Refinement Verification of OS Services based on a Verified Preemptive Microkernel

Ximeng Li[2], Shanyan Chen[1], Yong Guan[1,3], Qianying Zhang[2,3], Guohui Wang[2,3], Zhiping Shi[1,2](✉)

[1] College of Information Engineering, Capital Normal University, Beijing, China
shizp@cnu.edu.cn
[2] Beijing Key Laboratory of Electronic System Reliability and Prognostics, Capital Normal University, Beijing, China
[3] Beijing Advanced Innovation Center for Imaging Theory and Technology, Capital Normal University, Beijing, China

Abstract. An OS microkernel can be extended by implementing services upon it. A service could introduce an object that references a kernel object, and implement a group of functions that invokes the functions for manipulating the kernel object. We consider the scenario where the microkernel has been verified with machine-checkable proofs, while the services remain to be verified. Moreover, the verification of the microkernel is not performed with the verification of subsequent extension in mind. We address the problem of how to build sufficiently on the verification results for the microkernel, in achieving the verification of the services. Our methodology consists of enhancements to the verification framework for the microkernel, and the design of invariants for establishing the connection between the service-level objects and the kernel-level objects. Using the methodology, we have conducted a substantial formal verification of a group of services extending the inter-task communication functionalities of the preemptive microkernel μC/OS-II. Our verification uncovers dormant bugs and provides a level of correctness assurance for the services that is above what is achievable through extensive testing.

1 Introduction

Microkernels provide the most fundamental functionalities of operating systems such as task management, inter-task communication, and interrupt handling. Microkernels are relatively small in size and simple in structure. Compared with monolithic kernels, errors in microkernel-based systems are more likely to occur outside of the kernel. Thus, these errors are less likely to crash the entire system. A *preemptive microkernel* allows a task to be interrupted at any point of execution, as long as interrupts are enabled in the CPU. During interrupt handling, a higher-priority task can be switched to. This mechanism permits the timely processing of urgent workloads, increasing the responsiveness of the system.

D. Beyer and A. Cavalcanti (Eds.): FASE 2024, LNCS 14573, pp. 188–209, 2024.
https://doi.org/10.1007/978-3-031-57259-3_9

On the downside, the possibility of preemption results in a great number of inter-dependencies between tasks. This adds to the difficulty in correctly designing and implementing the microkernel. Out of concern for correctness, substantial efforts have been dedicated to achieving the formal verification of preemptive microkernels (e.g., [28]). These verification efforts lay a solid foundation for assuring the correctness of the software systems based on preemptive microkernels.

Since a microkernel only provides the core functionalities in abstracting and managing system resources, the extension of the functionalities for a microkernel is often required in a given application scenario. The functionality of a *kernel object* O_{knl} can be extended in the following way. Firstly, a data structure is introduced — an instance O_{srv} of this data structure contains a reference to O_{knl}, while maintaining some additional attributes. Secondly, the operations that can be performed on O_{srv} are implemented. In these operations, checks and updates are performed on the additional attributes in O_{srv}, and the operations for O_{knl} are invoked to complete the checks and updates on the internal attributes. The extension provides a service to the user. We shall refer to O_{srv} as a *service object*.

For instance, the mutexes in a microkernel might not support modes of operations such as recursive and non-recursive modes. This feature can be introduced in an extension of the microkernel, providing a modes-aware mutex service to the user. Firstly, a service-level mutex object can be introduced. Secondly, the mode of a mutex can be tracked by an attribute of this service object. Thirdly, in an operation that tries to obtain a service-level mutex that the current task already owns, the attribute is checked before deciding whether to invoke the kernel function for obtaining the mutex or not.

In safety-critical scenarios, the correctness of the services that extend the microkernel can be as important as the correctness of the microkernel itself. A reliable way to ensure the correctness of the services is formal verification. If the microkernel itself has been formally verified, the formal specifications and proofs for the functions of the microkernel could be used as a basis for this verification.

The formal verification of the services can still be non-trivial. This is true especially if the tasks executing the service functions (e.g., the function for obtaining a modes-aware mutex) can be preempted. In this case, it can be non-trivial even to ensure that a service object in use always references a corresponding kernel object that has been properly allocated and initialized. For the verification of the services, another problem is how to achieve good reuse of the specifications and proofs for the underlying microkernel. Moreover, if the proofs for the microkernel have been developed using a verification framework, it would be good to sufficiently leverage this verification framework, as opposed to requiring a great amount of modification to the verification framework.

In this article, we address the aforementioned challenges in the formal verification of OS services (in the above sense) that extend a preemptive microkernel. Specifically, we consider the case where *refinement verification* has been performed for the microkernel, using a variant of concurrent separation logic [9] called CSL-R [28, 27]. This is the program logic used in the first formal verification of a practical preemptive microkernel with machine-checkable proofs.

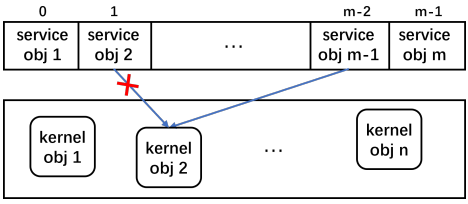

Fig. 1: The connection between service objects and kernel objects

The main contributions of this article include:

1. enhancements to the verification framework of CSL-R to support the compositional specification of the functions implementing the OS services
2. a design of invariants dependent on auxiliary variables for reasoning about the connection between service objects and their underlying kernel objects
3. results obtained by applying the extended verification framework and the invariants design to achieve the formal verification of inter-task synchronization and communication services that extend the corresponding functionalities of the preemptive microkernel μC/OS-II [3]

Specifically, the enhancements to the verification framework of CSL-R enables the integration of the specifications for the kernel functions as components for the specifications of service functions. The connection between the service objects and their underlying kernel objects is shown to satisfy structural properties that are generic to the specific purposes and contents of the services. The verification of the inter-task synchronization and communication services is performed in an industrial verification project in the aerospace domain, while these services also constitute a module of a system to be more widely used in other safety-critical scenarios. We devise the specification of each service function and prove that the specification is refined by the code of the function. The development is performed in the Coq proof assistant [1]. This verification is a substantial effort, in which we have uncovered problems in extensively tested code.

2 Challenges in Verifying an OS Service

We assume a service object (e.g., a service-level task, semaphore, or message queue) is implemented as a struct in C. The service object obj contains a pointer, obj.ptr, to a potential kernel object of the underlying microkernel. The service object contains a number of attributes that are managed outside of the microkernel. Moreover, we assume that all the service objects of the same kind are organized in the array obj_arr. This array is illustrated in the upper part of Fig. 1.

We consider a kernel object to be *active*, if the kernel object has been allocated and initialized. An active kernel object is expected to be in a consistent state. The set of active kernel objects is illustrated in the lower part of Fig. 1.

A desired integrity requirement about the connection between the service objects and the underlying kernel objects is:

Requirement 1 *If a service object is fully created, then the service object references a kernel object that is in a consistent state.*

This requirement is reflected by the arrow without a cross over it in Fig. 1. If the requirement is not met, then an operation on a service object could trigger an operation on an inconsistent kernel object. Hence, the proper completion of the kernel operation with correct results cannot be guaranteed.

