A Comprehensive Specification and Verification of the L4 Microkernel API

Leping Zhang[1], Yongwang Zhao[2,3(✉)], and Jianxin Li[1]

[1] School of Computer Science and Engineering, Beihang University, Beijing, China
[2] School of Cyber Science and Technology, College of Computer Science and Technology, Zhejiang University, Hangzhou, China
zhaoyw@zju.edu.cn
[3] State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China

Abstract. The L4 API (Application Programming Interface) is a core component of the operating system, which serves as the interface between user-level processes and the microkernel, facilitating communication and interaction. It is crucial to ensure the correctness and reliability of the API. This paper proposes a comprehensive formal specification and verification for the L4 microkernel API. The specification is reusable for all implementations on architectures supported by the microkernel. To further improve reusability (e.g., for the L4 family), a parameterized model is abstracted, which mainly includes variables related to L4 components and safety properties built on them. The desired properties are composed of 350 functional correctness and 39 safety properties, where the safety properties cover existing invariants of the microkernel. Several rewriting rules and reasoning steps are proposed for verification to improve proof efficiency. The proofs of the specification w.r.t these properties are accomplished in the theorem prover Isabelle/HOL, and the results show that all definitions, lemmas, and proofs pass the prover's check. During modeling and verification, 10 bugs in the source code are found, all of which are fixed in this paper.

Keywords: Formal Specification · Theorem Proving · L4 API · Correctness · Safety · Refinement · Isabelle/HOL.

1 Introduction

The L4 API is a fundamental part of an operating system (OS) that allows applications and user-level programs to interact with the kernel, provided by the L4 microkernel family of operating systems. Ensuring the correctness and reliability of the L4 API is paramount, as it forms the foundation for system stability, security, and the seamless operation of L4 microkernel-based operating systems in diverse and demanding environments. Although Klein et al. [3] formally verified the microkernel seL4, a specific implementation of the L4 microkernel, and achieved great results, L4 microkernels are diverse, such as Pistachio, Fiasco, OKL4, and each microkernel has its own API. Based on a standard L4 reference manual [14], this paper starts out from a functional model and is committed to

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 217–234, 2024.
https://doi.org/10.1007/978-3-031-57249-4_11

improving the correctness and reliability of the microkernel by formal verification.

So far, there has been substantial research on the formal verification of the microkernel [15,4,5,6]. Nevertheless, these studies predominantly mainly have the following problems. 1)The existing formal specifications [15,4,5,6] are incomplete. For example, Klein et al. built an abstract model for address spaces of the microkernel in Isabelle/HOL, then they formalized parts of the API using the B method, where the latter mainly supplemented threads and Inter-Process Communication (IPC) mechanism but did not include the key scheduling. 2)There are quite a few properties to formalize and verify for the kernel. Klein et al. verified three invariants about address spaces in [15,4], and no property on threads and IPC in [5,6]. 3)Klein's model [15,4] for address spaces is too one-dimensional, which is reflected in the lack of flexible page processing and access permission modeling. Flexibility is one of the design principles of L4, and permissions are an indispensable component in a practical kernel. It is necessary to model flexible pages and access permissions. 4)Following our modeling and verification experience, there are several errors in both specifications [15,5].

This paper aims to model and verify the comprehensive L4 API and make up for all the above shortcomings. We prioritize modeling based on the L4 reference manual to build a formal specification that involves all modules of the L4 microkernel. To verify enough properties (e.g., covering all invariants in Klein's model), we choose a concrete implementation built on the manual and involve more fine-grained modeling referring to the source code. By verification, we try to exclude these errors including the incomplete or unreasonable informal description in the manual, the inconsistency between the source code and the manual, the bugs in the source code, and so on.

During specification and verification for the API, we encountered three challenges. Firstly, the L4 API is quite complex, especially the address space with both tree structures and flexible pages. For this, the sel4 microkernel omits these two features of address spaces for simplicity. Moreover, the coupling of kernel modules is strong, and they form a hierarchical relationship. For example, all functions of address spaces are called by threads or IPC. Secondly, since the implementation of the API is slightly different under different CPU architectures, on the basis of fine-grained modeling, it is a challenge to build a specification that can be reused for all implementations on architectures supported by the microkernel. Thirdly, when facing non-trivial models and considerable properties, verification efficiency is often an important goal. It is necessary to present some reusable proof methods or frameworks.

On the premise of solving the above challenges, we conduct a formal verification for the L4 API in Isabelle/HOL [11], where the concrete implementation is based on the release version of the L4Ka::Pistachio [13]. To our knowledge, this work is the first effort at building the comprehensive specification and verification for the L4 microkernel. The contributions are described in detail as follows.

NO.	*Parameters*	*Functional Description*
1	*SpaceSpecifier ≠ nilthread, dest not existing*	*Creation. The space specifier specifies in which address space the thread will reside. Since address space do not have own IDs, a thread ID is used as SpaceSpecifier. Its meaning is: the new thread should execute in the same address space as the thread SpaceSpecifier.* *The first thread in a new address space is created with SpaceSpecifier = dest. This operation implicitly creates a new empty address space. Note that the new address space is created with an empty UTCB and KIP area. The space creation must therefore be completed by a SPACECONTROL operation before the thread(s) can execute.*
2	*SpaceSpecifier ≠ nilthread, dest exists*	*Modification Only. The addressed thread dest is neither deleted nor created. Modifications can change the version bits of the thread ID, the associated scheduler, the pager, or the associated address space, i.e., migrate the thread to a new address space.*
3	*SpaceSpecifier = nilthread, dest exists*	*Deletion. The addressed thread dest is deleted. Deleting the last thread of an address space implicitly also deletes the address space.*
.....		

