Implementation of an Orthogonally Persistent L4 μ-Kernel Based System

Christian Ceelen
ceelen@ira.uka.de
Supervisor: Cand. Scient. Espen Skoglund
Universität Karlsruhe

15th February 2002

Abstract

Orthogonal persistent systems open up possibilities for a wide number of applications. Even more, it is a very natural concept for the storage of information, since objects and information persists until the end of their lifetime. Most current commercial non-persistent systems have only an explicit storage model. Thus, an application has to care for the persistent storage of data itself. This has to be done by transforming the data structures into something that can be stored within a file. Furthermore the file has to be opened, written to and saved explicitly; a source of overhead for programmers. Moreover the programmer also has to estimate the life-time of all valuable data. Including the conversion and recovery of data, the amount of code needed to store data explicitly could easily take up a third or half of the actual programming work.

In order to support a convenient system environment, persistent storage could be handled implicitly by the operating system. The operating system has to store for each task an image of the user memory and all kernel internal data like page-tables, mapping structures, file descriptors and so on. This approach is very demanding and very error-prone for current monolithic systems. Therefore we propose a μ-kernel based system instead. The proposed work should provide an implementation base for further persistent systems by supplying the necessary mechanisms to build persistent applications on top of the μ-kernel. The stable storage of all data is achieved by regular checkpointing of all user-level memory and all needed kernel meta-information through user-level mechanisms.

Since μ-kernel are still a matter of in depth research, this thesis also pushes the μ-kernel idea to its limits by applying the concepts learned previously on a orthogonally persistent system. Further it analyze feasibility for a system structured this way.

Contents

Chapter 1

Introduction

This chapter gives a short introduction to the motivation for the thesis and describes related work in context of orthogonally persistent systems.

1.1 Motivation

In an orthogonally persistent system the persistence of an object is orthogonal to its type and implementation. Every object persists until it is no longer needed and after this end of life-time it is deleted explicitly (e.g., either by the user or a garbage collection system). Orthogonal persistence is therefore a natural concept because it is exactly what people expect when manipulating data. People not familiar with the explicit storage model of a computer would expect data to be orthogonally persistent. They would draw from their experience with existing "storage systems" like pen and paper, that after creation and writing the data on the paper the information is preserved through the special physical nature of the ink and the paper. In context of a computer, two different kind of papers exists. The first one persists, the other one looses its contents soon after usage. The first one is the natural behavior of paper, the second one is that of normal computer memory. One would use the latter for short-lived drafts and notes, but use the real paper for anything that has any worth. Since the amount of time needed to evaluate and write down the data and is much more precious than wasting a sheet of paper. Therefore one would even use the persistent paper when writing things that initially are not thought to last long, just because it might unexpectedly turn up as more valuable than initially expected. This way one does not have to make an explicit copy of the data. In terms of computer systems the amount of extra work needed for orthogonal persistence would pay for the convenience. There are a number of applications that do not need this kind of convenience, however, for instance shortlived schedules, notes, or scientific computations. Only while

developing optimized applications would a developer care about using memory that is not orthogonally persistent, since this feature does come at cost of some performance.

There are many ways to implement an orthogonal persistent system and guarantee the consistency of the stored objects. The one chosen for this thesis is the use of system wide checkpoints, because it could be understood easily and is simple to implement. Furthermore, it is possible to implement checkpointing through user-level mechanisms. This way the orthogonal persistence is an optional feature and not a mandatory one. In order to achieve persistence some of the kernel internal structures have to be accessed by the checkpointer. The needed meta-data needed additionally to the checkpoint could include page-tables, mapping structures, file descriptors and so on. After perfoming the checkpoint and gathering the necessary meta-data everything has to be written to stable storage. This is quite complex since the consistency of the system has to be ensured at any point in time. Therefore one motivation of this thesis is therefore to reduce the amount of work needed to build an orthogonally persistent system. Future research and development should be able to focus on problems that appear within a system supporting persistent and transient application.

1.2 Persistence On Top Of A μ-kernel

Many monolithic systems are designed around the file-abstraction. Monolithic systems like Unix or Plan 9 try to use this abstraction of files and directories as often as possible, even if it is sometimes inappropriate while dealing with structured data. The file model is good in the usage of hard-disks or other block-oriented devices and serial streams, but not structured objects, e.g., graphs. If an orthogonally persistent system is used, the usage of a persistent filesystem to organize data and memory is no longer necessary. Thus a new hierarchy to structure the information is needed. The new hierarchy could be a tree-like directory structure comparable to standard filesystems, or any other structure that binds objects to a human-readable naming scheme. This naming scheme is important for users of the system. The main difference to a normal filesystem now is that these objects may be raw data streams (files) or any kind of structured data used by an application.

The μ-kernel approach of structuring a system is to use very few well defined abstractions. These are carefully designed to allow any kind of policy to be implemented on top of the kernel. The abstractions and the μ-kernel concept itself, however, is still a matter of research. The practicability still has to be proven by implementing non-standard system semantics like orthogonal persistence on top of a μ-kernel. If it turns out that the implementation of these non-standard se-

mantics fit neatly into the existing abstractions, the μ-kernel concept comes closer to prove its flexibility and feasibility in practice. A further motivation for using a μ-kernel is the small number of well understood concepts. Dealing with, e.g., complex memory management and open files in a monolithic system pose much more problems and is much harder to implement correctly than on top of a μ-kernel. Persistence can easily be added to systems built on top of a μ-kernel since only a very small part of the system have to be modified.

1.3 Related Work

The concept of orthogonal persistence is not a new one. In L4's predecessor L3[11] it was an integral part of the kernel and persistence was system-wide. In L3, during the write-back of the checkpoint all TCBs were removed from all kernel internal queues and marked copy on write, thus blocking the whole system. However, in L4 persistence is not an integral part, but can be implemented on top of the kernel.

There are several other μ-kernels that integrate persistence. For instance EROS [9] and its predecessor KeyKOS [10] are capability based systems and implement persistence at kernel-level. During a snapshot EROS also performs a consistency check of critical kernel data structures. This catches possible implementation bugs and prohibits these from stabilize in the system, rendering the system unusable. This check is unnecessary in L4 since no critical kernel structures are included within the checkpoint.

Fluke [14] is a μ-kernel that exports user-visible, partly "pickled" kernel objects to the user. This way a user-level checkpointer "pickles" the remaining parts of the kernel objects and save these along with the memory image of the tasks.

Grasshopper [16] is a larger system, because it is not based upon a μ-kernel and uses abstractions specially designed with persistence in mind. These kernel abstractions are used to implement persistence on kernel level. Based on the experience with Grasshopper its designers later created a μ-kernel based operating system, Charm [17], to support persistent applications. In Charm, the kernel provides the application system with mechanisms to implement its own persistence policy by exposing all in-kernel meta-data to the application. The application has to ensure persistence itself by storing the data. As such, no specific persistence model is enforced.

There are also persistent single address space operating systems like Mungi [21], Opal and many other. In these systems protection and security is not implemented by address space boundaries but by capabilities or strong language environments. Orthogonal persistence in these systems is just the extension of the systems paging mechanism to store the main memory on a storage device. As

there are no different address spaces less meta-data like page-tables and mappings than in the case of a multi-address space system have to be stored besides the memory itself.

Another common way to achieve user-level checkpointing is to use the `fork()` UNIX system call to periodically create a snapshot of the task's image [15]. This allows the application to continue execution while the checkpoint is written to stable storage. The major disadvantage is the loss of almost all kernel state upon a system crash. Therefore, it is not possible to use every feature of the system and persistent application is mainly limited to special libraries. Moreover, the consistence of the checkpoint could not be guaranteed if the application interacts with the surrounding system, because it is a per task checkpoint. Therefore this checkpointing facility has only a limited number of applications mainly in the field of scientific computation.

1.4 Aim of Thesis

The goal of this thesis is to build a transparent orthogonal checkpointing mechanism on an existing μ-kernel system, the L4-μ-kernel. First the necessary additions should be discussed regarding the abstractions of the L4 μ-kernel. Then the proposed mechanisms should be implemented using the current L4Ka μ-kernel developed at the Universität Karlsruhe. This should provide the basis for further exploration of the concept of persistence.

The integration of the proposed mechanisms should prove to be a somewhat easy matter given that the right abstractions are used. Some problems concerning consistence of the system arise while dealing with legacy services not aware of persistent applications. But it is beyond the scope of this thesis to fully discuss the problems arising through orthogonal persistence in conjunction with transient applications, and describe all system components to implement orthogonal persistence. For a brief discussion of these issues see the paper[13] upon which this thesis is based. The main problems that have to be dealt with is the data consistence of the system at any point in time and the persistent storage of all application data. For instance, persistent applications may communicate with transient ones.

Several problems are expected in how to achieve a consistent snapshot of all kernel internal data and how the kernel could continue execution while a huge amount of its data is locked or read-only (e.g. read-only TCBs). In the L4 kernel this may have side effects on other parts of the kernel (e.g. communication, kernel memory allocation, kernel page fault handling).

1.5 Structure of the Thesis

Chapter 2 gives a brief introduction to the L4-μ-kernel. The concepts provided by this μ-kernel are adapted in Chapter 3 to support persistent application through user-level servers. The implementation of the necessary kernel mechanisms and the major problems that arose during the implementation due to the special design of L4Ka are discussed in Chapter 4. Thereafter some conclusions are briefly reported in Chapter 5 regarding portability and flexibility of the L4Ka-μ-kernel. Since the implementation of the user-level mechanisms and servers can be derived straight-forwardly from the conceptual design, they will not be discussed within this thesis.

