicate through IPC. Server-client relations from a complex service hierarchy.

may run concurrently on one system. Each subsystem applies policies that fit its particular needs. DROPS ([Härtig *et al.*]) demonstrates how this architecture allows untrusted legacy Linux applications and real-time applications to coexist.

Microkernel-based systems must divide resources (memory, CPU time, disk space, I/O bandwidth, network bandwidth ...) between several subsystems (for instance, concurrent OS personalities). They can allocate resources following a quota-based policy –no task can use more resources than its beforehand allotted quota– or with an admission test –the system monitors the global resource usage, and decides whether remaining resources are sufficient to satisfy incoming resource reservation requests. The Bastei OS architecture ([Feske, Helmuth 06]) is an example of a quota-based memory allocation system. The DROPS operating system uses an admission system to handle CPU reservation requests ([Hamann *et al.*]). The system must multiplex resources in a way that enforces isolation: A task must not be able to slow down, block or otherwise interfere with the execution of another subsystem, for instance by monopolizing resources in an unintended fashion. Enforcing isolation is the responsibility of all components that are common to several subsystems, user-level servers as well as the microkernel ([Hand]).

1.3 Rationale

In a multi-server system, applications requesting services from a server (IPC call) or from the kernel (system call) use resources indirectly. There often are several indirection levels because many servers use other services while processing a request. However, as explained in [Hand], shared usage of a server can break the isolation between tasks, because the tasks actually use (and depend on) the server's resources to perform their duties. The relevant information for accounting is the beneficiary of a resource, rather than the agent using it. An accurate accounting system should bill the service memory usage to the end user, rather than to the server. Such an accounting system helps enforcing independence principle: an application can deplete its own resources only, and cannot delay independent applications. The design of L4.sec ([Kauer]) applies this idea (each application should pay for what it directly or indirectly uses) to kernel memory management.

Moreover, diverse applications may have diverse resource requirement and resource usage patterns. For instance, some may have real-time requirements that do not allow some of their memory pages to be swapped out to disk, while other may require huge amounts of memory for which swapping make sense. Some applications may want to tailor how much memory they spend to cache data. Applications have specific knowledge, and thus can often take better decisions than the kernel or a server could. For instance, an application can predict which memory pages it will or will not write in the near future with more accuracy than common page replacement algorithms such as LRU[3] do. The server cannot easily integrate application-specific policies[4].

A robust way to implement accurate accounting is to let the client hand out the resources needed to provide a service to the server: The client trades its resources for a service. In such a system, the client would provide the memory that the server uses to provide the service. The server does not allocate any memory objects from its own memory pool on behalf of clients. Rather, it allocates them from client-backed memory. However, the server would have to deal with client-initiated resource revocations: When a session ends, that is, when a client does no longer need a service, the server must free the associated memory, or the memory may leak. The server memory footprint would not depend on the number of opened sessions: each client would provide the memory that the server uses on its behalf. The client would still retain some control over its memory usage because it is able to revoke the memory donations —at the price of a service degradation or disruption.

The goal of this work is to devise solutions that allow such memory trading, and to study the effect on server design. These solutions must include mechanisms with which clients can transfer memory access rights to the server, as well as mechanisms to revoke these rights. They should allow the server to securely use client-donated memory for most or all of its dynamic memory needs without hampering the robustness and the security of the system.

1.4 Related Work

The microkernel approach has triggered many efforts in the domain of user-level memory management. In this section, I shortly describe some research that relate to my subject.

[3]The LRU (least-recently-used) swapping algorithm bases on the assumption that a page that has been recently accessed is likely to be accessed again soon.

[4]Operating systems often resort to heuristics instead, many examples of which are found in Linux (I/O scheduler, CPU scheduler, memory swapping, see [Love]).

1.4.1 Capability-based Systems

Capability-based systems represent memory access rights through capabalities. By transfering these, tasks can implement a form of memory trade. The EROS system ([Shapiro]) uses an abstraction called a spacebank to represent a memory region. Client tasks invoke operations on spacebanks through capabilities that they can exchange. A single thread owns all storage space and handles all memory allocations. It implements a hierarchy of logical spacebanks (client tasks are not aware that all spacebanks are implemented by a single thread). Spacebanks are cheap to create and to destroy and can enforce allocation quotas. Each spacebank obtains memory from its parent. When the spacebank is destroyed, objects allocated from it are either deallocated or go under its parent's control. As the EROS system implements transparent persistence, all memory is mapped to disk. EROS views the physical memory as a cache for disk-stored objects. Objects allocated from a given spacebank are stored in contiguous region of the underlying disk. Thus, spacebanks help preserving locality properties from virtual memory regions to disk storage.

L4.sec, a version of L4 extended for security ([Kauer]), introduces the concept of untyped memory. Untyped memory has the advantage of separating memory control from access rights. To obtain a kernel service that requires memory allocation (such as the creation of a kernel object), a task must give a capability for untyped memory to the kernel. The kernel can then type this capability to create a fresh kernel-memory capability from it. A capability can be typed only once (unless it is later untyped and then typed again). Although the task supplied the kernel memory, it cannot access it. L4.sec uses this mechanism for kernel memory management. seL4 (for secure embedded L4, see [Elkaduwe *et al.*] or [Derrin *et al.*]) uses another type hierarchy, which includes untyped memory, user memory and kernel memory. seL4 extends the capability-typing mechanism for memory exchange between user-level tasks. Typed capabilities allow the exchange of memory access rights like the L4 inter-process communication does. Their advantage is that the task that donated the memory does not retain any access right to it (while still being able to revoke the donation). In comparison, the L4 memory mapping mechanism that I used in this work requires that the mapping task shares the access rights that it donates with the receiver.

1.4.2 Per-Client Memory Management

Some experimental operating systems such as CuriOS ([David, Chan *et al.*]) and the Bastei OS architecture ([Feske, Helmuth 06]) allow to build servers that use client memory, or at least memory that is conceptually assigned to a client, although it is used by the server. This approach implies that the memory of a server must be partitioned between its clients, as I discuss in Section 2.3. The rationale for such a partition can be fault isolation (as in CuriOS) or an accurate accountancy (as in Bastei).

The CuriOS operating system achieves fault-tolerance and enhances client isolation through server state regions (SSR). Each server of the OS devotes a memory region (a number of contiguous pages in its virtual memory) to each of its clients. These server state regions reside at the SSR manager. Whenever a client makes a request to a server, the SSR manager maps the relevant state region in the server memory, and unmaps it when the server has processed the request. This ensure that a software or hardware fault during a request cannot corrupt the server state related to other clients: Only the state of the requesting client is accessible during the request processing. Moreover, when a server crashes, it can restore its state with the help of the SSR manager. The server must be built in such a way that it can reconstruct its entire state from its state regions data. Thus, the SSR manager increases isolation between clients and the overall system robustness. The SSR manager fullfills a role comparable with the memory broker that I present in Section 3.2. However, CuriOS limits this architecture to trusted OS servers. Moreover, the memory is not provided by nor accounted to the clients on behalf of which the servers act.

In the Bastei OS architecture ([Feske, Helmuth 06]), user tasks are part of a trust hierarchy in which they can trade memory quota. The Core task, created at boot time, is the root of the tree, a parent task has control over its child tasks (which it created) and can destroy them. Every child trusts its parent. When creating a child, a task endows it with resources from its own quota. To open a service session (and use the service), a client must give the needed memory quota to the server. When the session ends, the server must give the quota back. The parents carry out the quota trade. If a child misbehave and does not transfer quota as required (for instance, a server does not free memory when a session ends), the parent can choose to kill the child, pay for the child with its own quota, or do nothing. The process goes on recursively, until a common parent of the server and the client arbitrate the conflict. Each subsystem is represented by a subtree of the task hierarchy. Bastei introduces the idea of quota trading (exchanging memory credits rather than memory), which the memory broker can implement as well (see Section 3.2.2), albeit with a different trust model.

1.4.3 Application-Specific Policies

To design servers that can fulfill their client's heterogeneous requirements, it is desirable to export policies from the server to the clients[5]. The Nemesis operating system ([Hand]) examplifies a concrete application of this principle to server design. The goal of NemesisOS is to run applications with varying requirements (real-time applications called Continuous Media, and best-effort applications). It avoids QoS cross-talks (interference between applications with different Quality-of-Service requirements) and allow each application to specify which hardware memory pages the kernel should remove when memory become scarce. NemesisOS guarantees a minimal set of resources to each application and introduces the concept of application self-paging: Each application is responsible for handling its own page faults. There are no external pagers to avoid race conditions while accessing the paging service: External servers are not aware of application deadlines and cannot schedule requests correctly. Therefore, applications *must* handle their page faults with their own resources (while in most microkernel systems, they *can* handle their page faults, but still can delegate this task to the underlying environment). The kernel allocates memory pages to each application. The application specifies which pages the kernel should remove first if memory become scarce: Applications can use their specific knowledge to estimate the cost of a page fault on a given memory region more accurately than

[5]Similarly, microkernel systems that export policies from the kernel to user-level servers offer more flexibility and better isolation than traditional systems.

the kernel can. Likewise, memory donation between two L4 tasks can bring policy specifications with it. In the direct mapping approach (Section 3.1), the client can remove memory pages and map them again at will. The memory broker model requires more elaborate mechanisms for policy specification.

Chapter 2

Design

Contents

In this chapter, I examine the memory utilization of servers. A server makes assumptions about the memory that it accesses, depending on the way it employs it. Therefore, a server should have recourse only to memory with given properties for given purposes (for instance, a disk driver needs DMA-addressable memory if it uses DMA). The mechanisms for memory trading must guarantee the properties of memory pages that servers expect from client-backed memory.