Fig. 1. Informal Functional Description of $ThreadControl$ in the L4 Reference Manual

1) We propose a comprehensive formal specification of the L4 API. The specification can be reused for all implementations on architectures supported by the microkernel.
2) We formalize 350 functional correctness and 39 safety properties. The safety properties on address spaces cover all invariants in Klein's model, simultaneously, a series of new key invariants are proposed in this model.
3) We use Isabelle/HOL to prove that the specification satisfies these properties, which improves the correctness and reliability of the L4 API. In addition, in this stage, we propose several rewrite rules and reasoning steps to improve proof efficiency.
4) We found that there are in total of 10 bugs in the manual and the source code of the microkernel. All of them are fixed in this paper.

2 Preliminaries

2.1 L4 Overview

The L4 microkernel mainly includes four core modules, i.e., thread, address space, IPC, and scheduling, where the thread is the execution unit of the L4 microkernel, the address space provides isolated execution environments, the IPC mechanism enables threads in different address spaces to communicate, and the scheduling mechanism is used to switch contexts.

Thread. L4 used the thread identifier *threadid* to identify a thread, almost all of whose information is recorded in TCB (Thread Control Block). The status of each thread is defined by the status field, and the transition relationship is shown in Fig.2(a). A special phenomenon is that L4 abstracts each interrupt as a thread. When an interrupt is triggered, the kernel notifies the corresponding thread to deliver the interrupt to its handler thread through IPC, ensuring that interrupt processing in a microkernel is completed in user mode.

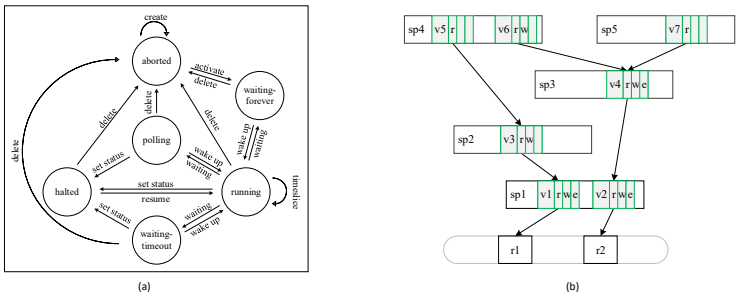

Fig. 2. The graph (a) shows the status transitions of the thread, and the graph (b) shows the tree address space structure.

Address Space. Address space is a logical concept that represents the range of virtual addresses that a thread can access. Each address space contains a page table, which realizes the conversion of the virtual address to a physical address and is a way to achieve memory isolation. The mapping mechanism is one of the features of the L4 microkernel. In addition to maintaining page table data, this mechanism also maintains the relationship between pages in different address spaces, which causes the address space to form a hierarchical structure (tree-shape) as shown in Fig.2(b), and adds difficulty to our proof.

IPC. The IPC mechanism is synchronous which means that a sender blocks until the receiver processes the message and responds. The communication of the sender and receiver can be specified as one or two phases through a special thread identifier called *nilthread*. If there is one phase, the sub-function is either sending or receiving, otherwise, sending then receiving. For the sub-function of receiving messages, the receiver can not only receive from a specific thread but from any thread that is specified by another thread identifier *anythread*.

Scheduling. L4 introduces a unique 256-level, fixed-priority scheduling system, combining time-sharing and round-robin (RR) principles. This scheduler prioritizes threads and executes them in order of their priority until certain conditions are met: a thread blocks in the kernel, gets preempted by a higher-priority thread, or consumes its allocated time quantum.

API. The L4 microkernel provides the API implemented in 10 system calls, shown in Table 1. The concrete sub-function is chosen by specifying the parameters of the system call. For example, Fig. 1 is a fragment of the L4 kernel reference manual, which informally describes three sub-functions of the system called $ThreadControl$. By controlling whether the values of $SpaceSpecifier$ are equal to *nilthread* and whether the target thread *dest* exists, the manual specifies $ThreadControl$ to complete the creation, modification, or deletion operation.

Table 1. L4 API Description

No.	Name	Functional Description
1	ThreadControl	Create, activate, modify, or delete a thread by privileged threads only.
2	ExchangeRegister	Exchange or read thread information such as IP, SP, etc.
3	Schedule	Set scheduling attributes of a thread.
4	SpaceControl	Initialize an address space by privileged threads only.
5	Unmap	Revoke pages that have been mapped or granted.
6	MemoryControl	Set the attributes of pages.
7	IPC	Transfer data from one thread to another.
8	ThreadSwitch	Switch to the specified thread or perform a normal thread scheduling.
9	SystemClock	Return the current clock.
10	ProcessorControl	Set the attributes of CPU by privileged threads only.

2.2 Related Work

Klein et al. modeled the virtual memory subsystem of an L4 kernel and verified three invariants [15,4]. Then they used the B method to model Application Programming Interface (API) as functional specifications without verification efforts [6]. Later, they conducted the refinement verification for the seL4 kernel w.r.t functional correctness [3]. Furthermore, they verified information-flow security properties for the kernel [8]. All scripts for modeling and proofs are implemented in Isabelle/HOL.

Costanzo et al. proposed the CertiKOS architecture for verifying the correctness of concurrent operating system kernels [1]. They implemented the verification by defining a series of logical abstraction layers and context refinement relations.

Nelson et al. [10] proposed the push-button verification for OS. They built three models for Hyperkernel in Python, a kernel with finite interfaces. They used the Z3 solver [7] to prove functional correctness and consistency between the abstract model and the implementation model. Later, they extended the verified properties to information flow security, in which they proposed a framework namely Nickel for verifying noninterference and used it to verify NiStar, NiKOS, and ARINC 653 standard [12]. In addition, they presented the Serval framework for developing automated verifiers [9].