Chapter 2

The L4-μ-Kernel

2.1 Motivation for a μ-Kernel based System

Growing complexity is a major concern of current systems. In terms of operating systems this complexity comes from the amount of supported services and control facilities (e.g. many supported file systems, emulation of other OSs and so on). In most current commercial operating systems these services are implemented as one central monolithic kernel. All supported services and control facilities are implemented within this single kernel, execute within the same context and therefore use the same resources. In Figure 2.1 an example for a monolithic system is pictured on the left side. All system services are concentrated within the monolithic kernel and execute with full privileges to control the hardware and the system. Therefore an implementation bug may show consequences in any other part of the kernel. This bug may cause an immediate reboot of the system or fail and settle silently. Even worse, the error could start a whole chain of further errors at completely unrelated locations in the kernel. These kind of errors are **very** difficult to debug. Therefore, as any complex system most monolithic systems have a number of bugs shipped with them, which could not be found due to the complexity and size of the kernel. E.g. Engler [18] applied a source-code checker to find program errors in Linux 2.4.1 (dereferences of null-pointers, missing unlock of a semaphore after a lock, etc.) and found more than 200 errors in code which checking for null-pointer and also 12 security errors. Despite this amount of errors, if implemented properly, a monolithic system is able to perform better than a μ-kernel-based one. However, the performance overhead induced by a μ-kernel can be decreased to the point where it is negligible. In a μ-kernel-based system the size of code of the kernel is manageable. Other OS-services are implemented outside the μ-kernel, thus the kernel is independent from these. Therefore the kernel might survive a failure of a single or several OS-services and is not influence

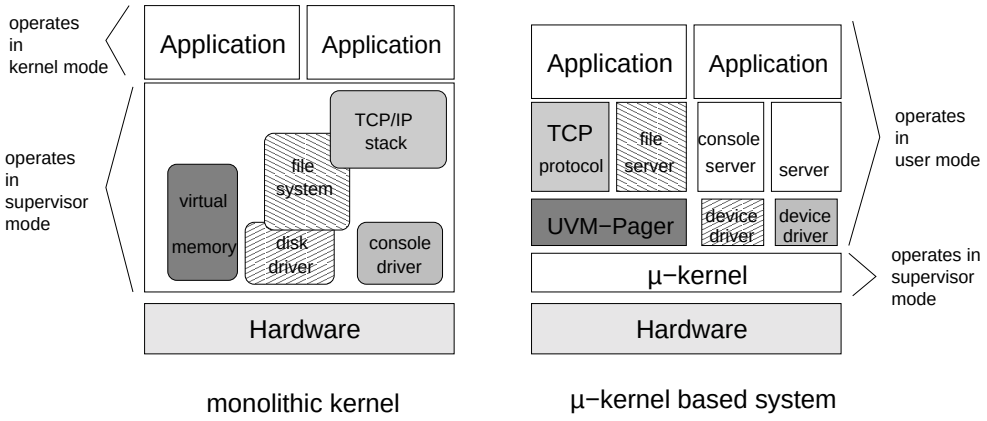


Figure 2.1: A monolithic kernel structure compared to a μ-kernel-based system with several services implemented in different tasks.

by other bugs present in the service system. Due to its small size it is manageable and easier to maintain than a larger kernel. It is even believed feasable to formally verify such a μ-kernel[22, 23].

2.2 μ-Kernels in General

Micro-kernels take a step in the other direction of the monolithic kernel architectures described above, and offer a set of well known basic mechanisms and abstraction. Most μ-kernels offer address spaces to implement a memory protection scheme to provide protection between tasks, means to manage hardware resources, and threads to manage the flow of control and time resources. They also provide some mechanism to communicate between threads in different address spaces, i.e., inter process communication (IPC). The IPC mechanism can used to implement, e.g., synchronization and shared memory. A detailed discussion about selecting appropriate abstractions for μ-kernels and how to implement them can be found in [3].

Using the abstractions described above, almost any kind of system structure can be built on top of the μ-kernel, even monolithic kernels (see L^4Linux in [12] or MkLinux [20]). The abstractions can also be used to build a component-based or multi-server system (see right sketch in Figure 2.1) as Sawmill Linux [24]. For each service a clean interfaces is specified to ensure compatibility between different versions and implementations of the same service. These components can now be implemented protected from each other in different tasks or in almost any other fashion. This allows bugs to be isolated much easier. By dividing the system into more and more fine grained services and components, communication

between the services gets much more frequent. Thus the IPC operation should be designed very carefully to allow fine grained system structuring without any performance overhead.

2.3 Short History of μ-Kernels

The first generation μ-kernels were built top-down by conceptually stripping a Unix-like system kernel from all unnecessary parts and services that could easily be implemented outside the kernel. In these μ-kernels the overhead to set up an IPC-message and deliver it was a major performance problem. A much worse problem was that the time needed to do an IPC operation did not scale with the processor speed. Thereby it is not surprising that the overall performance of systems using these μ-kernels has been devastatingly slow[1]. The main performance problems come from the complex asynchronous IPC-mechanism. During the sending of a message, the message must be copied multiple times because of the need for in-kernel buffering. This drastically slows down the message transfer and furthermore fills the processor caches. Since caches have become more and more performance critical these days, since system-calls generally require a message transfer, and since every message transfer is an IPC, the speed of the IPC operation dictates the overall performance. The second generation μ-kernels researched nowadays are built bottom-up and implement a much smaller subset of the previous kernels. The design has been focused particularly on the IPC operation to allow fine grained system structures with a high need for communication. For further details on μ-kernel design issues see [4] and [3].

2.4 The L4 μ-Kernel in General

The L4 kernel is a 2nd generation μ-kernel. It strives to be policy free by allowing as few concepts within the kernel as possible. This is achieved by only permitting concepts within the kernel if an implementation within user-level would be impossible or prevent other functionality to be implemented. Following this device drivers and memory management (paging, swapping, etc.) are implemented in user-level tasks. Based on this philosophy, the main abstractions used within the L4 μ-kernel are threads, address spaces and IPC (see [3, 4] for the rationale). All other concepts, abstractions and services have to be implemented outside the

[1] Even though Apple nowadays use Mach 3.0 for their new MacOS X and do get satisfactory performance, they archive performance by avoiding the usage of the Mach's IPC-operations as much as possible.

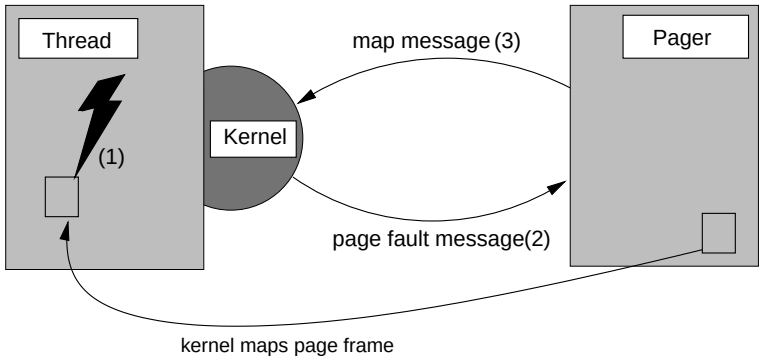


Figure 2.2: A page fault (1) is caught by the μ-kernel and translated into an IPC (2). The page fault handler thread maps a page (3) to the faulting location in the address space of the faulting thread.

μ-kernel in order to get an operational system. As a consequence of this minimalistic μ-kernel approach the L4 Version X.0 API [7] specifies only 7 system calls[2] that implement mechanisms to use the basic abstractions.

2.4.1 Address Spaces, Threads and IPC

An address space is L4's abstraction for memory and protection. It contains all memory objects an application can access directly. A thread is an activity executing entity inside an address space. The address space defines its execution context and can be shared with other threads. A set of threads sharing an address space and the address space itself is commonly referred to as a task or a process. Threads within different address spaces are protected from each other while threads within the same are not. Threads can communicate with each other by inter-process-communication (IPC).

The IPC primitive is the most carefully designed primitive and most fundamental mechanism in the L4-μ-kernel. It is the only means of communication between threads which does not share an address space, and is also used as an abstraction to signal exceptions and interruptions. For example all interrupts issued by I/O devices or other hardware components are sent as an IPC message to a thread that handles the interrupt.

All IPCs are synchronous operations. Thus, both the sending or receiving thread must agree on the message transfer and there is no need for buffering the message inside the kernel. In order to cope with faulting or not trustworthy communication partners, each IPC is bounded with a timeout. This timeout can be a zero, infinity or a value between approx. 1μs and 19 hours.

[2] System calls are kernel operations invoked by an application.

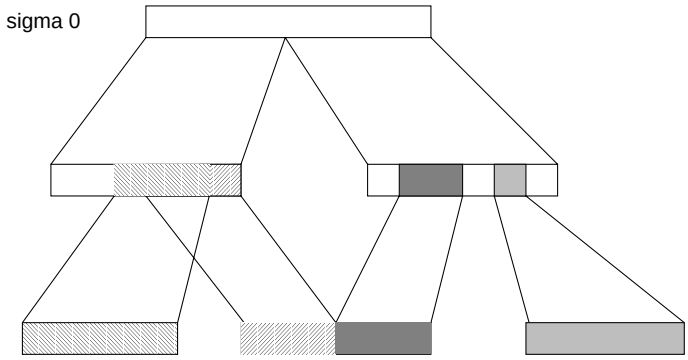


Figure 2.3: The address space of all pagers and applications is backed by σ_0

2.4.2 External Pagers in L4

The L4 kernel does not implement any kind of memory management policy such as (demand) paging, swapping and address space organization. It merely implements basic mechanisms to implement these different policies through user level pagers, and offers just the necessary hardware specific memory management mechanisms to construct further address spaces recursively from already existing ones. In L4 every thread has a specific pager that backs its address space. It has to trust this pager completely since the thread only "borrows" its memory from the pager. Whenever a page fault exception occurs, it is automatically translated by the μ-kernel into a page fault message and send to the pager of the faulting thread. The pager can then map a page in the address space of the faulting thread by sending a map message. The μ-kernel intercepts the map message and maps the page sent by the pager into the address space of the faulting thread (see Figure 2.2). During the page fault handling the faulting thread is suspended.

Because only recursive address space construction is allowed in L4, an initial address spaces is needed that resembles all physically addressable memory. This address space is called σ_0. Other address spaces are constructed hierarchically by layering address spaces on top of σ_0 (see Figure 2.3). σ_0 itself maps pages using physical addresses to other address spaces if a page fault occurs. To implement virtual memory a pager layered between the application and σ_0 is needed that translates a virtual a addresses to physical one. This way a hierarchical address space organization and protection scheme can be constructed (see Figure 2.4).

σ_0 hands out all memory initially and the pager below σ_0 usually hand out free pages downwards in the hierarchy. σ_0 just unconditionally hands out all pages on a first-come first-serve basis.