I will define the properties relevant to a memory trading system in the next section: These are the properties that memory *can* have. In Section 2.2, I will outline two patterns of memory usage commonly found in servers. I also will identify the constraints that each pattern entails: The properties that memory *must* have to be exploitable in these scenarios. I will conclude the chapter with a remark on the hypothesis that underlies any memory accounting system: that every memory object can be ascribed to a consumer.

2.1 Characterization of Memory Resources

2.1.1 Properties Independent of Mapping

Some memory properties are independent of the mechanisms that the server uses to trade memory, and to map memory in an address space. For instance, when memory is divided into classes according to intrinsic hardware properties (for instance, in systems with non-uniform memory access, or if only a fraction of the system memory is DMA-addressable), no mechanism can change the class of a given memory page. More generally, a memory trading mechanism maps memory in the server virtual address space. It cannot change any property that does not relate to the mapping. In such a case, the client has the responsibility to make sure that the pages that it donates possess the wanted property in the first place —and the server might have to check it.

2.1.2 Confidentiality

Some data are confidential towards external tasks, that is, only the client and the server should have access to them. Both the server and the client, rather than the memory donation mechanisms, have the shared responsibility to keep such data confidential, or *confined* (in the client-server relationship). The client must trust the server that it will not disclose any shared information (especially to other clients). Likewise, if the server shares data with the client, then the sharing mechanism alone cannot prevent the client from communicating this information.

If the server use client-donated memory pages, the client may retain access rights to pages that the server uses (depending on the donation mechanism used). Pages that contain data that are confidential with relation to the client backing the region are *confidential memory*. Some server data (a private encryption key, for instance) are intrinsically confidential and must not be read, even by the client to which they relate. Besides, the private data representing the server's state (for example, session data or state of requests currently under process) could be useful to a potential attacker while being virtually useless to a honest client. Although these data are not sensitive per se, they still are confidential and must be hidden from clients.

Confidentiality can be ensured either by encryption or hardware protection, that is, by setting the memory access rights of the page to prevent the client from accessing confidential memory. The former solution incurs a performance penalty. The latter solution limits the granularity of the memory region (discussed in Section 2.1.6) to the hardware protection unit, the page.

2.1.3 Availability and Integrity

Data corruption or loss has varying consequences depending on the region in which it occurs. An error in the client's data (for instance, the arguments of a request or the contents of a video frame buffer) would be noticed by the said client only, but would not crash the server nor interfere with other clients. The server does not need to care about the availability of memory regions that contain such data. On the other hand, loss or modification of the server's internal data is likely to crash it or cause an erroneous behavior. The memory trading system must guarantee the *availability* (that is, that the server can access the data) and the *integrity* (that is, that the client cannot alter the data without the server noticing) of client-backed memory regions that are essential for the server to work correctly.

Cryptography can assert the data integrity but cannot enforce availability. The server can ensure that its administrative data will always be accessible by either replicating it (a discouragingly expensive approach) or relying on pre-revocation signals (see Section 2.1.5). Swapping out memory pages does not threatens availability (but it does threaten real-time availability, see the next section): Even though a swapped page is not immediately accessible (because accessing it triggers a page fault), the server still can access the page data after some time (when the pager resolves the page fault).

2.1.4 Real-Time Availability

If the service must satisfy real-time constraints, then delays due to memory page faults and swapping may not be acceptable. On the other hand, enforcing strict real-time guarantees for all clients may require an overly conservative and costly design. To satisfy various timeliness requirements, the server should provide mechanisms that allow a client to control the server paging strategy. The server can do this by uploading, checking and executing a client-defined policy. This approach requires a language to define possible policies that the server should be able to interpret. The server must ensure that uploaded policies comply to its own security policies before applying them. Alternatively, the server can use clients as dataspaces and page fault handlers. In that way, each client can tune the paging policy of server memory that it backs to its needs (and pays the associated cost). For example, a client could pin memory, avoiding page faults at the cost of higher physical memory usage, while another could reduce its memory footprint with swapping. However, the client would have access to the server data, and availability would not be ensured as nothing forces the client to actually resolve page faults. In any case, the guarantees that the server offers depend on the guarantees that the client paging strategy provides: Whether a page fault can delay the request or not depends on the client.

2.1.5 Revocation Terms

A requisite of the trading system is that the client should be able to revoke the resources granted to a server, so that it retains control over its resources. This means that some memory pages will vanish from the server address space, usually through an unmap operation. The terms of the memory revocation depend on the trading system: A trusted entity can promise to send one or more messages (called a pre-revocation signal) before the memory is unmapped, or the unmapping could happen without any warning. In the later case, the data availability (see Section 2.1.3) cannot be guaranteed, and the server must be able to deal with missing memory pages. In the former case, a *revocation signaling protocol* must specify the message sequence and semantics.

2.1.6 Granularity

If memory trading is implemented using hardware access rights, the memory protection mechanisms constrain the size of a client-backed memory region to be a multiple of the hardware page (on the x86 platform, at least 4 KB). For smaller memory regions, this coarse granularity may entail a significant memory waste due to internal fragmentation (unused memory inside an allocation unit). While memory quotas can have an arbitrary fine granularity, they still must be translated to pages.

2.1.7 Interchangeability

Interchangeability is the property of a good whose individual units are capable of *mutual substitution.* This property has a strong influence on the way the good can be exchanged, because we can reason on quantities of an interchangeable resource, while we cannot add or subtract non-interchangeable things. Moreover, managing interchangeable resources brings flexibility: It is easier to find a free page than to free a given page, as the compensated pinning algorithm described in [Liedtke *et al.* 99] demonstrates. In this example, to pin memory pages during a critical section of a given duration, an application must prepare a pool of replacement pages (that are swappable) before entering the critical section. The kernel memory management system is modified so that if the Linux swapping algorithm selects a pinned page to swap out, a replacement page is swapped out instead. Hardware memory pages are interchangeable from the point of view of the Linux swapping algorithm.

While memory pages can appear interchangeable at first glance, this is not always true. Due to addressing limitations, pages in low memory (that can be used for DMA) cannot be replaced with pages in high memory. Shared memory pages are not easy to replace either.

Memory itself (with a finer granularity than the page) is interchangeable in the absence of external fragmentation only (unallocated memory fragments too small to be usable): the server sometimes will not be able to exchange two memory chunks of the same size situated on distinct pages if free memory is fragmented.

2.2 Server Memory Layout and Usage Patterns

An examination of user level servers found in microkernel systems and of a traditional OS ([Feske, Helmuth 05], servers from L4Env [TUD OS group], [Love]) led to the identification of two basic patterns of memory usage. Servers use memory either to keep track of their clients and of session-related data, or to exchange voluminous data with a client without copying them.

2.2.1 Shared Memory

A server may share memory pages with one (or more) client(s) for cheap broadband communication. Both client and server use the same data. Servers handling disk files or video frames commonly use shared memory. Per definition, shared pages contains data which both the client and the server are entitled to access. Confidentiality and integrity of these data is therefore not a concern for the memory trading mechanism (although it may be a concern for the client and the server, in relation with other tasks). Corruption of shared data may disturb the service to the sharing client, but will neither prevent the server from running nor interfere with other clients.

However, such data cannot be displaced in other memory regions like private (not shared) data can, because the sharing partners have agreed on which frame holds the data. Hence, shared memory is not easily interchangeable. Moreover, data sharing is restricted to the hardware protection granularity: in fact, memory pages rather than memory are shared.

Shared memory is used to manipulate data sets too large to be conveniently (or efficiently) copied, so that it usually makes an important part of the server memory usage. Backing shared memory regions with client memory is necessary to prevent Denial-of-Service attacks ([Feske, Helmuth 05]).

2.2.2 Administrative Data

Stateful servers hold session data about their clients and may group them into a global data structure. Some of these data are confidential (for instance, thread identifiers of clients), and the integrity of public data (for example, the names in a directory service, which are readable for at least some clients) still must be guaranteed. Corruption is likely to cause arbitrary errors, while data loss (either because a client overwrite the data or because it unmaps a page, preventing the server from accessing it) would crash the server. Administrative data are private to the server and can be displaced at other locations, at the cost of copying and updating pointers to the migrated data structures: The backing memory is interchangeable.

The session data pertaining to a client is usually small (some hundreds of bytes), so that allocating a whole memory page for a session descriptor leads to memory waste. On the other hand, the relatively small size of private information makes it possible to hold it in server-backed memory, at the price of a slightly less accurate accounting and of a number of concurrent sessions limited by the server resources.

2.3 Partition of the Server State

The hypothesis underlying a per-client memory accounting system is that every memory object of a server (or at least, any object allocated from its heap) can be ascribed to some client. To allocate an object from client-backed memory, the server must decide to which client the object belongs. This quota hypothesis is found in other works that address the management of client-related server state, such as Bastei ([Feske, Helmuth 06]), which offers a sliced heap and in which each client must donate the memory quota used to create a session, or CuriOS[1] ([David, Chan *et al.*]), which uses server state regions (SSR). Each heap slice or SSR contains the data pertaining to a given client session. Conversely, in Bastei, there must be a client that pays for each object, because dynamic allocation of server memory defeats the purpose of quota trading. The actual partition of the server data lies under the responsibility of the server. The authors of [David, Chan *et al.*] distinguish 3 types of servers:

stateless servers do not store client-related data, thus do not need client-backed memory

client-partitioned servers access session-exclusive data only when handling a request relating to a session, the server state is the union of all client-related states

servers with inter-client dependencies access global data or state related to third-party clients when handling a request, the server state comprises more than the union of client related states

[1]The rationale for the state partitioning in CuriOS is error isolation rather than accounting.

Client-partitioned servers (and stateless servers even more so) fulfill the quota hypothesis. Most servers that use sessions are trivially client-partitioned, because their heap only contains session data, which easily relates to a client. Some servers however may need to keep track of inter-client dependencies[2], or have global data structures (for instance, a hash table to cache reference to session objects) or for several clients (buffers for a file opened by several clients in a file system server). Partitioning the memory usage between clients without hindering independence is not trivial in such cases.