Another desired integrity requirement about the connection between the service objects and the underlying kernel objects is:

Requirement 2 *Each kernel object is referenced by at most one service object.*

This requirement is reflected by the arrow with a cross over it in Fig. 1. If a kernel object can be referenced by two or more service objects, then it is difficult to guarantee that all these service objects are consistent with the kernel object. An operation on one of these service objects would update the service object and the kernel object consistently. But this update could break the consistency of another service object with the kernel object.

It can be nontrivial to ascertain the satisfaction of Requirement 1 and Requirement 2 in a preemptive setting. Consider the function service_obj_create in Fig. 2. This function is used to create service objects. The dotted boxes reflect the areas of critical regions, in which the task executing the function cannot be preempted. Line 2 searches for an index idx in obj_arr using the internal function get_free_obj. This index identifies an array element that corresponds to an unused service object. Line 3 checks if the return value of get_free_obj is a valid index for obj_arr. If not, then the entries of obj_arr are used up, and the function service_obj_create returns. Otherwise, obj_arr[idx].ptr gets the special value Dummy at line 4. This value signals that the array entry obj_arr[idx] is reserved — it cannot be used by a different task attempting to create a service object. Then, the critical region is exited. Afterwards, the kernel function kernel_obj_create for creating a kernel object is invoked at line 5. Here, `katt` is the attribute value used to initialize the kernel object. The function returns the pointer to the kernel object that is allocated and initialized — `NULL` in case no kernel object can be allocated. This pointer is assigned to the kernel object pointer in the service object obj_arr[idx] at line 6. Then, it is checked whether the pointer is not `NULL`. The function service_obj_create returns if the kernel object pointer is `NULL`. Otherwise, the data attributes of the created service object obj_arr[idx] are initialized at line 8. The index idx for this created service object is then returned.

```
  service_obj_create(katt, satt):
1   local idx, p
2   idx <- get_free_obj(obj_arr)
3   Check valid(idx)
4   obj_arr[idx].ptr <- Dummy
5   p <- kernel_obj_create(katt)
6   obj_arr[idx].ptr <- p
7   Check p != NULL
8   obj_arr[idx].att <- satt
9   return idx
```

Fig. 2: The function service_obj_create

If Requirement 1 is to be satisfied, the following condition related to the function service_obj_create in Fig. 2 should be met.

Condition 1 *After the completion of the assignment* p<-kernel_obj_create(katt)*, the pointer* p *points to an active kernel object if* p *is not* NULL*.*

This condition guarantees that the pointer assigned to obj_arr[idx].ptr points to an active kernel object — thus a kernel object in a consistent state. This helps ensure that the service object obj_arr[idx] references a kernel object that is in a consistent state, once the service object is fully created. However, Condition 1 might not hold, since the data located at the return address of kernel_obj_create could be modified by preemptive tasks. Hence, dedicated reasoning is required to ascertain that the potential modification of data does not break Condition 1.

If Requirement 2 is to be satisfied, the following condition should be met.

Condition 2 *After the completion of the assignment* p<-kernel_obj_create(katt)*, no service object already references the kernel object pointed to by* p*.*

If Condition 2 is not met, then the service object obj_arr[idx] could start to reference the created kernel object, along with some other service object that originally referenced the same kernel object. It appears that the potential kernel object that is allocated in a call to kernel_obj_create must be free before the allocation. Given the code of service_obj_create, it is unlikely that a free kernel object would get referenced from a service object. However, the joint effects of all the functions supporting the creation, deletion, and use of the service object are more complicated than suggested by this observation. Hence, dedicated formal reasoning is required to ascertain the satisfaction of Condition 2.

In the remainder of the article, we will discuss how to ascertain the satisfaction of Condition 1 and Condition 2, thereby ascertaining the satisfaction of Requirement 1 and Requirement 2, in a refinement verification of OS services. A key ingredient of our methodology is the formulation of invariant conditions dependent on auxiliary variables in a separation logic (see Section 5).

Ultimately, the ability to show that Requirement 1 and Requirement 2 are fulfilled supports the formal verification of the service functions against their specifications. We will also discuss how to compose these specifications from the formal specifications of the underlying kernel functions (see Section 4). This enables the reuse of the specifications and proofs for the kernel functions, as previously developed in the formal verification of the microkernel.

3 Refinement Verification of OS Microkernels

To facilitate the understanding of our technical development, we briefly introduce the verification framework for the concurrent separation logic CSL-R [28, 27], as well as the formal verification of an OS microkernel using this framework.

3.1 The Big Picture

Through the refinement verification of an OS microkernel, a simulation is established between the execution of a concrete system and the execution of an abstract system. The concrete system consists of client programs, kernel functions, and interrupt handlers. The abstract system contains the same client programs

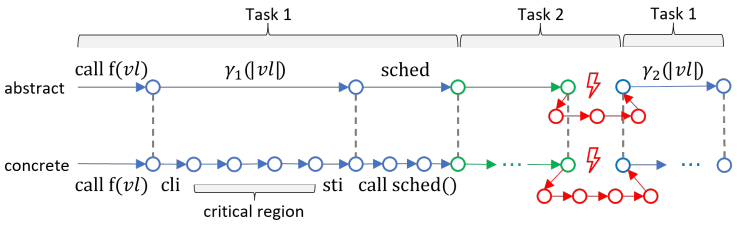

Fig. 3: Execution of a microkernel and simulation by a specification

as the concrete system. In addition, the abstract system contains the specifications for the kernel functions and the interrupt handlers. These specifications are in the form of *abstract programs*, as opposed to concrete C or assembly code.

An example of the simulation between the concrete system and the abstract system is illustrated in Fig. 3. In this figure, the concrete system runs two tasks. Task 1 calls the kernel function f with the list vl of argument values. This function executes a series of steps in a critical region. Then, it needs to wait on an event for a given time period. Hence, it calls the function sched() to trigger rescheduling. Suppose task 2 is scheduled for execution. After several steps taken by task 2, a tick interrupt comes. The arrival of the interrupt is illustrated by $\lightning$. After the interrupt is handled, the system looks for the highest-priority task that is ready for execution. Suppose task 1 has become ready and it is executed for another time. Task 1 then finishes the kernel function f and returns to user code. In the aforementioned scenario, task 2 is *preempted* by task 1.

The kernel function f is specified using the abstract program ω_{f} as given by

$$\omega_{\mathsf{f}}\ vl := \gamma_1(|vl|); \mathsf{sched}; \gamma_2(|vl|)$$

Here, γ_1 and γ_2 represent two atomic steps of execution. Each step has vl as the list of input values. In addition, sched is a primitive for the scheduling operation. Moreover, γ_1, sched, and γ_2 are sequentially composed. We will give further details about the language in which $\omega_{\mathsf{f}}\ vl$ is expressed in Section 3.2.