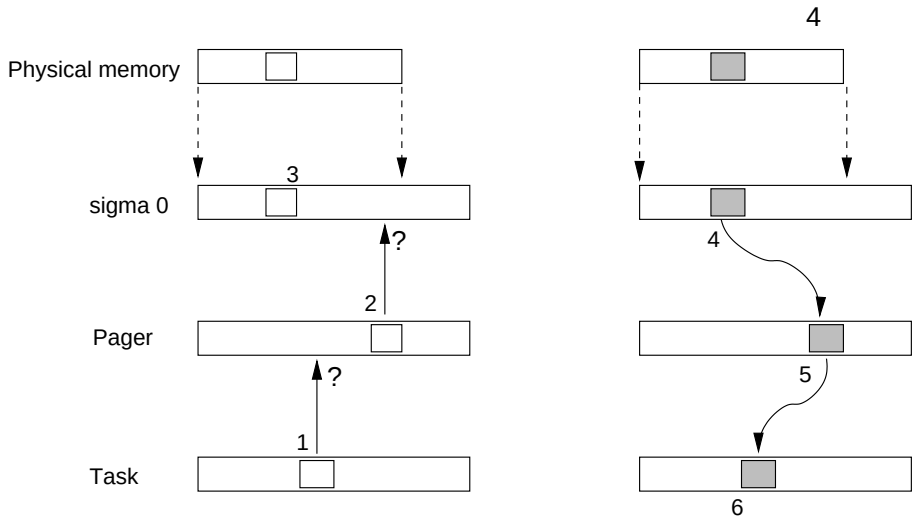


Figure 2.4: The page fault(1) that occurred within the a task is propagated to the pager of the task. The pager tries to allocate an empty page to resolve the page fault and sends a page fault message (2) to σ_0 to allocate a new page frame. σ_0 searches for an available page (3) and maps it to the pager (4). The pager maps the empty page frame to the task (5) and the task is restarted (6).

2.5 L4 API

The L4 API (actually an ABI) specifies 7 system calls to construct operating systems upon. It uses register-only parameters for as many system calls as possible and is therefore tightly coupled to the hardware architecture. A discussion of the system calls can be found in [5]. The register-only parameters are favored to speed up kernel calls by requiring fewer memory references. It also reduces the complexity of system calls because they do not have to cope with page faults while accessing the parameters. Therefore many system calls can run atomically. The IPC system call is a bit special in this regard. All message parameters are passed within the registers, but the message itself is copied via memory. However, parts of the message can be passed via registers as an optimization.

The amount of threads and address spaces in the L4 API is limited. In the experimental L4-X.0 API a thread ID is 32-bits, whereas the previous Version 2 of the API had 64-bit thread IDs. Within the 32 bit ID of the experimental API 8 bits (the task number field) identifies the address space and 6 bits (the local thread number field) identifies the thread within the address space. There is also a field to specify a version number to reuse thread IDs, and a chief field of 8 bits. The chief field is a relict of the clans and chiefs model [6]. It is a mechanism to control messages sent to a thread not within the same task group (called a clan). Each

message to an exterior task has to pass through each chief of each clan involved. This needs to be used in order to build secure systems. The clans and chief-model has been dropped for the more flexible IPC-redirection model [8], which will be part of the forthcoming API.

2.6 Implementation Notes on L4

Most system calls can be implemented so that they are time-bound. Therefore, the system calls can be executed with disabled interrupts since they are short enough to run atomically. There are some issues with this scheme, though. First, there is the long message transfer time when many message words (often referred to as long IPC). Then there is the time taken to unmap a page. Since a page frame can be mapped arbitrarily often, there could be an arbitrary number of page tables that have to be changed. These issues can prevent realtime systems with tight time constraints to be used on top of the μ-kernel because the maximal response time to signals or interrupts could be prolonged by the noted system calls.

The long IPC can be implemented in way that copying from one address space to another is preemptable. If the copy operation is interrupted, it is simply resumed at a later point in time. All other message transfers are relatively short and thus do not interfere with the time constraints. The un-mapping of a page is a bit more problematic because the operation is heavily accessing kernel internal structures. Even worse is the fact that the duration of the system call depends on the amount of existing mappings. It is possible to make the system call preemptable, though, if the accessed kernel structures are locked for the duration of the modification. One problem with locks is that it adds to the complexity of the kernel when, e.g., destroying a thread which is holding a lock. The usage of a lock-free mechanism to ensure consistent data structures would pay off here. Another possibility is to restrict the amount of mapping the system is allowed to do, but this requires changes to the API itself and is therefore not an option for an implementation using the L4-X.0 API.

Chapter 3

Design

3.1 Conceptual Design

There are many ways to achieve a persistent system. A conceptually simple and well-understood method is the periodical generation of a checkpoint. The main problems tied to this checkpoint model is the generation of consistent checkpoints and the recovery of the state after a system crash. This work proposes periodical checkpoint generation as a way to achieve a persistent system and describes how to implement it upon a modern μ-kernel. These checkpoints are per-machine and not synchronized with any system external entity. Also, synchronization between persistent and transient tasks, threads and further objects are not discussed.

The vision of this work is to implement a transparent user-level checkpoint server acting as a memory pager that makes regular snapshots of the persistent applications. Moreover, this should work orthogonal by other user-level services (pagers, file systems, security layers, etc.). Figure 3.1 gives an overview of the envisaged run-time environment. Every application that has all its memory backed by the checkpoint server is going to be persistent.

3.2 Persistent and Transient Objects

In a real system there will always be a mixture of persistent and transient objects. In Figure 3.1 transient objects are all components beneath or next to the checkpoint server or components which are explicitly marked transient. These include device drivers, which can not be made persistent since hardware usually has some implicit transient state, and other system services that implement the proper protocols for object handling in context of these devices. In case of a traditional file systems, all objects dealt with are explicitly persistent because they store their state on a stable device.

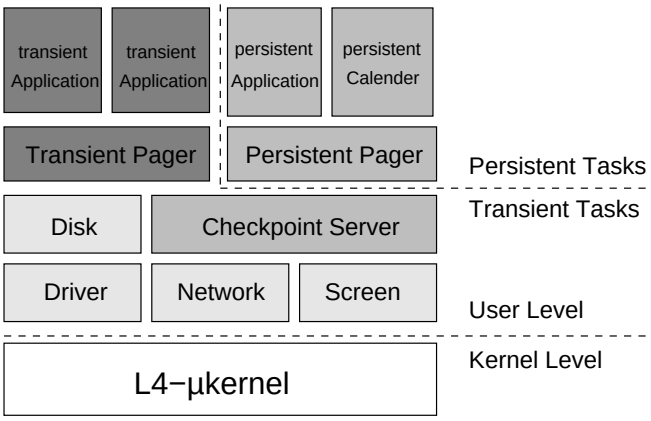


Figure 3.1: Architectural overview

Persistent applications might want to use these services and other transient services or applications (e.g. a time server or remote web site). In order to support interaction between transient and persistent applications, a system can only offer basic mechanisms to detect inconsistencies. Using this mechanism a protocol can be implemented that ensures the consistency of the shared object. A general method to recover from this inconsistency would be to invalidate all connections to transient applications and restart the corresponding operations. These problems are not discussed in this thesis.

3.3 Implicit Persistent Objects

Implicit persistent servers and applications are applications that originally do not include persistence. This is now added by transparently changing the set of system servers used by the application. The servers handle open connections, dangling references to transient services and consistence problems. Pagers and other legacy services that are explicitly persistent have to be exchanged and have either to be persistent themselves or recoverable. This new user-level environment takes care of generating synchronized checkpoints of the main-memory of the application, the kernel memory state, and backing-store state.

Most implicit persistent applications are going to use a legacy filesystem, since they are not aware of their persistent nature. The normal usage of a stable storage system could lead to consistency problems. For instance, an application reads some data a from a file, alters it to a' and writes a' back to the file. If the system crashed before a checkpoint includes the changed disc state, the previous checkpoint is flawed. Upon system recovery, an application state is recovered that is prior to the disk access and therefore resembles a state when the application ex-

pects a within the file, but gets a'. Therefore, a recoverable filesystem is needed for implicit persistent applications. This can be achieved using a recoverable-disk driver that maintains the disk state of the last checkpoint and the modifications made in between checkpoints. Using this, the previous disk state can then be recovered.

3.4 Main-Memory Checkpointing

In L4 address spaces are constructed recursively. Hence it is sufficient to have one initial persistent address space to construct further persistent ones. This initial address space offer persistent memory to other applications and is responsible to periodically store all used pages to stable storage. Since all memory frames given to persistent applications belong to the checkpointer, the checkpointer has full access to the memory of its clients. Therefore the clients have to trust the checkpointer. The checkpointer can be introduced somewhere within the pager hierarchy. If it is inserted above all persistent pagers, it automatically checkpoints all physical memory frames used by persistent applications including all mapping information stored by their pagers.

Kernel state of a thread is also stored in some kernel structure which is located within the main memory. Let's assume for now that this is magically included in the memory checkpointed by the user level checkpoint server. How this is going to work is explained in detail in Section 3.5.

3.4.1 Write-back of a checkpoint

As all external pagers, the checkpoint server itself is a regular user-level task. Therefore it is subject to interruptions. These interruptions may occur at any time during the checkpointing algorithm, even while writing a memory page to disk. During the interruption a persistent application might be scheduled that could try to change its state and thereby modify the memory currently checkpointed. This is not a desired behavior since the consistence of the checkpoint is violated. Therefore, the storing of the memory snapshot has to be atomic. Atomicity could be achieved by several means. One could virtually suspend all other threads, by modifying the priority of the running threads or telling the kernel to stop all other threads. This would ensure global serialization of the write-back and all other threads. Fortunately it is already sufficient to ensure that all persistent memory is write protected. This way transient applications can run concurrently to the checkpoint server. Furthermore it is possible to execute the persistent applications concurrently if they are not allowed to modify memory which has not been stored on a disk yet. The checkpoint server can now achieve a consistent snapshot of

