A Practical Verification Framework for Preemptive OS Kernels (Extended Version)

Abstract. We propose a practical verification framework for preemptive OS kernels. The framework models the correctness of API implementations in OS kernels as contextual refinement of their abstract specifications. It provides a specification language for defining the high-level abstract model of OS kernels, a program logic for refinement verification of concurrent kernel code with multi-level hardware interrupts, and automated tactics for developing mechanized proofs. The whole framework is developed for a practical subset of the C language. We have successfully applied it to verify key modules of a commercial preemptive OS μC/OS-II [20], including the scheduler, interrupt handlers, message queues, and mutexes *etc.* We also verify the priority-inversion-freedom (PIF) in μC/OS-II. All the proofs are mechanized in Coq. To our knowledge, our work is the first to verify the functional correctness of a practical *preemptive* OS kernel with mechanized machine checkable proofs.

1 Introduction

Operating system kernel is one of the most fundamental layers of software systems. Its correctness is crucial for the safety and security of the whole system. Verifying OS kernels has long been recognized as an important but also extremely challenging task. There have been exciting efforts for OS kernel verification in recent years, however, most of them are focused on sequential kernels, which disallow the kernel-level tasks to be preempted. This limitation restricts their applicability to real-time systems, where preemptive tasks are indispensable to achieve real-time guarantees.

Preemptive kernels require explicit invocation of schedulers inside interrupt handlers and careful interrupt management in the kernel code, which makes the kernel highly concurrent and complex. In this paper we propose a verification framework for preemptive OS kernels, and show its application in verifying key modules of μC/OS-II, a commercial preemptive real-time multitasking kernel for microprocessors and microcontrollers. To our knowledge, it is the first mechanized verification of the functional correctness of a preemptive OS kernel. Our work is based on existing theories on interrupt verification [11] and refinement verification of concurrent programs [23, 22, 24, 30, 31, 16], but makes the following new contributions:

First, we formulate and verify the correctness of the APIs of OS kernels as contextual refinement between their implementations and specifications. Although refinement verification has been used in earlier work on OS kernel

verification, it is used for non-preemptive (*i.e.*, sequential) kernels only. On the one hand, refinement of concurrent programs is much more challenging to verify than sequential ones, which has not been fully studied until recently [23, 22, 24, 30, 31, 16]. On the other hand, contextual refinement is more necessary in a concurrent setting because it enforces much stronger functional correctness than traditional Hoare-style specifications based on pre-/post-conditions, as we explain in Sec. 4.

Second, we provide a simple modeling language for specifying kernel primitives. The language strives to achieve balance between abstraction and expressiveness for scheduling. On the one hand we want the specification to abstract away implementation details. On the other hand it should provide enough details so that many important properties can be specified at the abstract specification level (*i.e.*, the model level). Our modeling language provides an abstract **sched** command, allowing us to specify explicitly when the scheduler is invoked in synchronization primitives or interrupt handlers. Semantics of **sched** is parameterized over scheduling policies (*e.g.*, priority-based or round-robin). Expressiveness about these details are necessary to specify system-wide scheduling properties.

Third, we propose a program logic for refinement verification of concurrent kernel programs. The logic supports multi-level nested hardware interrupts and configurable schedulers. It extends concurrent separation logic [26] (CSL) with relational assertions that relate program states at the implementation and the specification levels, as in [22, 24]. It also assigns ownership-transfer semantics to interrupt management operations and verify multi-level hardware interrupts in a realistic setting.

Fourth, our framework is developed for a practical subset of C. It has been successfully applied to verify key APIs of μC/OS-II [2], including the timer interrupt (and a pseudo interrupt to demonstrate the support of multi-level interrupts), the scheduler, the time management, and four synchronization mechanisms: message queues, mail boxes, semaphores, and mutexes. It is worth noting that, unlike existing works [4, 10, 19, 15] which are all focused on systems newly developed with verification in mind, we take a *commercial system developed by an independent third-party* and verify the code with minimum modification, which demonstrates the generality and applicability of our framework.

Fifth, in addition to the functional correctness of APIs, we also verify priority inversion freedom (PIF) of μC/OS-II. PIF is a crucial property for real-time systems and is worth verifying in its own right, but the verification has another two important purposes: (1) To show that contextual refinement seamlessly carries properties established at the abstract levels down to the concrete implementation level. As a system-wide property, we specify and verify PIF as a meta-property of the abstract model of the kernel. Then the contextual refinement ensures it indeed holds at the concrete implementation level. (2) To validate our model of system APIs. Specifying properties at the high abstract level is a useful approach to validate the model. As we explain above, many important properties cannot be specified if the model is too weak or overly abstract.

We have mechanized all the proofs in Coq [1]. The code is submitted as supplementary materials and is also available at the anonymous site [3].

2 Background and Overview of Our Work

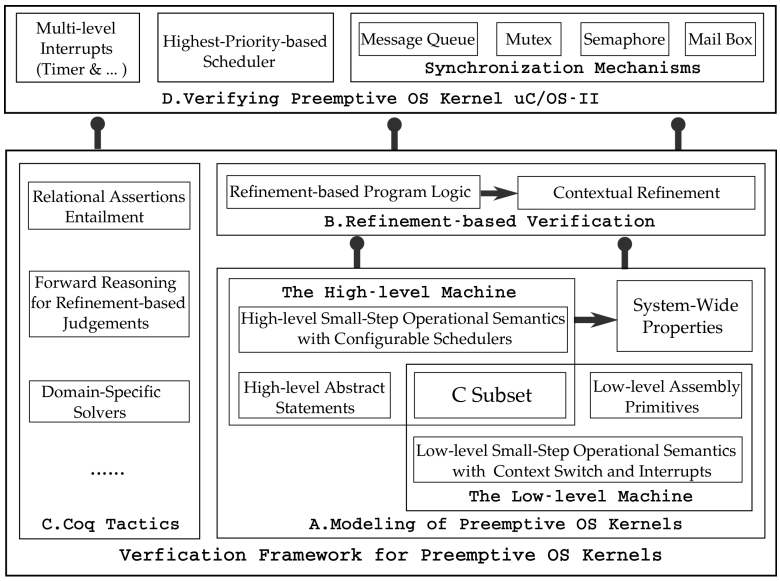


Fig. 1. Overview of Our Verification Framework

We first introduce some background knowledge about the implementation of preemptive OS kernels and the interrupt mechanisms. Then we give an overview of our framework consisting of components A, B and C, as shown in Fig.1.

2.1 Preemptive OS Kernels and Interrupts

In a preemptive OS, execution of a task can be interrupted at any program point (unless interrupts have been disabled) and the control flow can be switched to a different task. To allow preemption, we need these two conditions: (1) enabling interrupts at the kernel level; and (2) invoking the scheduler and context switching inside the interrupt handler.

As shown in Fig. 2, execution of Task A is interrupted and the control is switched to the interrupt handler (step (1)). Instead of returning to Task A directly when the interrupt request has been handled, we may execute the context switch routine (step (4)), which switches the control to another Task B. If we abstract away the interrupt handler, we say the execution of Task A is preempted by Task B in this case. Note that if we disallow one of the above conditions, the

only way to switch the control from Task A to task B is to let Task A volunteer to execute `switch` in its code (such as step (11)), resulting in a non-preemptive concurrency model.

Interaction between tasks and hardware interrupts. As we can see, interrupt handling and management are indispensable in preemptive OS kernels. Below we also give a simplified overview of the interrupt mechanism in x86 systems (based on the Intel 8259A interrupt controller).

The CPU has a flag bit `IF` indicating whether interrupts are enabled or not. The **cli**/**sti** instruction clears/sets the bit to disable/enable interrupts. In 8259A there is a register `isr`, each bit of which corresponds to a hardware interrupt and records if the interrupt is being served or not. Different priority levels are assigned to different sources of interrupts, with level-0 being the highest. When an interrupt request comes, we check `IF` and `isr`. If the interrupts are enabled and there is currently no interrupt of higher or the same priority being served, the request will be served. The corresponding bit in `isr` is set to 1 and the control jumps to the corresponding interrupt handler, shown as step (1) in Fig. 2.

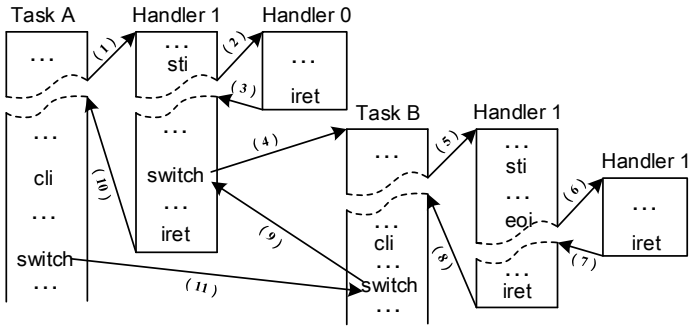


Fig. 2. Tasks and Multi-level Interrupts

On the invocation of the interrupt handler, the CPU flags (including the `IF` bit) are saved on the stack, and interrupts are disabled automatically. If the programmer enables interrupts again inside the handler, the interrupt hander could be further interrupted by interrupt requests with higher priorities (see step (2), where the level-0 interrupt has a higher priority than level-1 interrupt), causing nested interrupts.

The interrupt handler returns to the program being interrupted using **iret** (step (3)), which also recovers the flags (including the `IF` bit). Before the handler returns, we need to execute **eoi** . The command sends an "end of interrupt" signal to the interrupt controller, which clears the corresponding bit in `isr`. After **eoi** , if the interrupt is enabled (`IF` = 1), the interrupt handler could be further interrupted by interrupts at a lower or the same level (see step (6)).

2.2 Correctness of OS Kernels

An OS kernel provides an abstract programming model, which hides the details of the underlying hardware and simplifies the development of the high-level user applications. The correctness of the kernel requires that behaviors of user applications in the concrete machine model be consistent with their expected behaviors in the abstract model [17]. Thus the OS verification can be reduced to verifying refinement between the two programming models.

Correctness. We consider three entities, the application A, the abstract specification of the system routines and interrupt handlers $\mathbb{O}$, and their concrete implementations O. A is implemented in the C language. When system calls are made or interrupts are handled, routines in $\mathbb{O}$ are invoked at the abstract level, and those in O are invoked at the concrete level. Then the correctness of OS kernels requires O refines $\mathbb{O}$ under any context A:

$$\forall A. [\![A[O]]\!] \subseteq [\![A[\mathbb{O}]]\!]$$

where $[\![A[O]]\!]$ and $[\![A[\mathbb{O}]]\!]$ represent the sets of observable behaviors at the low level and the high level respectively.

Note that the correctness of OS kernels could be independent of the implementation language of applications A. Here we pick C as the implementation language mainly to simplify the verification, because the application and the kernel are now developed in the same language.

The high-level programming model. Viewing from the aspect of application programmers, we model the OS kernel as an extended C language with multi-tasking and system calls. As explained above, the C language is used to implement user applications A, and the system calls invoke an abstract version of system routines in $\mathbb{O}$, which are implemented using a simple specification language.

Programmers at this level have no control over interrupts (*e.g.*, enabling or disabling interrupts). Interrupts are modeled implicitly as abstract external events that may occur non-deterministically at any program points. Handlers of the events are also specified as part of $\mathbb{O}$.

Instead of following the interleaving semantics for multi-tasking, our high-level language semantics is parameterized over an abstract scheduler, which can be instantiated to model different scheduling strategies. Our specification language for $\mathbb{O}$ provides an abstract **sched** command, which can invoke the scheduler explicitly. Exposing these details at the high level allows us to specify and verify scheduling properties, such as PIF.

The low-level programming model. The low-level language is used to implement the kernel O. Most OS kernels are implemented in C and inline assembly. However, giving semantics directly to C with inline assembly requires us to expose stacks and registers, which make the semantics overly complex. To avoid this problem, we introduce built-in primitives to encapsulate the assembly code.

These primitives are executed atomically over abstract machine states at the low level.

Since we model the context switching routine (implemented in assembly) as a built-in primitive, the low-level language also has built-in multi-tasking, as in the high level. It also has primitives for explicitly managing interrupts (*e.g.*, enabling/disabling interrupts, or sending eoi signals).

The consistency between the original assembly code and assembly primitives are guaranteed by a certified C compiler, which compiles both the C code and assembly primitives to the target assembly code. The certified compilation is still ongoing work and is out of the scope of this paper.

Small-step semantics. To reflect the fact that execution of programs can be interrupted between any two *machine* instructions, we give a very small-step semantics to the C languages at both the high level and the low level, where the execution of expressions is also decomposed into small steps, following the semantics of ClightTSO [27].

Overview of μC/OS-II. μC/OS-II is a commercial preemptive real-time multi-tasking OS kernel developed by Micrium [2]. The kernel has 6000+ lines of C code and 300+ lines of assembly. It allows a fixed number of tasks, multi-level interrupts, and preemptive priority-based scheduling. The system APIs include "semaphores; event flags; mutual-exclusion semaphores that eliminate unbounded priority inversions; mailboxes; message queues; task, time and timer management; and fixed sized memory block management". μC/OS-II is developed for microprocessors and microcontrollers, and it does not support virtual memory. It has been deployed in many real-world safety critical applications, including Avionics (*e.g.*, the Mars Curiosity Rover) and medical equipments.

2.3 Refinement Verification and Coq Tactics

Compositional refinement verification has been a challenging problem until recently various techniques are proposed to address this problem. We propose a CSL-style *relational* program logic for refinement verification, which applies existing theories [23, 22, 24] but makes the following simplification and extensions:

- We drop the mechanisms (*e.g.*, speculation and helping) for fine-grained concurrency. Observing that in kernels most accesses of shared data are inside critical regions, we replace the rely/guarantee-style reasoning with the CSL-style, which specifies the well-formedness of shared data using invariants and assigns ownership-transfer semantics to synchronization primitives.
- We extend the refinement verification techniques with the support of multi-level interrupts and the explicit interrupt management. Following the idea of existing work [11], we give ownership-transfer-based axiomatic semantics to interrupt management primitives, including **cli**/**sti**, **eoi** and **iext**. Our interrupt model is also made more practical with the support of multi-level interrupts.

– We also adapt the refinement logic to support the low-level concurrency model, where thread scheduling and context switching are invoked explicitly in the task code. To verify the low-level implementation of the scheduler refines its high-level specification, we require in our logic that, whenever the low level makes context switching, the high-level scheduler must pick the same target task.

At the meta-level of our program logic, we have a compositional simulation relation, which is similar to RGSim [23] but is parameterized over CSL-style invariants instead of rely/guarantee conditions to describe interference. Similar to the program logic, the simulation is also extended to support multi-level interrupts, schedulers and context switching.