3 Formal Specification of the L4 API

This section details the formal specification of the L4 API. We first define the constants and types, then give the definitions of state and initial state, next show the state transitions according to modules, and finally, provide the parameterized abstract model that improves reusability.

3.1 Constants and Types

In the L4 kernel, constants mainly include several threads and address spaces. When the kernel starts up, two privileged threads (σ_0 and $rootserver$) and all interrupt threads ($IntThreads$) are created, where $IntThreads$ is a set of threads.

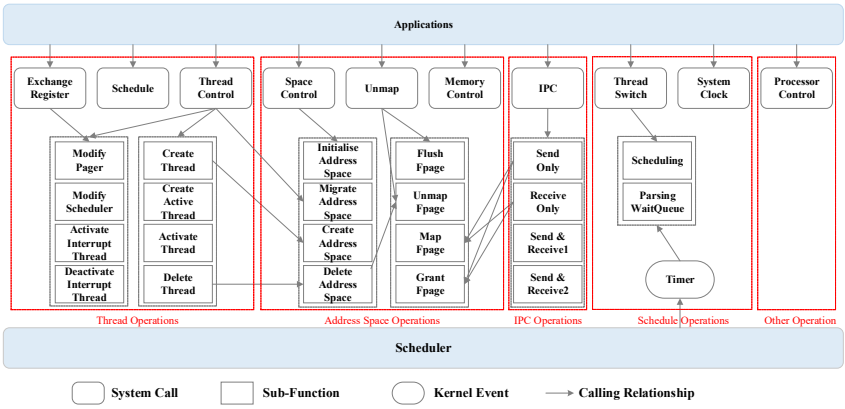

Fig. 3. L4 API, Sub-functions, and Their Relationship Graph.

These threads have their own address space. The σ_0's space is $Sigma0Space$, and $rootserver$'s space is $RootserverSpace$. We define the space of $IntThreads$ as $KernelSpace$ because they are running in kernel mode. There is a special thread namely *idle*, which starts running when the CPU is idle, and its space is $KernelSpace$.

Most types are defined according to the data structure in the source code. The following shows some special selections.

$$threadid\_t \ = \ Global \ globalid\_t \mid nilthread \mid anythread$$

The thread identifiers contain global identifiers, *nilthread*, and *anythread*, where a global identifier may represent a user thread, a kernel thread, or an interrupt thread, and the last two are special identifiers, usually used in the IPC mechanism.

$$fpage\_t \ = \ base \times size \times perms\_t \ set$$

A flexible page can be specified by type $fpage\_t$ that includes three fields of the base address, size, and permissions, where the size field makes the page size variable, and the permissions include read, write, and execute. In Klein's address space model, $fpage\_t$ is not taken into account resulting in all pages having the same size.

$$Space \ = \ spaceName\_t \rightharpoonup v\_page\_t \rightharpoonup page\_t \times perms\_t \ set$$

Where $spaceName\_t$ identifies address spaces, v_page_t identifies pages in address spaces (virtual pages for short), the type $page\_t$ is defined as:

$$page\_t \ = \ Virtual \ spaceName\_t \ v\_page\_t \mid Real \ r\_page\_t$$

Here, virtual pages and physical pages (identified by r_page_t) are unified into pages. Since $spaceName\_t$ appears in both parameters and results, $Space$ is a recursively constructed type with a tree structure. For a given virtual page, it can obtain the mapping page and permissions to access that page through $Space$. Actually, the type models the functionality of both page tables and Mapping DataBase (MDB, a core data structure used for maintaining relationships be-

Table 2. State Fields

Address Space: initialised_spaces, space_threads, space_mapping
Thread: current_thread, threads, active_threads, thread_space, thread_scheduler, thread_state, thread_pager, thread_handler, thread_message, thread_rcvWindow, thread_error, thread_priority, thread_total_quantum, thread_timeslice_length, thread_current_timeslice
IPC: thread_ipc_partner, thread_ipc_timeout, thread_recv_for, thread_recv_timeout, thread_incoming
Scheduling: current_time, wait_queuing, ready_queuing, current_timeslice
Other: heap, tlb

tween virtual pages), because 1) a valid virtual page can translate to a physical page; 2) a virtual page knows who mapped the physical page to it. The symbol $\rightharpoonup$ is the abbreviation of $\Rightarrow$ *option*, i.e., the return value is wrapped with the type *option*. In *Space*, the first $\rightharpoonup$ indicates that it can be known whether the given address space is created, and the second one indicates whether the given virtual page has a mapping.

3.2 State and Initialization

The fields of the state are shown in Table 2, which are built on the data structures in the source code, such as TCB, page tables, and so on. Some special phenomena include: each thread has a scheduler recorded by *thread_scheduler*. The scheduler is a special thread that modifies scheduling-related information for the specified thread. This may easily be confusing because there is a global scheduler used for managing scheduling modules in the source code. In addition, the fields related to the IPC module are defined in TCB.

The initialization operation mainly serves the threads created when the system starts up, including the privileged threads, interrupt threads, etc. For instance, the field *threads* is initialised as $\{\sigma_0,\ rootserver\} \cup IntThreads$.

3.3 State Transitions

In a state machine, state transitions are driven by events, where events refer to system calls shown in Table 1. The transition functions are built on both the Kernel Reference Manual and the source code. In order to reduce the complexity of each module API and the coupling between modules, we analyze and decouple the system calls into a series of sub-functions, shown in Fig. 3. The following shows the formal specification according to modules, including threads, address spaces, IPC, scheduling, and others.