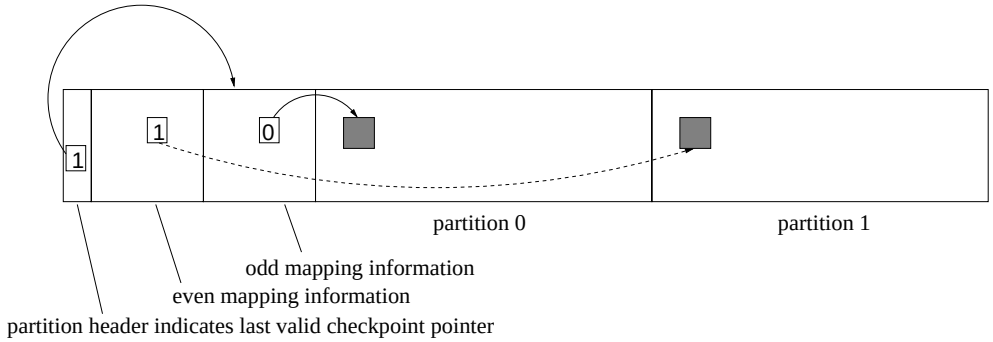
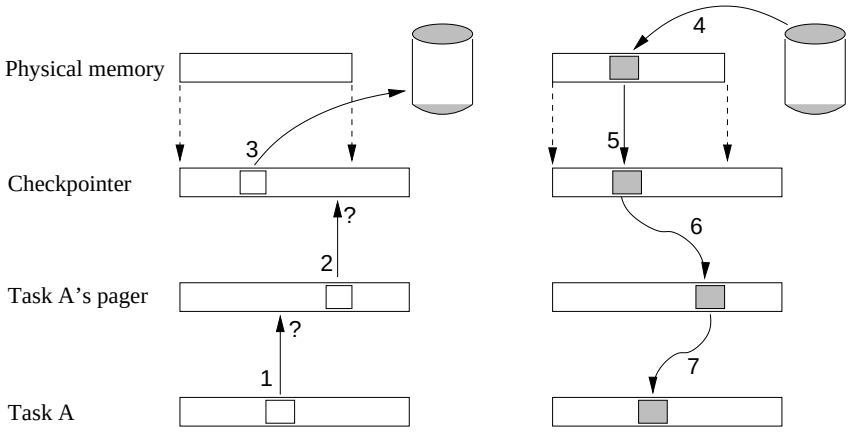


Figure 3.2: Partition Layout: The partition header indicates which block of $status$-bits has to be used. These bits tell which disk block has to be indexed.

the memory by marking all page frames given to persistent tasks read-only. At this point persistent threads are virtually suspended because every write access to memory causes now a page fault. Every page fault is propagated through the pager hierarchy and ends up at the checkpointer. Since the checkpointer is currently taking the checkpoint the page fault handling is delayed until the checkpointer has taken a consistent snapshot of the system image. Thus the checkpointer has control of all write accesses that might render the snapshot inconsistent. After the snapshot has been taken the pager could now either eagerly or lazily write any altered page frame to disk. Upon page faults it could either delay the mapping of the page until the snapshot has been written to stable storage, or copy it to a buffer and remap the page frame writable. This way persistent tasks do not have to be suspended fully during the write-back of the checkpoint, only during the copying of their pages.

The checkpoint mechanism used is based upon similar mechanisms as in Eros[9], KeyKOS[10] and L3[11] [1]. On the disk there are 2 disk blocks allocated for each page frame. These blocks are located on separate partitions and are used to store the data of separate checkpoints. A $partition$-bit exists per page frame that indicates which disk block contains the valid checkpoint of the corresponding page frame. If the contents of a dirty page should be written to disk the page is stored in the disk-block not indicated by the $partition$-bit. If a crash happens during the write operation there is a valid memory image on disk, but without the $partition$ bit-map it is not reconstructible. Therefore this bit-map has been stored on disk, too. Since the writing of a disk block can fail, an inconsistent disk image of the snapshot may occur. Thus a similar technique is needed for storing the $partition$-bits. For each checkpoint a disk area is allocated that stores all $partition$-bits of a

[1]The algorithm is a fair bit older, but the quoted ones are related to μ-kernels and therefore more similar to this than database systems.



(a) Page faults are propagated up to the checkpoint server which requests the page frame from stable storage.

(b) Once the page frame is copied into the main memory, the mappings are propagated down to the faulting task.

Figure 3.3: Cascading page faults. The numbers enumerate the order in which the steps are taken. The arrows in Figure (a) indicate page fault requests(IPCs) or data requests from disk to memory. In Figure (b) arrows indicate mapping of pages or data transfers from disk to memory

checkpoint. To distinguish in which of these areas the valid partition-bits reside, an overall checkpoint state is introduced (odd, even). The changed $partition$-bits are written to the disk-area which contains the outdated $partition$-bits of checkpoint prior to the last valid one. The principal partition layout is shown in Figure 3.2. An optimization of this checkpoint mechanism could write logs to minimize seek time whenever the amount of dirty pages is low. Since the amount of dirty pages depends on the checkpoint timing, it may pay off to reduce this time to benefit a log-structured disk usage.

3.4.2 Recovery of Memory Pages

After a system crash no applications are running and all address spaces are empty since all kernel internal data structures describing threads and address spaces are lost. These have to be rebuilt somehow. Now, if a persistent thread of a task is restarted magically by the kernel it would try to read its code. However, since the address space is empty the access generates a page fault exception. The page frame is backed by a thread-specific pager that manages this address space. Therefore, the page fault is propagated to the pager of the persistent thread (see Figure 3.3, label 1). The pager receives the message and tries to execute the necessary algorithms to remap the page to the persistent thread. Since its address space is

empty too, its code is missing and another page fault exception occurs. This page fault has to be resolved before the actual page fault of the persistent thread can be resolved. Since the pager is persistent and its address space was constructed from another one, its pager has to resolve this page fault. Once the pager is able to handle the request it accesses the requested page, thereby generating another page fault that is sent to the checkpointer (label 2). The checkpointer now tries to find the appropriate page-frame within its mapping data and read the corresponding page from the disk (label 3). The address space of the checkpointer is not implicitly persistent, but restored manually at startup time. Therefore no further page faults happen here. After the page has been transfered to memory (label 4) the page appears within the checkpointer's address space (label 5). The checkpointer then maps the page to the pager to answer the page fault request (label 6). The pager now continues the map operation started earlier and remaps the page received from the checkpointer to the faulting thread (label 7), allowing the thread to access its code and continue running. This procedure is now going to happen for every page mapped previously to the application address space.

3.5 Kernel Checkpointing

To restart a thread in a certain CPU state, a checkpointer needs access to the registers of a thread and its kernel state. This data is stored in the kernel memory. Fortunately the design of L4 allows us to relatively easily concentrate all necessary kernel information in the thread control block. Moreover, as seen in the previous section, there is no need to include any mapping information or page table structure, since page faults are handled by external pagers and these have to store the mappings themselves. Therefore, upon system restart, the mapping database and the page-tables are reconstructed dynamically during runtime. The kernel uses the mapping instructions in steps 5-7 in Figure 3.3 to rebuild the mapping database and the page-tables of the persistent task.

To make a consistent checkpoint of each persistent application's kernel state, we need to have access to the kernel memory containing all TCBs of persistent threads. This could easily be achieved if there was a trusted user-level pager from which we could allocate pages for TCB-memory. Since all persistent applications have to trust the main-memory checkpointer, to use it also as TCB pager is just the next logical step. Thus all page frames for TCBs are allocated from the checkpointer (see Figure 3.4).

The TCB pager takes regular snapshots of the TCB pages by mapping all pages read-only (see Figure 3.5). If the kernel tries to modify a write-protected TCB (label 1) a page fault is raised and sent to the TCB pager (label 2). The checkpointer looks up whether the page has been written to disk. If the page is not

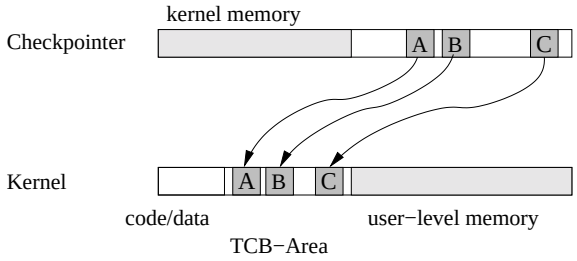


Figure 3.4: TCB paging. The user-level checkpointer owns the pages residing in the TCB area of the kernel. The mappings from the checkpointer's address space to the TCB-Area are indicated by arrows.

stored yet, the page is copied into a buffer (label 3) for later write-back. Thereafter the old page is mapped writable to the kernel (label 4) and the suspended kernel-internal operation is restarted.

Looking at the pager hierarchy in L4, it is easy to see that not all TCBs could be allocated from the TCB pager. For instance the TCBs for σ_0 and the TCB pager itself must be writable, otherwise the system could not schedule them. Therefore it makes sense to preallocate a small amount of TCB-pages for basic transient system services (σ_0, TCB pager, Disk driver, and so on), basically all threads started prior to the checkpointer. Since these TCBs are not needed for a consistent checkpoint the TCB pager does not need access to them. These TCBs could, for example, either be preallocated by the kernel and tagged with fixed thread IDs or be all threads and TCBs are tagged persistent or transient through a protocol with the checkpoint server.

To optimize response time of the system during the write back of the checkpoint, one should not blindly mark all TCB-pages copy-on-write. Since some of the mapped TCB pages might contain TCBs of transient threads and need not be mapped read-only. The amount of kernel-page faults can thus be reduced and the number of threads to schedule enlarged. In any case, the information about which TCB is persistent and about which not has to be stored somewhere by the TCB pager. Therefore this information could be used to sort out the persistent threads and specifically unmap the TCBs of these. This requires that the checkpointer has detailed knowledge of the TCB structure or mapping since it has to distinguish TCBs of persistent and transient threads.

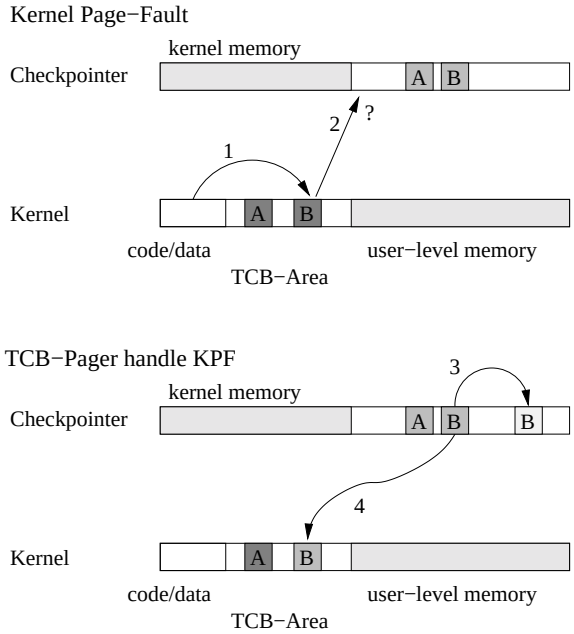


Figure 3.5: TCB copy-on-write-scheme: The TCB pages are marked copy (dark grey). The kernel raises a page fault while modifying the thread B's TCB. The page fault is sent as an IPC to the TCB pager which handle the page fault.