In practice, one has to pay great efforts for OS kernel verification. Proof automation is essential to improve the productivity. There have been a number of Coq tactics [8, 25, 5] implemented for semi-automated verification of C programs with separation logic. These tactics, however, cannot be readily applied in our relational program logic. Our framework provides a new set of Coq tactics developed based on existing techniques for refinement verification.

We develop tactics for automatically proving relational separation logic assertions and generating verification conditions. They are able to do forward reasoning for statements, such as function calls and the statements for entering and exiting critical regions, *etc.*. Also some domain-specific tactics are implemented for the rich data structures used in kernels. In our target kernel μC/OS-II, the scheduler uses bitmaps to quickly compute the target task to switch to. We develop some tactics for automatically proving the arithmetic properties of *Int32* and bitmaps.

3 Modeling of the Kernel

As explained above, the correctness of OS kernels is formalized based on three entities — user applications A, the concrete implementation O, and the abstract specification $\mathbb{O}$. In this section we introduce the programming (model) languages for the three entities.

3.1 The Low-Level Language

The low-level language consists of two parts for implementations of user applications and OS kernels, respectively.

Application language. The application language is shown in Fig. 3. It is a subset of the C language consisting of function calls, pointer operations (except pointer arithmetics), arrays, structs, bit operations, *etc.* The application code A maps function names to their function bodies. The function definition *cfd* for client consists of the type of the return value, the declaration of parameters, the declaration of local variables and the statements of the function body.

The command $f(\bar{e})$ calls the function f, which could be either an application function in A or an OS API (in O at the low-level or in $\mathbb{O}$ at the high-level, as we explain below).

$$
\begin{array}{llll}
(\mathit{Addr}) & a \in \mathit{Int32} & (\mathit{Ident}) & id \in \mathbb{Z} \\
(\mathit{FName}) & f \in \mathbb{Z} & (\mathit{Integer}) & k \in \mathit{Int32} \\
(\mathit{Type}) & \tau ::= \mathsf{Tvoid} \mid \mathsf{Tint32} \mid \mathsf{Tptr}(\tau) \mid \mathsf{Tarray}(\tau, n) \mid \mathsf{Tstruct}(id, \mathcal{D}) \mid \ldots & & \\
(\mathit{OpVal}) & \hat{v} ::= \bot \mid v & (\mathit{Value}) & v ::= \mathsf{Vundef} \mid \mathsf{Vnull} \mid \mathsf{Vint}(k) \mid \mathsf{Vptr}(a) \\
(\mathit{ValList}) & \bar{v} ::= \mathsf{nil} \mid v :: \bar{v} & (\mathit{ExprList}) & \bar{e} ::= \mathsf{nil} \mid e :: \bar{e} \\
(\mathit{TypeList}) & \mathcal{T} ::= \mathsf{nil} \mid \tau :: \mathcal{T} & (\mathit{DeclList}) & \mathcal{D} ::= \mathsf{nil} \mid (id, \tau) :: \mathcal{D} \\
(\mathit{UOP}) & \mathsf{uop} ::= \sim \mid ! \mid \ldots & (\mathit{BOP}) & \mathsf{bop} ::= + \mid - \mid \gg \mid \ll \mid \& \mid \ldots \\
(\mathit{CExpr}) & e ::= x \mid k \mid *e \mid \&e \mid e.id \mid e[e] \mid (\tau)e \mid \mathsf{uop}\ e \mid e\ \mathsf{bop}\ e & & \\
(\mathit{CltStmts}) & d ::= e = e \mid f(\bar{e}) \mid d; d \mid \textbf{if}\ (e)\ d\ \textbf{else}\ d \mid \textbf{while}\ (e)\ d \mid \textbf{return}\ e \mid \textbf{print}\ e \mid \ldots & & \\
(\mathit{CltFDef}) & \mathit{cfd} ::= (\tau, \mathcal{D}_1, \mathcal{D}_2, d) & (\mathit{CltCode}) & A ::= \{f_1 \rightsquigarrow \mathit{cfd}_1, \ldots, f_n \rightsquigarrow \mathit{cfd}_n\}
\end{array}
$$

Fig. 3. The Language for Applications

We use k for 32-bit integers and a for memory addresses (pointers). A value v is either undefined, null, a 32-bit word value or a pointer. We have rich C data types, including the type for the void type (Tvoid), the type of "int"($\mathsf{Tint32}$), the type of "pointers" ($\mathsf{Tptr}(\tau)$), the type of "array" ($\mathsf{Tarray}(\tau, \mathsf{k})$) and the type of "struct" ($\mathsf{Tstruct}(id, \mathcal{D})$),*etc.*. An expression e follows the syntax of the C programming language. It is either a constant integer, a program variable, memory reference, deference or standard arithmetic or logical operations over expressions. A statement d is either an assignment statement, function call statements, a sequence of statements, a branch statement, a loop statement, a return statement or a output statement **print**. The language is reasonably practical because it has been used to implement an executable operating system kernel μC/OS-II. The output command **print** e generates observable event, which is used to define observable event traces needed in our definition of refinement.

Note that the correctness of OS kernels are independent of the implementation language of A. Here we pick the C language for A to simplify the formalization because the applications and the kernel are now implemented in the same language and we do not have to consider the interaction between different languages when defining the whole system ($A[O]$) behaviors.

Low-level language for OS kernels. Figure 4 shows the low-level language for the concrete implementation of OS kernels. Usually the kernels are implemented in C with inline assembly. However, giving semantics directly to C with inline assembly requires us to expose stacks and registers, which makes the semantics overly complex. To avoid this problem, we extend the C statements with assembly primitives ι to encapsulate the assembly code. Semantics of these primitives will be given below.

switch x switches to the target task x. **encrt** enters a critical region by disabling interrupts. It also saves the old `IF` onto the stack to allow nested critical regions. Note we use *ie* to model the `IF` flag and abstract away other

$$
\begin{array}{lll}
(\mathit{LPrim}) & \iota ::= \mathbf{switch}\ x \mid \mathbf{encrt} \mid \mathbf{excrt} \mid \mathbf{eoi}\ k \mid \mathbf{iext} \mid \ldots & \\
(\mathit{LStmts}) & s ::= d \mid \iota \mid s; s \mid \mathbf{while}\ (e)\ s \mid \mathbf{if}\ (e)\ s\ \mathbf{else}\ s & \\
(\mathit{ItrpCode}) & \theta ::= [s_0, \ldots, s_{N-1}] & (\mathit{LFunDef})\ \mathit{ofd} ::= (\tau, \mathcal{D}_1, \mathcal{D}_2, s) \\
(\mathit{ProgUnit}) & \eta ::= \{f_1 \rightsquigarrow \mathit{ofd}_1, \ldots, f_n \rightsquigarrow \mathit{ofd}_n\} & \\
(\mathit{LOSCode}) & O ::= (\eta_a, \eta_i, \theta) & (\mathit{LProg}) \quad P ::= (A, O)
\end{array}
$$

Fig. 4. The Languages for Kernel Impl.

$$
\begin{array}{llll}
(\mathit{Memory}) & M \in \mathit{Addr} \rightharpoonup \mathit{Vaule} & (\mathit{SymTable}) & G, E \in \mathit{Var} \rightharpoonup \mathit{Addr} \\
(\mathit{CState}) & \Delta ::= (G, \Pi, M) & (\mathit{SymTblSet}) & \Pi ::= \{t_1 \rightsquigarrow E_1, \ldots, t_n \rightsquigarrow E_n\} \\
(\mathit{Cont}) & K ::= (\kappa_e, \kappa_s) & (\mathit{ExprCont}) & \kappa_e ::= \circ \mid \ldots \\
(\mathit{StmtCont}) & \kappa_s ::= \bullet \mid s \cdot \kappa_s \mid \mathbb{s} \cdot \kappa_s \mid (c, \kappa_e, E) \cdot \kappa_s \mid (f, \bar{v}, \bar{e}) \cdot \kappa_s \mid (f\ s\ E) \cdot \kappa_s \mid \ldots & & \\
(\mathit{CurEval}) & c ::= e \mid s \mid \mathbb{s} \mid \boldsymbol{fexec}(f, \bar{v}) \mid \boldsymbol{alloc}(\bar{v}, \mathcal{D}) \mid \mathbf{skip} \mid v \mid \ldots & & \\
(\mathit{TaskId}) & t \in \mathit{Addr} & (\mathit{TaskCode}) & C ::= (c, K) \\
(\mathit{CMem}) & m ::= (G, E, M) & (\mathit{TaskPool}) & T ::= \{t_1 \rightsquigarrow C_1, \ldots, t_n \rightsquigarrow C_n\}
\end{array}
$$

Fig. 5. The Common States

bits in the hardware EFLAGS register. **excrt** exits the current critical region by popping the stack to recover *ie*. Since we hide stacks in our state model, we use an abstract stack *cs* to save the historical *ie* bits (see Fig. 6, which is explained below). **eoi** k clears the k-th bit in *isr*, indicating that the k-th interrupt is no longer in service. **iext** enables interrupts and returns to the interrupted program.

The kernel implementation O consists of the system API implementation η_a, the internal functions η_i and the interrupt handlers θ. The internal functions are called only by code in η_a or θ. θ is a sequence of N interrupt handlers, where N is the maximum number of interrupts we support. The handler with the lower identifier has the higher priority. Then a complete low-level program P is defined as a pair of the application code A and the kernel code O.

Common Machine States. As shown in Fig. 5, we present the common machine states for the two levels. The memory M is modeled as a partial function from addresses to values. The global symbol table G and the local symbol table E map program variables to addresses. Note that we use a flat memory model to simplify the presentation. The basic memory operations follow the block-based memory model in CompCert [21]. Π maps the task identifiers to their local symbol tables. Δ consists of the global symbol table G, the set of local symbol tables Π and the memory M.

We give small-step operational semantics at the two levels. For each step, the processor picks the continuation of the current task and executes its current command or expression. To model fine-grained concurrency, both commands and *expressions* could be executed in multiple steps, where each step corresponds to the granularity of a single machine instruction (as in CompCertTSO [27], but we use the sequential consistent model instead of the x86-TSO memory model).

$$
\begin{array}{llll}
(\mathit{BitVal}) & b, ie \in \{0,1\} & (\mathit{ISRReg}) & isr ::= [b_0, \ldots, b_{N-1}] \\
(\mathit{CrtStk}) & cs ::= \mathsf{nil} \mid ie :: cs & (\mathit{ItrpStk}) & is ::= \mathsf{nil} \mid k :: is \\
(\mathit{ItrpTaskSt}) & \delta ::= (ie, is, cs) & (\mathit{ItrpSt}) & \pi ::= \{t_1 \rightsquigarrow \delta_1, \ldots, t_n \rightsquigarrow \delta_n\} \\
(\mathit{LOsFullSt}) & \Lambda ::= (\Delta, isr, \pi) & (\mathit{TaskLocalSt}) & \sigma ::= (m, isr, \delta) \\
(\mathit{LWorld}) & W ::= (P, T, \Delta, \Lambda, t) & &
\end{array}
$$

Fig. 6. The Low-level Machine States

The definition of expression and statement continuations are shown at Fig. 5. For the expression continuations κ_e, they can be $\circ$ which means that there is nothing left to be evaluated, or some other standard cases. For the statement continuations κ_s, $\bullet$ means that there is nothing left to be done. When the current task is interrupted, we use $(c, \kappa_e, E) \cdot \kappa_s$ to save the execution context for the current task. Since the interrupt may happen when evaluating an expression, we need to record the current evaluation c, the expression continuation κ_e and the current local symbol table E for resuming the execution context in the future. We use $(f, \bar{v}, \bar{e}) \cdot \kappa_s$ to save the intermediate results of calculating the function arguments. $(f, s, E) \cdot \kappa_s$ is used to save the context when doing functions calls.

We also introduce some runtime statements. For instance, $\boldsymbol{fexec}(f, \bar{v})$ is an intermediate statement for calling a function. $\boldsymbol{alloc}(\bar{v}, \mathcal{D})$ is used to do memory allocations for local variables and parameters of function. More details about the usage of these statements can be seen in Fig 8, which gives some key operational semantics rules for the low-level language.

Low-level Machine States. The language is concurrent, with multiple continuations (*i.e.*, control stacks) in the state, each corresponding to a task. All tasks share memory, but each has its own local variables and local interrupt states (see δ in Fig. 6, which is explained below). We also separate the program state (including memory and variables) into two disjoint parts, one for the application code A and the other for the kernel code O. The only way for A to access kernel states is to call system APIs in O, and O cannot access application states.

As explained in Sec. 2.1, *ie* is a boolean flag used to turn on/off interrupts. *isr* is a sequence of boolean flags, one for each interrupt. The stack *cs* records the historical values of *ie*, which are pushed whenever the execution enters a critical region. It is used to support nested critical regions. The task-local stack *is* records the sequence of interrupts that interrupt the execution of *this task*. It is auxiliary data introduced for verification purpose. Then the task-local interrupt status δ is defined as a triple (ie, is, cs). π records the δ of each task. The kernel-level state Λ consists of the general C state Δ, the global *isr* register and the set π of task-local interrupt status.