Part of the simulation between the concrete system and the abstract system is concerned with the simulation of the execution steps for the function f. The abstract statement $\omega_{\mathsf{f}}\ vl$ is executed in the abstract system after the function f is called with the list vl of arguments. The concrete execution steps in the critical region are simulated by the atomic step γ_1. Furthermore, the concrete execution steps for sched() are simulated by the execution step of sched. In addition, the concrete execution steps taken by task 1 after it is resumed are simulated by the atomic step γ_2. The simulation between the concrete system and the abstract system is required to preserve a global invariant. The global invariant is used to relate the states of the two systems — further details will be given in Section 3.3.

The simulation of the concrete system by the abstract system is established by reasoning about each kernel function separately. This reasoning is performed using the rules of the CSL-R logic. For the kernel function f, the goal of the reasoning is to establish the correspondence between the concrete code of f and the abstract program ω_{f}. The reasoning goes forward (in the sense of [16]) in

the concrete code of f, performing symbolic execution of the abstract statement $\omega_{\mathsf{f}}\ vl$ at appropriate points. Thus, the goal is turned into establishing the correspondence between the remainders of f and the remainders of $\omega_{\mathsf{f}}\ vl$, i.e., the abstract statements $\gamma_1(\!|vl|\!); \mathsf{sched}; \gamma_2(\!|vl|\!)$, $\mathsf{sched}; \gamma_2(\!|vl|\!)$, and $\gamma_2(\!|vl|\!)$.

3.2 The Specification of Kernel Functions

As illustrated in Section 3.1, a kernel function is specified using a mathematical function ω. This function maps each list vl of argument values to an *abstract statement* $\mathbf{s}$. This abstract statement is expressed using the values in vl. The syntax for abstract statements is given below.

$$\begin{aligned} \mathbf{s} &::= \gamma(\!|vl|\!) \mid \mathsf{sched} \mid \mathsf{end}\,\hat{v} \mid \mathbf{s}_1; \mathbf{s}_2 \mid \mathbf{s}_1 + \mathbf{s}_2 \\ \hat{v} &::= \mathsf{Some}\ v \mid \mathsf{None} \\ &\text{where } v \in \mathit{Val}, vl \in \mathit{Val}^*, \gamma \in \mathit{Val}^* \times \mathit{AState} \times \mathit{Val}^? \times \mathit{AState} \end{aligned}$$

Here, Val is the set of values, Val^* is the set of value lists, and $\mathit{Val}^?$ is the set of optional values. An optional value is represented by the meta-variable $\hat{v}$. Furthermore, AState is the set of *abstract states*. In the atomic operation $\gamma(\!|vl|\!)$, γ relates the list vl of input values and an initial abstract state to an optional output value and a resulting abstract state. Furthermore, $\mathsf{end}\,\hat{v}$ signals the completion of execution for an abstract statement. In addition, $\mathbf{s}_1; \mathbf{s}_2$ is a sequential composition. Lastly, $\mathbf{s}_1 + \mathbf{s}_2$ is a nondeterministic choice.

An abstract state $\Sigma \in \mathit{AState}$ captures as mathematical objects the memory content that is relevant to the abstract programs of the kernel functions. For example, a C $\mathsf{struct\ s}$ with the members $\mathsf{s.a}$ and $\mathsf{s.b}$ in the memory can be abstractly represented as a pair (a, b) in the abstract state. Overall, an abstract state could contain the representations of typical kernel objects such as kernel-level tasks, semaphores, mutexes, and message queues. The formal semantics of the abstract statements is defined based on reads and updates of the abstract state. We omit the definition of this semantics here.

3.3 Invariants and Fractional Permission

In a concurrent separation logic, the well-formedness of global resources is expressed using a global invariant. Examples of these global resources include the kernel data structures for tasks, synchronization objects, etc. In a concurrent separation logic that supports refinement verification, the global invariant I is interpreted over a concrete state and an abstract state. Thus, I can be used to assert the well-formedness of the global resources in concrete and abstract representations and the relation between the two. Hence, if the $\mathsf{struct\ s}$ mentioned in Section 3.2 is global, then I can be used to assert the well-formedness of s in the memory, the well-formedness of the tuple (a, b) in the abstract state, and the fact that a and b properly represent the memory values of $\mathsf{s.a}$ and $\mathsf{s.b}$.

In reasoning about a kernel function, the global invariant I can be asserted to hold after entering a piece of code that has exclusive access to the global resources (e.g., a critical region in which a task cannot be preempted). The auxiliary information provided by this assertion of I can be used in the subsequent

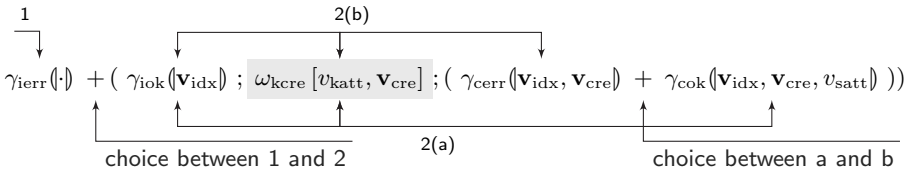

Fig. 4: The abstract statement for service_obj_create

reasoning. The well-formedness of the global resources may be temporarily broken in the code, but it must be re-established at the point where exclusive access to the global resources is given up. At this point (e.g., where a critical region is exited), I must be shown to hold again. Intuitively, a critical region consumes well-formed global resources and gives back well-formed global resources again.

Consider an auxiliary variable that represents the current program location for a task. If the global invariant is formulated to depend on such a variable, then the variable should be treated as a global resource. However, the variable is then modifiable at any point outside of a critical region, by another task that preempts the current one. Nonetheless, the current program location of a task should not be modifiable by a different task. This is where *fractional permission* [8] can be employed to facilitate verification using a concurrent separation logic.

More concretely, an auxiliary variable x can be introduced for a task t, such that t has $\frac{1}{2}$ permission, and the global invariant has $\frac{1}{2}$ permission, over x. A task is allowed (by the program logic) to read a variable, as long as the task has $\frac{1}{2}$ permission over the variable. On the other hand, a task is allowed to modify a variable, only if the task has full permission over the variable. Hence, the task t is allowed to modify the variable x, when the other $\frac{1}{2}$ permission over x is obtained from the global invariant, e.g., in a critical region. The variable x cannot be modified by any preemptive task t'. This is because t' is allowed to obtain at most $\frac{1}{2}$ permission over the variable from the global invariant.

4 Compositional Specification of Service Functions

4.1 Composing Service Specification from Kernel Specification

To enable the refinement verification of the function service_obj_create in Fig. 2, the function should be specified using an abstract statement. This abstract statement should reflect the following cases about the execution of service_obj_create.