3.6 Kernel Recovery

3.6.1 Recovering TCBs

After a system crash the TCB area is empty except of the TCBs of the basic system services like disk driver, σ_0, the checkpointer and so on. In order to recover persistent tasks, the kernel has to ask the checkpointer, using a simple protocol, which TCBs it should try to recover. The kernel then tries to access the virtual address of the TCBs he should recover. This raises a page fault which is sent to the TCB pager as explained in the previous section.

After the kernel received a TCB from the TCB pager, it has to integrate it into its kernel-internal structures and recover the previous thread state. All kernel internal queues are restructured at this point of time. That is, all TCBs are enqueued in a present queue and, depending on their thread state, in further queues. Moreover, each queue can be reconstructed in any internal order since applications can make no assumptions about the order within the queue.

3.6.2 Recovering In-Kernel Threads

Ideally no persistent thread should be "inside the kernel" during the generation of a checkpoint. However, due to preemptable and ongoing IPC-operations, many threads could be inside the kernel. Strictly speaking, all threads besides the currently running one are inside the kernel because there can be only one thread executing at a time on a single-processor machine. Therefore, all system calls must either be restartable, or the thread has to be put into a consistent state during system recovery. Even though the L4 kernel was not designed with persistence in mind, all system calls except IPC are restartable. Thus, if a TCB is found to be checkpointed during a system call, it is restarted by setting it back to user-level onto the instruction that started the system call. The TCB would look like the TCB of a thread that had been preempted right before entering the kernel. This scheme, however, would only work if all system call parameters are stored in a well known place. Fortunately, all system calls have register-only parameters. Therefore, by using the register values saved in the TCB at kernel entry, a kernel stack frame is constructed that starts the thread right on the instruction that entered the kernel.

3.6.3 Recovering IPC

The IPC system call cannot be restarted unconditionally because it may consist of two phases: a send phase and a subsequent receive phase. Each phase on their own is restartable since the indication that the thread is still doing the single-phase IPC-operation leads to the conclusion that the message has not been sent/received yet. An interrupted long IPC operation would simply be restarted, which would just result in copying the whole message again. This is not serious, however, since system performance at recovery of the system is not a crucial issue. The assumption that a thread still doing an IPC has not sent/received the message holds, even if a long IPC has been interrupted by a page fault, and the checkpoint has been taken during the page fault handling. As such, the kernel just has to determine whether the interrupted IPC was a two-phase operation or not. If the checkpoint is taken after a completed send phase, but before the combined receive phase has been completed, restarting the system call would repeat the send phase a second time and send a message even though the checkpointed recipient has received the first message. This message duplication can break any protocol relying on the assumption that IPC does not duplicate messages. The problem of duplication could be prevented by evaluating the thread state of each thread checkpointed during an IPC. If the thread has already sent its message and is thus waiting to receive a message, the send phase of the IPC is skipped by altering the stored parameters of the system call (i.e., altering the parameters to not include a send-phase). On a restart of the system call the thread enters the receive phase directly. Uncom-

pleted send operations of two-phase IPCs are simply restarted in the same manner as single-phase IPCs.

One open issue remain here. What should be done with IPCs that bounded their period of validity with a timeout? All IPC with zero or infinite timeouts pose no problems. The former IPC would be aborted with a timeout-error before the checkpoint could be taken, and the latter is just restarted since the timeout is still valid. Non trivial timeouts are a problem however. Basically there are two models of timeouts that have to be discussed. (1) Timeouts given relative to the current point of time (e.g. in five seconds) and (2) timeouts that specify a fixed point in time (e.g. Monday 11:00 AM). The first timeout is based upon the current system time which does not tick during a power-off or system crash. This time model describes computation time. Since the computation time does not tick during a system downtime the relative timeout might not be exceeded. The second time model is based on the absolute time of the external environment (i.e. wall clock). This time ticks during the system downtime and therefore timeouts could have been due much prior to the system restart or could be still valid. The desired behavior for these timeouts is to continue all IPCs with absolute timeouts that are still valid and return a timeout error-code in all other cases. Since the current L4 Version X.0 and L4 Version 2 API do not support absolute timeouts, the current implementation return timeout errors for all IPCs waiting on a non-trivial timeout. Since applications using timeouts have to be able to deal with timeout errors in the first place this behavior is transparent to persistent applications. One might think of sending a further error code to the application explaining the timeout in detail, because this might help persistent applications that are aware of their persistence. But this behavior is not transparent to implicit persistent applications. Furthermore, if the persistent application wants to know whether the timeout occurred due to system crash, it could simply contact the checkpoint server or another system component that keeps track of this.

Chapter 4

Implementation

4.1 Implementation on L4Ka/Hazelnut

The envisaged system should be added transparently to user-level threads, thus should make no or only transparent changes in the L4 X.0 API. Furthermore the proposed system should allow the coexistence of transient and persistent tasks. All new mechanisms that implement the persistent tasks should work orthogonally to the normal kernel operations and rely as far as possible on existing mechanisms to show the flexibility of the L4 μ-kernel and the ease of adapting it to new needs.

4.2 L4Ka: An L4 compatible μ-kernel

L4Ka is an implementation of the experimental L4 API Version X.0. It has been implemented at the System Architecture Group of the Universität Karlsruhe and is currently in active development (see [1]). It is written in C/C++ and assembler. The assembler parts in L4Ka handle hardware specific setup and startup routines, generate a kernel stack layout at kernel entry compatible to the C-calling convention, implement hardware specific kernel entry/exit points for system calls and exceptions/interruptions, perform the thread switch and an experimental hand optimized IPC path. The C routines implement the portable kernel abstractions and mechanisms like scheduling, page fault handling and system calls.

4.2.1 Thread Control Blocks

Threads are implemented in L4Ka through thread control blocks (TCBs). The TCB of a thread contains all thread context and kernel data needed to administer it. The context information consists of a thread state which specifies which threads are `running`, `aborted` or `blocked`, the kernel stack pointer, a thread ID (TID),

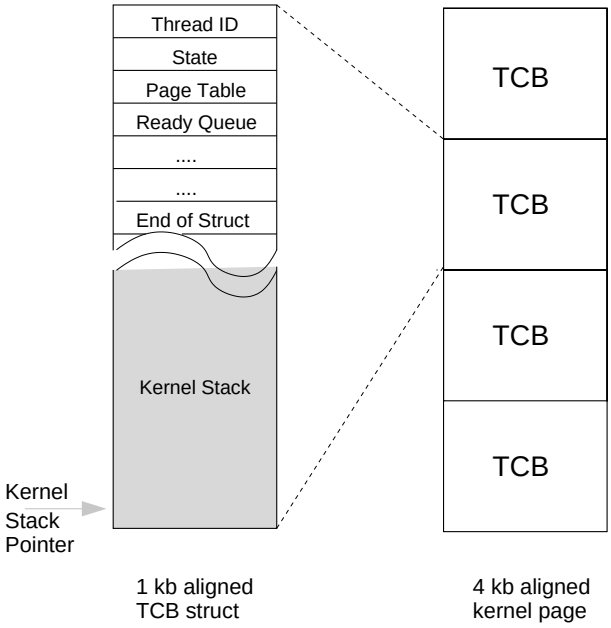


Figure 4.1: Alignment of 4 TCBs with each 1 KB size within a 4 KB memory frame.

the TID of its pager, a reference to the page table of this address space and several scheduling parameters as well as all pointers for kernel internal thread management queues. All these kernel queues are implemented by doubly linked lists. The *present queue* contains all TCBs of existing and valid threads. The *ready queue* holds all runnable threads ordered by scheduling priority. The *wakeup queue* stores all threads that blocked on an IPC timeout. These queues are all global ones. A further local queue exists per thread which stores all threads wanting to send an IPC to this thread.

The TCB structure is co-located with the kernel stack of the corresponding thread (there is a kernel stack per thread). The kernel stack and the TCB have a combined size of 1024 bytes and are aligned to 2^{10} byte borders. Thus, on the x86 there are 4 TCBs within on page frame (see Figure 4.1). The TCBs are located within a special virtual area within the kernel memory, the “TCB Area”. Having the kernel stack reside inside the TCB allows easy lookup of a TCB if the kernel stack pointer (KSP) of a thread is known. A pointer to the top of the TCB-structure could be obtained by simply clearing the lower 10 bits of the kernel stack pointer.

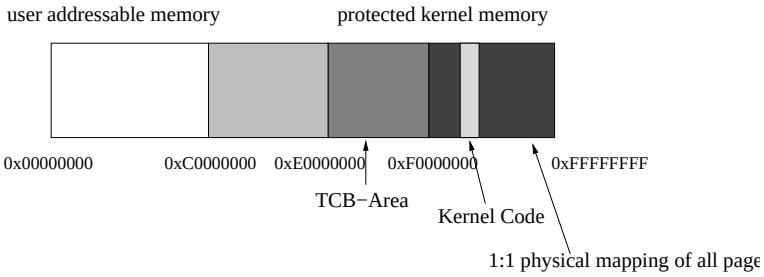


Figure 4.2: Mapping of the kernel memory. Starting from address `0xF0000000` all physical memory frames (starting from address `0x00000000`) are contiguously mapped.

4.2.2 Scheduling and Dispatching

In the TCB there are several fields containing information for scheduling purposes. Each thread has a specific priority level between 0 and 255. This implies a precedence relation. A runnable thread with a higher priority is always scheduled before any runnable thread with a lower priority. Several threads can have the same priority level. Within each priority level the threads are treated equally and scheduled in a round-robin fashion.

To reduce the number of special cases within the scheduling code, a runnable idle-thread always exists. The idle thread has priority 0 meaning that it is only scheduled if there is no other runnable thread in the system. It does not have a user-context and always runs in kernel-mode.

4.2.3 Kernel Memory Mapping

In every virtual address space L4Ka reserves the highest 1 GB (on a 32 bit systems) for internal usage. This memory area is called kernel memory. It is not user-accessible, thus protected from modifications of malicious or buggy user threads. The kernel memory is shared in all virtual address spaces. Certain kernel memory areas are dedicated for special use (see Figure 4.2). On the x86 the address area `0xE0000000-0xEFFFFFFF` is the TCB area, where all TCBs are stored. Starting from address `0xF0000000` L4Ka maps all lower 256 MB physical memory frames contiguously. Within this area the kernel code is linked.