The whole program configuration W now consists of the task pool T, the client state Δ, the kernel state Λ, and the identifier t of the current task. Note that W contains two pieces of Δ, one for user applications (clients), the other inside Λ for the kernel. Separating the data into two parts prevents user applica-

tions from accessing kernel data. Applications trying to access data unavailable in the client Δ will trigger a runtime error in our operational semantics.

$$\lfloor \kappa_s \rfloor_c \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } \kappa_s = \bullet \\ \kappa_s & \text{if } \kappa_s = (f, s, E) \cdot \kappa'_s \\ \lfloor \kappa'_s \rfloor_c & \text{otherwise} \end{cases} \quad \lfloor \kappa_s \rfloor \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } \kappa_s = \bullet \\ \kappa_s & \text{if } \kappa_s = (c, \kappa_e, E) \cdot \kappa'_s \\ \lfloor \kappa'_s \rfloor & \text{otherwise} \end{cases}$$

$$f \perp g \stackrel{\text{def}}{=} dom(f) \cap dom(g) = \emptyset \qquad f \uplus g \stackrel{\text{def}}{=} \begin{cases} f \cup g \text{ iff } f \perp g \\ undef \quad \text{otherwise} \end{cases}$$

$$\mathsf{InOS}(C, (A, O)) \stackrel{\text{def}}{=} \exists c, \kappa_s. C = (c, (\_, \kappa_s)) \wedge ((\exists f. f \in dom(O.\eta_a \uplus O.\eta_i) \wedge (c = \textbf{\textit{fexec}}(f, \_) \vee \lfloor \kappa_s \rfloor_c = (f, \_, \_) \cdot \_)) \vee \lfloor \kappa_s \rfloor \neq \bot)$$

$$(G, \Pi, M)|_t = (G, E, M) \stackrel{\text{def}}{=} \Pi(t) = E$$

$$(\Delta, isr, \pi)|_t = (m, isr, \delta) \stackrel{\text{def}}{=} \Delta|_t = m \wedge \pi(t) = \delta$$

$$\Lambda' = \textsc{upd}\mathrm{TS}(\Lambda, t, \sigma') \stackrel{\text{def}}{=} \Lambda'|_t = \sigma' \wedge \forall t' \neq t. \Lambda'|_{t'} = \Lambda|_{t'}$$

$$\Delta' = \textsc{upd}\mathrm{CS}(\Delta, t, m') \stackrel{\text{def}}{=} \Delta'|_t = m' \wedge \forall t' \neq t. \Delta'|_{t'} = \Delta|_{t'}$$

$$[\![x]\!]_{(G,E,M)} = t' \stackrel{\text{def}}{=} \exists a. (E(x) = a \wedge M(a) = t') \vee (G(x) = a \wedge x \notin dom(E) \wedge M(a) = t')$$

$$[\![x]\!]_{(t,((G,\Pi,M),isr,\pi))} = t' \stackrel{\text{def}}{=} [\![x]\!]_{(G,\Pi(t),M)} = t'$$

$$((G, \Pi, M'), isr, \pi) = \textsc{upd}\mathrm{G}(((G, \Pi, M), isr, \pi), \mathsf{OSTCBCur}, t') \stackrel{\text{def}}{=} \exists a. G(\mathsf{OSTCBCur}) = a \wedge M' = M\{a \rightsquigarrow t'\}$$

Fig. 7. Auxiliary Definitions

Low-level Operational Semantics. We give the low-level operational semantics in Fig. 8. For the low-level program steps, denoted as "$P \vdash W \Longrightarrow_L W'$", may execute a regular command in a task (the PTASK rule), or execute **switch** x to do context switch (the PSW rule), or be interrupted and transfer the control to the corresponding interrupt handler (the PITRP rule). When executing a regular command in a task, it may either belong to the kernel (the TKERNEL rule) or the client (the TCLT rule).

The PTASK rule is used to lift the task-local step of the current task t to program step. We project the local data σ of t from Λ (σ defined in Fig. 5), execute a task-local step, and then update the program configuration accordingly. Here $\textsc{upd}\mathrm{TS}(\Lambda, t, \sigma')$ (defined in Fig.7) updates the local data of t in Λ with the new σ'.

The assembly implementation of the context switch routine is abstracted into the primitive **switch** x. It switches the execution from the current task to the target task x, where x stores the task identifier. The PSW rule simply resets the current thread identifier and updates the global variable $\mathsf{OSTCBCur}$ accordingly.

$\boxed{P \vdash W \Longrightarrow_L W'}$

$$\frac{\Lambda|_t = \sigma \quad P \vdash (C, \sigma, \Delta) \longrightarrow_L (C', \sigma', \Delta') \quad T(t) = C \quad T' = T\{t \rightsquigarrow C'\} \quad \Lambda' = \textsc{updTS}(\Lambda, t, \sigma')}{P \vdash (T, \Delta, \Lambda, t) \Longrightarrow_L (T', \Delta', \Lambda', t)} \ (\textsc{Ptask})$$

$$\frac{T(t) = (\mathbf{switch}\ x, K) \quad [\![x]\!]_{(t,\Lambda)} = t' \quad \Lambda' = \textsc{updG}(\Lambda, \mathsf{OSTCBCur}, t')}{P \vdash (T, \Delta, \Lambda, t) \Longrightarrow_L (T\{t \rightsquigarrow (\mathbf{skip}, K)\}, \Delta, \Lambda', t')} \ (\textsc{Psw})$$

$$\frac{\begin{array}{c} P = (A, (\eta_a, \eta_i, \theta)) \quad T(t) = (c, (\kappa_e, \kappa_s)) \quad \Lambda|_t = ((G, E, M), isr, (1, is, \mathsf{nil})) \\ \forall k'.k' \leq k \rightarrow isr(k') = 0 \quad C' = (\theta(k), (\circ, (c, \kappa_e, E) \cdot \kappa_s)) \quad T' = T\{t \rightsquigarrow C'\} \\ \sigma' = ((G, \emptyset, M), isr\{k \rightsquigarrow 1\}, (0, k :: is, \mathsf{nil})) \quad \Lambda' = \textsc{updTS}(\Lambda, t, \sigma') \end{array}}{P \vdash (T, \Delta, \Lambda, t) \Longrightarrow_L (T', \Delta, \Lambda', t)} \ (\textsc{Pitrp})$$

$\boxed{P \vdash (C, \Delta, \sigma) \longrightarrow_L (C', \Delta', \sigma')}$

$$\frac{\mathsf{InOS}(C, P) \quad P \vdash (C, \sigma) \bullet\!\!\longrightarrow_L (C', \sigma')}{P \vdash (C, \Delta, \sigma) \longrightarrow_L (C', \Delta, \sigma')} \ (\textsc{TKernel})$$

$$\frac{\neg\mathsf{InOS}(C, (A, (\eta_a, \eta_i, \theta))) \quad \Delta|_t = m \quad A \uplus \eta_a \vdash (C, m) \longmapsto (C', m') \quad \Delta' = \textsc{updCS}(\Delta, t, m')}{(A, (\eta_a, \eta_i, \theta)) \vdash (C, \Delta, \Lambda) \longrightarrow_L (C', \Delta', \Lambda)} \ (\textsc{TClt})$$

$$\frac{\sigma = (m, isr, (ie, is, cs)) \quad \sigma' = (m, isr, (0, is, ie :: cs))}{P \vdash ((\mathbf{encrt}, K), \sigma) \bullet\!\!\longrightarrow_L ((\mathbf{skip}, K), \sigma')} \ (\textsc{EnterCrt})$$

$$\frac{\sigma = (m, isr, (ie, is, ie' :: cs)) \quad \sigma' = (m, isr, (ie', is, cs))}{P \vdash ((\mathbf{excrt}, K), \sigma) \bullet\!\!\longrightarrow_L ((\mathbf{skip}, K), \sigma')} \ (\textsc{exitcrt})$$

$$\frac{0 \leq k < N \quad \sigma = (m, isr, (ie, is, cs)) \quad \sigma' = (m, isr\{k \rightsquigarrow 0\}, (ie, is, cs))}{P \vdash ((\mathbf{eoi}\ k, K), \sigma) \bullet\!\!\longrightarrow_L ((\mathbf{skip}, K), \sigma')} \ (\textsc{eoi})$$

$$\frac{\sigma = ((G, E, M), isr, (ie, k :: is, cs)) \quad \sigma' = ((G, E', M), isr, (1, is, cs)) \quad \lfloor \kappa_s \rfloor = (c, \kappa_e, E') \cdot \kappa_s'}{P \vdash ((\mathbf{iext}, (\circ, \kappa_s)), \sigma) \bullet\!\!\longrightarrow_L ((c, (\kappa_e, \kappa_s')), \sigma')} \ (\textsc{iret})$$

$$\frac{\eta_i \vdash (C, m) \longmapsto (C', m')}{(A, (\eta_a, \eta_i, \theta)) \vdash (C, (m, isr, \delta)) \bullet\!\!\longrightarrow_L (C', (m', isr, \delta))} \ (\textsc{KCstep})$$

$\boxed{\eta \vdash (C, m) \longmapsto (C', m')}$

$$\frac{}{\eta \vdash ((f(\mathsf{nil}), (\circ, \kappa_s)), m) \longmapsto ((\boldsymbol{fexec}(f, \mathsf{nil}), (\circ, \kappa_s)), m)} \ (\textsc{Fn})$$

$$\frac{}{\eta \vdash ((f(e :: \bar{e}), (\circ, \kappa_s)), m) \longmapsto ((e, (\circ, (f, \mathsf{nil}, \bar{e}) \cdot \kappa_s)), m)} \ (\textsc{Fa})$$

$$\frac{}{\eta \vdash ((v, (\circ, (f, \bar{v}, (e :: \bar{e})) \cdot \kappa_s)), m) \longmapsto ((e, (\circ, (f, (v :: \bar{v}), \bar{e}) \cdot \kappa_s)), m)} \ (\textsc{Feval})$$

$$\frac{}{\eta \vdash ((v, (\circ, (f, \bar{v}, \mathsf{nil}) \cdot \kappa_s)), m) \longmapsto ((\boldsymbol{fexec}(f, v :: \bar{v}), (\circ, \kappa_s)), m)} \ (\textsc{FEnter})$$

$$\frac{\eta(f) = (\tau, \mathcal{D}_1, \mathcal{D}_2, s) \quad m = (G, E, M) \quad m' = (G, \emptyset, M)}{\eta \vdash ((\boldsymbol{fexec}(f, \bar{v}), (\circ, \kappa_s)), m) \longmapsto ((\boldsymbol{alloc}(\bar{v}, \mathsf{rev}(\mathcal{D}_1) +\!\!+ \mathcal{D}_2), (\circ, (f, s, E) \cdot \kappa_s)), m')} \ (\textsc{FAlloc})$$

$$\frac{}{\eta \vdash ((\boldsymbol{alloc}(\mathsf{nil}, \mathsf{nil}), (\circ, (f, s, E) \cdot \kappa_s)), m) \longmapsto ((s, (\circ, (f, s, E) \cdot \kappa_s)), m)} \ (\textsc{FBody})$$

Fig. 8. The Low-level Operational Semantics

The PITRP rule says that the task t can be interrupted by a level-k interrupt request if *ie* is 1 (thus we are not in critical regions and *cs* must be nil) and there is currently no interrupt of higher or the same priority being served, according to *isr*. Then we switch the control to the interrupt handler $\theta(k)$, and saves the execution context (c, κ_e, E) onto the statement continuation (see Fig. 8). We also set the k-th bit of *isr*, clear the *ie* bit, and push k onto the *is* stack.

We use "$P \vdash (C, \Delta, \sigma) \multimap_L (C', \Delta', \sigma')$" to define task-local semantics of the assembly primitives for interrupt management. The TKERNEL rule means to execute a kernel command. It checks whether it is executing the kernel code using $\mathsf{InOS}(C, P)$ (defined in Fig.7 by checking the current continuation). The TCLT rule executes the client code. Here $\text{UPDCS}(\Delta, t, m')$ (defined in Fig.7) updates the local data of t in Δ with the new m'.

The assembly primitives ι except **switch** are all related to interrupts management and handling. To model their semantics, we introduce interrupt states in the state model, as shown at Fig. 6. The *global* register *isr* is shared by all tasks. It models the *isr* register in 8259A interrupt controller, as explained in Sec. 2.1. In addition, there are *local* interrupt states δ for each task. It contains a local copy *ie* of the `IF` flag in the EFLAGS register (see Sec. 2.1) recording whether interrupts are enabled, a stack *cs* consisting of the historical values of *ie* to support nested critical regions, and another stack *is* recording the sequence of interrupts that interrupt the execution of *the task*. The stack *is* is auxiliary data introduced mainly for verification purposes. π records the δ of each task.

encrt enters a critical region by disabling interrupts (*i.e.*, clearing the *ie* bit using **cli**). It also saves the old *ie* onto the *cs* stack. **excrt** exits the critical region by popping off the top value on *cs* and using it to restore *ie* (executing **sti** if the value is 1).

Interrupt requests may arrive non-deterministically after each step if $ie = 1$. A level-k request is served only if there is no request at higher or the same level being served (*i.e.*, $\forall k'.k' \leq k \rightarrow isr(k') = 0$). Then the processor clears *ie*, sets $isr(k)$ to 1, pushes the number k onto the logical stack *is*, saves the execution context and the local variables onto the abstract control stack (*i.e.*, the continuation), and finally jumps to the interrupt handler $\theta(k)$.

eoi k clears the k-th bit in *isr*, indicating that the k-th interrupt is no longer in service. **iext** is an abstraction of the **iret** instruction. It resets the *ie* bit to 1 to enable interrupts, pops out the topmost interrupt number on the *is* stack, and returns to the interrupted program. The IRET rule pops the execution context from the statement continuation, and sets the *ie* bit. We use $\lfloor \kappa_s \rfloor$ defined in Fig.7 to pop the execution context from the statements continuations.

In addition to the above rules, we use "$\eta \vdash (C, m) \longmapsto (C', m')$" to define the operational semantics for C steps. Here we only give the FN ,FA, FEVAL, FENTER, FALLOC and FBODY rules for executing a function call, other rules for standard C semantics are in our Coq code and omitted here. When invoking a function, it firsts uses the FA rule to evaluate the expressions of function arguments. Then the FEVAL rule is applied for evaluating the next expression. After all the arguments are evaluated to a value list, we use FALLOC to allocate

$$
\begin{array}{llll}
(\mathit{HAbsPrim}) & \gamma & \in & \mathit{ValList} \to \mathit{HAbsSt} \to (\mathit{OpVal} \times \mathit{HAbsSt}) \to \mathit{Prop} \\
(\mathit{HStmts}) & \mathbb{s} & ::= & \mathbf{sched} \mid \gamma(\bar{v}) \mid \mathbf{assert}\ \mathbb{b} \mid \mathbf{end}\ \hat{v} \mid \mathbb{s}_1; \mathbb{s}_2 \mid \mathbb{s}_1 + \mathbb{s}_2 \\
(\mathit{HAPISet}) & \varphi & ::= & \{f_1 \rightsquigarrow \mathbb{s}_1, \ldots, f_n \rightsquigarrow \mathbb{s}_n\} \qquad (\mathit{HEvtSet})\ \varepsilon ::= [\mathbb{s}_0, \ldots, \mathbb{s}_{N-1}] \\
(\mathit{HSched}) & \chi & \in & \mathit{HAbsSt} \to \mathit{TaskId} \to \mathit{Prop} \qquad (\mathit{HBExpr})\ \mathbb{b} \in \mathit{HAbsSt} \to \mathit{Prop} \\
(\mathit{HOSCode}) & \mathbb{O} & ::= & (\varphi, \varepsilon, \chi) \qquad (\mathit{HProg})\ \mathbb{P} ::= (A, \mathbb{O}) \\
(\mathit{HWorld}) & \mathbb{W} & ::= & (\mathbb{P}, T, \Delta, \Sigma)
\end{array}
$$

Fig. 9. High-Level Spec. Language

memory blocks for local variables and arguments. Finally we apply the FBODY rule to execute the function body after the local allocation. Note that when client code invokes a kernel API, the dynamical statement $\boldsymbol{fexec}(f, \bar{v})$ can be considered as the execution boundary between client-steps and and kernel-steps.