The server design can address the problem by splitting the global data structure among several sessions using pointers, a natural solution for dynamic structures. Another option is to redundantly bill the complete resource usage of a structure to each client using it, although this results in conservative memory reservation. Memory backed by a client can contain data that do not directly relate to the client: for instance, if the session objects are member of a linked list, each session object contains a pointer to the next object.

In the following work, I will take for granted that the server state is client partitioned, without examining further how to solve this problem in the general case.

[2]The memory broker described in Section 3.2 falls in this category.

Chapter 3

Mechanisms for Memory Trading

Contents

In this chapter, I describe the design of two memory trading mechanisms that I have implemented on top of the Fiasco microkernel, as well as their implications on server design. The first one uses the L4 flexpage IPC mechanism and lets the server deal with page faults. The second uses a trusted entity, as an intermediary between the client and the server, the broker.

3.1 The Direct Mapping Approach

3.1.1 Principle

The most straightforward way to trade memory in L4 is to directly use IPC mechanism built in the microkernel for this purpose: flexpage IPC (see Section 1.1). In fact, as flexpage IPC is the only way to exchange memory access rights, any trading mechanism uses it either directly or indirectly. With the direct mapping approach, I investigate what this mechanism can achieve, as well as its shortcomings.

In this approach, the client maps its own memory pages in the server address space. The client can then employ the service as long as the shared memory stays available to the server. Since the pages are mapped, the client can unmap them at any time.

The client closes the session and unmaps its memory when it does no longer need the service. The server has nevertheless no guarantee that the client will not withdraw the mapped memory before actually closing the session. In fact, the client could unmap the memory while the server is processing a request from the said client (if the client is multi-threaded or if it does not wait for a server response after sending a request). In the absence of any enforced pre-revocation signal, the server must deal with pages that may vanish from its address space due to a revocation. There are optimistic solutions (using memory and handling page faults) as well as pessimistic ones (locking pages in memory before using them). I will discuss this problem in the Section 3.1.3.

3.1.2 Session Negotiation

To use a service, a client opens a session to the server and maps the required memory. The two steps do not need to happen simultaneously: the client can open a session and map the memory lazily when page faults occur at the server (see Section 3.1.3 for details about page fault handling).

For the flexpage IPC rendezvous to succeed, the client must send an IPC specifying the pages to be mapped while the server has a receive window opened. The size of the window may be negotiated during the session opening, or fixed by the server protocol. Usage of client-independent receive windows (a receive window can be attributed to any client) allow the server to perform an open wait[1] operation and inspect the received message to determine the nature of the request. The memory pages required for a session are sent in the very message requesting the session creation. Thus, a session can be created with a single IPC call operation without waiting phase. This solution is well-suited for single-threaded servers. On the other hand, if the server needs knowledge about the client (especially the size of the client-backed memory region) to decide to which region of its address space the client memory should be mapped, then the opening of a session requires at least two IPC operations: the first for session negotiation, the second for the actual memory mapping. The server must wait for the client to initiate the second IPC call after it sent the response to the first one. While the client needs an unknown (but presumably small) time to issue the IPC system call, the server cannot wait indefinitely for a message that may never come —the client is untrusted. The protocol must therefore arrange for a time span during which the server waits for the client to map its memory after a message from the server. A multi-threaded server that dispatches incoming requests (including further communication) to worker threads continues to receive and process requests while worker threads wait for a client flexpage IPC. The protocol can therefore specify a long or even infinite timeout for the waiting phase without hampering the server response time.

3.1.3 Revocation

The revocation of a page occurs when either the client or the server remove it from the server address space. Although the server protocol can include an

[1]During the receive phase of a L4 IPC operation, the receiver chooses to wait for a message either from a specific partner only (closed wait) or from any partner (open wait).

operation to signal a page revocation, there is no way to force the client to correctly notify the server before unmapping the memory. The server therefore must be able to deal with absent pages in client-backed regions of its address space.

In the pessimistic approach, the server locks a page for the duration of the read/write operation (and ensures it is present) before accessing it, in the sense that a locked page cannot be unmapped. This would ensure that no page faults can occur. The delayed preemption mechanism is a candidate for page locking on *single-processor* systems: as long as the server thread cannot be preempted, the client thread does not have a chance to issue an unmap system call. The server could perform data manipulations on the pages within short critical sections protected by delayed preemption. However, L4 does not offer any mechanism to prevent the unmapping of a given page, so that this approach requires a quite important alteration of the microkernel.

This is why this design takes the optimistic path: The server processes incoming requests under the assumption that client-backed memory pages are present. Page faults occur when they are not. A dedicated pager thread sharing the server task's address space distinguishes page faults occurring in client-backed regions and those occurring in regular memory regions by inspecting the page fault address. The pager can also identify the faulty session through the faulting thread and the region in which the page fault address occurred. It needs knowledge about the layout of client-backed memory regions in the server address space to relate the fault address to the faulting session. Regular page faults are forwarded to the server task's region mapper, while client-related faults are handled by the server itself. The server can take several courses of actions when a page is missing. Options include:

- aborting the current operation, and return an error to the client
- aborting the current operation, and destroy the faulting session
- requesting the client to map memory in the server address space and retry the operation

The latter solution leads to lazy memory mapping: instead of receiving all memory at the creation of a session, the server requests client-backed pages when they are accessed. This makes the session creation simpler and avoid mapping memory that is never used. The argument on timeout of the 2-steps session creation in the preceding section applies to lazy mapping as well, because the server must wait for the client to send pages after a page fault. Moreover, the lazy-mapping mechanism allow the server to actually delegate the memory paging to the client. The pager then acts as a region manager, dispatching page faults to the responsible dataspace (the client). The client can unmap pages that it donated to the server if its paging strategy requires so, it will get an opportunity to map them again when they are used.

Letting the faulting worker thread execute the page fault handling code has the advantage of preserving the RPC semantics, because the response to the request stems from the thread that received the request, a requirement of the IPC call operation. The client is not forced to use the IPC call operation (it could use a send), but call is the operation of choice for server protocols. Another way to preserve the RPC semantic in a multi-threaded server is to use a proxy. The proxy thread handles all IPC operations with other tasks, dispatching incoming requests to worker threads, and forwarding request results to the clients. In this architecture, it does not matter whether a worker thread or a pager thread produced the response.

I will discuss four possible implementations of the pager thread in Chapter 4.

3.1.4 Evaluation

Direct memory mapping is a simple solution. Its implementation does not require modifying the microkernel or the environment. The added complexity is supported solely by the server, due to the added pager thread and page-fault handling code. Moreover, it allows the server to delegate the paging of client-backed memory to each backing client, which acts as a dataspace for the server. Clients can use this flexibility to tune the service to their needs. The price of this simplicity is that the (untrusted) client retains full control over the memory lent to the server. This limits the usability of client-mapped memory for the server: There is no pre-revocation signal, and the client has at least the same access rights as the server for the shared pages. This later problem can be addressed by data encryption, although the computational cost can be high. The concept of untyped memory implemented in L4.sec, an extension of L4 towards enhanced security, would also solve the problem by allowing tailored access right definition ([Kauer], [Elkaduwe *et al.*]). In such a scenario, the client would send a capability for untyped memory to the server. The server would then convert (type) this capability into one for user memory for its own use. While retaining the old capability (and being able to revoke it, and thus the memory), the client would have no access to the typed capability, and therefore, no access to the memory.

The absence of pre-revocation signals implies that data availability (in the non real-time sense) cannot be assured. The pages exchanged in this scheme are therefore suitable for client-specific data only. Private server data[2], which loss would crash the server, must reside in the server's own memory. This system is well-suited to manage memory regions that are shared with one client anyway (such as video buffers). It does not easily extend to the case where a memory region must be shared between the server and several clients. It is not suited for memory regions containing private server data due to the absence of a pre-revocation protocol, which is indispensable to ensure data availability.

The direct mapping scheme does not allow memory exchanges below the granularity of a page. This is however not a problem for broadband communication with shared memory, because the amount of shared memory usually is much higher than the page size, so that the waste due to the slack space is negligible.

[2]Private server data includes at least the structure keeping track of client-backed memory regions (which client has mapped how much memory at which address of the server address space). Servers using client-backed memory cannot be stateless, unless the memory is mapped for the duration of a request only.

3.2 The Memory Broker

3.2.1 Intermediation

As the memory supplier (client) and the memory consumer (server) do not trust each other, pre-revocation signals require a trusted third entity (the kernel or a server) that intermediates between these mutually untrusted agents. A trusted user-level server (which will be part of both trusted computing bases) can take over this role and implement pre-revocation signals with IPC operations.

In order to avoid confusion with the client and the server trading memory, I will use the term broker for the trusted third entity. Server, client and broker are merely roles endorsed by L4 tasks in the memory exchange protocol. A task can act as a server (and receive memory) and later act as a client (and grant memory). There could be several memory brokers running concurrently, to isolate subsystems or to manage distinct memory types (a broker for pinned memory and one for paged memory, for instance). The only constraint is that the agents of a transaction must trust the broker executing it on their behalf.

As an intermediary, the broker transmits pages ascribed to a client from a trusted source (like the primary pager σ_0 or the quota system) to a server. A page is deemed clean when no untrusted task retains access rights to it. To ensure data confidentiality, the broker must manipulate only clean pages. Pages granted by a client are not clean, because the client could have mapped the page several times in its address space before granting it to the broker. Although one of the mappings disappears through the grant, the client still can access the page through the remaining mappings. Such manipulations are impossible if the memory broker bypasses the client and gets pages directly from a trusted source of the underlying system.