4.3 Copy on Write Scheme

In conjuction with the TCB pager there are many possibilities to implement a snapshot mechanism for TCB pages within the L4Ka kernel. A kernel internal

implementation of the copy-on-write scheme could check each TCB on every access whether it is copy-on-write or not. This imposes a high overhead due to the added checks, in particular for preemptable system calls, since all checks have to be done again on every preemption. Furthermore the user-level pager backing the TCB area should decide to some extent which checkpoint policy to implement. It might chose to write all TCB pages eagerly or lazily to disk. The kernel should work as efficient and flexible as possible regarding all these situations and this should be reflected in the implementation.

A simpler way to implement the copy-on-write scheme would be to remap all persistent pages read-only during a snapshot. This imposes almost no extra checking overhead in the kernel, since the hardware performs all necessary checks. The kernel just has to cope with the possibly generated page faults. One might think the kernel has now to assume a page fault on every TCB write access but this assumption can be relaxed.

A TCB pager might choose not to implement a copy-on-write scheme this way, or at all. After all, from the kernels point of view TCB pages are either mapped read-only or read/writable. Therefore, in the following discussion the term “marked copy-on-write” is not used since the semantical point of view lies at the implemented user-level pager.

4.4 Resolving Kernel Page Faults

After writing to a write protected TCB a page fault is triggered. The CPU jumps to the exception handler and the kernel tries to resolve the page fault. In order to resolve the kernel page fault the TCB pager has to be notified somehow. This is handled by sending a page fault message to the TCB pager through the normal page fault handler mechanism. This has a drawback, since the IPC mechanism implemented in L4Ka heavily relies on the usage of the participated thread’s TCBs and IDs to send and receive the message. However, the kernel does not have a TCB nor a thread ID of its own. Furthermore the communicating threads are blocked until the message is delivered.

4.4.1 General Solutions

In general the faulting thread might look as a good choice to send the page fault message from since it cannot continue with its operation until the exception is handled and the accessed TCB is writable again. Strictly speaking whenever a thread interacts with a write-protected thread this is its own problem. For instance, if thread A wants to change the scheduling parameters, IP, SP or pager of thread B while B’s TCB is write-protected the access raises a page fault. It is not possible

to send a page fault message in the context of B because its TCB is not accessible. But since thread A's program relies upon the assumption that the state transition of thread B has taken place, it has to be suspended during the page fault handling anyway. Therefore thread A's TCB could be used to send the page fault message to TCB pager.

An alternative to this solution is to handle all kernel page faults with a dedicated kernel thread[1] and thereby allocate a TCB and a valid thread ID as a proxy for the kernel. All faulting TCB pages are then queued into a special queue that is cleared by this kernel thread. As long as this queue is empty the thread sleeps. Upon a page fault the handler enqueues the TCB page in which the page fault occurred into the queue and wakes up the proxy thread if it is sleeping. This looks up a faulting TCB page from the queue, sends a page fault message to the TCB pager and waits to receive a new mapping for the TCB page. After a new message is established it removes the TCB page from the queue. Since there is more than one TCB in a TCB page the page fault handler should ensure that a page only gets enqueued once. However, all threads originally faulting on the TCB page have to be blocked during the page fault handling and have to be resumed afterwards. It is therefore necessary to implement some kind of event system so that each blocked thread is resumed after the TCB page it accessed gets available.

4.4.2 Special Cases

The first proposed solution sounds quite reasonable regarding most user level threads. But there also exist several kernel threads which also access TCBs and therefore could raise a page fault on these. One of these is the idle thread. The idle thread runs if there are no further runnable threads in the system. This is a critical thread since the kernel relies upon that it is always runnable and never blocked. But if the idle thread raises a page fault and one applies the solution proposed above, the idle thread would block. Furthermore the TCB pager cannot send a map messages to the idle thread since the thread ID of the idle thread is invalid. That is, it is not possible to block the idle thread and send a page fault message. Therefore it is not possible to use the context of the faulting thread unconditionally. A further example is the TCB pager itself because it is not able to transparently resolve its own page faults and send a page fault message to itself. Since the TCB pager might depend on disk-access, the same argument is applicable to device drivers used to access the disk.

If the idle thread or another critical thread accesses a write protected TCB the page fault message has to be send differently. Since the number of these excep-

[1] A kernel thread is a normal thread without a user-level context or user address-space. It is able to run only in kernel mode and never switches to user mode.

tional cases is relatively small they could be handled within the page fault handler. To do so, the handler has to check the ID of the faulting thread to determine whether the thread can be blocked or not. This check has to be performed on every page fault, but since handling of a page fault is not a time critical operation the additional overhead is neglected. To resolve the special cases a dedicated kernel thread K could be used to resolve all TCB page faults of critical threads. K sends the page fault messages to the TCB pager and resumes the thread blocking on the page fault of the corresponding TCB page.

Except for the special case with system critical threads noted above, there seems to be nothing that objects to the idea of using the context of the faulting threads. However, at a closer look there are several other cases which do not permit the blocking of the faulting threads (e.g., transient realtime threads and so forth). Strictly speaking, any thread with no active relationship to the write-protected TCB should not be blocked. Thus only threads actively accessing the read-only TCB should be blocked. Therefore realtime threads could fulfill their time constrains by avoiding all contacts to persistent threads.

4.5 TCB Allocation

The current L4Ka deals with non-existent TCBs very simply. A special write-protected zero-page is mapped at every empty TCB location. There are a number of reasons for this. It removes the need to keep track of allocated TCBs in a table since every write access to an empty TCB location results in a page fault and a new TCB could be mapped into this. Given a thread ID of a non existing thread no special case handling has to be introduced since a comparison of the thread ID stored within the TCB and the given ID reveals whether the thread exists and is active or not. The page fault handling code catches every write fault within the TCB area and recognizes this fault as a page allocation for TCBs. It fetches a page from a central page pool that is allocated from σ_0 for kernel internal structures at startup and maps it to the empty TCB location.

For a kernel supporting persistent threads, to map a TCB page the user-level TCB pager must be involved. This is achieved by having a page fault IPC sent to the TCB pager. Following the discussion of chapter 4.4 the context of the thread that tried to start the new thread is used to send the IPC. Through the usage of an existing TCB the standard IPC-mechanism and mapping-mechanism can be used. There is no problem in suspending the current thread, because it has to wait until the TCB is allocated and its system call that resulted in the TCB page fault can finish.

4.6 Changes to the Mapping Implementation

The mapping mechanism maps pages from a given address v_a within an address space σ_a to a second address v_b in an address space σ_b. It simply copies the page-table entry e_a of the virtual address v_a into the page-table entry e_b of the virtual address v_b of σ_b. Depending on the mapping parameter, the access-rights of the page-table entry e_b are set to either read-only or read-write. If the map operation is a grant, the page-table entry e_a is removed from the address space σ_a. It is not possible to map within the same address space ($\sigma_a = \sigma_b$) since this would allow easy DOS-attacks[2], but it is possible to grant a page within the same address space. The mapping is also stored within a kernel-internal mapping database to support later unmapping of the pages.

During the normal mapping of a page several access restrictions apply. The receiver of a mapping can specify the virtual address to accept the mapped page. Furthermore normal user-level threads are obviously not allowed to map pages into the kernel memory. This constraint has to be relaxed for the checkpointer, however, since we have to trust the checkpointer to map into the TCB area. Currently this check is enforced by allowing a thread with a specific thread ID (set at compile time) to map and unmap pages within the TCB area. To set this parameter at compile time is a bit inflexible, but it could also be passed via the kernel info page. Anyhow, the kernel needs to know the ID of the TCB pager anyway, or it would be unable to generate TCB pagefaults.

4.7 Kernel Internal Structures

As mentioned earlier all queue pointers are located within the TCBs. Thus, no further memory constructs have to be allocated for thread management. At first this does not look like a problem, but it complicates all handling of queue structures since all memory frames of persistent TCBs are mapped read-only during the write-back of a checkpoint, disallowing even the kernel to perform write access to the TCBs. If a writable TCB of the current thread (e.g., a transient thread) has to be dequeued from a queue, any of its neighbor TCBs could be a persistent thread with a write-protected TCB. Figure 4.3 illustrates the problem. To dequeue the TCB found in the middle of the picture it is necessary to modify the `next` pointer within the previous TCB and the `previous` pointer within the next TCB. Thus the dequeue-operation is going to raise a page fault whenever any of the adjacent TCBs are within a write-protected page frame. As such any queue operation could

[2]The denial of service attack could consume kernel internal memory by mapping a page recursively within its own space, eating up memory for both page tables and mapping database structures.

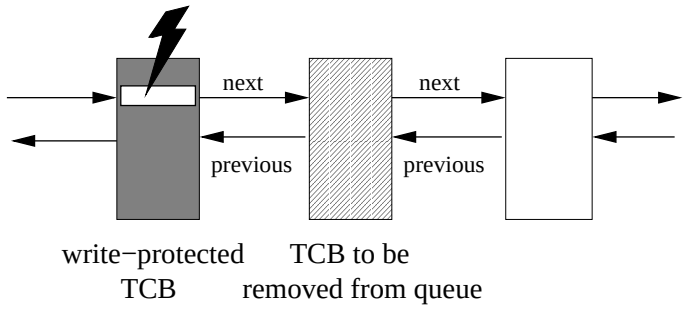


Figure 4.3: The TCB to remove from the queue is in the middle. The access to next pointer of the previous TCB raises a page fault.

result in a page fault. It is not an option to postpone the queue modification until a page fault message is sent to the TCB pager and the page fault is resolved because the kernel (e.g. the IPC code) depends on queue operations to be performed. However, the queue-pointers need not be persistent since all kernel queues are rebuilt at startup. Hence we can implement a special case handling to allow write access to a write-protected TCB for the duration of the queue operation.

A possible solution to handle this special case would be to extend the access rights for the page and to diminish them after the queue operation. This could be done right before and after every modification within the queues. Since queuing-operations are very frequent operations within the L4Ka μ-kernel, performance would seriously suffer. Therefore it is not an option to modify the page-table entries within every queue-modification. It may be possible to do a special case handling within the page fault path to resolve the exception when needed. This could happen in different ways. The value to store could be extracted from the exception frame, the access right extended, the store operation executed on behalf of the faulting and the access right diminished again.

For L4Ka there is a simple way to solve this problem. As described in chapter 4.2.3, there is a writable mapping of the first 256 MB physical page frames in the kernel starting from address `0xF0000000`. If all queuing operations are done using this second mapping no page fault occurs at all. By storing a pointer to the second mapping within each TCB, the complete overhead to access the queue-field can be reduced to an indirect addressing. The problem of this solution is the limited scalability of main memory since the size of the mapping area in the kernel is currently 256 MB. This problem can be solved in the TCB pager by preallocating enough physical memory below 256 MB border for TCBs.