3.2 The High-Level Specification Language

$$
\begin{array}{llll}
(\mathit{HAbsSt}) & \Sigma & ::= & \{\mathbb{a}_1 \rightsquigarrow \Omega_1, \ldots, \mathbb{a}_n \rightsquigarrow \Omega_n\} \qquad (\mathit{HDataNm})\ \mathbb{a} ::= \mathsf{tcbls} \mid \mathsf{ecbls} \mid \mathsf{ctid} \mid \ldots \\
(\mathit{HData}) & \Omega & ::= & \alpha \mid \beta \mid t \mid \ldots \qquad (\mathit{HEvtId})\ \mathit{eid} \in \mathit{Addr} \\
(\mathit{HTCBLs}) & \alpha & ::= & \{t_1 \rightsquigarrow (pr_1, ts_1), \ldots, t_n \rightsquigarrow (pr_n, ts_n)\} \\
(\mathit{HECBLs}) & \beta & ::= & \{\mathit{eid}_1 \rightsquigarrow \mathit{ed}_1, \ldots, \mathit{eid}_n \rightsquigarrow \mathit{ed}_n\} \\
(\mathit{Priority}) & \mathit{pr} & \in & \mathit{int32} \qquad (\mathit{WaitType})\ \mathit{wt} ::= \mathsf{mtx}(\mathit{eid}) \mid \mathsf{tm} \mid \ldots \\
(\mathit{WaitQ}) & Q & \in & \mathsf{nil} \mid t :: Q \qquad (\mathit{HStatus})\ \mathit{ts} ::= \mathsf{rdy} \mid \mathsf{wait}(\mathit{wt}, k) \\
(\mathit{MtxOwner}) & w & ::= & \bot \mid (t, \mathit{pr}) \qquad (\mathit{HECBData})\ \mathit{ed} ::= \mathsf{mutex}(\mathit{pr}, w) \mid \ldots
\end{array}
$$

Fig. 10. The High-level Abstract Machine

Viewing from the aspect of application programmers, we model the OS kernel as an extended C language with multi-tasking and system calls. As explained above, the C language is used to implement user applications A, and the system calls invoke an abstract version of system routines in $\mathbb{O}$, which are implemented using a simple specification language. Correspondingly, the low-level concrete representation of kernel states is modeled as algebraic abstract states at the high level. This section presents the high-level language and its semantics.

As shown in Fig. 9, the whole high-level program $\mathbb{P}$ consists of the application code A and the abstract specification of the kernel $\mathbb{O}$. The application code A is the same as in the low-level language (see Fig. 4). $\mathbb{O}$ contains the specifications φ for kernel APIs, ε for interrupt handlers, and χ for the scheduler. The high-level program configuration $\mathbb{W}$ consists of the task pool T, the client state Δ, and the abstract kernel state Σ.

Programmers at this level have no control over interrupts (*e.g.*, enabling or disabling interrupts). Always enabled, interrupts are modeled implicitly as abstract external events that may occur non-deterministically at any program

$\boxed{\mathbb{P} \vdash \mathbb{W} \Longrightarrow_H \mathbb{W}'}$

$$\frac{\Sigma(\mathsf{ctid}) = t \quad T(t) = C \quad T' = T\{t \rightsquigarrow C'\} \quad \mathbb{P} \vdash (C, \Delta, \Sigma) \longrightarrow_H (C', \Delta', \Sigma')}{\mathbb{P} \vdash (T, \Delta, \Sigma) \Longrightarrow_H (T', \Delta', \Sigma')} \ (\textsc{Htask})$$

$$\frac{\mathbb{P} = (A, (\varphi, \varepsilon, \chi)) \quad \Sigma(\mathsf{ctid}) = t \quad \chi\ \Sigma\ t' \quad T(t) = (\mathbf{sched}; \mathbb{s}, K) \quad T' = T\{t \rightsquigarrow (\mathbb{s}, K)\}}{\mathbb{P} \vdash (T, \Delta, \Sigma) \Longrightarrow_H (T', \Delta, \Sigma\{\mathsf{ctid} \rightsquigarrow t'\})} \ (\textsc{Schd})$$

$$\frac{\Sigma(\mathsf{ctid}) = t \quad T(t) = (c, (\kappa_e, \kappa_s)) \quad T' = T\{t \rightsquigarrow (\varepsilon(k), (\circ, (c, \kappa_e, \emptyset) \cdot \kappa_s))\}}{(A, (\varphi, \varepsilon, \chi)) \vdash (T, \Delta, \Sigma) \Longrightarrow_H (T', \Delta, \Sigma')} \ (\textsc{Pevent})$$

$\boxed{\mathbb{P} \vdash (C, \Delta, \Sigma) \longrightarrow_H (C', \Delta, \Sigma')}$

$$\frac{(\mathbb{s}, \Sigma) \bullet\!\!-\!\!-_H\!\!-\!\!\triangleright (\mathbb{s}', \Sigma')}{\mathbb{P} \vdash ((\mathbb{s}, K), \Delta, \Sigma) \longrightarrow_H ((\mathbb{s}, K), \Delta, \Sigma')} \ (\textsc{InApi})$$

$$\frac{\Delta|_t = m \quad A \vdash (C, m) \longmapsto (C', m') \quad T' = T\{t \rightsquigarrow C'\} \quad \Delta' = \textsc{upd}\mathrm{CS}(\Delta, t, m')}{(A, \mathbb{O}) \vdash (C, \Delta, \Sigma) \longrightarrow_H (C', \Delta', \Sigma)} \ (\textsc{HTClt})$$

$$\frac{\varphi(f) = \omega \quad C = (\textit{fexec}(f, \bar{v}), K)}{(A, (\varphi, \varepsilon, \chi)) \vdash (C, \Delta, \Sigma) \longrightarrow_H ((\omega\ \bar{v}, K), \Delta, \Sigma)} \ (\textsc{EnApi})$$

$$\frac{}{\mathbb{P} \vdash ((\mathbf{end}\ v, K), \Delta, \Sigma) \longrightarrow_H ((v, K), \Delta, \Sigma)} \ (\textsc{EndApi1})$$

$$\frac{}{\mathbb{P} \vdash ((\mathbf{end}\ , K), \Delta, \Sigma) \longrightarrow_H ((\mathbf{skip}, K), \Delta, \Sigma)} \ (\textsc{EndApi2})$$

$$\frac{C = (\mathbf{end}\ , (\circ, (c, \kappa_e, \emptyset) \cdot \kappa_s))}{\mathbb{P} \vdash (C, \Delta, \Sigma) \longrightarrow_H ((c, (\kappa_e, \kappa_s)), \Delta, \Sigma)} \ (\textsc{EndEvt})$$

$\boxed{(\mathbb{s}, \Sigma) \bullet\!\!-\!\!-_H\!\!-\!\!\triangleright (\mathbb{s}', \Sigma')}$

$$\frac{\gamma\ \bar{v}\ \Sigma\ (\hat{v}, \Sigma') \quad \Sigma \perp \Sigma_f \quad dom(\Sigma(\mathsf{tcbls})) = dom(\Sigma'(\mathsf{tcbls})) \quad dom(\Sigma) = dom(\Sigma') \quad \Sigma(\mathsf{ctid}) = \Sigma'(\mathsf{ctid})}{(\gamma(\bar{v}), \Sigma \cup \Sigma_f) \bullet\!\!-\!\!-_H\!\!-\!\!\triangleright (\mathbf{end}\ \hat{v}, \Sigma' \cup \Sigma_f)} \ (\textsc{Prim})$$

$$\frac{\mathbb{b}\ \Sigma \qquad \Sigma \perp \Sigma_f}{(\mathbf{assert}\ \mathbb{b}, \Sigma \cup \Sigma_f) \bullet\!\!-\!\!-_H\!\!-\!\!\triangleright (\mathbf{end}\ \perp, \Sigma \cup \Sigma_f)} \ (\textsc{Assert})$$

$$\frac{}{(\mathbf{end}\ \hat{v}; \mathbb{s}, \Sigma) \bullet\!\!-\!\!-_H\!\!-\!\!\triangleright (\mathbb{s}, \Sigma)} \qquad \frac{(\mathbb{s}_1, \Sigma) \bullet\!\!-\!\!-_H\!\!-\!\!\triangleright (\mathbb{s}_1', \Sigma')}{(\mathbb{s}_1; \mathbb{s}_2, \Sigma) \bullet\!\!-\!\!-_H\!\!-\!\!\triangleright (\mathbb{s}_1'; \mathbb{s}_2, \Sigma')}$$

$$\frac{}{(\mathbb{s}_1 + \mathbb{s}_2, \Sigma) \bullet\!\!-\!\!-_H\!\!-\!\!\triangleright (\mathbb{s}_1, \Sigma)} \qquad \frac{}{(\mathbb{s}_1 + \mathbb{s}_2, \Sigma) \bullet\!\!-\!\!-_H\!\!-\!\!\triangleright (\mathbb{s}_2, \Sigma)}$$

Fig. 11. The High-level Operational Semantics

points. Handlers of the events are also specified as ε in $\mathbb{O}$. At the high level an incoming level-k event is always handled by executing $\varepsilon(k)$.

The system APIs and interrupt handlers are specified as an abstract statement $\mathbb{s}$, which forms a simple but expressive specification language. **sched** does scheduling. Its semantics is determined by the abstract scheduler specification χ. As defined in Fig. 9, χ is a binary relation between abstract states and task identifiers. That is, given an abstract state Σ (defined at the bottom of Fig. 9), χ finds a related task identifier as the next task to execute. Note that χ is a relation instead of a function, therefore the abstract scheduler does not have to be deterministic. Since χ is provided by users as part of the kernel specification, the semantics of **sched** in our language is configurable (by users). Specifying details of the scheduling policies (instead of using a more abstract non-deterministic scheduler that may pick *any* task) allows us to specify and verify scheduling properties such as PIF at the high level.

$\gamma(\bar{v})$ is a meta-level relation (defined in Coq) that takes $\bar{v}$ as arguments and maps an abstract state to another. Users can instantiate it to specify any *atomic* transitions over abstract states. **assert** $\mathbb{b}$ asserts that the predicate $\mathbb{b}$ holds over the current abstract state. **end** $\hat{v}$ represents the end of abstract APIs with optional return values or interrupt handlers. $\mathbb{s}_1; \mathbb{s}_2$ and $\mathbb{s}_1 + \mathbb{s}_2$ are statements for sequential composition and non-deterministic choices respectively.

Abstract states. The kernel state is represented as the abstract state Σ at the high level. As defined at Fig. 10, Σ is a mapping from names $\mathbb{a}$ to the abstract data Ω. Here tcbls is the name for the high-level abstract TCB list α, which maps task identifiers to abstract tasks, including the priority pr (a natural number), the task status (ready, waiting, *etc.*) and so on, depending on the low-level implementations. ctid is the name for the current task identifier t. wt represents the type of waiting, including waiting for a mutex $\mathsf{mtx}(eid)$ and waiting for a duration tm *etc.*. ecbls is the name for the high-level abstract ECB list β, which maps the event identifiers to abstract events, including abstract mutexes, abstract message queues and so on. The abstract mutex can be formalized as $\mathsf{mutex}(pr, w)$, in which pr is the priority of the mutex and w is a pair of task identifiers and priorities. ecbls is used to implement mutexes.

The High-level Operational Semantics. Figure 11 shows some selected rules of the high-level operational semantics, which are defined following the same structure as the low level. We use $\mathbb{P} \vdash \mathbb{W} =\!\!=\!\!\Rightarrow \mathbb{W}'$, $\mathbb{P} \vdash (C, \Delta, \Sigma) \multimap\!\!\!\rightarrow (C', \Delta, \Sigma')$ and $(\mathbb{s}, \Sigma) \bullet\!\!\multimap\!\!\triangleright (\mathbb{s}', \Sigma')$ to represent the high-level program steps, the high-level task-steps, and the high-level kernel-steps, respectively. The operational semantics rules for program-steps, correspond to an execution step of the abstract kernel specification, a step of the **sched** command, a step to handle events (abstractions for interrupts) and a step of client execution.

The high-level machine applies the Api rule to execute the specification code of the function f. As shown in Fig.12, when the client invokes an API at two levels, both the low-level and high-level machines first execute with the exactly

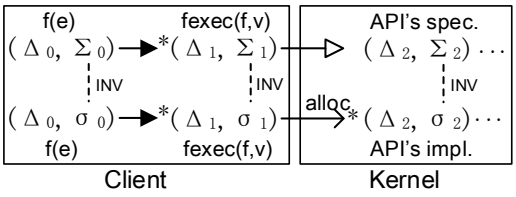


Fig. 12. Correspondence of Invoking APIs at two Levels

the same client-steps to the statement $\boldsymbol{fexec}(f, \bar{v})$. Then the low level first allocates memory for arguments and local variables and enters the concrete function body of f as we explained before, while the high level enters the abstract specification code. Note that we ignore the steps of interactions, which ensure that client states are identical at each step and the related kernel states satisfies an invariant. The InApi and EndApi rules are used to execute and return from the abstract function body. The Evt and EndEvt rules are used for responding the external events (corresponding to low-level interrupts) at the high level. The HPrim and Assume rules give the semantics for two basic abstract statements. The operational semantics of the sequential statements and non-deterministic choice statements are standard.

3.3 Modeling μC/OS-II in Our Framework.

Encoding C code of μC/OS-II in Coq. We do deep encoding of the code by defining the abstract syntax tree of the C language (subset) as an inductive datatype in Coq, and manually write the code as an expression of this type. With the help of Coq notations, the syntactic representations of C code in Coq look similar as the original C code. Actually the encoding could be done by an automatic transformer. Figure 13 (b) shows our encoding in Coq for the C source code of the scheduler of μC/OS-II (Fig. 13 (a)). All the code definitions are located in the directory of our Coq source code "CertiOS/certiucos/code/".

Specifying OS kernels. To demonstrate the use of our specification language, we show in Fig.14 the implementation and specification of the μC/OS-II API OSTimeDly, which blocks the current task for certain number of time ticks. The main job is finished in the critical region (lines 3-7), which deletes the current task from the ready table OSRdyTbl (a bitmap) and update the TCB of the current task to record the delay. After exiting the critical region, the current task is no longer ready, and it calls the scheduling function OSSched() (line 9) to switch to the ready task with the highest priority.