The broker maps each clean page (from the client's quota) to the address space of a server specified by the client. To revoke a page grant to a server, the client informs the broker that it wants to reclaim its memory pages. The broker then notifies the server, which can take any measure needed to free the memory, and replies to the broker notification when ready. The broker can then unmap the memory pages without risking to crash the server.

The broker must determine and implement a resource reclaiming policy (when to actually unmap memory). As the broker does not trust the server, it cannot rely on a timely response. A malicious or erroneous server could delay its response (or not respond at all). Moreover, in a non real-time environment, there can be no guarantee whatsoever that the server actually runs while the memory broker waits for the memory to be freed: a thread with a priority higher than the server could prevent it from responding. Thus, even an honest and error-free server could be unable to free its memory and delay its response for an arbitrary long duration. No timeout value alone allows to distinguish between a misbehaving server and an overloaded system.

Unmapping memory before the server replies to the notification is likely to harm the server, crashing it or provoking unpredictable behavior. On the other hand, waiting for the server response gives malicious servers an opportunity to retain memory forever. My sample implementation of a memory broker favors the server (and waits for a response) for the sake of system stability, but does not allow a server to grant credits or request memory as long as it has a revocation signal pending. However, the broker could easily be modified to implement a different policy. In an interactive system for instance, the broker could notify the user and ask for a decision. In a system supporting time donation, the client could give the time needed to free memory to the server (the very same way that it gives the memory needed for a session). This presupposes that at least the server can predict the maximum time required to free a memory region.

The memory broker must keep track of the currently active transactions (which task granted which pages to which server). It must be able to decide whether a revocation request does match a previous grant and may have to store some information relative to the revocation terms (for instance, a timeout value). The data structures used to realize this are an implementation detail.

3.2.2 Credit Trading

In the frequent case where the client does not require access to the memory pages that it donates, the server does not really need that the client actually *provides* memory pages: It only requires the memory *usage* to be *accounted* to the client. Therefore, trading memory credits (right to use memory) is sufficient, as tasks running on the seL4 microkernel do when they exchange capabilities for untyped memory rather than for memory pages.

Trading credits rather than memory gives flexibility: when a client reclaims a page, the broker can choose which page to free —or leave this choice to the server. The broker can use this flexibility to trade memory at a finer granularity than the page, thus reducing the memory waste due to internal fragmentation (unused memory inside an allocation unit). Credit trading requires memory to be interchangeable (see Section 2.1.7). This implies the absence of external fragmentation (memory regions that cannot be allocated).

The memory broker can take up the mapping from credit to memory pages in addition to its intermediation role. To ensure that there is no fragmentation, servers can demand credit grants of a fixed size[3]. Allocation of fixed-size blocks ensures that the server can reuse the memory that a client reclaims for another client (deleted data leaves no hole, so to say). Two servers can have different block sizes depending on their needs. If the block size is larger than a page, then the advantage of a finer granularity is lost (credit trading is then equivalent to page-wise memory trading).

Each server that wants to use the memory broker must create a session. In fact, it also will need *one* (1) memory page available because the broker rounds down the sum of each server's credits to a number of pages —the server gets the *rounded down* amount of memory that its clients granted, rather than the *exact* amount. For instance, with a page size of 4 Kb, a server with a credit of 18 Kb (say, six clients having granted 3 Kb each) would receive four pages from the broker (amounting to 16 Kb), and the two last kilobyte of data would be stored in a private server page. The rounding cost never exceeds one page for each server session. As this cost depends only on the server architecture and not on the number of clients, the server can reasonably support it. Clients wanting to donate memory to a server must open a session as well. The memory pages that clients donate to the memory broker are converted into credits, that can be

[3]Note that the memory broker does not need to care about how the tasks actually handle fragmentation. This is the sole responsibility of the task that will actually use the memory, and a server designer could decide to take grants of the size of a page only

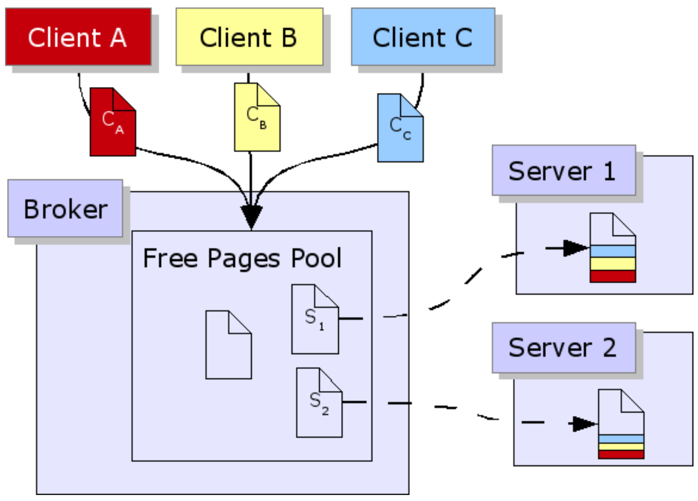

Figure 3.1: Principle of the Memory Broker.
Each task donates memory quota to the broker. A server gets from each of its clients the credit for the fraction of a page used on its behalf.

granted to servers. As a task can act both as a client and a server, each session might be a server session, a client session, or both simultaneously.

Each server receives and gives back memory pages as the sum of its credits varies. The server is notified when a client revokes a grant (so that it has an opportunity to deallocate the client's data and to defragment its memory) and before a page is unmapped. It is up to the server to check that the client has granted the necessary credit before allocating memory for this client. The broker scatters and gathers credits to satisfy requests from memory users, as illustrated in Figure 3.1.

To convey an idea of how this works, I will sketch a proof that the memory broker always has enough available pages to satisfy legitimate requests at any given point of time.

Let us use the following notations:

- $N \in \mathbb{N}$ the number of opened sessions at the memory broker
- $C_i \geq 0, i \in [\![1, N]\!]$ is the number of pages that session i (as a client) donated to the memory broker
- $S_j \geq 0, j \in [\![1, N]\!]$ is the number of pages that the memory broker mapped in the address space of session j (as a server)
- $g_{i \to j}, (i, j) \in [\![1, N]\!]^2$ is the amount in bytes of the credit grant from the session i to the session j ($g_{i,j} = 0$ in the frequent case where there is no such grant)
- P the size of a memory page in bytes

From these definitions, we can express the invariant that must hold for each session: The session owner's balance is positive (there are more contributed pages and received credit than consumed pages and donated credits):

$$\forall i \in [\![1, N]\!] : C_i \times P + (\sum_{j=1}^{N} g_{j\to i}) - S_i \times P - (\sum_{j=1}^{N} g_{i\to j}) \geq 0$$

On receiving a request, the memory broker ensure that satisfying it would not break the invariant of the involved sessions. If it would, the request is not acceptable and must not be executed. By adding the inequalities for all sessions, we find that the number F of pages available to the memory broker is always positive or null (the grant terms cancel out because each grant appears as a debt and as a credit):

$$F = \sum_{i=1}^{N} (C_i - S_i)$$

$$F \times P = \sum_{i=1}^{N} [C_i \times P + (\sum_{j=1}^{N} g_{j\to i}) - S_j \times P - (\sum_{j=1}^{N} g_{i\to j})] \geq 0$$

Thus, the memory broker never runs out of memory as long as it ensures that each session does not spend more memory than it put in the system. This arithmetic proof makes sense only if memory credits and memory can indeed be summed (that is, if these resources are interchangeable). This is the case for anonymous memory (or memory of a fixed type) with the block-wise allocation scheme, which prevents external memory fragmentation (any other solution that avoid fragmentation would do as well). Moreover, as the memory broker maps a number of pages lower than the sum of memory credits (and lets the servers pay the rounding cost), there is no rounding problem. In cases where the block-wise allocation scheme is not tractable (for instance, with shared memory), the broker must deal with pages rather than bytes.

3.2.3 Evaluation

The memory broker fulfills two distinct roles: It intermediates between the client and the server (mandatory for signaled revocation), and can optionally handle credit trading to refine the granularity of memory grants. Compared with direct mapping, the page broker (without credit handling) offers additional functionality: pre-revocation signaling protocol, and separation of memory credits from access rights and revocation terms. It can therefore guarantee the confidentiality and the availability of client-backed memory. A server using a page broker can use client-backed pages for any dynamic memory allocation (including private data) without risk. Although my implementation of a broker does not support memory sharing, it is easy to build a page broker that does so.

The memory model of the broker (memory comes from the same dataspace for all clients) makes it difficult to delegate the paging policy to the client, or even impossible for servers that use credit trading to store information pertaining to several clients on the same page. Servers that use the broker rely on it (as a dataspace) for client-backed regions, while they rely on the client in the direct approach.

On top of the intermediation role, a broker handling memory credits rather than pages significantly reduces the memory footprint of servers that use small amounts of memory per session. This optimization comes at the price of a more complex memory management for the server: It must be able to defragment its memory by moving objects, which is not necessary with a sufficiently flexible page broker.

The complexity of the memory broker trading system resides mainly in the server that is added to the environment, but also includes overhead on the side of the client and the server, due to the additional communication necessary to trade memory before using a service. Moreover, the new server is part of the TCB of all tasks using it. Trading memory credit makes the server's memory management more complex than it would be if the server used its own memory, especially when using memory credits. In order to free a memory page when a client revoke a credit grant, the server may have to move data around from one page to another. This memory management scheme is very similar to the slab cache widely used in the Linux kernel (see [Love]). The slab cache manages pages filled with objects of a fixed size (and type), but in the client-backed slab cache, the server has to consolidate memory pages so that the memory broker can unmap pages without harming the server when a client revokes a donation. The slab management system might have to copy objects from one page to another to fulfill this requirement. This copy implies updating all pointers and references to the object, as well as preventing race condition during the copy in a multi-threaded environment. The code complexity can be hidden in a library, but the server still has to pay the communication overhead, the synchronization and the copy costs at runtime. The slight memory overhead (due to added code and added data structures) is negligible compared to the memory gain if the allocation granularity is small compared to the page size. As it uses memory backed by its clients, the memory broker itself exemplifies how a server can implement a memory management scheme suitable for memory trading. The costs and benefits of credit trading are discussed in details at the example of the broker in Section 5.5.