4.8 System Call Mechanism

L4Ka uses 2 different hardware mechanisms on the x86 to enter the kernel. Widely used there is the "software interrupt" mechanism. Software interrupts are synchronous exceptions generated by the user application. The x86 hardware in this case pushes an exception frame[3] onto the kernel stack and jumps to a specific entry point yielding control to the kernel. Depending on the exception generated, the kernel calls the appropriate system call function. As mentioned in Chapter 3.6.2, it is necessary to store all parameters passed via registers. This is done by pushing all registers onto the kernel stack before the kernel tries to reorganize the stack[4] to call the appropriate C-function for this system call. After successful execution of the system call the C-function return to a special return function which is an assembler macro loading the returned parameters into the registers and returning to the user application.

Since the usage of software interrupts is expensive on the x86, a special mechanism to trap into kernel was introduced with the Pentium Pro. The new mechanism uses two instructions, `sysenter` and `sysexit`, to enter and exit the kernel. Using the sysenter/sysexit mechanism fewer cycles are needed to enter the kernel, but the kernel code needed to invoke the system call and return to user-level is more complex. The mechanism is used in the fastest implementation of the IPC-path in L4Ka. It is possible to use the mechanism in a persistent L4Ka too, but was not implemented due to the complexity of the mechanism itself.

4.9 Thread Switch

In order to make a thread switch, the context of the thread currently running has to be stored on the kernel stack. Since in L4KA the kernel stack is within the TCB this stack could also be write-protected. As such, the kernel stack of the thread we would like to switch to could be write-protected, too. This poses a problem because the x86-hardware relies on the assumption that the kernel stack is always writable. The stack is needed by the hardware to store the reason for the fault and the last context before the fault occurred. If this write fails, another fault is raised. The hardware tries again to store the reason for the fault and the last context as well which raises just the next fault. To avoid endless recursion the processor reboots on three consecutive faults not handled properly (triple fault).

[3] Within the exception frame the user-level stack pointer and return address are stored, along with code-segment and stack-segment selector.

[4] This is has to be done in order to achieve a stack compatible to the C-calling convention used within the kernel.

Thus it is crucial to always have a writable kernel stack on the x86. Therefore the current context is stored on the kernel stack and, before the real switch is done, the TCB of the next thread is accessed before changing the stack pointers like in normal thread switching. If the next TCB is write protected the page fault is handled specially within the page fault handler. To distinguish the fault from a normal kernel page fault, different methods could be used (e.g. setting a global variable). The implemented solution uses a writable temporary stack when accessing the TCB of the next thread. If no page fault occurs, the write access is successful and the kernel just completes the thread switch by loading the current kernel stack pointer from the new TCB. If a page fault occurs the exception frames is pushed on the temporary stack and the page fault handler code is executed. The special kernel stack pointer can then be detected, indicating that the new TCB is write-protected. Since the context of the thread the kernel just left is secured on its kernel stack there is no further clean-up necessary. The cost for this are four additional memory accesses and one additional load of the stack pointer register. It is possible to reduce this overhead to only two memory accesses, but this would make the special case handler much more complicated.

Since a thread doing a system call should experience it as an atomic operation the thread-switch operation should block the last thread and use its context to send the page fault message if it tried to make a thread switch to a write-protected destination thread using a system call as `l4_thread_switch` or as part of an IPC operation. In the case of a thread switch due to an end of timeslice the context of the current thread should not be blocked. How to send a page fault message in this case was discussed in Chapter 4.4.

4.10 Changes to the IPC implementation

The IPC mechanism accesses TCBs on several different occasions depending on the type of the message being sent or received. This may raise page fault, because the TCB of a thread could be unmapped while the thread is waiting for its partner. There are two cases that have to be carefully analyzed. (1) Communicating with a persistent thread could block the sender or receiver of a message, and (2) the interruptible long IPC messages might be interrupted by a page fault originating from user-memory of either the sender or receiver. The thread state changes multiple times within this process. If the checkpoint is taken at an inappropriate time, the recovery mechanism cannot distinguish the thread state of the long IPC and the page fault IPC.

4.10.1 Communicating with persistent threads

Message transfer between threads where at least one thread is persistent is a critical operation since the TCB of the persistent thread may be read-only and thus delay the IPC operation until the TCB is writable again. The scheme described above to send the page fault message within the context of the faulting thread is suitable for almost all cases of IPC. On a closer look, all IPC modes except the open receive call specify a distinctive communication partner to wait for. In the open receive, a thread receives the message of any thread that waits to deliver a message. The receiving thread does not known from whom it is going to receive a message. It has agreed upon receiving a message from anyone. For instance, if a very busy server application tries to receive a new request from a new requestor and this sender has been waiting for the server some time, it is possible, that the sender TCB is read-only. In this situation it is questionable to block the receiving thread since it is not responsible for its communication partner. This happens quite frequently if a persistent application sends a page fault message to the TCB pager which does not receive the message yet but instead currently maps all pages read-only to take a checkpoint. If now the TCB pager tries to handle the next page fault sent by any thread, a page fault happens during the receive phase of the IPC. This page fault can not be handled by blocking the faulting thread and sending a message to the TCB pager since it cannot send a message to itself. Therefore this case must be handled explicitly. These situations do not occur very frequently and no checks for this special cases should therefore be included within the IPC-code since it would degrade IPC performance. Rather, the special case should be detected and resolved within the page fault handler. The handler can distinguish threads faulting in an open receive and a plain receive by looking at the receive parameter. Since this has been stored upon the kernel stack at kernel entry it can be extracted from there. This is necessary because the receive descriptor is not stored within the TCB.

After detection of the special case it is necessary to remove the write-protected TCB from the send-queue of the faulting thread and restart the receive phase of the IPC, much like recovering a thread which is in an IPC after system crash. The page fault itself can be resolved by the dedicated kernel thread discussed in Section 4.4. After handling the page fault the send phase of the sending thread has to be restarted. It is not sufficient just to enqueue it again in the send queue of the receiver because this would imply that the receiver is not ready to receive when the thread may indeed be blocked, waiting to receive. Restarting the whole IPC is much simpler than programming and maintaining a special trampoline code to enter the normal IPC-code again at several different places. Since this is a very infrequent event it is not a critical point for the overall system performance.

4.10.2 Long-IPC Messages

In L4Ka the message transfer of a long copy message is interruptible. The message transfer can be interrupted by a page fault in one of the address spaces or the end of the current time slice. The transfer is then suspended and another thread is scheduled. The communicating threads are not runnable until the message has been transfered. Therefore the kernel does not schedule the receiver of the message, but in order to progress it eventually schedules the sender which continues copying the message[5]. To achieve this behavior L4Ka has a special thread state model in which threads currently transferring the message change into a special state indicating that they are doing a message transfer. Since a checkpoint can be taken during the message transfer, the kernel has to handle this properly. Fortunately this is not very difficult as the whole message is simply transfered again.

The message transfer may also be interrupted by a page fault within one of the address spaces. In this situation the thread belonging to the faulting address space sends the page fault message to its pager and waits for a page to be mapped into its address space. Note that the thread which was previously not runnable (due to a message transfer) is now transferring a different message, and is put into a different state. If a checkpoint is taken during the message transfer of the page fault message then the thread state stored within the TCB can be completely unrelated to the actual IPC initiated by the application. For instance, if a page fault occurs during the message transfer in the senders address space, the kernel sends the page fault message for the sender and thereafter receives a message from its pager (see Figure 4.4). Thereby the senders thread state is changed to receiving. Now consider that a checkpoint is taken after sending the page fault message, but before the page fault is resolved by the pager and a map message is sent back to the faulting thread. Then the recovery algorithm would assume that the send phase of the first IPC has already been executed, and would skip the uncompleted message sent earlier. Therefore, the thread state of the faulting thread has to be stored within the TCB prior to sending the page fault message and erased afterwards. Following this scheme the recovery mechanism can now correctly recover the last thread-state by checking the saved thread state.

4.11 Changes to the page fault implementation

The page fault handler is invoked whenever a page fault occurs. The modifications to the page fault handling code and the implementation of the special case handling described in the previous chapters are straight forward. The only modi-

[5]This message transfer model is called “active sender” since the sender is transferring the message while the receiver is waiting.

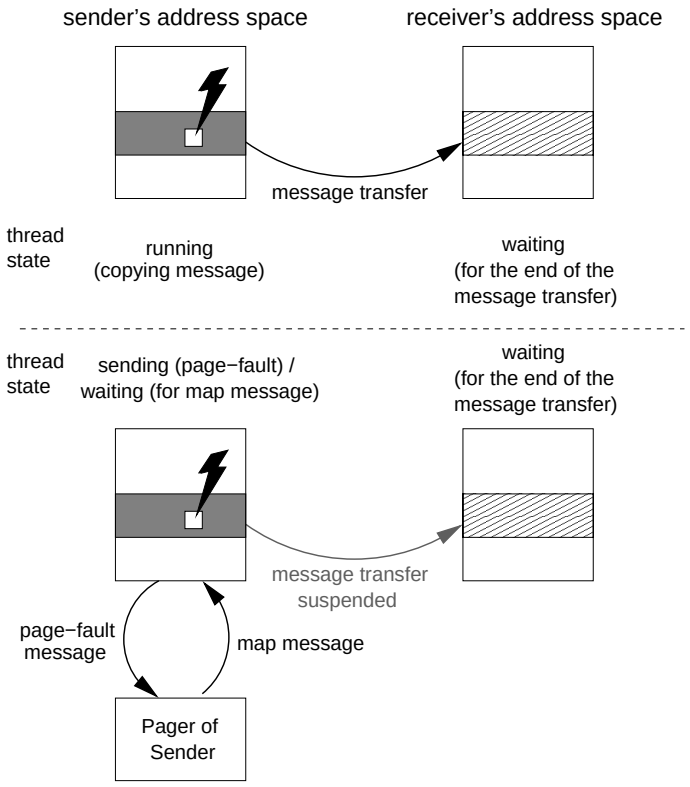


Figure 4.4: In the upper diagram the message transfer is interrupted by a page fault in the address space of the sender. It is resolved in the lower picture by sending a page fault message to its pager and receiving a mapping.

fication of the existing code was the sending of a page fault message to the TCB pager in case of page fault exception within the TCB area. Since this automatically uses current context no further changes were necessary.