The specification for OSTimeDly is defined as $\omega_{\mathrm{OSTimeDly}}$. We use the abstract statement $\gamma_{\mathrm{dly}}(\bar{v})$ to specify the *atomic* behaviors of the critical region, *i.e.*, the update of the status of the current task from rdy to waiting for a duration of ticks.

```c
void  OS_Sched (void)
{

  INT8U      y;

  OS_ENTER_CRITICAL();
  y             = OSUnMapTbl[OSRdyGrp];
  OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
  if (OSPrioHighRdy != OSPrioCur) {
    OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
    OSCtxSwCtr++;
    OSCtxSw();
  }
  OS_EXIT_CRITICAL();
  return;
}
```

(a) The C code of Scheduler

```
Definition OS_Sched_impl :=
Void ·OS_Sched·(␣ ␣)··{
       ␣y @ Int8u␣;

        ENTER_CRITICAL;ₛ
        y' =ₑ OSUnMapTbl'[OSRdyGrp'];ₛ
        OSPrioHighRdy' =ₑ  ((y' ≪ '3) +ₑ OSUnMapTbl'[OSRdyTbl'[y']]);ₛ
        If (OSPrioHighRdy' !=ₑ OSPrioCur') {
            OSTCBHighRdy' =ₑ OSTCBPrioTbl'[OSPrioHighRdy'];ₛ
            ++ OSCtxSwCtr';ₛ
            SWITCH
        };ₛ
        EXIT_CRITICAL;ₛ
        RETURN
}·.
```

(b) The Coq encoding of Scheduler

Fig. 13. C Source code vs. Coq encoding

The subsequent call to OSSched() is specified by $\mathbb{s}_{\text{sched}}$. The abstract statement $\gamma_{\text{err}}(\bar{v})$ is for the error case for invalid arguments.

The critical section in OSSched() (See Fig. 13(b)) presents the code of the scheduling algorithm in μC/OS-II. It employs a constant table (OSUnMapTbl) to rapidly calculate the highest priority (OSPrioHighRdy) in the ready table (OSRdyTbl), then compare it with the priority of the current running task, if they are different, then we use the table (OSTCPrioTbl mapping priorities to the corresponding tasks) to get the ready task (OSTCBHighRdy) with the highest priority and switch to it, otherwise it does nothing because the current running task still has the highest priority among ready tasks. Since the context switching is done depends on whether the current running task has the highest priority or not. Thus the specification code for OSSched() is defined as $\mathbb{s}_{\text{sched}}$, which says that **sched** is invoked if the scheduler picks a different task (as specified in $\mathbb{b}_{\text{s}}$). Here we also give the specification $\chi_{\mu\text{C/OS-II}}$ for the priority-based scheduling

void OSTimeDly (Int16u ticks) :

```
1  if (ticks > 0){
2    OS_ENTER_CRITICAL();
3      if((OSTCBCur->OSTCBStat==OS_STAT_RDY)&&
           (OSTCBCur->OSTCBDly==0)){
4        OSRdyTbl[OSTCBCur->OSTCBY]=
             OSRdyTbl[OSTCBCur->OSTCBY]& (OSTCBCur->OSTCBBitX);
5        if(OSRdyTbl[OSTCBCur->OSTCBY]==0){
6           OSRdyGrp= OSRdyGrp&(OSTCBCur->OSTCBBitY)}
7        OSTCBCur->OSTCBDly=ticks;
8        OS_EXIT_CRITICAL();
9        OSSched();
10     }else{ OS_EXIT_CRITICAL();} }
11 return;
```

$$\text{OSTimeDly: } \omega_{\text{OSTimeDly}} \stackrel{\text{def}}{=} \lambda\bar{v}.(\gamma_{\text{err}}(\bar{v}) + (\gamma_{\text{dly}}(\bar{v}); \mathbb{s}_{\text{sched}}; \mathbf{end}))$$

$$\gamma_{\text{err}} \stackrel{\text{def}}{=} \lambda\bar{v}, \Sigma, (\hat{v}, \Sigma').\exists \mathsf{ticks}.\bar{v} = \mathsf{ticks}::\mathsf{nil} \wedge \Sigma = \Sigma' \wedge (\mathsf{ticks} \leq 0 \vee (\exists t.\Sigma(\mathsf{ctid}) = t \wedge \Sigma(\mathsf{tcbls})(t) = (\_, ts) \wedge ts \neq \mathsf{rdy}))$$

$$\gamma_{\text{dly}} \stackrel{\text{def}}{=} \lambda\bar{v}, \Sigma, (\hat{v}, \Sigma').\exists \mathsf{ticks}.\bar{v} = \mathsf{ticks}::\mathsf{nil} \wedge \mathsf{ticks} > 0 \wedge (\exists t, pr.\Sigma(\mathsf{ctid}) = t \wedge \Sigma(\mathsf{tcbls})(t) = (pr, \mathsf{rdy}) \wedge \Sigma' = \Sigma\{\mathsf{tcbls} \rightsquigarrow \{t \rightsquigarrow (pr, \mathsf{wait}(wt, \mathsf{ticks}))\}\})$$

$$\text{OSSched: } \mathbb{s}_{\text{sched}} \stackrel{\text{def}}{=} (\mathbf{assert}\ \mathbb{b}_{\text{s}}; \mathbf{sched}) + \mathbf{assert}\ \mathbb{b}_{\text{n}}$$

$$\mathbb{b}_{\text{s}} \stackrel{\text{def}}{=} \lambda\Sigma.\exists t, t'.\Sigma(\mathsf{ctid}) = t \wedge \chi_{\mu\text{C/OS-II}}\ \Sigma\ t' \wedge t \neq t'$$

$$\mathbb{b}_{\text{n}} \stackrel{\text{def}}{=} \lambda\Sigma.\exists t.\Sigma(\mathsf{ctid}) = t \wedge \chi_{\mu\text{C/OS-II}}\ \Sigma\ t$$

$$\chi_{\mu\text{C/OS-II}} \stackrel{\text{def}}{=} \lambda\Sigma, t.\exists \alpha, pr.\Sigma(\mathsf{tcbls}) = \alpha \wedge \alpha(t) = (pr, \mathsf{rdy}) \wedge \forall t', pr'.\ (t \neq t' \wedge \alpha(t') = (pr', \mathsf{rdy})) \rightarrow pr' \prec pr$$

Fig. 14. Specification Code for μC/OS-II

strategy. It requires that the target task have the highest priority among all the ready tasks.

As more examples, Table. 1 shows the specifications for some μC/OS-II interrupt handlers and APIs, with definitions for atomic operations γ omitted which can be found in our Coq code [3]. To demonstrate that our framework supports multi-level interrupts, we introduce a pseudo handler named as ToyISR for the level-1 interrupt. It increments a counter, a global variable shared with tasks. The specification code for interrupts ends with a non-deterministic choice of OSSched's specification code $\mathbb{s}_{\text{sched}}$. That is because, before exiting the interrupts, it calls to OSSched() or not, depending on whether the interrupt is topmost. γ_{tickisr} specifies the behaviors of how the time interrupt updates the status of each task. The APIs OSQPend and OSQPost are used to receive and send messages among tasks. In OSQPend's specification code, it says the current task might successful receives a message from the queue, or it might be blocking (γ_{qpdblk}) due to the empty queue. If the task gets blocking, it calls the scheduler. When the task is switched back, it is either timeout (γ_{qpdto}) or successfully obtaining a message ($\gamma_{\text{qpdblkget}}$). The specification code of OSQPost means that

Scheduler	$\chi_{\mu\text{C/OS-II}}$			defined in Fig.14
Multi-level Interrupts	$\varepsilon_{\mu\text{C/OS-II}}$	0	TimeTickISR	$\mathbb{s}_0 \stackrel{\text{def}}{=} \gamma_{\text{tickisr}}(\mathsf{nil}); ((\mathbb{s}_{\text{sched}}; \textbf{end}\ ) + \textbf{end}\ )$
		1	ToyISR	$\mathbb{s}_1 \stackrel{\text{def}}{=} \gamma_{\text{toyisr}}(\mathsf{nil}); ((\mathbb{s}_{\text{sched}}; \textbf{end}\ ) + \textbf{end}\ )$
Synchronization Mechanism	$\varphi_{\mu\text{C/OS-II}}$	Message Queue	OSQPend	$\omega_{\text{qpend}} \stackrel{\text{def}}{=} \lambda\bar{v}.\gamma_{\text{qpderr}}(\bar{v}) + \gamma_{\text{qpdsucc}}(\bar{v}) + (\gamma_{\text{qpdblk}}(\bar{v}); \mathbb{s}_{\text{sched}}; (\gamma_{\text{qpdto}}(\bar{v}) + \gamma_{\text{qpdblkget}}(\bar{v})))$
			OSQPost	$\omega_{\text{qpost}} \stackrel{\text{def}}{=} \lambda\bar{v}.\gamma_{\text{qpsterr}}(\bar{v}) + \gamma_{\text{qpstsuccnowt}}(\bar{v}) + (\gamma_{\text{qpstsuccwt}}(\bar{v}); \mathbb{s}_{\text{sched}}; \textbf{end}\ \mathsf{NO\_ERR})$
			OSQCreate	$\omega_{\text{qcrt}} \stackrel{\text{def}}{=} \lambda\bar{v}.\gamma_{\text{qcterr}}(\bar{v}) + \gamma_{\text{qcrtsucc}}(\bar{v})$
			OSQDelete	$\omega_{\text{qdel}} \stackrel{\text{def}}{=} \lambda\bar{v}.\gamma_{\text{qdelerr}}(\bar{v}) + \gamma_{\text{qdelsucc}}(\bar{v})$
			OSAccept	$\omega_{\text{qacc}} \stackrel{\text{def}}{=} \lambda\bar{v}.\gamma_{\text{qaccerr}}(\bar{v}) + \gamma_{\text{qaccsucc}}(\bar{v})$
		Mutex, Semaphore, Mail Box, ...		

Table 1. The Specification Code for the Main Components of μC/OS-II

$$(\mathit{EvtTrace})\ \xi ::= \mathsf{nil} \mid \lightning \mid \varsigma :: \xi$$

$$\frac{}{\mathit{LETr}(P, W, \epsilon)} \qquad \frac{P \vdash W =\!\!=_L\!\!\Rightarrow \textbf{abort}}{\mathit{LETr}(P, W, \lightning)} \qquad \frac{P \vdash W =\!\!=_L\!\!\Rightarrow W' \quad \mathit{LETr}(P, W', \xi)}{\mathit{LETr}(P, W, \xi)}$$

$$\frac{P \vdash W =\!\!=_L^{\varsigma}\!\!\Rightarrow W' \quad \mathit{LETr}(P, W', \xi)}{\mathit{LETr}(P, W, \varsigma :: \xi)}$$

$$\frac{}{\mathit{HETr}(\mathbb{W}, \epsilon)} \qquad \frac{\mathbb{P} \vdash \mathbb{W} =\!\!=_H\!\!\Rightarrow \textbf{abort}}{\mathit{HETr}(\mathbb{P}, \mathbb{W}, \lightning)} \qquad \frac{\mathbb{P} \vdash \mathbb{W} =\!\!=_H\!\!\Rightarrow \mathbb{W}' \quad \mathit{HETr}(\mathbb{P}, \mathbb{W}', \xi)}{\mathit{HETr}((\mathbb{P}, \mathbb{W}), \xi)}$$

$$\frac{\mathbb{P} \vdash \mathbb{W} =\!\!=_H^{\varsigma}\!\!\Rightarrow \mathbb{W}' \quad \mathit{HETr}(\mathbb{P}, \mathbb{W}', \xi)}{\mathit{HETr}(\mathbb{P}, \mathbb{W}, \varsigma :: \xi)}$$

$$(P, W) \preccurlyeq (\mathbb{P}, \mathbb{W}) \stackrel{\text{def}}{=} \forall \xi. \mathit{LETr}(P, W, \xi) \Longrightarrow \mathit{HETr}(\mathbb{P}, \mathbb{W}, \xi)$$

Fig. 15. Event Trace Refinement

the current task sends a message to the queue, and makes a waiting task become ready if there exists one ($\gamma_{\text{qpstsuccwt}}$), then it calls the scheduler and return with NO_ERR. If there is nobody else waiting it just inserts the message to the queue ($\gamma_{\text{qpstsuccnowt}}$) without calling to the scheduler.

3.4 OS Correctness

As we explain in Sec.2.2, the correctness of OS kernels can be defined in terms of contextual refinement. Below we give its formal definition.

Definition 3.1 (OS Correctness). $O \sqsubseteq_\psi \mathbb{O}$ iff
$\forall A, W, \mathbb{W}.\, \mathsf{Match}(\psi, W, \mathbb{W}) \Longrightarrow ((A, O), W) \preccurlyeq ((A, \mathbb{O}), \mathbb{W})$
where $\psi \in \mathit{LOSFullSt} \to \mathit{HAbsSt} \to \mathit{Prop}$ and
$\mathsf{Match}(\psi, (T, \Delta, \Lambda, t), (T, \Delta, \Sigma)) \stackrel{\text{def}}{=}$
$(t \in \mathit{dom}(T)) \wedge (\psi\ \Lambda\ \Sigma) \wedge (t = \Sigma(\mathsf{ctid})) \wedge (\mathit{dom}(T) = \mathit{dom}(\Sigma(\mathsf{tcbls})))$

The low-level kernel code O refines its high-level abstract specifications $\mathbb{O}$ with constraints ψ over initial kernel states, denoted as $O \sqsubseteq_\psi \mathbb{O}$, if and only if for any client code A, *low-level state* W and *high-level state* $\mathbb{W}$, if W and $\mathbb{W}$ satisfy certain consistency constraint (w.r.t. ψ), then the set of observable behaviors of the low-level configuration $((A, O), W)$ is a subset of $((A, \mathbb{O}), \mathbb{W})$ (*i.e.*, $(P, W) \preccurlyeq (\mathbb{P}, \mathbb{W})$, following the event trace refinement in [23]). The definition of $(P, W) \preccurlyeq (\mathbb{P}, \mathbb{W})$ is given in Fig. 15 and will be explained below.

The constraint Match requires that: (1) initially W and $\mathbb{W}$ have the same task pool T and client state Δ; (2) the current task t is in T; (3) the low-level kernel state Λ and the high-level abstract state satisfy ψ; (4) the *current* task at the low level and the high level are the same; and (5) the set of tasks in the abstract TCB list should be the same as those in the low-level task pool.