Threads. The operations of thread modules include *ThreadControl*, *Schedule*, and *ExchangeRegister*. Note that *Schedule* is not a traditional schedule function (like switching context), but a function for modifying the scheduling-related fields in TCB of a thread by its *scheduler* thread. Taking *ThreadControl* as an example, the formal specification is shown in Fig. 4. For the sub-functions of

```
1  definition ThreadControl :: "Sys_Config ⇒ State ⇒ threadid_t ⇒ threadid_t ⇒ threadid_t ⇒
       threadid_t ⇒ State" where
2  "ThreadControl SysConf S destNo spaceSpec schedNo pagerNo =
3  (if ThreadControl_Cond SysConf S destNo spaceSpec schedNo pagerNo # basic check
4   then
5    (if ¬ dIsPrivilegedSpace (GetCurrentSpace S) # check the current space
6     then SetError S (current_thread S) eNoPrivilege
7     else
8      (if spaceSpec = nilthread # try to delete a thread
9       then
10      (if ThreadControl_Delete_Cond S destNo
11       then WeakDeleteThread SysConf S destNo
12       else SetError S (current_thread S) (SOME e. e ∈ {eUnavailableThread,eNoPrivilege}))
13     else
14      (if (spaceSpec ≠ nilthread) ∧ (destNo ∉ GetThreadsTids S) # try to create a thread
15       then
16        (if ThreadControl_Create_Cond SysConf S destNo spaceSpec schedNo pagerNo
17         then WeakCreateThread SysConf S destNo spaceSpec schedNo pagerNo
18         else SetError S (current_thread S) (SOME e. e ∈ {eInvalidSpace,eUnavailableThread,
               eInvalidScheduler, eUnavailableThread,eOutOfMemory}))
19      else
20       (if (spaceSpec ≠ nilthread) ∧ (destNo ∈ GetThreadsTids S) ∧ (TidToGno destNo ∉
            kIntThreads) # try to modify a thread
21        then
22         (if ThreadControl_Modify_Cond SysConf S destNo spaceSpec schedNo pagerNo
23          then WeakModifyThread SysConf S destNo spaceSpec schedNo pagerNo
24          else SetError S (current_thread S) (SOME e. e ∈ {eInvalidSpace,eUnavailableThread,
                eInvalidScheduler,eOutOfMemory}))
25        else
26         (if (spaceSpec ≠ nilthread) ∧ (destNo ∈ GetThreadsTids S) ∧ (TidToGno destNo ∈
              kIntThreads) # try to handle an interrupt thread.
27          then
28           (if pagerNo ∈ GetThreadsTids S
29            then IntThreadControl SysConf S destNo pagerNo
30            else SetError S (current_thread S) eUnavailableThread)
31          else S)))))
32 else S)"
```

Fig. 4. Formal Definition of the System Call *ThreadControl*

creation, modification, and deletion, the specification corresponds exactly to the informal manual shown in Fig. 1. Since there is no clear description for handling interrupt threads, we refer to the source code and add a conditional branch (shown in Lines 26-30) to supplement information from the manual.

Address Spaces. The operations of address spaces include *SpaceControl*, *Unmap*, and *MemoryControl*. The complex sub-functions focus on the mapping mechanism including unmap, flush, map, and grant for pages. Before formalizing these functions, we introduce several key definitions. We define $s \vdash x \rightsquigarrow^1 y$ to represent that both pages are in one path, and the page x can reach the page y by one step. The terms $s \vdash x \rightsquigarrow^+ y$ and $s \vdash x \rightsquigarrow^* y$ represent transitive, and reflexive and transitive paths, respectively. If a page x is a physical page or x can reach another page in a given state s, then x is a valid page denoted as $s \vdash x$. Leveraging these definitions, we give the specification for the function *map*, shown in Fig. 5. In the above definition, we add the formalization of access permissions in terms of Klein's model. Lines 3 and 5 show the conditions of successfully executing the operation. According to our experience for proving subsequent invariants, even

```
definition map :: "State ⇒ spaceName_t ⇒ v_page_t ⇒ spaceName_t ⇒ v_page_t ⇒ perms_t set
      ⇒ State" where
"map s sp_from v_from sp_to v_to perms =
(if (sp_to ≠ Sigma0Space)
 then
  (if s ⊢ (Virtual sp_from v_from) ∧ perms ≠ {} ∧ perms ⊆ get_perms s sp_from v_from ∧
       sp_from ≠ sp_to ∧ (∀v. ¬s ⊢ (Virtual sp_from v_from) ⇝+ (Virtual sp_to v)) ∧
       space_mapping s sp_to ≠ None ∧ v_to < page_maxnum
   then s(|space_mapping:= λsp'.
    (if space_mapping s sp' = None
     then None
     else Some (λv_page1.
      (if the (space_mapping s sp') v_page1 = None ∧ Virtual sp' v_page1 ≠ Virtual sp_to v_to
       then None
       else
        (if s ⊢ (Virtual sp' v_page1) ⇝* (Virtual sp_to v_to)
         then
          (if (Virtual sp' v_page1) = (Virtual sp_to v_to)
           then Some ((Virtual sp_from v_from), perms)
           else None)
         else the (space_mapping s sp') v_page1))))|)
    else s)
 else s)"
```

Fig. 5. Formal Definition of the Sub-function *map*

if conditions related to permissions are not considered, there is a lack of these conditions in Klein's model, causing their first invariant (corresponding to our Invariant 1) to not hold. Based on the above definition, we follow the method of iteratively processing pages of the same size in the source code and define a recursive function to handle flexible pages.