1. the execution of service_obj_create could fail, in case there is no usable service object in the system, or
2. service_obj_create could obtain an index $\mathbf{v}_{\text{idx}}$ for a usable service object, attempt at kernel object creation as implemented in kernel_obj_create, obtain the return value $\mathbf{v}_{\text{cre}}$ from kernel_obj_create, and then proceed as follows:
   (a) if $\mathbf{v}_{\text{cre}}$ is the address of a newly allocated and initialized kernel object, then service_obj_create sets the kernel object pointer in the $\mathbf{v}_{\text{idx}}$-th service object to $\mathbf{v}_{\text{cre}}$, sets the data attribute in this service object to the given attribute value v_{satt}, and returns the index value $\mathbf{v}_{\text{idx}}$
   (b) if $\mathbf{v}_{\text{cre}}$ is `NULL`, then service_obj_create returns an invalid index value

We intend to formulate the abstract statement for service_obj_create using the specification language presented in Section 3.2. A potential formulation is given in Fig. 4. At the top level, this abstract statement is a nondeterministic choice between the part expressing the meaning of item 1 and item 2 above. The meaning of item 1 is expressed using the atomic operation γ_{ierr}. The meaning of item 2 is expressed with two sequential compositions. Here, the atomic operation γ_{iok} is used to express the operation of obtaining $\mathbf{v}_{\text{idx}}$. Furthermore, ω_{kcre} is the abstract program for kernel_obj_create. In addition, the nondeterministic choice between γ_{cerr} and γ_{cok} is used to express a choice between the sub-items 2(b) and 2(a) above. This particular choice is deterministic because of the conditions about $\mathbf{v}_{\text{cre}}$ as expressed in 2(a) and 2(b). The correspondence between the informal expression of the functional requirements for service_obj_create and the formal counterpart is illustrated by the annotations in Fig. 4.

The specification of service_obj_create in Fig. 4 is composed of the abstract program for kernel_obj_create. This compositional aspect enables the reuse of the specification for the functions of the underlying microkernel. This reuse implies that the formal proofs for these kernel functions (as developed in verifying the microkernel) can also be reused. However, a technical problem was encountered with specifications like the one in Fig. 4. The function service_obj_create has two formal parameters (see Fig. 2). According to the CSL-R framework, if the abstract program of the function service_obj_create is ω_{scre}, then the result of calling the function with the arguments v_{katt} and v_{satt} in the abstract system is the abstract statement $\omega_{\text{scre}}\,[v_{\text{katt}}, v_{\text{satt}}]$. This cannot be the abstract statement in Fig. 4, because the additional parameters $\mathbf{v}_{\text{idx}}$ and $\mathbf{v}_{\text{cre}}$ are not introduced.

To solve the aforementioned problem, we modify the semantics of the specification language such that a call to a function could nondeterministically result in an abstract statement $\omega\,(vl{+}{+}vl')$, where ω is the mathematical function representing the abstract program for the callee, vl is a list that contains exactly the actual arguments for the callee, and vl' is an arbitrary list of values. Intuitively, the list vl' can be used to accommodate the intermediate values generated in the abstract program. For the above example with service_obj_create, we define ω_{scre} such that $\omega_{\text{scre}}\,([v_{\text{katt}}, v_{\text{satt}}]{+}{+}vl')$ yields the abstract statement in Fig. 4. We use the first value of vl' for $\mathbf{v}_{\text{idx}}$, and use the second value of vl' for $\mathbf{v}_{\text{cre}}$.

With this abstract statement, we intend to express that the atomic operation γ_{iok} identifies a specific index $\mathbf{v}_{\text{idx}}$ — the $\mathbf{v}_{\text{idx}}$-th service object is unused in the abstract state from which the operation is performed. Afterwards, the atomic operation γ_{cok} initializes exactly the $\mathbf{v}_{\text{idx}}$-th service object. However, $\mathbf{v}_{\text{idx}}$ is arbitrary if it is the first value of the arbitrary list vl'. How to ensure that $\mathbf{v}_{\text{idx}}$ is the index found by γ_{iok} at the point where the operation γ_{cok} is performed?

We solve this problem by permitting the execution of an abstract statement to reach an *error state*. From the error state no further execution of the abstract statement is permitted. We adjust the refinement condition to express that the concrete system should be simulated by the abstract system *unless the abstract system is in an error state*. In the abstract program for service_obj_create, we define the atomic operation γ_{iok} such that an error state results if the parameter

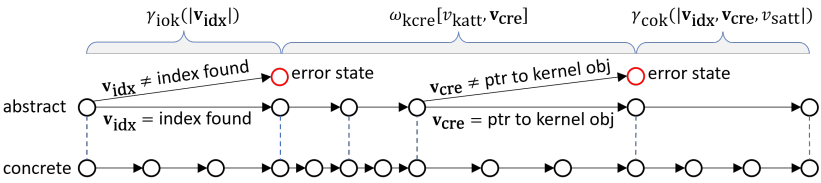

Fig. 5: Simulation for service_obj_create in the extended verification framework (potential preemption before/after atomic operations omitted)

$\mathbf{v}_{\text{idx}}$ is not equal to the found index (see Fig. 5). Hence, if γ_{cok} is executed to simulate the concrete execution of service_obj_create, the previous execution of γ_{iok} could not have ended up in an error state. Thus, $\mathbf{v}_{\text{idx}}$ as used in γ_{cok} is equal to the index of the unused service object found by γ_{iok}.

By admitting the error states in the abstract computation, and extending the notion of refinement in CSL-R correspondingly, we permit using the output of operations in the subsequent abstract computation. In particular, this enables the compositional specification of the service functions — where the abstract programs of the kernel functions may produce results that are used in the abstract programs of the service functions. For sound reasoning about the new notion of refinement, we have also introduced new rules into the program logic. Formally, we have re-established the soundness of the verification framework.

Remark 1. In the μC/OS-II microkernel, the computation result of a critical region is rarely passed to another critical region via local variables or return values of functions. Correspondingly, it is unnecessary to capture the output value of an operation and pass this value to another operation in the abstract program of a function. Hence, the CSL-R framework for the verification of μC/OS-II was not originally designed to accommodate additional parameters like $\mathbf{v}_{\text{idx}}$ and $\mathbf{v}_{\text{cre}}$.

4.2 Expressing Assumptions about the User

A second use of the error states in the abstract computation (as discussed in Section 4.1) is to support the expression of assumptions about user data in the formal specification of the service functions.

For an example of these assumptions, consider a variant of the service function service-obj-create in Fig. 2 that works properly only if the argument satt satisfies a well-formedness condition. More concretely, suppose satt is intended to be a pointer to a struct. This struct contains several attributes for initializing the service object. However, the C language does not provide a feature to check whether satt really points to a well-formed struct that contains these attributes (like instanceof in Java). Hence, this check might not be implemented in the code of this variant of service_obj_create. Then, service_obj_create should be verified under the assumption that satt points to the right type of struct.

The above assumption can be naturally expressed in the pre-condition for a function, if the function is to be verified using an ordinary Hoare-style program logic. However, a service function is specified using an abstract program instead of pre/post-conditions in a refinement verification. Then, the assumption should

be expressed in this abstract program. We express such an assumption in the definition of an atomic operation in the abstract program. More concretely, this atomic operation gives the error state if the assumed condition about user data is not satisfied. With our adjusted definition of simulation, the abstract system is required to simulate the concrete system only if the abstract system is not in an error state (see Section 4.1). This corresponds to the meaning of assumptions — the refinement of the abstract programs by the concrete code is only required if the assumptions about user data are satisfied.

5 Reasoning about Service-Kernel Connection

Through refinement verification of an OS service, we establish the simulation between the execution of the service functions and the execution of their abstract programs (see Section 4.1). This simulation preserves the global invariant.