On the other hand, the environment of tasks that do not use the memory broker does not change —their memory allocation path is unchanged. Like any added server, the memory broker is service-neutral in the sense defined in [Liedtke *et al.* 99].

The memory broker design implies a policy choice that binds every application using it: the arbitration when a server does not respond to a revocation signal. This rigidity can be alleviated by running several brokers (one per policy).

3.2.4 Kernel-Enforced Signaled Revocation

As every task in the system must trust the microkernel, it makes sense to extend its communication primitives (especially page mappings) to make them suitable for memory trading, so that a task can safely use memory mapped by an untrusted task as if it were its own. The microkernel would then play the role of the intermediate third entity. The concept of untyped memory implemented in L4Sec ([Kauer]) already solve the confidentiality problem. Extending the unmap operation with a pre-revocation signal would solve the availability problem. A microkernel implementing these two features would offer the same service as a page broker (a memory broker without credit trading), without the overhead of session negotiation.

The downside is that modifying the microkernel changes (and enlarges, as a microkernel with more features is likely to be more complex) the trusted computing base of every tasks in the system. Moreover, modifying the kernel is not necessarily service-neutral: it may affect tasks which do not use the added functionality.

Whether the trading functionality is implemented in the kernel or in a user-level task does not change the implications for the memory management of servers using client-backed memory, however. The server design constraints that emerge from signaled revocation are the same, regardless whether the signal comes from the microkernel or from a user-level server.

Chapter 4

Implementation of Direct Mapping

Contents

4.1 The Encryption Server

4.1.1 Principle

Direct mapping (presented in Section 3.1) is a simple memory trading system where the server uses a simple flexpage IPC to get memory from the client. I used a simple encryption server as an example for my implementation of direct mapping. The sole purpose of this server is to illustrate a typical service protocol and operations on shared memory. The encryption service takes two client-backed memory regions (for plain text and encrypted text) defined at the creation of a session, and offers two (very similar) operations: encryption and decryption. Both regions are shared with the client and have the same size (up to 4 Mb), which is a multiple of the system's page size. Although this is not required for such a simple service, the server is multi-threaded, following the classical worker pool architecture: one thread handles the communication with other tasks, a fixed number worker thread processes requests and a dedicated pager handles page faults triggered by worker threads. The usse of client-backed memory does not restrict the server architecture, the optimal choice depends on the characteristics of the service. The point of multi-threading here is to investigate how random page faults in client-backed regions and synchronization issues interplay.

The server code is divided into three modules. The module for memory management implements the isolated server operations (session creation, session destruction, encryption and decryption). Simple spin locks protect the server structures from concurrent access. This module implements the management of the virtual address space, discussed in 4.1.2.

The threading module defines the server's control flow (the server loop) and its communication protocol. The main module brings together the other modules and contains the page fault handling code. As I have tested several solutions to handle page faults (detailed in Section 4.2), there are four slightly different implementations of the encryption server.

4.1.2 Memory Layout

To receive memory from a client, the server must assign a region of its virtual address space to this client by opening a receive window. Thus, server virtual addresses, a resource from the encryption server's point of view, are not plentiful[1]. The encryption server dedicates 1.6 GB of its address space to client-backed memory.

An important implementation goal of the encryption server is to restrict dynamic memory allocation to client-backed memory regions. As threads use server memory (mainly due to their stack), I use a fixed number of worker threads. All private data structures (including thread stacks, the address space manager and the session registry) are declared statically and are hence part of the data section.

4.1.3 Detailed Protocol

The encryption protocol relies on RPC and is implemented with the IPC call operation. Server requests take an operation code and a valid session number as arguments (except session creation requests, for which the second argument is the region size). The server response contains an acknowledgement (if the operation completed successfully), an error or a request for memory (in case a page fault occurred in a client-backed page). In the latter case, the client must send some memory within a time span defined in the server protocol. If it does not comply, the requested operation fails and the server sends an error as response. If the client maps memory timely, then the operation continues.

4.1.4 Communication and Synchronization

The RPC communication semantics of the IPC call operations force the response to come from the very thread to which the request was addressed —a response

[1]A plentiful resource is a resource of which the system never lacks. In most cases, virtual address space is considered plentiful because the system does not have enough physical memory to populate the whole virtual address space. The virtual address space available to a user task is finite however, and the encryption server cannot create more than a fixed number of sessions due to this limitation.

from another thread with the same task number will not do. When a page fault occurs, the request for memory must come from the thread that accepted the request, not from the pager thread. I have chosen to handle all communication with a proxy thread. Another option is to have the proxy thread redirect a client to a worker thread that handles the whole client request (including communication and memory errors). The second option greatly enhances parallelism because a worker can wait for client memory without stopping the whole server.

4.2 The Pager Thread

4.2.1 Introduction

The encryption server can trigger a page fault either because of lazy mapping or copy on write mechanisms, or because a client failed to map pages to a server address space region. The former case occur when a page mapping is delayed until the page is actually accessed, a mechanism that the program loader uses to avoid mapping unneeded pages. It is the responsibility of the underlying environment (usually the loader) to provide the missing page. Each L4 task has a special task, the region manager, that forwards page fault messages to the server responsible for the memory region where the fault occurred.

Client-backed memory regions are a special case because contrarily to trusted environment servers, an untrusted client may not provide the needed memory page.

Rather than modifying the default region mapper, I added a new pager thread on top of it. This thread, registered as the worker thread's pager, filters out and handles page faults occurring in the client-backed memory. It forwards other page faults occurring to its own pager, the region manager.

When a page fault occurs, the faulting thread is blocked waiting for an IPC from its pager thread. There are two basic ways for the pager to unblock the faulting thread: to send an IPC to it[2], or to issue a `l4_thread_ex_regs()` system call targeting the faulting thread. The `l4_thread_ex_regs()` system call modifies the state of a thread of the same task, and has the side effect of canceling any ongoing IPC operation of the target thread. The pager can use this system call to have the worker thread jump to error-handling code by changing its instruction pointer and stack pointer[3].

In the following section, I will discuss 4 alternative solutions to handle client-related page faults. The first one maps memory, while the three others use `l4_thread_ex_regs()` to change the worker's state.

4.2.2 The Joker Page

The easiest solution to keep the server running when a client-backed page is absent, is to map a surrogate server-owned page: the joker page. While the server will read from or write to the joker page (which is not mapped in the

[2]the IPC should map some memory at the fault address, or the re-execution of the faulting instruction will cause a fault again

[3]Using `l4_thread_ex_regs` is not equivalent to a function call when using most calling conventions, because no return address is pushed on the stack, and no register is saved. See Section 4.2.3 for a discussion of this problem

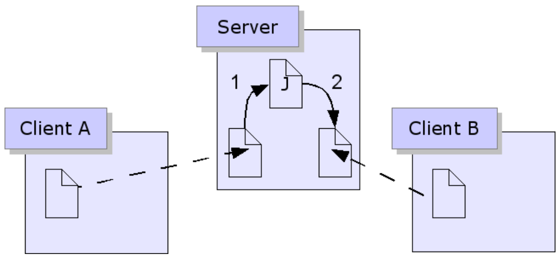

Figure 4.1: The Joker Page as Hidden Canal.
Clients A and B map their pages in the server address space. Client A issues a request to have the server copy its data to the joker page. Client B then issues a request that cause the server copy the data from the joker page to B's page.

client's address space), it still can complete the operation and take any corrective action afterwards.

The pager uses a reserved memory page (the joker page) to replace every missing page in turn. At each page fault from a worker thread, the pager sets a flag in the request state and maps the joker page at the fault location. Before mapping the joker page, the pager undoes the previous mapping to avoid to reach the maximal number of simultaneous mappings that the mapping database imposes. The pager could alternatively enforce mutual exclusive use of the joker page, and unmap the joker page after the worker thread that uses it finished handling its request.

After processing a request, the worker thread checks the page fault flag. If it is clear, the request completed successfully. If it is set, (garbage) data was read from the joker page, or the request data were written to the joker page and thus lost (or both). The worker thread closes the invalid session and sends an appropriate error code back to the client.

This approach has a weakness: If implemented carelessly, the joker page may open a hidden canal through which unrelated tasks can communicate, thus lowering the system security standards. Figure 4.1 shows how two malicious client can use an encryption server with one joker page as an hidden communication canal.

To avoid such hidden canals, the pager can zero joker pages after each request (which costs bus bandwidth), or use separate pages for write and read access. On the x86 architecture and other architectures that does not offer write-only page protection, the latter approach only works for a restricted set of server that never read and write from a given client-backed page (at least not during a single request).

Whatever solution is chosen to prevent hidden canals, the joker pager does not disturb the server's control flow (the server code can therefore use external libraries without any restriction), and is simple to implement. It has an important overhead when pages are missing because of the multiple mappings (and the page zeroing if needed).

4.2.3 The Control Flow Problem

If the pager cannot retrieve the missing memory page from the client, it must interrupt the worker's control flow to branch it to error handling code. The worker thread's currently executing function does not run to its completion and never returns, neither do the enclosing functions. The error handling code has to do their cleanup (freeing memory, releasing locks and so on). Explicit cleanup of dynamic resources is error prone and requires care: Failing to correctly release resources can lead to memory leaks and deadlocks.

Recovery from a client-related page fault is similar to the recovery from exceptions in C++[4]: Exceptions can occur at many places and abort the functions enclosing the `throw()` statement. The Resource Acquisition Is Initialization (short RAII) pattern suggested by Bjarne Stroustrup and commented in [Stroustrup] is a classical solution for dynamic resource management (including cleanup after an exception) in C++. It rests upon the stack unwinding mechanism[5] specified by the C++ standard. In short, the C++ standard guarantees that the destructor of every any local (that is, allocated on the stack) object is executed when leaving a function. By moving every resource allocation in constructor (and symmetrically, the release in the destructor) of local objects, the programmer can easily ensure that no resource leaks, no matter how complex the control flow is.