IPC. The IPC mechanism is implemented by the system call IPC. By specifying parameter values, the call may involve both the sending phase and the receiving phase. Only after the sending phase is completed, the receiving phase can be executed. According to two phases, we decouple the operation into four subsections, i.e., *send_only*, *receive_only*, *send_receive*1, and *send_receive*2, where the difference between the last two definitions is whether the sending operation is performed successfully. If not, the data of the receiving phase must be saved in a stack. For instance, the parameters used for specifying the receiver's information are saved in state fields *thread_recv_for* and *thread_recv_timeout* in our model.

Scheduling. The operations of the scheduling module include *ThreadSwitch* and *SystemClock*. For the former, the user thread can use it to actively switch current context. To make our specification more complete, we model the unique operation guided by the kernel, i.e., timer interrupt. The operation is used to complete thread scheduling and its strategy is either preemption or a common scheduling.

Others. In addition to the above operations, the behaviors of the Memory Management Unit (MMU) and Translation-Lookaside Buffer (TLB) also are involved in our specification. For these, we just provide a model, and the proofs of the functional correctness and safety properties are not in the scope of this work.

3.4 Parameterized Abstract Model

In addition to the above concrete model, we build a parameterized abstract model to improve reusability using the locale system provided by Isabelle/HOL. The system generally includes variables and assumptions, in which variables are parameters of the system, and assumptions describe the relationships between variables. In our model, the variables involve the initial state s_0, the state transition function *step*, and some key L4 components such as σ_0, *rootserver*, and so on, denoted as:

$$M_{abs}\ \{s_0,\ step,\ \sigma_0,\ rootserver,\ \cdots\}$$

where M_{abs} is the system name, and the types of its parameters are abstract to improve reusability, because data structures in different implementations of L4 microkernels vary slightly. The assumptions contain only some general safety properties. For example, an active thread must be a created thread, i.e., $active\ s \subseteq threads\ s$, which applies to almost all L4 microkernels. If a property relies too much on the implementation of the API or sub-functions, then it will be difficult to reuse. This is also why we do not consider adding functional correctness to the assumptions. In subsequent refinement proofs, these assumptions must be proven to be true.

4 Formalizing Functional Correctness and Safety Properties

The section depicts the formalization of the functional correctness and safety properties. In general, functional correctness means that a program has a correct output for a given input, which can be easily described by the Hoare Triple [2], i.e., $\{P\}\ c\ \{Q\}$, where P is the pre-condition, Q is the post-condition, and c represents the program. Safety means that there is no unsafe state in the whole state space, which can be represented as a series of invariants. The form of expressing an invariant lemma is similar to the Hoare triple, which can be defined as $\{I\}\ c\ \{I\}$ or $\{I_1 \wedge I_2 \wedge \cdots\}\ c\ \{I\}$. In total, we formalized 350 functional correctness and 39 safety properties.

Functional Correctness. Almost of functional correctness lemmas are built on sub-functions and auxiliary definitions (the sum of the two quantities is 50). These lemmas describe the changes in all state fields which are divided into 7 parts (current, UTCB, TCB, address space, mapping, IPC, scheduling). For a given sub-function f, the correctness lemmas include two cases: 1) After executing f, the changed fields are set to correct values. 2) The values of unchanged fields in the original state and the new state are equal. The advantage of constructing lemmas in this way is that the functional correctness is relatively complete. In addition, these lemmas built on sub-functions provide convenience for subsequent proofs, e.g., we can exploit these lemmas and eliminate the function through a substitution strategy instead of directly unfolding its definition causing the structure to be destroyed.

The following shows one of the functional correctness lemmas of the sub-function *map* whose function is to add mapping to a page.

Lemma 1. $\neg s \vdash (Virtual\ sp\ v) \rightsquigarrow^* (Virtual\ sp\_to\ v\_to) \Longrightarrow$
$s \vdash (Virtual\ sp\ v) \rightsquigarrow^+ page \Longrightarrow$
$(map\ s\ sp\_from\ v\_from\ sp\_to\ v\_to\ perms) \vdash (Virtual\ sp\ v) \rightsquigarrow^+ page$

The lemma means that if a virtual page $Virtual\ sp\ v$ is not on the path to $Virtual\ sp\_to\ v\_to$, then the operation has no effect on $Virtual\ sp\ v$, and its accessibility to other pages remains unchanged.

Safety Properties. Safety properties are represented as invariants. Parts of invariants with cumbersome proofs are mainly related to the address space module, and they are shown as follows.

Invariant 1 *Pages do not form rings in the address space structure.*

$$\forall s\ sp\ v1.\ (\nexists v2.\ s \vdash (Virtual\ sp\ v1) \rightsquigarrow^+ (Virtual\ sp\ v2))$$

In Klein's model, the invariant is defined as $\forall s.\ (\nexists x.\ s \vdash x \rightsquigarrow^+ x)$, which only ensures that for a given virtual page there is no loops on this page, and is a corollary of Invariant 1. In fact, it is naturally unreasonable if the page can reach another page in its address space because they will eventually be translated to the physical page. Our definition solves this problem by allowing that $v1$ is not equal to $v2$.

Invariant 2 *A page is valid if and only if there is a physical page translated from the valid page.*

$$\forall s\ x.\ (s \vdash x \longleftrightarrow (\exists r.\ s \vdash x \rightsquigarrow^* (Real\ r)))$$

Invariant 2 improves the corresponding invariant in Klein's model from implication to equivalence.