4.12 Recovering TCBs in L4Ka

In Section 4.8 the necessity was discussed for storing all relevant information (i.e., user registers) within the TCB at kernel entry to recover a thread. Therefore the recovery mechanism proposed in Section 3.6.3 can be implemented almost unchanged. The only addition to the recovery mechanism is the saved thread state discussed in chapter 4.10.2 and a dedicated exit-point that recovered threads use to return to user-level. The main advantage of storeing all registers on kernel entry is that all threads have the same kernel stack layout. The exception frame on the kernel stack just has to be modified slightly during system recovery (i.e.,

modifying the instruction pointer to point to the instruction that started the system call before). Thereafter the TCB is enqueued into all necessary kernel queues.

To recover all persistent threads at startup, a user-visible dedicated kernel thread active only at startup is introduced. Its primary purpose is to send page fault messages to the TCB pager and thereby initiate the recovery process. Furthermore it recovers all TCBs located within the page frame after the reply of the TCB pager with the appropriate mapping and thereby restarts the blocked threads. After the startup this thread could also be used to send the page fault messages as discussed in Section 4.4.

4.13 Miscellaneous

4.13.1 Floating-Point Registers

In L4Ka there are several kernel-internal operations that are performed lazily, e.g. storing and restoring of the floating point unit (FPU) state. Since the size of FPU-state is 128 to 512 bytes on the x86, IPC performance would suffer if it is stored on every thread switch. Fortunately the FPU-state can be stored on demand. The x86 hardware supports this by allowing locking the FPU. If a thread tries to access the locked FPU an exception is raised. The kernel searches for the thread that used the FPU last and stores the FPU-state in its TCB. Thereafter it restores the FPU-state from the accessing thread.

Having lazily stored FPU state the checkpoint server must ensure the storage of the FPU-context right before making a snapshot. Otherwise the last state of the FPU-registers would not be saved, rendering the checkpoint inconsistent if an application used floating-point operations. It is not sufficient that the checkpointer simply accesses the FPU and let the kernel do the rest since the checkpointer could be preempted right before the following unmap of the TCB pages. Therefore the `unmap` system call should be modified to automatically store the FPU-context before unmapping all TCB pages.

4.13.2 σ_1 and L4Ka

In the original design of L4 there is a thread ID reserved for σ_1. This should be used as the TCB pager that backs all memory needed for TCBs within the kernel. There are also fields reserved within the kernel-info-page to set up σ_1. In the current implementation this is not used, since L4Ka does not implement support for σ_1.

4.13.3 Kernel Memory Management

The L4Ka kernel preallocates a page pool at startup time for kernel internal use. The pages are used in the non-persistent version of the kernel for TCBs, page tables and mapping nodes in the mapping database. Since in the persistent L4Ka kernel TCBs are allocated from the TCB pager, the page pool is only used for the transient page tables and mapping database. The remaining kernel memory management is independent of the TCB paging and can therefore be used unmodified.

Chapter 5

Conclusions

5.1 Achievements

We have seen that the basic concepts and abstractions can easily be modified to the need of persistent systems on top of the kernel. The mechanisms for user-level paging of TCB pages and checkpointing has been implemented on top of the L4Ka and provide the environment proposed in Section 1.4. So far, a user-level TCB pager and checkpointer is implemented that use an IDE-driver from the Sawmill system to access the disk. An implementation of a recoverable disk driver as described in [13] and more thorough testing is needed, however. The implemented checkpoint server and TCB pager is a bit immature and not optimized, but provide basic functionality. Therefore no performance measurements have been done yet. Since this is crucial for a full evaluation of the design decisions the system could only be partly evaluated.

5.2 Discussion

One open issue with the current implementation is how well it work with other architectures than the x86. Since the L4Ka is divided into architecture dependent and independent parts and most of the changes discussed in Chapter 4 are implemented in an architecture dependent part due to their connection to the kernel stack layout, it may be possible to tackle general issues, discussed in Chapter 3, in the architecture independent part. The ARM architecture (the only other architecture supported by L4Ka) has a very different exception model (which banks the kernel registers during a system call), therefore some problems associated with the kernel stack model of the x86 disappear, but others may arise.

Symmetric-multiprocessor (SMP) systems have been around for many years and are used in various server and scientific scenarios. If one likes to use system-

wide persistence in such a system one has to ensure proper synchronization between the processors and the checkpointer. The problems that arise through multiple processors are manyfold and their implications are beyond the scope of this thesis. At a first glance, however, nothing seems to prevent a orthogonally persistent SMP system. However, preemptable and reentrant system calls can provide serious further synchronization problems.

Future versions of L4 will include a mechanism to control communication known as IPC redirection. In this model every IPC a thread is sending could be rerouted to a different thread. As such it is questionable whether every is able to send a message to the TCB-pager or/and the checkpointer. In these upcoming μ-kernel it should be considered to save a specific thread ID for the kernel just for this purpose.

The combination of hard realtime or soft realtime constrains with persistence is still an open issue. There are several introduced critical sections and bottlenecks that might end up in priority inversion (e.g. the resolving of a TCB page fault). Since reentrant and preemptable system calls present problems this issue may never be tackled.

5.3 Final Conclusions

We have seen that the support for user-level transparent orthogonal checkpointing on top the L4-μ-kernel can easily be implemented upon the existing code base of L4Ka. The existing mechanisms are easily adapted in implementing the same abstractions within a persistent context. The key feature of L4 in context of persistence is the concept of recursive address spaces that enables the implementation of memory management by user-level servers.

5.4 Future Work

The next step to be taken after this thesis is to do a preliminary performance measurement with micro-benchmarks to evaluate the performance impact of the implemented mechanisms on normal μ-kernel system calls invoked with and without persistent applications.

Different implementations of the checkpoint write-back mechanism in the user-level checkpointer and TCB pager should also be studied in detail. The mechanism should be optimized regarding disk-access and latency on remapping a copy-on-write page. The partition-layout should be revisited with, i.e., a log-structured block storage in mind.

More future work is to implement the system server and system components

which allow the access to legacy file system services (i.e., a recoverable system driver).

Still an open issue is the feasibility to integrate persistence into L^{4}Linux [1]. How far this integration could be pushed without touching the Linux core itself should be conducted in a short survey and the major problems identified. In this context more performance measurements should be conducted to analyze the impact of persistent applications on the performance of the whole system, upon single applications and transparent applications running concurrently to persistent ones.

[1] L^{4}Linux is a single server Linux compatible operating system on top of L4.

Bibliography

[1] L4-Ka Team, *The L4-Karlsruhe*, http://l4ka.org

[2] Uwe Dannowski, Espen Skoglund, *L4/Ka Design Manual*, available along with the L4/Ka-sources from http://l4ka.sourceforge.net, (manual still under construction)

[3] Jochen Liedtke, *On μ-kernel Construction*, Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95), Copper Mountain Resort, CO, December 3-6 1995.

[4] Jochen Liedtke, *Improving IPC by Kernel Design*, Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP'93), Asheville, NC, December 1993.

[5] Jochen Liedtke, *μ-kernels Must And Can Be Small*, Proceedings of the 5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS), Seattle, WA, October 1996.

[6] Jochen Liedtke, *Lava Nucleus (LN) Reference Manual, 486, Pentium, Pentium Pro, Version 2.2*, IBM T. J. Watson Research Center, March 1998.

[7] Jochen Liedtke, *L4 API/ABI Version X.0*, http://l4ka.org/documentations/files/l4-86-x0.ps

[8] T.Jaeger, K. Elphinstone, J. Liedtke, V. Panteleenko and Y. Park, *Flexible access control using IPC redirection*, Proceedings of the 7th Workshop on Hot Topics on Operating Systems (HotOS'99), 1999.

[9] Jonathan S. Shapiro, Jonathan M. Smith, David J. Farber, *EROS: a fast capability system*, Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP '99), Kiawaha Island, USA, December 1999.

[10] Charles R. Landau, *The checkpoint mechanism in KeyKOS*, Proceedings of the 2nd International Workshop on Persistent Object Systems (POS2), Paris, France, September 1992.

[11] Jochen Liedtke, *A persistent system in real use: experiences of the first 13 years*, Proceedings 3rd International Workshop on Object-Orientations in Operating Systems (IWOOOS '93), Asheville, NC, December 1993.

[12] Michael Hohmuth, *Linux-Emulation auf einem Mikrokern*, Diploma Thesis, TU Dresden, August 1996.

[13] Espen Skoglund, Christian Ceelen and Jochen Liedtke, *Transparent Orthogonal Checkpointing Through User-Level Pagers*, In 9th International Workshop on Persistent Object Systems (POS9), Lillehammer, Norway, September 2000.

[14] Patrick Tullmann, Jay Lepreau, Bryan Ford and Mike Hibler, *User-level checkpointing through exportable kernel state*, Proceedings of the 5th International Workshop on Object-Orientations in Operating System (IWOOOS'96), Seattle, WA, October 1996.

[15] James S. Plank, Micah Beck, Gerry Kingsley and Kai Li, *Libckpt: transparent checkpointing under UNIX*, Proceedings of the USENIX 1995 Technical Conference, New Orleans, LA, January 1995.

[16] Alan Dearle, Rex di Bona, James Farrow, Frans Hensken, Anders Lindström, John Rosenberg and Francis Vaughan, *Grasshopper: an orthogonally persistent operating system*, Computing Systems, 7(3):289-312, Summer 1994.

[17] Alan Dearle and David Hulse, *Operating system support for persistent systems: past, present, future*, Software - Practice and Experience, Special Issue on Persistent Object Systems, 30(4):295-324, 2000.

[18] Dawson Engler, David Yu Chen, Andy Chou, *Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code*, Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP '01), Chateau Lake Louise, Canada, October 2001.

[19] http://www.tunes.org/Glossary/paper/index.html#Persistence

[20] http://www.mklinux.org/

[21] Kevin Elphinstone, Stephen Russell, Gernot Heiser, and Jochen Liedtke, *Supporting Persistent Object Systems in a Single Address Space*, In 7th International Workshop on Persistent Object Systems (POS7), Cape May, NJ, USA, May 1996.

[22] Verifiable L4-Fiasco, http://os.inf.tu-dresden.de/vfiasco/

[23] Michael Hohmuth, Hendrik Tews, *Work-in-Progress Report: VFiasco - Towards a Provably Correct Microkernel* USENIX Annual Technical Conference, 2001

[24] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J.E. Tidswell, L. Deller, and L. Reuther, *The SawMill Multiserver Approach*, In 9th SIGOPS European Workshop, Kolding, Denmark, September 2000.