We express Requirement 1 and Requirement 2 (see Section 2) in the global invariant to show that the satisfaction of both requirements is preserved in the simulation. As explained in Section 2, the establishment of Condition 1 and Condition 2 is supportive of showing the fulfillment of Requirement 1 and Requirement 2. The two conditions can be established if they are also formulated in the global invariant, and are shown to be preserved in the simulation. However, these two conditions involve the program location that is local to a task, as well as a task-local pointer to a kernel object. These parameters cannot be directly expressed in the global invariant. In this section, we explain how to capture the program location and the kernel object pointer for each task using auxiliary variables with fractional permission (Section 5.2). We then present a design of invariant conditions that depends on these auxiliary variables (Section 5.3). We are able to show that Condition 1 and Condition 2 are preserved by the execution of each service function, with the help of the invariant conditions.

The satisfaction of Condition 1 and Condition 2 depends on the way each service function affects the connection between a service object and its underlying kernel object. Hence, we will first present a series of code patterns for service functions that capture a proper way to handle this connection (Section 5.1).

5.1 Creation, Deletion, and Use of Service Objects

We assume that the service functions for creating, deleting, and using a service object possess the code patterns in Fig. 6. The scope of critical regions is represented by the dashed boxes. A line with the content Check cond represents a conditional that checks the condition cond. A return from the function is triggered if the check fails. Before each return from inside a critical region, the critical region is exited first. A line in the non-bold face represents an assignment to an auxiliary variable. These assignments will be explained later.

Creation of Service Objects. The function service_obj_create is used to create a service object. The code pattern of this function is shown in Fig. 6a. This code pattern is the same as in Fig. 2, except for containing two extra assignments to auxiliary variables. In addition, the code pattern for the underlying kernel function kernel_obj_create is given in the upper part of Fig. 6b.

```
service_obj_create(katt, satt):
 local idx, p
 idx <- get_free_obj(obj_arr)
 Check valid(idx)
 obj_arr[idx].ptr <- Dummy
 p <- kernel_obj_create(katt)
 obj_arr[idx].ptr <- p
 Check p != NULL
 obj_arr[idx].att <- satt
 CurTask->_loc <- _Loc_normal
 CurTask->_ptr <- NULL
 return idx
```

(a) creation of service objects

```
kernel_obj_create(katt):
 local p
 // p <- allocate kernel obj.
 // initialize kernel obj.
 CurTask->_loc <- _Loc_cre
 CurTask->_ptr <- p
 return p
kernel_obj_delete(p):
 // delete kernel obj. at p
 CurTask->_loc <- _Loc_normal
 CurTask->_ptr <- NULL
 return
```

(b) kernel obj. creation/deletion

```
service_obj_oper(idx):
 local p, err
 Check valid(idx)
 Check cond(obj_arr[idx].att)
 p <- obj_arr[idx].ptr
 Check p != NULL && p != Dummy
 CurTask->_ptr <- p
 err <- kernel_obj_oper(p)
 CurTask->_ptr <- NULL
 return err
```

(c) use of service objects

```
service_obj_delete(idx):
 local p
 Check valid(idx)
 p <- obj_arr[idx].ptr
 Check p != NULL && p != Dummy
 obj_arr[idx].ptr <- NULL
 CurTask->_loc <- _Loc_del
 CurTask->_ptr <- p
 kernel_obj_delete(p)
 return
```

(d) deletion of service objects

Fig. 6: The patterns for creation/deletion/use of service/kernel objects

Deletion of Service Objects. The function service_obj_delete (Fig. 6d) is used to delete a service object. The deleted service object is the one represented by the array element obj_arr[idx]. Here, idx is the argument of the function. The function first checks to ensure that idx is within the array bound for obj_arr. Then, the function remembers the kernel object pointer obj_arr[idx].ptr in the local variable p. Afterwards, the function checks if the pointer p is neither NULL nor Dummy. If so, then obj_arr[idx] should represent a valid service object. The function then sets obj_arr[idx].ptr to NULL. Finally, the function invokes the kernel function kernel_obj_delete (Fig. 6b) to free the kernel object pointed to by p.

Use of Service Objects. The function service_obj_oper (Fig. 6c) outlines the general pattern for an operation on a service object. First, the validity of the index for the target service object is checked. Then, it is checked whether the attribute value of the service object satisfies the conditions for performing the intended operation. Next, it is checked whether the pointer to the kernel object obj_arr[idx].ptr is valid. If so, the kernel function kernel_obj_oper performing the corresponding operation on the underlying kernel object is invoked.

5.2 Auxiliary Variables with Fractional Permission

We introduce an auxiliary variable, _ptr, for each task. This auxiliary variable reflects the value of the local pointer p at key program locations in the functions of Fig. 6. We employ fractional permission for _ptr. Half of the permission over _ptr is given to the global invariant. Hence, _ptr can be read in the global invariant. Half of the permission over _ptr is retained by the task for which _ptr is introduced. Hence, _ptr can be used to reflect the value of a local pointer.

Via built-in mechanisms of CSL-R, we ensure that whenever a task enters a service function, the value of _ptr is NULL. This captures that the task is not working with a kernel object when entering a service function. When the task running a service function gets hold of a kernel object via p, we set _ptr of the task to the value of p. For service_obj_create, this is at the end of the critical region in the underlying kernel function kernel_obj_create — when the kernel object has just been created. For service_obj_delete and service_obj_oper, this is at the end of their first critical regions. We reset _ptr to NULL when the task loses hold of the kernel object. For service_obj_delete, this is at the end of the critical region in the kernel function kernel_obj_delete — when the kernel object has just been freed. For service_obj_create and service_obj_oper, this is at their end.

We introduce an auxiliary variable, _loc, for each task. This auxiliary variable reflects the current program location of the task. We employ fractional permission for _loc. Half of the permission over _loc is given to the global invariant. Hence, this variable can be read in the global invariant. Half of the permission over _loc is retained by the task for which _loc is introduced. Hence, the program location of each task cannot be modified by a different task.

Via built-in mechanisms of CSL-R, we ensure that whenever a task enters a service function, the value of _loc is _Loc_normal. This reflects that the task is not at a special program location concerning object creation or deletion when entering a service function. When a task running a service function starts to work with a kernel object, we distinguish between the cases for object creation and object deletion, by setting _loc to different values. We set _loc to _Loc_cre for object creation (see Fig. 6b). We set _loc to _Loc_del for object deletion (see Fig. 6d). We reset _loc to _Loc_normal when the task stops working with the underlying kernel object. If the service function executed is service_obj_oper, then _loc remains at the value _Loc_normal through the execution of the function.