Invariant 3 *A page has a subset of the permissions of its direct parent page.*

$$\forall s\ sp1\ sp2\ v1\ v2.\ s \vdash (Virtual\ sp1\ v1) \rightsquigarrow^1 (Virtual\ sp2\ v2) \longrightarrow get\_perms\ s\ sp1\ v1 \subseteq get\_perms\ s\ sp2\ v2)$$

Invariant 4 *The permissions of valid pages are not empty.*

$$\forall s\ sp\ v.\ s \vdash (Virtual\ sp\ v) \longrightarrow get\_perms\ s\ sp\ v \neq \{\}$$

Invariants 3 and 4 are proposed to ensure properties on the additional permission fields.

Invariant 5 *A created thread must have an address space, and the space has been created.*

$$\forall s\ t.\ t \in threads\ s \longrightarrow (\exists sp.\ sp \in spaces\ s \wedge thread\_space\ s\ t\ =\ Some\ sp)$$

Invariant 6 *For an arbitrary created address space sp, the set of threads in sp is equal to that of threads whose space is sp.*

$$\forall s\ sp.\ sp \in spaces\ s \longrightarrow the\ (space\_threads\ s\ sp)\ =\ \{t.\ thread\_space\ s\ t\ =\ Some\ sp\}$$

Invariants 5 and 6 are used to associate threads with address spaces.

Invariant 7 *The identifier of the thread that has not been created must be within the configured range.*

$$\forall\, s\ x.\ x \notin threads\ s \longrightarrow x \in Threads\_Gno\ SysConf$$

Following the source code, the global identifier does not exceed $2^{18} - 1$. We use $SysConf$ to define the system environment, and $Threads\_Gno$ obtains the set of global identifiers from $SysConf$.

Invariant 8 *A created thread must have a scheduler, and the scheduler has been created.*

$$\forall\, s\ t.\ t \in threads\ s \longrightarrow \\ (\exists\, sche.\ sche \in threads\ s \wedge thread\_scheduler\ s\ t\ =\ Some\ sche)$$

When invariants 7 and 8 are proved, some bugs in the source code are discovered, and they are discussed in detail in Section 6.

5 Formal Verification

The section illustrates the formal verification for the L4 API. The proof task consists of three parts: functional correctness, safety properties, and refinement between the abstract model and the concrete model. The following first introduces the rewrite rules and reasoning steps that improve verification efficiency, then separately shows proofs of the three parts.

5.1 Rewrite Rules and Reasoning Steps

Since different strategies can be used and the order of proof can also be different, the properties can be proven by various proof methods. However, whether the proof method is excellent can greatly affect the verification efficiency. A good method generally wishes proof steps to be concise and reusable. A typical counterexample is the abuse of automatic tactics provided by Isabelle/HOL. For example, the tactic *auto* tries its best to make the proof goal as simple as possible, but the extent of simplification is unclear, in other words, the subgoal obtained must be re-analyzed every time, as long as the original goal has not been proven completely. Even worse, some of the structure in the original goals has been destroyed.

To avoid these problems, we first propose 21 rewrite rules to simplify the goal and obtain the desired subgoals. These rules are constructed for the three expressions of if, let, and case, which are common in our specification. Some typical rules are shown as follows:

- $if.\ (Q \Longrightarrow P\ x) \Longrightarrow (\neg Q \Longrightarrow P\ y) \Longrightarrow P\ (if\ Q\ then\ x\ else\ y)$
- $let.\ P\ s\ t \Longrightarrow (\bigwedge s\ t.\ P\ s\ t \Longrightarrow P\ (f\ s)\ (f\ t)) \Longrightarrow P\ (Let\ s\ f)\ (Let\ t\ f)$
- $case.\ (opt\ =\ None \Longrightarrow P\ f1) \Longrightarrow ((opt \neq None) \Longrightarrow P\ (f2\ (the\ opt))) \Longrightarrow P\ (case\ opt\ of\ None \Rightarrow f1 \mid Some\ x \Rightarrow f2\ x)$

```
lemma "¬s⊢(Virtual sp v)⇝*(Virtual sp_to v_to) ⟹ s⊢(Virtual sp v)⇝+y ⟹ (map s
    sp_from v_from sp_to v_to perms)⊢(Virtual sp v)⇝+y"
proof-
  assume a1:"s⊢(Virtual sp v)⇝+y" and a2:"¬s⊢(Virtual sp v)⇝*(Virtual sp_to v_to)"
  then show ?thesis
  proof(induction rule:tran_path.induct)
    case (one_path x y) # the case of direct path
    then have "(map s sp_from v_from sp_to v_to perms)⊢x⇝1y"
      using map_not_path_direct FatherIsVirtual by metis
    then show ?case using tran_path.intros by blast
  next
    case (tran_path x y z) # the case of transitive path
    then have "¬s⊢y⇝*(Virtual sp_to v_to)"
      using refl_tran by blast
    then have h1:"(map s sp_from v_from sp_to v_to perms)⊢y⇝+z"
      using tran_path by simp
    have "(map s sp_from v_from sp_to v_to perms)⊢x⇝1y"
      using tran_path map_not_path_direct FatherIsVirtual by metis
    then show ?case using h1 tran_path.intros by simp
  qed
qed
```

Fig. 6. Formal Proof for Correctness of the Sub-function *map*

To our knowledge, the theory library provided by Isabelle/HOL does not include the rules we proposed, although many of them look very similar, especially for the if rule above. But a tiny difference such as replacing $\Longrightarrow$ with $\longrightarrow$ will produce different results.