Event trace refinement for OS correctness. We give the definition of event trace refinement in Fig 15. nil means a empty trace. A trace is a sequence of externally observable events ς, and may end with a fault marker $\lightning$. $LETr(P, W, \xi)$ means ξ can be generated by low-level machine (P, W). $HETr(\mathbb{P}, \mathbb{W}, \xi)$ means ξ can be generated by high-level machine $(\mathbb{P}, \mathbb{W})$. $(P, W) \preccurlyeq (\mathbb{P}, \mathbb{W})$ means that all the observable event trace generated by the low-level machine (P, W) can also be generated by high-level machine $(\mathbb{P}, \mathbb{W})$.

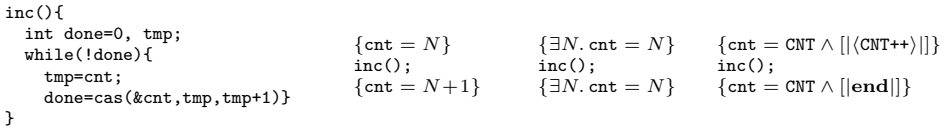
4 Relational Program Logic for Refinement Verification

In this section, we present a CSL-style *relational* program logic for refinement verification. The logic uses relational assertions to prove refinement between an implementation and its specification. It also follows the ownership-transfer semantics in CSL to reason about multi-level hardware interrupts.

4.1 Refinement of concurrent programs, and relational reasoning.

For concurrent programs, refinement establishes stronger functional correctness than traditional Hoare triples. As an example, the function `inc` shown in Fig. 16(a) increments the counter `cnt`. It may be called simultaneously by concurrent tasks. Figure 16(b) gives pre-/post-conditions to specify `inc`, which would be valid in a sequential setting and is sufficient to describe the functionality. However, they cannot be used in a concurrent setting because they are not stable with respect to concurrent behaviors of other tasks. To make them stable, we may need the specifications in Fig. 16(c), which is too weak to capture the functionality.

Figure 16(d) gives a relational specifications to show that `inc` refines an abstract operation $\langle$`CNT++`$\rangle$ [24], where $\langle C \rangle$ represents an *atomic* operation C. The relational assertions specify three important entities, the concrete state (`cnt`), the abstract state (`CNT`) and the abstract operation ($\langle$`CNT++`$\rangle$) that the program refines (which could be non-atomic in general) [24]. The precondition requires that initially `cnt` has the consistent value with its abstract counterpart `CNT`, and



(a) Implementation of `inc` (b) A wrong spec. (c) A weak spec. (d) Refinement spec.

Fig. 16. Specification of Concurrent Programs

the abstract operation that `inc` needs to refine is $\langle$`CNT++`$\rangle$. The postcondition ensures `cnt` and `CNT` remain consistent and the remaining abstract operation that needs to be refined is **end** (*i.e.*, $\langle$`CNT++`$\rangle$ has been accomplished).

Our refinement proofs for OS kernels follow the same kind of relational reasoning, where the assertions now relate the concrete kernel state, the abstract kernel state (Σ) and the API specifications ($\mathbb{s}$).

4.2 The Relational Assertions Language

Figure 17 gives the relational assertion language. Its semantics is given in Fig. 18.

$$
\begin{array}{lll}
(Asrt) & p, q, r & ::= \mathsf{emp} \mid \mathsf{empE} \mid x \mapsto v \mid \mathsf{ISR}(isr) \mid \mathsf{IE}(ie) \mid \mathsf{IS}(is) \mid \mathsf{CS}(cs) \mid \llcorner k \lrcorner \mid \chi \triangleright t \\
 & & \quad \mid \mathbb{a} \rightarrowtail \Omega \mid [|\mathbb{s}|] \mid p * p \mid p \wedge p \mid \ldots \\
(InvAsrt) & I & ::= [p_0, \ldots, p_N]
\end{array}
$$

Fig. 17. Relational Assertions

As explained above, the assertions are interpreted over relational states Θ, which consist of the low-level task-local states σ, the high-level abstract states Σ, and the abstract statements $\mathbb{s}$ that the low-level code needs to refine. Σ and $\mathbb{s}$ are defined in Fig. 9. σ, as shown in Fig. 18, consists of a task-local view m of program variables and memory, and also the global *isr* register and the task-local interrupt states δ (see Fig. 4). Here m contains the global and local variables (G and E respectively) and the memory M, whose definitions are omitted.

Assertion emp says the low-level memory and the high-level abstract state are both empty. empE further requires that the local variable environment be empty too. $x \mapsto v$ specifies a singleton memory cell with v stored in the global program variable x. $\mathsf{ISR}(isr)$, $\mathsf{IS}(is)$, $\mathsf{IE}(ie)$ and $\mathsf{CS}(cs)$ specify the value of the corresponding interrupt status (see Fig. 4). $\llcorner k \lrcorner$ means that the currently running interrupt handler is at level k (or $k = N$, meaning no running handlers).

$\chi \triangleright t$ says that, based on the high-level abstract state, the abstract scheduler χ picks t as the target task. $\mathbb{a} \rightarrowtail \Omega$ specifies a singleton high-level abstract state mapping the data name $\mathbb{a}$ to the abstract data Ω. $[|\mathbb{s}|]$ means the current abstract statement remaining to be refined is $\mathbb{s}$. The separating conjunction $p_1 * p_2$ means p_1 and p_2 hold over disjoint parts of a relational state.

$$(\mathit{RelState})\ \Theta ::= (\sigma, \Sigma, \mathbb{s})$$

$$\begin{array}{lll}
(\sigma,\Sigma,\mathbb{s}) \models \mathsf{emp} & \text{iff} & \sigma.m.M = \emptyset \wedge \Sigma = \emptyset \\
(\sigma,\Sigma,\mathbb{s}) \models \mathsf{empE} & \text{iff} & \sigma.m.E = \emptyset \wedge (\sigma,\Sigma,\mathbb{s}) \models \mathsf{emp} \\
(\sigma,\Sigma,\mathbb{s}) \models x \mapsto v & \text{iff} & \exists a.(\sigma.m.G)(x) = a \wedge \sigma.m.M = \{a \rightsquigarrow v\} \wedge \Sigma = \emptyset \\
(\sigma,\Sigma,\mathbb{s}) \models \mathsf{ISR}(isr') & \text{iff} & \sigma.isr = isr' \wedge (\sigma,\Sigma,\mathbb{s}) \models \mathsf{emp} \\
(\sigma,\Sigma,\mathbb{s}) \models \llcorner k \lrcorner & \text{iff} & ((k{=}N \wedge is{=}\mathsf{nil}) \vee \exists is'.(\sigma.\delta.is{=}k :: is')) \wedge (\sigma,\Sigma,\mathbb{s}) \models \mathsf{emp} \\
(\sigma,\Sigma,\mathbb{s}) \models \chi \triangleright t & \text{iff} & \chi\ \Sigma\ t \\
(\sigma,\Sigma,\mathbb{s}) \models [|\mathbb{s}'|] & \text{iff} & \mathbb{s} = \mathbb{s}' \wedge (\sigma,\Sigma,\mathbb{s}) \models \mathsf{emp} \\
(\sigma,\Sigma,\mathbb{s}) \models \mathbb{a} \mapsto \Omega & \text{iff} & \Sigma = \{\mathbb{a} \rightsquigarrow \Omega\} \wedge \sigma.m.M = \emptyset
\end{array}$$

$$f \perp g \stackrel{\text{def}}{=} dom(f) \cap dom(g) = \emptyset \qquad \Sigma_1 \uplus \Sigma_2 \stackrel{\text{def}}{=} \begin{cases} \Sigma_1 \cup \Sigma_2 & \text{iff } \Sigma_1 \perp \Sigma_2 \\ \mathit{undef} & \text{otherwise} \end{cases}$$

$$\sigma_1 \uplus \sigma_2 \stackrel{\text{def}}{=} \begin{cases} ((G,E,M_1 \cup M_2), isr, \delta) & \text{iff } M_1 \perp M_2 \wedge \sigma_1 = ((G,E,M_1),isr,\delta) \\ & \qquad \wedge \sigma_2 = ((G,E,M_2),isr,\delta) \\ \mathit{undef} & \text{otherwise} \end{cases}$$

$$\Theta_1 \uplus \Theta_2 \stackrel{\text{def}}{=} (\sigma_1 \uplus \sigma_2, \Sigma_1 \uplus \Sigma_2, \mathbb{s}) \qquad \text{where } \Theta_1 = (\sigma_1,\Sigma_1,\mathbb{s}) \wedge \Theta_2 = (\sigma_2,\Sigma_2,\mathbb{s})$$

$$\Theta \models p_1 * p_2 \qquad \text{iff } \exists \Theta_1,\Theta_2.\Theta = \Theta_1 \uplus \Theta_2 \wedge \Theta_1 \models p_1 \wedge \Theta_2 \models p_2$$

Fig. 18. Semantics of Relational Assertions

Ownership-transfer semantics for multi-level interrupts. CSL [26] prevents data races by enforcing disjoint ownership of resources among tasks. Synchronization is modeled in terms of ownership transfer. Feng *et al.* [11] extended CSL and assigned ownership-transfer semantics to interrupt operations, but it supports only single-level interrupt. We extend their work to support multi-level interrupts.

Figure 19 shows the *logical* memory model (where the number N of interrupts is 6). Interrupt handlers at levels 0 to $N-1$ are assigned with resource blocks $B_0, \ldots, B_{N-1}$ respectively, and the resource shared *only by non-handler code* is represented as B_N. The block B_k is specified by $I(k)$, where I is defined as a sequence of $N+1$ assertions (see Fig. 17). Handlers' priorities to pick their required resources are consistent with their interrupt priority levels. That is, B_0 satisfies all the need of the level-0 (highest priority) handler, while the level-k handler may need to access $B_0, \ldots, B_{k-1}$, in addition to B_k. The non-handler code has the lowest priority.

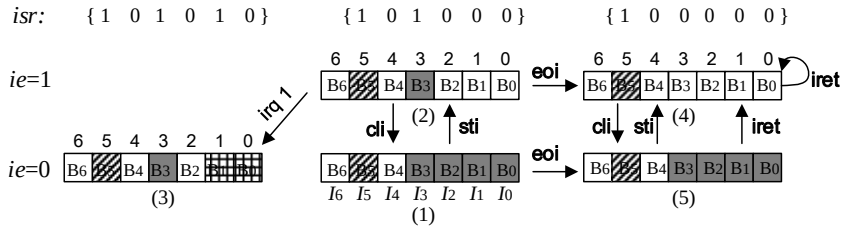


Fig. 19. Ownership-Transfer for Multi-level Interrupts

Figure 19 demonstrates the ownership transfer of resource caused by interrupt operations under different conditions. The grey blocks (with different textures) represent resources being exclusively owned in different interrupt handlers, while the white ones mean resources are *available* for share. Suppose initially we are at state (1), where the level-3 handler is being executed. Since interrupts are disabled, the handler owns $B_0 - B_3$, knowing no requests of levels 0 to 3 could be served. Enabling interrupts (**sti**) loses $B_0 - B_2$, as shown by state (2), but B_3 is remained because $isr(3) = 1$ and requests of the same (or lower) level are not handled. However, if $isr(3) = 0$ instead (as in state (5)) executing **sti** loses B_3 as well. Ownership transfer caused by **cli** is the dual of **sti**.

Executing **eoi** at state (1) reaches state (5), but it causes no ownership transfer because interrupts are disabled anyway. If interrupts are enabled instead, as in state (2), **eoi** loses the ownership of B_3 because another level-3 request may be handled in state (4). **iret** can be executed only after **eoi**. If interrupts are disabled (as in state (5)), it transfers $B_0 - B_3$ from local resources to shared resources. Otherwise (as in state (4)) there is no ownership transfer because the handler has lost the ownership of $B_0 - B_3$ already.

At state (2), interrupts with higher priority can be served. The "**irq** 1" step sets the bit $isr(1)$, disables interrupts, and transfers B_0 and B_1 from shared resources to local resources of the level-1 handler, as in state (3).

4.3 Inference Rules

The top rule. We show some selected program logic rules in Fig. 22. The TopRule establishes the judgment $\vdash_\psi O : \mathbb{O}$, ensuring the correctness of O w.r.t. $\mathbb{O}$ if the initial concrete and abstract kernel states satisfy ψ (explained in Def. 3.4).

$$(\mathit{FunPre})\ \ fp \in \mathit{Vallist} \to \mathit{Asrt} \qquad (\mathit{FunPost})\ fq \in \mathit{Vallist} \to \mathit{Asrt}$$
$$(\mathit{FunSpec})\ \ \Gamma\ \in \mathit{FName} \rightharpoonup \mathit{FunPre} \times \mathit{FunPost}$$

Fig. 20. Function Specifications

To verify the kernel, we need to come up with a specification Γ for the internal functions η_i in the low-level code, and a sequence of invariants I for kernel states. Γ defined in Fig. 20 assigns a pair of pre-/post-conditions to each internal function. The pre-/post-conditions is a mapping from value lists to assertions. The value list is used to specify the value of parameters.

Then we prove that the internal functions, the API implementations and the interrupt handlers in the low-level kernel satisfy their specifications, respectively (the last three premises in the first line of the TopRule rule). The proof of each component carries the abstract scheduler specification χ and the invariant I.