Unfortunately, no stack unwinding takes place when the pager changes a worker's thread instruction pointer (unless the error handling code consists in throwing an exception, see Section 4.2.6). The burden of dynamic resources management thus lies on the server programmer's shoulders. I will discuss in more details the concrete constraints that the control flow problem places on the server design in the next sections.

4.2.4 Long Jumps

The long jump is a feature of the C library that allows jumps from a callee function to a caller function (as opposed to `goto` statements which are limited to the local scope). Note that long jumps can go only *up* the stack: You cannot jump back to a function *after* you have returned from it. Long jumps are implemented by mean of the `setjmp()` and `longjmp()` functions. `setjump(env)` saves the thread's current execution context in an architecture-dependent structure (you can use a local or a global variable for this purpose). Moreover, the return of `setjmp()` sets the point where the execution resumes after a long jump. The return value is 0 on the `setjmp()` call, and the value specified by a subsequent `longjmp(env, value)` call when it returns from a long jump. The `setjmp()` return value is the only difference between the initial context and the context that `longjmp()` restores.

As long jumps require explicit cleanup (see the preceding section), they are deprecated (like `goto`) and require careful server coding.

[4]C++ exceptions are abstractions meant to represent error conditions, which client-related page faults and other processor exceptions are.

[5]When an exception is thrown, the compiler destroys all local variables (using destructors for objects) and jumps to the calling function, repeating the process until it finds an enclosing `try{...} catch()` block matching the exception type. If a destructor throws an exception during the stack unwinding, the program exits. This is why throwing exceptions in a destructor is strongly discouraged.

The pager thread can make the worker thread long jump (by mean of a dedicated function with a known address) to a state that the worker saved before it attempted to run the risky operation. The worker thread can detect that a page fault occurred by examining the return value of `setjmp()`, and run error handling code instead of the faulting operation. This result in the following code structures:

For the worker thread:

```
//(1) get and lock session data (cannot fault)
  client = registry.lock(session_id);
  client->status = err_unknown;

//(2) execute operation
  if ((pf_address = setjmp(env)) != 0)
  {                                     //page fault
    if (!proxy.get_memory(client))      //try getting memory
      client->status = err_no_mem;      //(cannot fault)
  }
                                        //request processing
  if (client->status != err_no_mem)     //may fault
    client->execute(opcode);            //and longjmp to (2)

//(3) send end response (cannot fault)
  proxy.response(client);
  registry.unlock(client);
```

The pager thread runs the following code:

```
while(1)
{
  l4_ipc_wait(&pagefaulter, &address, ...);

  if (address < start_client_space)
  {
    //forward and wake up faulter
    l4_touch_rw((void *) address, 1);
    l4_ipc_send(...);
  }
  else //have the faulting thread jump
  {
   Worker_thread * worker = workers_pool[pagefaulter];
   worker->jump(address);
  }
}
```

The `worker->jump(address)` method make the worker thread long jump, as the following definitions show (this works only if the compiler uses the standard C calling convention for `void run_jump(l4_addr_t)`):

```
void run_jump(l4_addr_t pf_address)
{ longjmp(env, (int) pf_address); }

void
Worker_thread::jump(l4_addr_t pf_address)
{
  //do not use the thread stack
  l4_umword_t* esp = special_jump_stack_start;

  *(--esp) = (l4_umword_t) pf_address;  //argument
  *(--esp) = 0;                         //return address

  Thread::start(run_jump, esp);
}
```

The use of long jump is not without disadvantages. First of all, the very structure of `setjmp()/longjmp()` forces the error-handling code and request-processing code (that may fault) to follow the conditional statement that tests `setjmp()` return value. This leads to an unclear code structure. Moreover, when a page fault occur, the long jump aborts some functions (represented by `execute()` in my code). Either these functions must not use dynamic resource (for instance, I lock the client before rather than during the risky section), or the error handling code must do the cleanup.

The server designer must identify and carefully adapt the affected functions. For instance, she could store the function state outside of the stack (as is done in the method discussed in the next section). Another option is to save the execution context often (for instance, before each client-backed memory access) in order to minimize the impact of the long jump. All these measures require code modifications however, and are therefore not applicable to external libraries with a reasonable effort, if at all.

4.2.5 Procedures with Reified State

As the control flow problem imposes drastic restrictions on the stack usage, it makes sense for worker threads to use a dedicated static object rather than their stack to store the state that they need to continue their work. This reified state —which in fact represents the request continuation— does not only describe the request (operation code, originating client), but also keeps track of its progress and of all the data that must be preserved across page faults (and are thus not safe on the stack). This can include information about acquired locks, reserved memory, but also a running counter that allows the worker to resume a loop at the interrupted iteration.

I divided the request processing into steps which are implemented as procedures, that is, function that take a reference to a request object as argument and return no value. The request object includes, among other data, the step sequence that constitutes the request processing (which depends on the operating code), and the next step to execute. The only effect of a procedure is to change the continuation, that is, to modify the request object passed as argument.

I represented the procedures with methods of the request object, the request processing as an array of method pointers and the next step as an index in the array. The procedure sequences could be equally well represented by a list of function pointers, or be hard coded in the procedures themselves. A worker thread process the request by applying procedures to the request object. In fact, several workers could relay to process a request. When the request is done, the worker thread marks itself as idle and waits for another request to process.

As an example, here are the procedures I defined for the encryption server:

- do_create_client: creates a new session object
- do_get_client: fetches the session object and locks it
- do_populate: requests the client to map pages
- do_encrypt/decrypt: executes the eponymous operation
- do_unsubscribe: fetches and destroys the session object
- do_respond: sends a response to the client and unlocks the session object. This procedure ends every request.

These procedures are arranged into a 4-row matrix (one row per operation: session creation, session dismissal, encryption and decryption), as follow:

```
action _steps[4 /*op*/][2 /*max. steps per op*/] = {
 /*subscribe*/   {do_create_client, do_populate},
 /*unsubscribe*/ {do_unsubscribe  , 0},
 /*encrypt*/     {do_get_client   , do_encrypt},
 /*decrypt*/     {do_get_client   , do_decrypt}
};
```

Moreover, I added a special method for page-fault handling, that the pager invokes after a page fault. If it fails to get the missing memory pages from the client, it aborts the request (which mean reclaiming resources and bumping the request object to the final step: send a response). Otherwise, it does not change the request object and the processing resumes where it was interrupted.

Procedures that may cause a page fault must be idempotent (that is, they should change an object only once even if they are applied to it repeatedly), so that they can run again after a fault without risking data inconsistency. This can be achieved by transfering all the execution state to the request object, so that the continuation is always up-to-date. For instance, the encryption procedure use an offset to access memory buffers. This offset is a member of the request object rather than a local variable and therefore keeps its value across page faults. Even if the encryption procedure is run twice on a given request object, the actual encryption operation is executed at most once on each datum.

Maintaining a continuation object has a low overhead but requires an important code refactoring. The structured code architecture imposed by a reified state makes resource management easy, and leads to a neat separation between error handling and server logic. However, the rewriting is not possible or sensible for any libraries accessing client-backed memory that the server designer may use in a real server.

4.2.6 C++ Exceptions

The exception construct has been introduced in C++ to deal with error conditions and is therefore tailored to deal with system exceptions, as explained in [David, Carlyle *et al.*]. In addition to fostering separation of error handling and server logic, turning client-related page faults in exceptions would allow the server to take advantage from the built-in stack unwinding of C++ compilers. The stack unwinding is a powerful tool for resource management that, when applicable (in conjunction with RAII, see Section 4.2.3), performs duties comparable to those of the garbage collection and the finally clause in the Java language ([Stroustrup]). It is an important part of the C++ language.

The C++ exception concept is synchronous and intrinsically related to that of a thread (much like long jumps): When handling an exception, the compiler-generated code looks for a matching `catch()` clause along the thread's control path. Therefore, the pager thread must indirectly inject an exception in the worker thread by having it call the compiler-defined `throw()` function. The details of the internal exception representation and handling depend on the compiler. For the g++ compiler which I used in this work, the implementations[6] are discussed in [David, Carlyle *et al.*]. The built-in `throw()` function hides such details from the server code.

In fact, turning a client-related page fault in an exception in the faulting thread boils down to tricking the C++ runtime library into believing that the function that caused a page fault threw an exception. Rather than simulating a call to the `throw()` function by manipulating the thread's stack, I have written a function (see the following code listing) that merges its stack frame with the previous frame on the stack (using an assembler instruction), and then throws an exception. As the C++ runtime uses the stack (and thus the value of the ebp register) to identify the throwing function, it unwinds the stack assuming that the exception originated from the function that caused the page fault (which is the wanted result) rather than from `throw_exception()`.

```
void throw_exception() {
  //merge the 2 upper stack frames
   asm("movl (%ebp), %ebp;");

   //fault_address is a global variable
   throw (fault_address);
}
```

The rationale of this choice is that the exact stack layout that a function call produces depends on the calling convention used by the compiler, which is in turn architecture dependent. While merging two stack frames also is architecture dependent, it is much simpler and probably easier to port on other architectures. The pager thread branches the faulting worker at the `throw_exception()` function as it would branch it to the jump function in the long jump approach. The server code catches these exceptions and handles them as C++ programs usually do.