Second, we construct some general reasoning steps to improve verification efficiency. Our construction follows two principles: 1)If the goal is not fully proven in the current proof step, then this step must be deterministic (i.e., producing specific subgoals); 2)Allows the use of automated tactics that only work on the current goal if the current goal can be directly proven; 3)Allows the use of any automated proof tactic in the last step. Since the reasoning procedure is quite similar for a given invariant or function, these steps are mainly used for proving invariants and they are especially effective when the state fields involved in the invariant do not appear in the functions being proved. Leveraging these steps, we often only need to replace auxiliary lemmas without modifying proof tactics. Indeed, in our experience, most lemma proofs are written through a copy-replace-paste procedure. In Section 5.3, we take proving a concrete safety property as an example to show the use of reasoning steps.

5.2 Functional Correctness Proofs

Sophisticated proofs of functional correctness focus on mapping operations. On the one hand, these operations involve the reflexive and transitive path, causing that for almost every correctness lemma, we must use the **induction** tactic introduced by hand; On the other hand, it is not easy to clarify the relationship between virtual pages in the tree address space structure. Fig. 6 shows the proof of Lemma 1 related to mapping operations. Except for mapping operations,

```
1  lemma DeleteThread_Inv_Space_Perms_IsNot_Empty:
2    assumes p1:"Inv_Space_Perms_IsNot_Empty s"
3    shows "Inv_Space_Perms_IsNot_Empty (DeleteThread s gno)"
4    apply(subst DeleteThread_eq)
5    apply(rule elim_if)
6    subgoal
7      apply(rule SetError_Inv_Space_Perms_IsNot_Empty)
8      apply(rule delete3_Inv_Space_Perms_IsNot_Empty)
9      apply(rule delete2_Inv_Space_Perms_IsNot_Empty)
10     apply(rule delete1_Inv_Space_Perms_IsNot_Empty)
11     using assms by simp
12   using assms by simp
```

Fig. 7. Formal Proof for Invariant 4 on *DeletingThread*

other operations are proven to be functionally correct mainly by unfolding their definitions.

5.3 Safety Proofs

To verify safety properties, we prove that the specification satisfies all of the invariants. An invariant usually is proved by induction, i.e., both the initial state and the transition step are established on the invariant. The former is proved by unfolding the definition of the initial state s_0_def, while the latter is demonstrated by an example, shown in Fig. 7. The lemma describes that the transition of deleting a thread satisfies Invariant 4. Here, due to the complexity of the function *DeleteThread*, we equivalently replace this function with an execution sequence [detele1, delete2, delete3, SetError] to simplify the proof. Line 5 shows the application of the rewrite rule $elim\_if$ for if expressions, which decouples the goal to two subgoals. The proof task is concentrated on the first that was brought out by the keyword **subgoal** in Line 6. We leverage the lemma that every element in the sequence satisfies the invariant to reduce our conclusion to the hypothesis in reverse order. The second subgoal is the case when the execution condition is not met (the state is unchanged), which is proved by *simp*. The whole proof process forms the general reasoning steps, which is firstly applicable to almost all proofs of *DeleteThread* on invariants, and secondly can be reused for other functions but only requires modification of the auxiliary definitions or lemmas used to assist the proof.

5.4 Refinement Proofs

The refinement proofs are used to ensure consistency between the abstract level and the concrete level. Recalling the abstract model M_{abs} using the locale system, we must instantiate it into the concrete model, which can be organized by the keyword **interpretation** as follows.

$$\textbf{interpretation}\ M_{abs}\ \{s_0',\ step',\ \sigma_0',\ rootserver',\ \cdots\}$$

The elements within the brackets are defined in the concrete model, and they replace corresponding parameters of the system M_{abs}. Thus, an instantiation

Table 3. Efforts for Specification and Proofs

Item	Specification		Correctness		Invariants	
	LOC	PM	LOP	PM	LOP	PM
Threads	∼600	1	∼2500	1	∼9000	2
Address Space	∼300	1	∼4500	2	∼1300	1.5
IPC	∼200	0.5	∼1000	0.5	∼3000	1
Scheduling	∼150	0.5	∼2000	1	∼5000	2
Others	∼300	0.5	—	—	∼500	0.5
Total	∼1550	3.5	∼10000	4.5	∼18800	7

theorem is defined if there is no type conflict. Next, our task is to prove that the assumptions on these concrete variables hold. Since these assumptions are invariants proved in Section 5.3, the theorem can be easily derived through the tactic *auto*.

6 Discussion

Result. We leverage Isabelle/HOL to build a comprehensive formal specification for the L4 API. The specification completely covers all modules in the kernel, including address spaces, threads, IPC, scheduling, and others. We prove that the formal specification satisfies all functional correctness and invariants we proposed. To improve the readability of formal proofs, most formal proofs are written in a structured language Isabelle/Isar [16], especially some complex lemmas. In total, this work produces about 1.5K lines of code(LOC) for specifications and about 29K lines of proofs(LOP) and takes about 15 person-months(PM). Details of this work are described in Table 3.

Verified Issues. During specification and verification, we found 10 bugs that violate functional correctness and safety properties. They are classified into 6 categories and reported as follows.

- **Out-of-Bounds Access.** When we prove the invariant 7, we found that in the system calls *ThreadControl*, *IPC*, and *ExchangeRegister*, there is no policy to limit the range of the destination thread's identifier *dest_tid*. This bug allows threads to access non-TCB areas. We recommend adding a condition expression into the source code, ensuring that these system calls work only if *dest_tid* does not exceed the maximum.
- **Illegal Deletion.** The source code allows the privileged threads to delete any unprivileged thread, which may cause exceptions to occur. We know that once a thread is created, it is assigned a scheduler used for managing its scheduling-related fields. Thus, invariant 8 needs to be guaranteed in the whole state space. However, the invariant on the sub-function of deleting thread cannot be proven to be true, this is because there will be no scheduler to serve the thread if the deleted object is the scheduler. Our recommendation is to only allow privileged threads such as *rootserver* to serve

as the scheduler for a thread. The advantage is that it can be implemented by modifying very little code because there is no need to check the dependencies between unprivileged threads.