The rule also requires that ψ ensures the initial states satisfy the invariant $I[0, N]$, the interrupt-related states are properly initialized, and the initial local

$$
\begin{array}{ll}
(\sigma, \Sigma, \mathbb{s}) \models x \mapsto_l v & \text{iff } \exists a.(\sigma.m.E)(x) = a \wedge \sigma.m.M = \{a \rightsquigarrow v\} \wedge \Sigma = \emptyset \\
\mathsf{getD}(\eta, f) & \stackrel{\text{def}}{=} \begin{cases} \mathsf{rev}(\mathcal{D}_1) +\!\!+ \mathcal{D}_2 & \text{iff} \quad \eta(f) = (\tau, \mathcal{D}_1, \mathcal{D}_2, s) \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{BuildP}(\mathcal{D}, \bar{v}) & \stackrel{\text{def}}{=} \begin{cases} x \mapsto_l v * \mathsf{BuildP}(\mathcal{D}', \bar{v}') & \text{iff } \mathcal{D} = (x, \tau) :: \mathcal{D}' \wedge \bar{v} = v :: \bar{v}' \\ x \mapsto_l \_ * \mathsf{BuildP}(\mathcal{D}', \mathsf{nil}) & \text{iff } \mathcal{D} = (x, \tau) :: \mathcal{D}' \wedge \bar{v} = \mathsf{nil} \\ \mathsf{emp} & \text{iff } \mathcal{D} = \mathsf{nil} \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{BuildR}(\mathcal{D}) & \stackrel{\text{def}}{=} \mathsf{BuildP}(\mathcal{D}, \mathsf{nil}) \\
\mathsf{BuildAPIPre}(\eta_a, f, \omega, \bar{v}) & \stackrel{\text{def}}{=} \mathsf{OS}[\bar{0}, 1, \mathsf{nil}, \mathsf{nil}] * \mathsf{BuildP}(\mathsf{getD}(\eta_a, f), \bar{v}) * [|\omega(\bar{v})|] \\
\mathsf{BuildAPIRet}(\eta_a, f) & \stackrel{\text{def}}{=} \mathsf{OS}[\bar{0}, 1, \mathsf{nil}, \mathsf{nil}] * \mathsf{BuildR}(\mathsf{getD}(\eta_a, f)) * [|\mathbf{end}\ |] \\
\mathsf{BuildFunPre}(\eta_i, f, \bar{v}, fp) & \stackrel{\text{def}}{=} fp(\bar{v}) * \mathsf{BuildP}(\mathsf{getD}(\eta_i, f), \bar{v}) \\
\mathsf{BuildFunRet}(\eta_i, f, \bar{v}, fq) & \stackrel{\text{def}}{=} fq(\bar{v}) * \mathsf{BuildR}(\mathsf{getD}(\eta_i, f)) \\
\mathsf{BldItrpPre}(k, \varepsilon, isr, is, I) & \stackrel{\text{def}}{=} \mathsf{OS}[isr\{k \rightsquigarrow 1\}, 0, k :: is, \mathsf{nil}] * I[0, k] * [|\varepsilon(k)|] * \mathsf{empE} \\
\mathsf{BldItrpRet}(k, isr, is, I) & \stackrel{\text{def}}{=} \exists ie.\, \mathsf{OS}[isr\{k \rightsquigarrow 0\}, ie, k :: is, \mathsf{nil}] * \\
& \qquad ((ie = 1 \wedge \mathsf{emp}) \vee (ie = 0 \wedge I[0, k])) * [|\mathbf{end}|] \\
I[n, m] & \stackrel{\text{def}}{=} \begin{cases} I(n) * I(n+1) * \ldots * I(m) & \text{if } 0 \le n \le m \le N \\ \mathsf{emp} & \text{otherwise} \end{cases} \\
\mathsf{OS}[isr, ie, is, cs] & \stackrel{\text{def}}{=} \exists k.\, \mathsf{ISR}(isr) * \mathsf{IE}(ie) * \mathsf{IS}(is) * \mathsf{CS}(cs) * \llcorner k \lrcorner * \\
& \qquad (\forall k'.\, 0 \le k' < k \rightarrow isr(k') = 0) \\
\lfloor \psi \rfloor & \stackrel{\text{def}}{=} \lambda(\sigma, \Sigma, \mathbb{s}).\ \forall \Lambda.\, \psi\ \Lambda\ \Sigma \wedge \exists t.\Lambda|_t = \sigma \\
\mathsf{INV}(I, k) & \stackrel{\text{def}}{=} \exists isr.\ \mathsf{ISR}(isr) * \\
& \qquad ((isr(k) = 1 \wedge \mathsf{emp}) \vee ((isr(k) = 0 \vee k = N) \wedge I(k))) \\
\mathsf{SWINV}(I) & \stackrel{\text{def}}{=} \mathsf{ISR}(\bar{0}) * \mathsf{IE}(0) * (\exists\ k.\, \llcorner k \lrcorner * I[0, k])
\end{array}
$$

Fig. 21. Auxiliary Definitions of Inference Rules

variable environment is empty. $I[n, m]$ defined in Fig. 22 is the separating conjunction of invariants from level n to m. $\mathsf{OS}[isr, ie, is, cs]$ specifies the status of interrupts, and requires that the currently executing handler (on top of is) have the highest priority among those in service (as recorded in isr). $\lfloor \psi \rfloor$ lifts ψ to relational assertions (defined in Fig. 21). More details about side conditions in the rule can be seen in Coq code[3].

Verifying interrupt handlers. The ITRP rule proves the correctness of interrupt handlers. It requires that each individual interrupt handler is correct with respect to its specification. The judgment for statements is in the form of $\Gamma; \chi; I; r; p_i \vdash \{\, p \,\}\, s\, \{\, q \,\}$. We follow the CSL-style reasoning, where I specifies shared resource blocks, and the pre-/post-conditions specify *local* resources that are accessed exclusively by the current task. The precondition is p, while q, r and p_i are all post-conditions for different exits, *i.e.*, sequential composition, return from functions, and return from interrupts, respectively. For the whole body of interrupt handlers, we disable the other two exits by setting r and q to false.

$$\frac{O = (\eta_a, \eta_i, \theta) \quad \mathbb{O} = (\varphi, \varepsilon, \chi) \quad \chi; I \vdash \eta_i : \Gamma \quad \Gamma; \chi; I \vdash \eta_a : \varphi \quad \Gamma; \chi; I \vdash \theta : \varepsilon \quad \lfloor \psi \rfloor \Rightarrow I[0, N] * \mathsf{OS}[\bar{0}, 1, \mathsf{nil}, \mathsf{nil}] * \mathsf{empE} \quad \text{other side conditions}}{\vdash_\psi O : \mathbb{O}} \ (\textsc{TopRule})$$

$$\frac{p = \mathsf{BldItrpPre}(k, \varepsilon, isr, is, I) \quad p_i = \mathsf{BldItrpRet}(k, isr, is, I) \quad dom(\theta) = dom(\varepsilon) \quad \Gamma; \chi; I; \mathsf{false}; p_i \vdash \{\ p\ \}\ \theta(k)\ \{\ \mathsf{false}\ \} \quad \text{for all } k \in \{0, \ldots, N-1\}}{\Gamma; \chi; I \vdash \theta : \varepsilon} \ (\textsc{Itrp})$$

$$\frac{dom(\eta) = dom(\varphi) \quad \mathsf{BuildAPIPre}(\eta, f, \varphi, \bar{v}) = p \quad \mathsf{BuildAPIRet}(\eta, f) = r \quad \eta(f) = (\_, \_, \_, s) \quad \varphi(f) = \omega \quad \Gamma; \chi; I; r; \mathsf{false} \vdash \{\ p\ \}\ s\ \{\ \mathsf{false}\ \}}{\Gamma; \chi; I \vdash \eta : \varphi} \ (\textsc{WfAPI})$$

$$\frac{dom(\eta) = dom(\Gamma) \quad \mathsf{BuildFunPre}(\eta, f, \bar{v}, fp, \omega) = p \quad \mathsf{BuildFunRet}(\eta, f, fq) = r \quad \eta(f) = (\_, \_, \_, s) \quad \Gamma(f) = (fp, fq, \omega) \quad \Gamma; \chi; I; r; \mathsf{false} \vdash \{\ p\ \}\ s\ \{\ \mathsf{false}\ \}}{\chi; I \vdash \eta : \Gamma} \ (\textsc{WfFun})$$

$$\frac{p \Rightarrow p' \quad \Gamma; \chi; I; r; p_i \vdash \{\ p'\ \}\ s\ \{\ q'\ \} \quad q' \Rightarrow q}{\Gamma; \chi; I; r; p_i \vdash \{\ p\ \}\ s\ \{\ q\ \}} \ (\textsc{conseq})$$

$$\frac{\Gamma; \chi; I; r; p_i \vdash \{\ p_1\ \}\ s_1\ \{\ p_2\ \} \quad \Gamma; \chi; I; r; p_i \vdash \{\ p_2\ \}\ s_2\ \{\ p_3\ \}}{\Gamma; \chi; I; r; p_i \vdash \{\ p_1\ \}\ s_1; s_2\ \{\ p_3\ \}} \ (\textsc{seq})$$

$$\frac{\Gamma; \chi; I; r; p_i \vdash \{\ p_1\ \}\ s\ \{\ p_2\ \} \quad q \text{ does not specify } ie, is, cs, isr \text{ and } \mathfrak{s}}{\Gamma; \chi; I; r * q; p_i * q \vdash \{\ p_1 * q\ \}\ s\ \{\ p_2 * q\ \}} \ (\textsc{frm})$$

$$\frac{p \Rrightarrow p' \quad \Gamma; \chi; I; r; p_i \vdash \{\ p'\ \}\ s\ \{\ q'\ \} \quad q' \Rrightarrow q}{\Gamma; \chi; I; r; p_i \vdash \{\ p\ \}\ s\ \{\ q\ \}} \ (\textsc{absCSQ})$$

$$\frac{}{\Gamma; \chi; I; r; p_i \vdash \{\mathsf{OS}[isr, 1, is, cs] * \llcorner k \lrcorner * [|\mathfrak{s}|]\}\ \mathbf{encrt}\ \{\mathsf{OS}[isr, 0, is, 1 :: cs] * \mathsf{INV}(I, k) * I[0, k-1] * [|\mathfrak{s}|]\}} \ (\textsc{encrt})$$

$$\frac{}{\Gamma; \chi; I; r; p_i \vdash \{\ \mathsf{OS}[isr, 0, is, cs] * [|\mathfrak{s}|]\ \}\ \mathbf{encrt}\ \{\ \mathsf{OS}[isr, 0, is, 0 :: cs] * [|\mathfrak{s}|]\ \}} \ (\textsc{encrt-0})$$

$$\frac{}{\Gamma; \chi; I; r; p_i \vdash \{\mathsf{OS}[isr, 0, is, 1 :: cs] * \llcorner k \lrcorner * \mathsf{INV}(I, k) * I[0, k-1] * [|\mathfrak{s}|]\}\ \mathbf{excrt}\ \{\mathsf{OS}[isr, 1, is, cs] * [|\mathfrak{s}|]\}} \ (\textsc{excrt})$$

$$\frac{}{\Gamma; \chi; I; r; p_i \vdash \{\ \mathsf{OS}[isr, 0, is, 0 :: cs] * [|\mathfrak{s}|]\ \}\ \mathbf{excrt}\ \{\ \mathsf{OS}[isr, 0, is, cs] * [|\mathfrak{s}|]\ \}} \ (\textsc{excrt-0})$$

$$\frac{}{\Gamma; \chi; I; r; p_i \vdash \{\ \mathsf{OS}[isr, 1, k :: is, cs] * I(k) * [|\mathfrak{s}|]\ \}\ \mathbf{eoi}\ k\ \{\ \mathsf{OS}[isr\{k \rightsquigarrow 0\}, 1, k :: is, cs] * [|\mathfrak{s}|]\ \}} \ (\textsc{eoi})$$

$$\frac{p \Leftrightarrow \mathsf{SWINV}(I) * \mathsf{IS}(is) * \mathsf{CS}(cs)}{\Gamma; \chi; I; r; p_i \vdash \{\ (p * [|\mathbf{sched}; \mathfrak{s}|]) \wedge \chi \triangleright x\ \}\ \mathbf{switch}\ x\ \{\ p * [|\mathfrak{s}|]\ \}} \ (\textsc{switch})$$

$$\frac{p \Rightarrow p_i}{\Gamma; \chi; I; \mathsf{false}; p_i \vdash \{\ p\ \}\ \mathbf{iext}\ \{\ \mathsf{false}\ \}} \ (\textsc{iext}) \qquad \frac{p \Rightarrow r}{\Gamma; \chi; I; r; \mathsf{false} \vdash \{\ p\ \}\ \mathbf{return}\ \{\ \mathsf{false}\ \}} \ (\textsc{ret})$$

Fig. 22. Selected Inference Rules

We build the pre-/post-conditions of interrupt handlers with the auxiliary definitions $\mathsf{BldItrpPre}$ and $\mathsf{BldItrpRet}$ given in Fig. 21. The precondition says that, when entering the level-k handler, $isr(k)$ is set to 1, the interrupt is disabled and k is pushed onto the interrupt stack is (therefore $\mathsf{OS}[isr\{k \rightsquigarrow 1\}, 0, k :: is, \mathsf{nil}]$). Since there is no handler of higher-priority in service, the handler has exclusive access to the resource $I[0, k]$ (see Fig. 19). It also needs to refine the high-level specification code $\varepsilon(k)$. empE requires there are no local variables at the beginning. The built post-condition requires that: (1) the corresponding isr bit has been cleared; (2) if interrupts are enabled ($ie = 1$), the handler has no access to the shared resources; otherwise it needs to ensure that its owned resources are well formed w.r.t. $I[0, k]$ (see the two **iret** steps in Fig. 19); and (3) there is no high-level specification code remaining to be refined (*i.e.*, the abstract specification code $\varepsilon(k)$ specified in the precondition has been fulfilled).

Similarly, we use $\mathsf{BuildAPIPre}$ and $\mathsf{BuildAPIRet}$ defined in Fig. 21 to construct the pre-/post-conditions for kernel APIs. The local states before and after calling to the API f are specified by $\mathsf{BuildP}(\mathsf{getD}(\eta_a, f), \bar{v})$ and $\mathsf{BuildAPIRet}(\eta_a, f)$, which specify the memory locations of the arguments and local variables of the API. For the internal functions, in addition to these local states, we need to add the local states specified by the functions specifications. The rules of proving $\chi; I \vdash \eta_i : \Gamma$ and $\Gamma; \chi; I \vdash \eta_a : \varphi$ are similar to the rules for interrupt handlers.

In the middle of Fig.22, we give the SEQ rule for sequential compositionality, and CONSEQ rule to strengthen and weaken the precondition and postcondition respectively. Also as in separation logic, the FRM is designed for modular reasoning, in which the side-condition requires that the framed assertion should not say anything about interrupt states and abstract statements.

Rules for commands. The IEXT rule simply requires that the post-condition p_i holds when we reach the end of the interrupt handler. The RET rule requires that the post-condition r holds when we reach the end of the non-handler function. The ENCRT rule shows the ownership transfer when interrupts are disabled. Suppose we are at the level-k handler ($k = N$ means we are executing the non-handler code). Disabling interrupts prevents interrupt requests from level 0 to $k - 1$, therefore the current task gains the ownership of $I[0, k - 1]$. The transfer of the k-th block is specified by $\mathsf{INV}(I, k)$ in Fig. 22. If the bit $isr(k)$ is 0 (or $k = N$), the task also gains the ownership of $I(k)$, otherwise it already has the ownership of the k-th block and there is no extra ownership transfer. The two scenarios are also demonstrated by the two **cli** steps in Fig. 19. If interrupts are already disabled when **encrt** is executed, there is no ownership transfer, as shown by the ENCRT-0 rule.

The EXCRT rule is the dual of the ENCRT rule (see the two **sti** steps in Fig. 19). Correspondingly there is a EXCRT-0 rule. The EOI rule says, if interrupts are enabled, the task loses the ownership of $I(k)$ after **eoi** k. Otherwise there is no ownership transfer and the corresponding rule is omitted (see the two **eoi** steps in Fig. 19).