5.3 Invariant Conditions Dependent on Auxiliary Variables

Via the auxiliary variables, _loc and _ptr, we are able to formalize Condition 1 and Condition 2. The formulation of these conditions is simpler if the abstract representations of data are used instead of the concrete counterpart. We use *locmp* to represent a function from each task identifier to an optional value of the auxiliary variable _loc for the task. We use *ptrmp* to represent a function from each task identifier to an optional value of the auxiliary variable _ptr for the task. We also introduce the abstract representations of the service objects and the kernel objects. We use *sobjmp* to represent a function that maps each index value i to an optional tuple. The tuple represents the service object obj_arr[idx] if idx

$$
\begin{aligned}
&sobj\_kobj\_aux(locmp, ptrmp, sobjmp, kobjmp, fkobjs) :=\\
&\forall t, a : ptrmp(t) = \mathsf{Some}\ (Vptr\ a) \Rightarrow\\
&\left(\begin{array}{l}
\left(\begin{array}{l} locmp(t) = \mathsf{Some}\ \_\mathsf{Loc\_cre} \wedge \\ \boxed{kobjmp(a) \neq \mathsf{None}}^{①} \wedge \boxed{\neg obj\_ref(sobjmp, a)}^{②} \wedge \boxed{\neg ptr\_in\_fkobj\_pool(a, fkobjs)}^{③} \end{array}\right)\\
\vee \left(\begin{array}{l} locmp(t) = \mathsf{Some}\ \_\mathsf{Loc\_del} \wedge \\ \boxed{kobjmp(a) \neq \mathsf{None}}^{④} \wedge \boxed{\neg obj\_ref(sobjmp, a)}^{⑤} \end{array}\right)\\
\vee \left(\begin{array}{l} locmp(t) = \mathsf{Some}\ \_\mathsf{Loc\_normal} \wedge \\ (\boxed{kobjmp(a) \neq \mathsf{None}}^{⑥} \vee \boxed{ptr\_in\_fkobj\_pool(a, fkobjs)}^{⑦}) \end{array}\right)
\end{array}\right)
\end{aligned}
$$

where $obj\_ref(sobjmp, a) := \exists i, att : sobjmp(i) = \mathsf{Some}\ (KObj\ a, att)$
and $ptr\_in\_fkobj\_pool(a, fkobjs)$ means a is the address of some free kernel object

---

$$
\begin{aligned}
&cre\_del\_mut\_ex(locmp, ptrmp) :=\\
&\quad \forall t_1, t_2, a : (locmp(t_1) \in \{\_\mathsf{Loc\_cre}, \_\mathsf{Loc\_del}\} \wedge ptrmp(t_1) = \mathsf{Some}\ (Vptr\ a)) \Rightarrow\\
&\qquad\qquad (locmp(t_2) \in \{\_\mathsf{Loc\_cre}, \_\mathsf{Loc\_del}\} \wedge ptrmp(t_2) = \mathsf{Some}\ (Vptr\ a)) \Rightarrow\\
&\qquad\qquad t_1 = t_2
\end{aligned}
$$

Fig. 7: The invariant conditions $sobj\_kobj\_aux$ and $cre\_del\_mut\_ex$

has the value i. More concretely, we have $sobjmp(i) = \mathsf{Some}\ (KObj\ a, att)$ if the value of obj_arr[idx].ptr is a, and the value of obj_arr[idx].att is att. Furthermore, we use $kobjmp$ to represent a function that maps the address of each active kernel object to the abstract representation of the kernel object. Hence, the expression $kobjmp(a) \neq \mathsf{None}$ means that there is an active kernel object at the address a.

We devise the condition $sobj\_kobj\_aux(locmp, ptrmp, sobjmp, kobjmp, fkobjs)$ as shown in Fig. 7. We make this condition a part of the global invariant. According to this condition, if a task with the identifier t is working with the kernel object at the address a (i.e., $ptrmp(t) = \mathsf{Some}\ (Vptr\ a)$), then the task could be at a special program location for object creation, at a special program location for object deletion, or not at one of these special program locations. These three cases are reflected by a disjunctive normal form in $sobj\_kobj\_aux$.

***The Use of the Invariant Condition* sobj_kobj_aux.** The invariant condition $sobj\_kobj\_aux$ becomes available to the reasoning task after each critical region is entered. The contents of the parameters $locmp$, $ptrmp$, $sobjmp$, $kobjmp$, and $fkobjs$ correspond to the concrete data they represent. The specific parts ①-⑨ can be exploited depending on the values of the auxiliary variables.

We are able to capture Condition 1 and Condition 2 in Section 2 using $sobj\_kobj\_aux$. If a task t has just completed the assignment p<−kernel_obj_create(katt) in the function service_obj_create, then the task is at a special program location for object creation (i.e., $locmp(t) = \mathsf{Some}\ \_\mathsf{Loc\_cre}$). Hence, Condition 1 in Section 2 is captured by the condition ① in Fig. 7. Furthermore, Condition 2

in Section 2 is captured by the condition ② in Fig. 7. Condition ② is expressed using the predicate *obj_ref*. The definition of this predicate is given below the definition of *sobj_kobj_aux* in the upper part of Fig. 7.

We next explain the use of the condition ④. When a task is in the function kernel_obj_delete (hence at _Loc_del), the task resets the members of the kernel object pointed to by p to their initial values. Condition ④ says that p points to an active kernel object. This helps ensure the safety of the dereferencing operation on p. The condition ⑥∨⑦ serves an analogous purpose. When a task is in the function kernel_obj_oper (hence at _Loc_normal), the task dereferences the pointer p to access the members of the kernel object. The condition ⑥∨⑦ says that p points to a kernel object that is either active or in the pool of the free kernel objects. Thus, the safety of the dereferencing operation is ensured. Here, the disjunction of ⑥ with ⑦ is necessary. This is because before the task enters kernel_obj_oper, the task can be preempted by another task. The latter task could invoke service_obj_delete, obtain the pointer to the kernel object, and free the kernel object in kernel_obj_delete. This deletion does not cause trouble to the execution of kernel_obj_oper — a sensible design of kernel_obj_oper would check whether the kernel object to be used has been freed. This check can be implemented using a data member of kernel objects.

***The Proof Obligations for* sobj_kobj_aux.** Since *sobj_kobj_aux* is specified as a part of the global invariant, a proof obligation in the verification of the service functions is to establish *sobj_kobj_aux* where a critical region is exited. Further invariant conditions are supplied for fulfilling this proof obligation.

Suppose a task with identifier t is about to return to the service function service_obj_create from the kernel function kernel_obj_create. There, we have $locmp(t) = \mathsf{Some}\ \_\mathsf{Loc\_cre}$. In addition, if the local pointer p has the value a, then we have $ptrmp(t) = \mathsf{Some}\ (Vptr\ a)$. Hence, condition ① in *sobj_kobj_aux* requires that there be an active kernel object at the address a. Consider a potential case where the task t is preempted by a different task t', which happens to be entering the function kernel_obj_delete, with the address a as the value for the parameter p. At the point where t' exits from the critical region in kernel_obj_delete, condition ① cannot be established for t. This is because the kernel object at a would have been freed by the task t' — this kernel object is no longer active.

To show that the aforementioned scenario involving the tasks t and t' is impossible, we introduce another condition, *cre_del_mut_ex*, into the global invariant (see bottom part of Fig. 7). The condition says that the actual accesses of the special program locations marked by _Loc_cre and _Loc_del are mutually exclusive, among all the accessing tasks that deal with the same kernel object at some address a. Consider the point where task t' enters the critical region in kernel_obj_delete. The task is then at the program location _Loc_del. If task t is about to return from kernel_obj_create, the task is at the program location _Loc_cre. Hence, the kernel object dealt with by t cannot be the kernel object that is dealt with by t', according to the invariant condition *cre_del_mut_ex*. While task t' is in the critical region of kernel_obj_delete, no other task can execute. Hence, the kernel object dealt with by t cannot be the kernel object dealt with (deleted) by t', when task t' exits the critical region of kernel_obj_delete.