This approach makes an assumption about the way the compiler manages the stack —namely, that the first word of each stack frame contains a pointer to

[6]The GCC g++ compiler offers two implementations of exception handling, one based on long jumps and a table-driven one, hence the plural.

the previous frame, and that the ebp register contains a pointer to the first word of the current frame (ebp is the start of the list, so to speak). This assumption is weaker (and simpler) than the assumptions that would be necessary to simulate a function call, but it can be false even on the x86 architecture, depending on the compiler options (and the way it manages the stack). Before using this method, it is necessary to examine how the compiler work on the particular platform to be used, and to adapt the `throw_exception` function accordingly.

Although `throw_exception()` may require modifications depending on the compilation environment, this method allows a great flexibility in the design and the structure of the server code. It can use client-backed memory in conjunction with external libraries only if these are built to deal safely with exceptions. While most library programmers would not expect a simple memory access to trigger an exception, libraries that manage resources with the RAII pattern are exception-safe anyway.

Chapter 5

Implementation of the Memory Broker

5.1 Introduction

In this chapter, I present an implementation of the memory broker discussed in Section 3.2. Aside from demonstrating the feasibility of the memory broker, this implementation also illustrates how a server can manage its virtual address space and heaps of client memory, as the broker itself uses memory from its clients for the transaction bookkeeping.

In this implementation, I assume that threads cannot forge their identity when sending an IPC and that clients grant clean pages to the memory broker. While the current version of Fiasco does not provide such guarantees, the next version will provide unforgeable thread identifiers (thus making IPC spoofing impossible). Moreover, the successor of L4Env, which will run on the next version of Fiasco, should offer a mechanism to test the trustworthiness of memory allocators.

5.2 Detailed Protocol

My memory broker implementation offers 6 operations, which are implemented as IPC call:

Session creation: A L4 task can create several session (which allow to manage pages with different allocation granularities). The message requesting a new session must contain an initial memory grant. The memory credit necessary to hold the session descriptor is withdrawn from the initial grant, the credits left are available for later use. The task must also specify the thread to which the memory broker will send pre-release notification messages. The memory broker respond to the request with the session identifier (an unique integer value), which is used in subsequent requests..

Page grant: A session owner, that is, any thread of the task that created the session, can increase the session's balance by granting memory pages to the memory broker. It can also ask to get the pages back (in which case the amount is negative). Although the granted pages are not trustworthy, this mechanism is an acceptable approximation of a quota granting mechanism for a test implementation.

Page map: The owner of a session can ask the memory broker to map memory pages in its address space. The balance of the session is decreased of the corresponding credit amount. If the amount is negative, pages are unmapped. The pages are unmapped in a LIFO manner (the pages mapped last are unmapped first).

credit grant or revoke: A session owner can grant some credits from its session to another session, or revoke it (using a negative amount).

check credit: A session owner can check whether it got a credit grant from another session given as argument. Servers can use this operation to check whether they got the credits from a client before allocating memory on its behalf. Each task is responsible for keeping track of the credits it granted, the pages it granted to the memory broker and the pages the memory broker mapped to it.

Session dismissal: A session owner can close the session. Before the session is actually closed, all grants are revoked, and pages are unmapped.

On receiving a request, the memory broker checks the session's balance, and executes the operation if the credits are sufficient. When the balance of a session would become negative after a client revokes its credit grant to the session, the memory broker sends a notification message at the thread registered in the session before unmapping a page (thus decreasing S_i and increasing the balance). As the server owning the session can revoke grants it made to other tasks (or do any operation that increase its balance) before answering to the notification, there is no guarantee about the number of exchanged messages before the memory is actually freed. The memory broker also checks that C_i and S_i stay positive, and that the session has not exhausted its dedicated address space.

My implementation unmaps pages in their reverse mapping order (LIFO). The LIFO unmapping scheme is simple to implement (and thus has a low overhead) but makes sharing of client-backed memory very complex and costly. A somewhat more elaborate server could leave the choice of the page to give back to the server task, which would simplify memory management (and sharing) for servers that trade only pages.

While the broker allows a task to open several sessions (it does not check for duplicate task identifiers), there is little benefit in doing so. A task that wants to manage several heaps with different granularities, as the broker itself does (see Section 5.4), would need to reserve one slack page per heap due to the rounding problem. It could nonetheless use a single broker session to back the heaps, at the price of displacing pages from one heap to the other due to the FILO unmapping order.

5.3 Memory Layout

Like the encryption server, the memory broker must manage its address space. A region of the address space, the free pages pool, receives the donated memory pages from all clients (I use page grants as simulation of quota donation). Moreover, the broker assigns an address space region to each opened session. This region contains the pages mapped in the session owner's address space. Before mapping a page in a client's address space, the broker moves it from the free pool to the appropriate session region by mean of a grant IPC. A statically allocated bitmap keeps track of the address space usage. The number of regions limits the number of simultaneous sessions. The fixed region size limits the memory quantity that a server can get from the memory broker. If memory donations to the broker could be represented by quota donations rather than by granted page, there would be no need for a physically present free page pool.

The memory broker also manages two heaps for its own internal data structures. They are described in the next section.

5.4 Data Structures and Heap Management

The memory broker uses a session object (56 bytes) to represent each opened session and a credit object (28 bytes) to represent each credit grants between two sessions. Using a heap *per object type* eliminates inner fragmentation and makes its management simple: The heap contents are an homogeneous array of objects. The downside is that the server must supply the first page of each heap, due to the broker rounding down credits.

The session heap (which is equivalent to a dynamic array) stores all sessions objects. Whenever an object inside the array is deleted, the last object of the array moves in its place (Figure 5.1) so that there is no hole in the array. When a request arrives, the broker searches a session object with the matching identifier in the array. The sessions could be inserted in a binary tree to optimize the research. However, the pointer would have to be doubled (like in doubly-linked lists) to allow a fast displacement of session objects. In addition to S_i, C_i, and its balance, each session object maintains a circular doubly-linked list of the grants it received (credits) and a list of the grants it made (debts). The list pointers are part of the credit object themselves.

Like the session objects, the credit objects are stored in the credit object heap, and can be displaced to keep the array dense. Moreover, each of them is inserted in the credit list of the donating session (the creditor) and the debt list of the donating session (the debtor). The credit objects thus form a logical (sparse) matrix, as Figure 5.2 illustrates. Each logical row (stored as a linked list) of the matrix represents the memory credits that a given client received. Each column (which is also stored as a list) stands for the credits that a client granted. Each credit, situated at the crossing of a row (its creditor's credit list) and a column (its debtor's debt list), is member of two lists and represents a non-zero coefficient of the logical matrix. This way of representing the credit matrix has two main advantages over a vectorized representation: It is more memory-efficient because it leaves out the many null coefficients, and it allows to easily add or remove rows and columns.

The credit object contains a pointer to its creditor and to its debtor. The

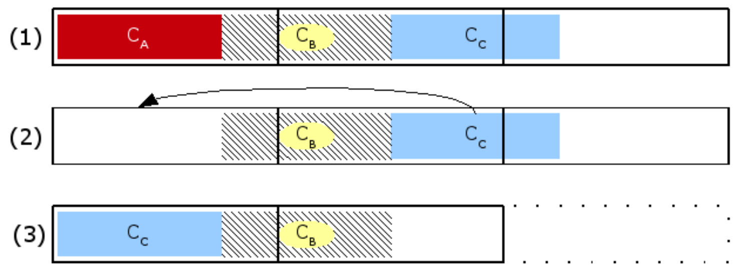

Figure 5.1: Defragmentation of the Memory Heap.
After the client C_A closes its session and the corresponding object is destroyed (2), the server has credits for two memory pages only (represented by the black rectangles). It must move C_C in the available slot to free a page (3).

broker must update these pointers for all the credits related to a session object when moving it. Another option would be to use session identifiers rather than pointers. However, searching the session array to find a grant partner is more costly than updating the (usually few) members of the credits and debts lists when a session object moves (which occurs when a session is closed, a rather rare event). Of course, an optimization of the session search could change this trade-off. The memory cost of each session object is deduced from the balance of the session that it represents. Although credits are related to two sessions (the creditor and the debtor), the debtor alone pays for the credit.

In a multi-threaded server, there is a risk that a worker thread moves a session object while another references it. Each time a worker closes a session, it moves the last session of the array, which may have no relation with the closed session. To avoid concurrent modification, each session object contains a lock that protects the session object itself as well as its credits and debts list. Before modifying or moving a session object, a worker must lock it. Before modifying a credit object, a worker must acquire both its creditor's lock and its debtor's lock in ascending identifier order. Moreover, each session object has a reference counter. A worker thread must wait for the reference counter to be 1 before moving a session object, to ensure that it is the only one referencing it.

5.5 Evaluation of Credit Trading

In this section, I evaluate the impact of credit trading on the server design, using the example of the broker. For an evaluation of the broker as a memory trading mechanism, see Section 3.2.3. Credit trading, that is, exchanging memory credits rather than pages, allows memory allocation of private server data with a finer granularity than a page. It relies on two properties of the memory allocation system that the server uses:

quota trading: the broker must be able to allocate (trusted) memory from its client's quotas.

signaled revocation: the server needs a pre-revocation signal to ensure that it can copy data before pages are removed from its address space.

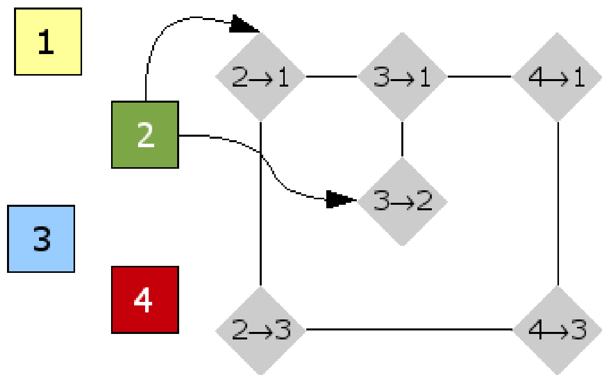

Figure 5.2: Representation of a Sparse Matrix.
Squares are session objects, lozenges are credit objects. Each credit is part of two circular doubly-linked lists (its row and its column) and has a pointer to its debtor and to its creditor. All coefficients are positive (the zero coefficients are left out). Each session object holds a reference to the list of its credits and its debts (shown for the session object 2 only). Reciprocal donations (between 2 and 3 in this example) are allowed and are not substracted from each other.