- **Lack of Validity Checks.** The system call *ThreadControl* does not check whether the identifier of the parameter *scheduler_tid* is valid, where validity means the thread represented by *scheduler_tid* has been created. The lack of these checks still violates the invariant 8 when reasoning about the sub-function of creating threads. Following our specification of *Create_Thread*, the bug can be fixed by determining whether *scheduler_tid* exists in the created thread collection (*threads*) that is defined as a ring queue called *present_list* in the source code.
- **Unfinished Definitions.** When modeling the functionality of the activating thread, we found there is no handling of the case when the activation operation fails. We recommend that before activating a thread, make a backup of the fields that need to be changed during activation, and restore the values of these fields if the activation fails. Some similar bugs include a lack of handling failure to allocate address space; and a lack of implementation for the system call *ProcessorControl*.
- **Incomplete Initialization.** In the header file *schedule.h* of the source code of the Pistachio0.4 version, the initialization function *init* does not initialize the last priority queue. In detail, in the code snippet of $for(int\ i\ =\ 0;\ i\ <\ MAX\_PRIO;\ i++)\{\cdots\}$, the loop condition should be set as $i\ <=\ MAX\_PRIO$. It was discovered when we compared the initial state of our model with that of the source code. Fortunately, this bug is fixed in the latest version.
- **Inconsistent Implementation.** In the source code, the handler of each interrupt thread is recorded by the field *scheduler* in TCB, while the advice given in the manual is to take the field *pager* in UTCB. In our experience, it is reasonable to record the handler by *scheduler*, because the interrupt threads are executed in the kernel mode, and can be viewed as kernel threads, thus, there is no need to assign them UTCB areas.

7 Conclusion and Future Work

This paper proposes a comprehensive formal specification and verification for an L4 microkernel API. The formal specification makes up for the missing core components of the existing models and fixes the errors in these models. To improve the correctness and reliability, 350 functional correctness and 39 safety properties are formalized. After machine-checking proof in Isabelle/HOL, the formal specification strictly satisfies the proposed 350 functional correctness and 39 safety properties. Through decoupling functionalities into some sub-functions, abstracting the parameterized model, rewriting proof rules, and building reason patterns, we solve the challenges in this work on complexity, reusability, and efficiency. During specification and verification, we found 10 bugs in the kernel reference manual and source code of the L4 microkernel, and we provided solutions to fix them. In the future, we will expand the verification for the API to

that for the entire source code and may pay more attention to the automation technology in refined verification.

Acknowledgment

This work has been supported in part by the Natural Science Foundation of China under Grants No. 62132014 and No. U2341212, and Zhejiang Science and Technology Plan Project under Grant No.2022C01045.

References

1. Gu, R., Shao, Z., Chen, H., Wu, X.N., Kim, J., Sjöberg, V., Costanzo, D.: CertiKOS: An extensible architecture for building certified concurrent os kernels. In: Keeton, K., Roscoe, T. (eds.) 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. pp. 653–669. USENIX Association (2016), https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu
2. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (1969)
3. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D.A., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an os kernel. In: Matthews, J.N., Anderson, T.E. (eds.) Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009. pp. 207–220. ACM (2009). https://doi.org/10.1145/1629575.1629596, https://doi.org/10.1145/1629575.1629596
4. Klein, G., Tuch, H.: Towards verified virtual memory in l4. TPHOLs Emerging Trends **4**, 16 (2004)
5. Kolanski, R.: A formal model of the μ-kernel api using the b method. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney **2052** (2004)
6. Kolanski, R., Klein, G.: Formalising the l4 microkernel api. In: Proceedings of the Twelfth Computing: The Australasian Theory Symposium-Volume 51. pp. 53–68 (2006)
7. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24, https://doi.org/10.1007/978-3-540-78800-3_24
8. Murray, T.C., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., Klein, G.: sel4: From general purpose to a proof of information flow enforcement. In: 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013. pp. 415–429. IEEE Computer Society (2013). https://doi.org/10.1109/SP.2013.35, https://doi.org/10.1109/SP.2013.35

9. Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E., Wang, X.: Scaling symbolic evaluation for automated verification of systems code with serval. In: Brecht, T., Williamson, C. (eds.) Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019. pp. 225–242. ACM (2019). https://doi.org/10.1145/3341301.3359641, https://doi.org/10.1145/3341301.3359641
10. Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., Wang, X.: Hyperkernel: Push-button verification of an OS kernel. In: Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017. pp. 252–269. ACM (2017). https://doi.org/10.1145/3132747.3132748, https://doi.org/10.1145/3132747.3132748
11. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A proof assistant for higher-order logic. Springer-Verlag (2002)
12. Sigurbjarnarson, H., Nelson, L., Castro-Karney, B., Bornholt, J., Torlak, E., Wang, X.: Nickel: A framework for design and verification of information flow control systems. In: Arpaci-Dusseau, A.C., Voelker, G. (eds.) 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018. pp. 287–305. USENIX Association (2018), https://www.usenix.org/conference/osdi18/presentation/sigurbjarnarson
13. Team, L.: Pistachio microkernel. https://www.l4ka.org/65.php (2010)
14. Team, L.: L4 experimental kernel reference manual version x.2 (2011), https://www.l4ka.org/l4ka/l4-x2-r7.pdf
15. Tuch, H., Klein, G.: Verifying the l4 virtual memory subsystem. In: Proc. NICTA FM Workshop on OS Verification. pp. 73–97 (2004)
16. Wenzel, M., et al.: The isabelle/isar reference manual (2004)