***The Proof Obligations for* cre_del_mut_ex.** Since *cre_del_mut_ex* is specified as a part of the global invariant, a proof obligation in the verification of the service functions is to establish *cre_del_mut_ex* where a critical region is exited.

For instance, when a task t exits from the critical region in service_obj_delete, the task gets to the program location _Loc_del. Hence, it should be ascertained that there is no other task at the program location _Loc_cre, and working with the kernel object pointed to by the local pointer p in service_obj_delete. Consider the point where task t has just completed the assignment p<−obj_arr[idx].ptr in the aforementioned critical region. There, the kernel object O_{knl} pointed to by p is referenced from a service object. From ② in the invariant condition *sobj_kobj_aux*, if a task t' is at the program location _Loc_cre and working with a kernel object O'_{knl}, this O'_{knl} is not referenced from any service object. Hence, O'_{knl} must be different from O_{knl}. Since the other tasks do not execute while the task t is in a critical region, there is still no task at _Loc_cre and working with the kernel object O_{knl}, when the task t exits from the critical region in service_obj_delete. In addition, conditions ③ and ⑤ in the definition of *sobj_kobj_aux* are also used to establish the condition *cre_del_mut_ex* where some of the critical regions are exited. We do not expand on the details.

Summary of Invariant Design. The invariant conditions dependent on auxiliary variables enable the establishment of structural integrity properties about the connection from service objects to kernel objects. This provides a solid foundation for formally verifying the service functions (if they are implemented with the expected code patterns) based on a microkernel that is already verified in CSL-R. We provide the formalized code, formal specifications, and correctness proofs for the functions in Fig. 6 as part of the accompanying artifact.

6 Experimental Evaluation

We apply our methodology in the formal verification of a group of inter-task synchronization and communication services implemented as an extension to the preemptive microkernel μC/OS-II. These services are developed by a separate group of people for safety-critical usage scenarios (e.g., in aerospace vehicles, self-driving cars, etc). The services provide functions for manipulating mutexes, semaphores, and message queues. These service objects extend the corresponding kernel objects of μC/OS-II. For instance, a service-level mutex can be recursive or non-recursive, a service-level semaphore can be binary or counting, and a service-level message queue can be blocking or non-blocking. This fine-grained distinction of object types is not supported by the corresponding kernel objects of μC/OS-II. We discuss some key aspects of our formal verification below.

Application of the Methodology. Almost all the interface functions for the inter-task synchronization and communication services invoke the underlying functions of μC/OS-II to complete operations on kernel objects. This invocation is usually performed outside of critical regions. For instance, the service function could be pthread_mutex_lock for obtaining a service-level mutex, and the corresponding kernel function of μC/OS-II would be OSMutexPend. We are able to

compose the specifications of the service functions from the specifications of the corresponding kernel functions in the extended CSL-R verification framework (see Section 4.1). In addition, the service objects are often initialized with pointers to dedicated structs containing attribute values. Our extension to the CSL-R framework also enables us to express the assumption that each of these pointers points to a well-formed struct of the appropriate type.

Almost all the service functions are implemented following the code patterns in Fig. 6. For each kind of service (for mutexes, semaphores, and message queues), we use the method in Section 5 to establish the structural properties about the connection between service objects and kernel objects. A complication arises because μC/OS-II has a common pool for kernel objects of different kinds. On the other hand, each kind of service object is represented using a different struct, and organized in a separate array. In the verification, we establish that each kind of service object in use references a kernel object *of the same kind*, and each kernel object is referenced by at most one service object *of the same kind*.

Verification Efforts. The source code for the interface functions and the newly implemented internal functions totals 1561 lines. Our proof code for these functions totals approximately 49k lines. The statistics about the lines of source code and the lines of proof code for our verification of the interface functions for the mutex service are given in Table 1. The corresponding statistics for the verification of the other two services are omitted for space reasons. The overall ratio between the verified code and the verification code is about 1:31. This ratio is on par with that in the formal verification of μC/OS-II [28, 27]. Owing to the compositional specification of the service functions, we did not need to redevelop the proofs for the microkernel. Hence, we were able to devote more efforts to establishing the structural properties of the connection between the service level and kernel level, which made the verification of the services possible. It took approximately 3 person years to complete the verification. This included 6 person months for extending the CSL-R framework as well as designing and stabilizing all the invariants that connect the service level and the kernel level.

Table 1: The statistics about the formal proofs for the mutex service

Service Function	Source LOC	Proof LOC	Service Function	Source LOC	Proof LOC
pthread_mutex_init	76	1986	pthread_mutexattr_init	60	1150
pthread_mutex_destroy	33	605	pthread_mutexattr_destroy	21	506
pthread_mutex_lock	99	2514	pthread_mutexattr_gettype	36	654
pthread_mutex_trylock	96	2457	pthread_mutexattr_settype	38	705
pthread_mutex_timedlock	106	2765	pthread_mutexattr_getprioceiling	38	726
pthread_mutex_unlock	97	2563	pthread_mutexattr_setprioceiling	39	732

Problems and Fixes. Through formal verification, we uncovered several problems in the code of the inter-task synchronization and communication services. This code had been extensively tested before our verification started. The most common cause for the uncovered problems is the absence of big enough critical regions that ensure the uninterruptible execution of code. The problem with

the most complicated cause is: If four tasks create and delete service objects concurrently, service objects that are out-of-sync with their corresponding kernel objects can be brought into existence. For instance, a service-level mutex could start to reference a kernel-level message queue, and a binary service-level semaphore could start to reference a kernel-level semaphore with a value of 10. We uncovered part of the problems after realizing that the services could not be shown to preserve some of the conditions in the global invariant — but these conditions captured the required or intended behaviors of the services.

We reported the uncovered problems to the developers of the OS services. They performed three main types of modifications to the code. The first was enlarging a critical region. The second was adjusting the order of operations. The third was introducing dedicated mechanisms to avoid races over global resources. An example modification to the code was the following. The initial implementation of the service function mq_delete invoked the kernel function OSQDel before it set the pointer from a service queue to the underlying kernel queue to NULL. This order was later reversed such that it agreed with the code pattern of service_obj_delete in Fig. 6d. The reason for this reversion was that the original order was found to cause the existence of service objects that are inconsistent with their underlying kernel objects in a highly concurrent setting.

7 Related Work

Our focus is the formal verification of functional correctness for OS services, building on the verification results for an underlying OS kernel. However, our methodology is also applicable if the service functions are implemented inside the kernel. Hence, one type of related work is the formal verification of OS kernels.

In the literature, there are several developments about the formal verification of OS kernels at the implementation level. The seL4 operating system is formally verified in terms of functional correctness and information security [21, 20]. In the Verisoft project, an operating system kernel encompassing assembly code and device drivers is formally verified [5, 4]. CertikOS [18, 17] is a formally verified concurrent OS. It is carefully organized in layers to facilitate verification. The commercial preemptive microkernel μC/OS-II is formally verified in terms of the functional correctness of the API functions [28, 27]. In [11], queue data structures for inter-process communication are verified using the Iris framework [2].