To benefit from credit trading, a server must have an adequate memory management system which allocates memory through the broker and prevents memory fragmentation, as does the block-wise memory allocation scheme implemented in the memory broker. However, the server designer must implement the migration of objects allocated from client-backed memory. Block-wise memory management makes the allocation of virtually contiguous memory blocks costly: The server can allocate two or more blocks on behalf of a single client, but it may need to migrate one of the blocks (and therefore, all of them to keep them contiguous) to defragment its memory. Hence, data structures that need important quantities of continuous memory such as arrays or stacks are costly to implement. Moreover, the server designer should choose data structures that facilitate moving objects around (for instance, linked lists rather than arrays) to reduce the defragmentation overhead. Data structures that have low deletion and insertion costs are well suited, because moving an element is at most as costly as deleting and re-inserting it[1]. On the other hand, an unbounded number of pointers to an object makes its copy cost unpredictable. This is the case of the broker session objects, which are referenced by many credit objects. As mentioned before, objects could use unique identifiers (which never need to be updated) rather than address to reference other objects. However, actually accessing the referenced object would require a costly search instead a simple pointer dereference. A general study of the data structures that are most suitable to use in this context remains to be done. A generic memory allocation framework in the broker client library can remove the burden of memory management implementation from the server designer. Such a framework would

[1]In the case of balanced trees, an element can be moved by copying its content and updating inbound pointers. This is less costly than deletion and insertion, because there is no need to search the insertion location and to re-balance the tree.

fulfill the same role as the slab layer does in the Linux kernel, with the added requirement that it must move objects from one slab to another to consolidate slabs.

Credit trading can significantly reduce the memory footprint of servers that have many clients and allocate relatively small memory quantities (in comparison with the page size) per client. For instance, with a page size of 4KB, it is unlikely that the memory broker allocates more than a page for a given client (a page can contain one session object and up to 144 credit objects). We can thus assume that, without credit trading, its memory consumption would amount 4KB for each opened session . With credit trading and enough clients, its memory consumption tends toward its real memory needs. Assuming an average of 12 credit objects and one session object for each client, this would amount to 392 bytes per client. For a server that allocates several MB per client however, the gain would be negligible.

My memory broker implementation does not allow the server to choose which page to give back when credit grants are revoked. However, a more elaborate broker could keep track of which pages are mapped in which address space[2], and let the server unmap any of its client-backed pages. The server would have the guarantee that it can keep a given page as long as it has sufficient credits for this page only (in the FIFO mapping scheme, the server must have enough credits for older pages as well). The server can take advantage of this added flexibility in several (not mutually exclusive) ways:

shared memory: The server could map pages that it got from the broker to its clients, thus sharing client-backed memory. The sharing is not limited to two partners and is fully under the control of the server —as long as the clients do not revoke their credit grants.

multiple heaps: The server could use one heap per object type. If a heap shrink, the server can unmap a page devoted to this heap, even if the page is not the youngest one.

virtually contiguous memory pages: the server can allocate client-backed memory in contiguous pages instead of block-wise, and use large arrays or stacks. This is a special case of the one heap per object type strategy, with a granularity that is the size of a page.

[2]The broker maps a given page at most once, so it does not need to keep track of recursive mappings like the microkernel does.

Chapter 6

Conclusion

The L4 microkernel exports memory management policies from the kernel to user-level tasks. Letting a server use client-backed memory amounts to exporting the memory management policy from the server to its clients. In this approach, each client directly or indirectly plays the role of a dataspace for the server.

Directly using untrusted clients as dataspaces requires fault handling from the server side because clients may not correctly handle faults, as the implementation of direct mapping shows. This fault handling mechanism could apply to other kinds of system faults that require application-specific handling. It does not allow the server to securely use untrusted dataspaces to store confidential or critical data, however. Using a trusted entity that has access to the client's quota (the broker) as a dataspace solves security problems and allows memory credit trading (rather than page trading), which can optimize memory usage. As clients provide physical memory, memory becomes plentiful —it no longer limits the number of concurrent opened sessions— and virtual address space takes the role of the limiting resource that requires management.

This work opens the way to several extensions. Firstly, a more accurate evaluation of the costs and benefits of credit trading would require a systematic analysis of the cost of memory defragmentation depending on the server data structures. The actual development of servers that trade memory credit with their client would greatly benefit from an integration into an environment and a memory management library that transparently handles broker sessions, quota exchange, credit trading and memory defragmentation. This library could include implementations of data structures that reside in client-backed memory. As the benefits to be expected from memory trading increase with the number of clients using it, such a library would contribute to improve the memory efficiency in addition to making the usage of client-backed memory simpler.

Moreover, while the direct approach exports some of the memory management policy from the server to the memory-providing client (the paging policy), the indirect approach does not allow the client to control how the server uses the client-backed memory. As an extension of memory trading mechanisms, the servers could offer an interface that allows clients to specify the usage policy of the memory that they lend. To that end, the client would send a specification to the server. The server would interpret and apply it after checking that the policy complies to its own policies.

Alternatively, extensions of the L4 IPC mapping mechanisms, such as untyped memory and kernel-enforced pre-revocation signaling (that is, having the kernel notify a task before unmapping a capability), could bring some of the advantages of the indirect approach to the direct approach. It would allow a server to use client memory without concern for confidentiality. While untyped memory is already implemented in capability-oriented L4 systems like seL4, determining the exact semantics and the implementation complexity of pre-revocation signals requires further work.

Lastly, the resource-for-a-service approach could extend to other resources that tasks can exchange. CPU time and kernel resources such as threads would be interesting candidates.

Bibliography

[David, Carlyle *et al.*] Francis M. David, Jeffrey C. Carlyle, Ellick M. Chan, David K. Raila, and Roy H. Campbell. Exception Handling in the Choices Operating System, in C. Dony et al. (Eds.):Exception Handling, LNCS 4119, pp. 42-61, 2006 Springer-Verlag Berlin Heidelberg 2006

[David, Chan *et al.*] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, Roy H. Campbell. CuriOS: Improving Reliability through Operating System Structure, 8th USENIX Symposium on Operating System Design and Implementation, December 2008

[Derrin *et al.*] Philip Derrin Dhammika Elkaduwe Kevin Elphinstone. seL4 reference Manual:
http://www.ertos.nicta.com.au/research/sel4/sel4-refman.pdf

[Elkaduwe *et al.*] Dhammika Elkaduwe, Gerwin Klein and Kevin Elphinstone. Verified protection model of the seL4 microkernel, Proceedings of VSTTE 2008 Verified Software: Theories, Tools and Experiments, Toronto, Canada, October 2008

[Feske, Helmuth 05] Norman Feske and Christian Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI, Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC), 2005

[Feske, Helmuth 06] Norman Feske, Christian Helmuth. Design of the Bastei Architecture, TU Dresden technical report TUD-FI06-07 September 2006, ISSN 1430-211X

[Hamann *et al.*] Claude-J. Hamann, Lars Reuther, Jean Wolter, Hermann Hrtig. Quality Assuring Scheduling, TU Dresden technical report TUD-FI06-09, December 2006

[Hand] Steven M. Hand. Self-paging in the Nemesis operating system, Proceedings of the third symposium on Operating systems design and implementation, 1999

[Härtig *et al.*] H. Härtig and R. Baumgartl and M. Borriss and Cl.-J. Hamann and M. Hohmuth and F. Mehnert and L. Reuther and S. Schönberg and J. Wolter. DROPS: OS Support for Distributed Multimedia Applications, Proceedings of the Eighth ACM SIGOPS European Workshop, September 1998

[Kauer] Bernhard Kauer. L4.sec Implementation - Kernel Memory Managment, TU Dresden, May 2005

[L4Linux Web page] L4Linux Official Web page:
http://os.inf.tu-dresden.de/L4/LinuxOnL4

[Liedtke 93] Jochen Liedtke. Improving IPC by kernel design, ACM SIGOPS Operating Systems Review, December 1993

[Liedtke 95] Jochen Liedtke. On -Kernel Construction, 15th ACM Symposium on Operating System Principles (SOSP), December 1995

[Liedtke *et al.*] J. Liedtke, K. Elphinstone, S. Schonberg, H. Härtig, G. Heiser, N. Islam, T. Jaeger. Achieved IPC performance (still the foundation for extensibility), The Sixth Workshop on Hot Topics in Operating Systems, May 1997

[Liedtke *et al.* 99] Jochen Liedtke, Volkmar Uhlig, Kevin Elphinstone, Trent Jaeger, Yoonho Park. How To Schedule Unlimited Memory Pinning of Untrusted Processes Or Provisional Ideas About Service Neutrality, The Seventh Workshop on Hot Topics in Operating Systems, March 1999

[Love] Robert Love. Linux Kernel Development (2. ed., 3. print. with corr.), Novell Press, 2007, ISBN 0-672-32720-1

[Mach Web page] The Mach project Web page:
http://www-2.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html

[Rozier *et al.*] M. Rozier, A. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard and W. Neuhauser. CHORUS Distributed Operating System, Computing Systems 1(4), volume 1, 305370, 1988

[Shapiro] J.S Shapiro. EROS: A Capability System, University of Pennsylvania, April 1999

[Stroustrup] Bjarne Stroustrup's Homepage at AT&T Research:
http://www2.research.att.com/ bs/bs_faq2.html

[TUD OS group] L4Env documentation page on the TU Dresden OS Group web page:
http://www.inf.tu-dresden.de/index.php?node_id=1582