Like our work, the aforementioned developments verify operating system code using a proof assistant such as Isabelle [23] or Coq [1]. Unlike our work, these developments are not focused on the formal verification of code that builds on an OS kernel, by building on prior verification results for the kernel. Our verification is performed for a group of inter-task synchronization and communication services. On the other hand, the verification performed in the aforementioned related developments either has a comprehensive coverage of the functionalities of an OS, or targets a different component than our verification does.

Apart from the aforementioned related work, several developments (e.g. [25, 12, 13, 24, 22, 6, 7, 29]) formally verify operating systems at a more abstract level than we do, or via an approach that is different from ours – such as through

model checking or requiring trust in external solvers (e.g., Z3 [15]). In addition, some of the existing works [20, 14, 30] verify the security properties of operating systems, instead of functional correctness as we verify in the present work.

Our work is about the formal verification of concurrent programs in a broad sense. Notable verification frameworks in this regard include Iris [19] and VST [10]. These frameworks have no builtin support for the type of concurrency in a preemptive OS kernel, where the switch between threads is triggered via interrupt handling. Our use of the auxiliary variables with fractional permission helps express a protocol followed by the concurrent tasks that manipulate the service objects. In the literature, there exist techniques with dedicated abstractions for expressing the protocols followed by concurrent threads. An example abstraction is a state transition system [26]. In the present work, our focus is to achieve the required verification by maximally exploiting the features of the verification framework for the underlying microkernel. Hence, we have not introduced further abstractions for the expression of protocols. Due to space limits, we stop here in our discussion about related work in concurrent program verification.

8 Conclusion

We address the problems in formally verifying a group of OS services that build on a preemptive microkernel, in case the microkernel itself has been formally verified. Specifically, the verification of the microkernel has been a refinement verification performed using a concurrent separation logic that supports fractional permission. Our aim is to build sufficiently on the verification framework and verification code for the microkernel, in verifying the code of the services. Our methodology consists of enhancements to the verification framework that enable the compositional specification of the service functions, as well as a design of invariants for establishing structural integrity properties about the connection between the service level and the kernel level. We use the methodology to accomplish a substantial verification task targeting a group of inter-task synchronization and communication services based on the preemptive microkernel μC/OS-II. The verification uncovers dormant bugs and provides a level of correctness assurance that is above what can be achieved through extensive testing.

A potential direction for future work is the design of deductive systems that facilitate the verification of global properties for a service, based on the abstract programs of all the interface functions of a service. Another direction for future work is the verification of progress properties for the functions of a service.

Data-Availability Statements. The mechanized extension to the CSL-R verification framework and proofs for the OS service in abstract form (as described in Section 4 and Section 5) are published at Zenodo (`10.5281/zenodo.10456998`).

Acknowledgments. This work was partially supported by the National Natural Science Foundation of China (62002246, 62272322, 62272323, 62372311, 62372312). We thank Xinyu Feng for help with the CSL-R verification framework. We thank Qinxiang Cao and Bohua Zhan for advices on some of the technical ingredients facilitating the completion of our work. We thank the anonymous reviewers for providing valuable feedback that helped improve our presentation.

References

1. The Coq proof assistant. `https://coq.inria.fr/`. Accessed: 2023-10-08.
2. Iris – a higher-order concurrent separation logic framework, implemented and verified in the Coq proof assistant. `https://iris-project.org/`. Accessed: 2023-10-12.
3. μC/OS-II. `https://www.osrtos.com/rtos/uc-os-ii/`. Accessed: 2023-10-08.
4. Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert Schirmer, and Artem Starostin. The Verisoft approach to systems verification. In *Proceedings of Second International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, pages 209–224, 2008.
5. Eyad Alkassar, Wolfgang J. Paul, Artem Starostin, and Alexandra Tsyban. Pervasive verification of an OS microkernel - inline assembly, memory consumption, concurrent devices. In *Proceedings of Third International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, pages 71–85, 2010.
6. June Andronick, Corey Lewis, and Carroll Morgan. Controlled Owicki-Gries concurrency: Reasoning about the preemptible eChronos embedded operating system. In *Proceedings of Workshop on Models for Formal Analysis of Real Systems, (MARS)*, pages 10–24, 2015.
7. Bernhard Beckert and Michal Moskal. Deductive verification of system software in the Verisoft XT project. *Künstliche Intell.*, 24(1):57–61, 2010.
8. John Boyland. Checking interference with fractional permissions. In *Proceedings of 10th International Symposium on Static Analysis (SAS)*, pages 55–72, 2003.
9. Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, 2016.
10. Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning*, 61(1-4):367–422, 2018.
11. Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W. O'Hearn, and Francesco Zappa Nardelli. Applying formal verification to microkernel IPC at Meta. In *Proceedings of 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 116–129, 2022.
12. Shu Cheng, Jim Woodcock, and Deepak D'Souza. Using formal reasoning on a model of tasks for FreeRTOS. *Formal Aspects of Computing*, 27(1):167–192, 2015.
13. Nathan Chong and Bart Jacobs. Formally verifying FreeRTOS' interprocess communication mechanism. In *Embedded World Exhibition & Conference*, 2021.
14. David Costanzo, Zhong Shao, and Ronghui Gu. End-to-end verification of information-flow security for C and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 648–664, 2016.
15. Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *Proceedings of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Held as Part of ETAPS*, pages 337–340, 2008.
16. Mike Gordon and Hélène Collavizza. Forward with Hoare. In A. W. Roscoe, Clifford B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C. A. R. Hoare*, pages 101–121. Springer, 2010.
17. Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. Building certified concurrent OS kernels. *Communications of the ACM*, 62(10):89–99, 2019.

18. Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 653–669, 2016.
19. Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.
20. Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, 2014.
21. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proceedings of 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, 2009.
22. Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 252–269, 2017.
23. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
24. Evgeny Novikov and Ilja S. Zakharov. Verification of operating system monolithic kernels without extensions. In *Proceedings of 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), Part IV*, pages 230–248, 2018.
25. Leandro Batista Ribeiro, Florian Lorber, Ulrik Nyman, Kim Guldstrand Larsen, and Marcel Baunach. A modeling concept for formal verification of OS-based compositional software. In *Proceedings of 26th International Conference on Fundamental Approaches to Software Engineering (FASE), Held as Part of ETAPS*, pages 26–46, 2023.
26. Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 343–356, 2013.
27. Fengwei Xu. *Design and Implementation of A Verification Framework for Preemptive OS Kernels*. PhD thesis, University of Science and Technology of China, 2016.
28. Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A practical verification framework for preemptive OS kernels. In *Proceedings of 28th International Conference on Computer Aided Verification (CAV)*, pages 59–79, 2016.
29. Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 99–110, 2010.
30. Yongwang Zhao, David Sanán, Fuyuan Zhang, and Yang Liu. Refinement-based specification and security analysis of separation kernels. *IEEE Transactions on Dependable and Secure Computing*, 16(1):127–141, 2019.