The SWITCH rule requires that the invariant $\mathsf{SWINV}(I)$ holds before switching away and it is preserved after switching back. $\mathsf{SWINV}(I)$, defined in Fig. 22, says

that interrupts must be disabled, and all the bits of *isr* are 0 (*i.e.*, either we are running non-handler code or we are in the outmost layer of nested invocation of interrupt handlers and have already executed **eoi**). Also if we are running level-k code (either handler or non-handler if $k = N$), the resource blocks 0 to k acquired before should satisfy $I[0, k]$, so that the target task could access them. The rule also says that the task-local states *is* and *cs* are not changed by **switch**.

To establish refinement, the precondition also requires that the high-level abstract scheduler χ picks the same task with the one in x, and **switch** x at the low level correspond to the **sched** step at the high level. Therefore in the post-condition **sched** is no longer in the remaining abstract operations.

Following [24], the ABSCSQ rule looks like a regular consequence rule but allows us to consume the abstract code. The implication $p \Rrightarrow p'$ is defined below.

$$\forall \sigma, \Sigma, \mathbb{s}.(\sigma, \Sigma, \mathbb{s}) \models p \longrightarrow \exists \Sigma', \mathbb{s}'. \left((\mathbb{s}, \Sigma) \bullet\!\!-\!\!H\!\!-\!\!\triangleright^{*} (\mathbb{s}', \Sigma')\right) \wedge (\sigma, \Sigma', \mathbb{s}') \models p'$$

That is, starting from related states satisfying p, the abstract code could execute zero or multiple steps so that the resulting related states satisfy p'. This rule allows us to establish simulation between the concrete and the abstract code, which then ensures refinement. Theorem 4.11 gives the soundness of the framework. The proofs are based on a compositional simulation following [23], we will show the proof sketch in the next section, and have been formalized in Coq.

4.4 Soundness via. Simulations

In this section, we give the semantics of our logic judgments and show the proof sketch of proving the soundness of the CSL-style relational logic.

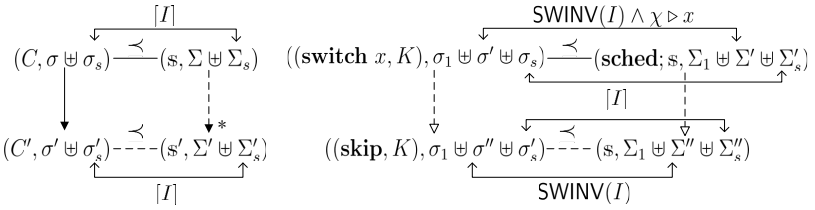


Fig. 23. Simulation Digraphs

Judgment Semantics. In this section, we show the semantics of inference rules. The semantics of $\Gamma; \chi; I; r; p_i \vdash \{\, p \,\}\ s\ \{\, q \,\}$ is defined in Def. 4.1 via a compositional simulation (defined in Def 4.2). Def. 4.3, 4.4 and 4.5 are the semantics of WFINT rule, WFAPI rule and WFFUN rule, respectively.

Our compositional simulation is defined by adapting RGSim [23]. Fig. 23 shows the main idea. We use $[I]$ (defined in Fig 24), instead of Rely/Guarantee

conditions, to specify the interleavings between interrupt handlers and non-handler code. The definition of $[I]$ follows the ownership-transfer semantics given in Fig. 19. It precisely specifies the well-formed shared resource blocks (while blocks) and allows the ownership of these blocks to be transferred in terms of switching on/off *ie* and *isr*.

For the task steps (self steps), as shown in Fig 23(a), if the shared relational state (the white blocks in Fig. 19) $(\sigma_s, \Sigma_s, \_)$ satisfies $[I]$ (defined in Fig. 24) and the low-level kernel code C can make one step, then the high-level specification code $\mathbb{s}$ could make zero-or-multiple steps with maintaining the resulting shared part $(\sigma_s', \Sigma_s', \_)$ satisfying $[I]$, and the remained low-level code C' simulates the abstract specification code $\mathbb{s}'$. By enforcing the ownership transfer semantics with $[I]$ it allows us to do compositional reasoning for interrupts.

For switch steps (environment/other steps), which make the concurrency model be different from the idealized parallel composition $C1 \parallel C2$ in the exiting theoretical work. As shown in Fig 23(b), we modularly establish the correspondence for the context switch between the two levels. The interrupt is disabled when doing context switch, based on the ownership transfer semantics, the resource blocks specified by $\mathsf{SWINV}(I)$ are owned by the current task while some of them specified by $[I]$ are shared. Note that we have $\mathsf{SWINV}(I) * [I] \Rightarrow I[0, N]$, then we know that all the resource blocks $B_0, \ldots B_{N-1}$ and A are well-formed with respect to $I[0, N]$, which ensures the safe execution of the task switched to. When switching back from another task, we also have that all the resource blocks are well-formed. To achieve the compositionality, we require there exists one particular scheduling (among all possible ones) at the high level that picks the same task as the low level, which is specified by $\chi \triangleright x$.

To support modular reasoning of internal function calls, at the entry point of a internal function, we use the function specification stored in Γ to avoid step into the function body, as shown in the **Function Steps** in Def. 4.2.

Skip and **IRet** case are used to deal with the ending of the code and other ending cases are omitted in Def. 4.2.

For each step in the kernel method, we require it to be safe.

$$
\begin{array}{ll}
\sigma \perp \sigma' & \stackrel{\text{def}}{=} \sigma = ((G, E, M), isr, \delta) \wedge \sigma = ((G, E, M'), isr, \delta) \wedge M \perp M' \\
I\{n, m\} & \stackrel{\text{def}}{=} \begin{cases} \mathsf{INV}(I, n) * \mathsf{INV}(I, n+1) * \ldots * \mathsf{INV}(I, m) & \text{if } 0 \le n \le m \le N \\ \mathsf{emp} & \text{otherwise} \end{cases} \\
[I] & \stackrel{\text{def}}{=} ((\mathsf{IE}(1) * I\{0, N\}) \vee (\mathsf{IE}(0) * (\exists\, k. \llcorner k \lrcorner * I\{k+1, N\}))
\end{array}
$$

Fig. 24. Auxiliary Definitions of Simulations

Definition 4.1 (Judgment Semantics). $\Gamma; \chi; I; r; p_i \models \{p\} s \{q\}$ holds, iff for any σ, Σ and $\mathbb{s}$, if $(\sigma, \Sigma, \mathbb{s}) \models p$, then $\Gamma; \chi; I; r; p_i; q \models ((s, (\circ, \bullet)), \sigma) \preceq (\mathbb{s}, \Sigma)$.

Definition 4.2 (Method Simulation). $\Gamma; \chi; I; r; p_i; q \models (C, \sigma) \preceq (\mathbb{s}, \Sigma)$ holds, whenever:

- **Normal Steps:** for any $P, C', \sigma_s, \Sigma_s, \sigma_1$ and σ_1', if $C \neq (\textbf{\textit{fexec}}(\_,\_),\_)$, $(\sigma_s, \Sigma_s, \_) \models [I]$, $\sigma_1 = \sigma \uplus \sigma_s$, $\Sigma \perp \Sigma_s$ and $P \vdash (C, \sigma_1) \bullet\!\!-\!\!\!-_L\!\!\!\rightarrow\!\triangleright (C', \sigma_1')$, then there exist $\Sigma_s', \mathbb{s}', \Sigma', \sigma'$ and σ_s', such that the followings hold:
  - $\sigma_1' = \sigma' \uplus \sigma_s'$, $(\sigma_s', \Sigma_s', \_) \models [I]$,
  - $(\mathbb{s}, \Sigma \uplus \Sigma_s) \bullet\!\!-\!\!\!-_H\!\!\!\rightarrow\!\!\!\!\triangleright^{*} (\mathbb{s}', \Sigma' \uplus \Sigma_s')$,
  - $\Gamma; \chi; I; r; p_i; q \models (C', \sigma') \preceq (\mathbb{s}', \Sigma')$.
- **Function Call:** for any $\sigma_s, \Sigma_s, \kappa_s, f$ and $\bar{v}$, if $C = (\textbf{\textit{fexec}}(f, \bar{v}), (\circ, \kappa_s))$, $\sigma \perp \sigma_s$, $(\sigma_s, \Sigma_s, \_) \models [I]$ and $\Sigma \perp \Sigma_s$, then there exist $\sigma_1, \sigma_f, \Sigma_1, \Sigma_f, \Sigma'$, $\Sigma_s', \mathbb{s}', fp$ and fq , such that the followings hold:
  - $\Gamma(f) = (fp, fq)$,
  - $(\mathbb{s}, \Sigma \uplus \Sigma_s) \bullet\!\!-\!\!\!-_H\!\!\!\rightarrow\!\!\!\!\triangleright^{*} (\mathbb{s}', \Sigma' \uplus \Sigma_s')$, $(\sigma_s, \Sigma_s', \mathbb{s}') \models [I]$,
  - $\sigma = \sigma_1 \uplus \sigma_f$, $\Sigma' = \Sigma_1 \uplus \Sigma_f$, $(\sigma_1, \Sigma_1, \mathbb{s}') \models fp\ (\mathsf{rev}(\bar{v}))$,
  - for any σ', σ_1', Σ'', Σ_1' and $\mathbb{s}''$, if $\sigma.m.G = \sigma'.m.G$, $\sigma.m.E = \sigma'.m.E$, $(\sigma_1', \Sigma_1', \mathbb{s}'') \models fq\ (\mathsf{rev}(\bar{v}))$, $\sigma' = \sigma_1' \uplus \sigma_f$, and $\Sigma'' = \Sigma_1' \uplus \Sigma_f$, then $\Gamma; \chi; I; r; p_i; q \models ((\textbf{skip}, (\circ, \kappa_s)), \sigma') \preceq (\mathbb{s}'', \Sigma'')$.
- **Context Switch:** for any σ_s, κ_s, x and Σ_s, if $\sigma \perp \sigma_s$, $C = (\textbf{switch}\ x, (\circ, \kappa_s))$, $(\sigma_s, \Sigma_s, \_) \models [I]$ and $\Sigma \perp \Sigma_s$, then there exist $\sigma_1, \sigma', \Sigma_1, \Sigma'$, Σ_s' and $\mathbb{s}'$, such that the followings hold:
  - $(\mathbb{s}, \Sigma \uplus \Sigma_s) \bullet\!\!-\!\!\!-_H\!\!\!\rightarrow\!\!\!\!\triangleright^{*} (\textbf{sched}; \mathbb{s}', \Sigma_1 \uplus \Sigma' \uplus \Sigma_s')$,
  - $(\sigma_s, \Sigma_s', \_) \models [I]$, $\sigma = \sigma_1 \uplus \sigma'$,
  - $(\sigma', \Sigma', \_) \models \mathsf{SWINV}(I) \wedge (\chi \triangleright x)$
  - for any σ'', σ''', Σ'' and Σ''', if $\sigma''' = \sigma_1 \uplus \sigma''$, $\Sigma''' = \Sigma_1 \uplus \Sigma''$ and $(\sigma'', \Sigma'', \_) \models \mathsf{SWINV}(I)$, then $\Gamma; \chi; I; r; p_i; q \models ((\textbf{skip}, (\circ, \kappa_s)), \sigma''') \preceq (\mathbb{s}', \Sigma''')$.
- **Skip:** for any σ_s and Σ_s, if $C = (\textbf{skip}, (\circ, \bullet))$, $\sigma \perp \sigma_s$, $(\sigma_s, \Sigma_s, \_) \models [I]$ and $\Sigma \perp \Sigma_s$, then there exist Σ', Σ_s' and $\mathbb{s}'$, such that $(\mathbb{s}, \Sigma \uplus \Sigma_s) \bullet\!\!-\!\!\!-_H\!\!\!\rightarrow\!\!\!\!\triangleright^{*} (\mathbb{s}', \Sigma' \uplus \Sigma_s')$, $(\sigma_s, \Sigma_s', \_) \models [I]$ and $(\sigma, \Sigma', \mathbb{s}') \models q$;
- **IRet:** for any κ_s, σ_s and Σ_s, if $C = (\textbf{iext}, (\circ, \kappa_s))$, $\lfloor \kappa_s \rfloor = \perp$, $\lfloor \kappa_s \rfloor_c = \perp$, $\sigma \perp \sigma_s$, $(\sigma_s, \Sigma_s, \_) \models [I]$ and $\Sigma \perp \Sigma_s$, then there exist Σ', Σ_s' and $\mathbb{s}'$, such that $(\mathbb{s}, \Sigma \uplus \Sigma_s) \bullet\!\!-\!\!\!-_H\!\!\!\rightarrow\!\!\!\!\triangleright^{*} (\mathbb{s}', \Sigma' \uplus \Sigma_s')$, $(\sigma_s, \Sigma_s', \_) \models [I]$, and $(\sigma, \Sigma', \mathbb{s}') \models p_i$;
- **ReturnE** and **Return** cases are similar to the **IRet** case, we omit them here;
- **Abort:** for any P, Σ_s, σ_s and σ', if $C \neq (\textbf{\textit{fexec}}(\_,\_),\_)$, $\sigma' = \sigma \uplus \sigma_s$, $(\sigma_s, \Sigma_s, \_) \models [I]$ and $\Sigma \perp \Sigma_s$, then $\neg(P \vdash (C, \sigma') \bullet\!\!-\!\!\!-_L\!\!\!\rightarrow\!\triangleright \textbf{abort})$.

Definition 4.3 (Well-Formed Interrupts). $\Gamma; \chi; I \models \theta : \varepsilon$ holds, iff for any k, isr, is, G, p and p_i, if $\varepsilon(k) = \mathbb{s}$, $p = \mathsf{BldItrpPre}(k, \mathbb{s}, isr, is, I)$, $p_i = \mathsf{BldItrpRet}(k, isr, is, I)$, then there exists s, such that $\theta(k) = s$ and $\Gamma; \chi; I; \mathsf{false}; p_i \models \{p\}s\{\mathsf{false}\}$

Definition 4.4 (Well-Formed APIs). $\Gamma; \chi; I \models \eta_a : \varphi$ holds, iff for any f, $\bar{v}$, ω, p and r, if $\varphi(f) = \omega$, $p = \mathsf{BuildAPIPre}(\eta_a, f, \omega, \bar{v})$, $r = \mathsf{BuildAPIRet}(\eta_a, f)$, then there exists s, such that $\eta_a(f) = (\tau, \mathcal{D}_1, \mathcal{D}_2, s)$ and $\Gamma; \chi; I; r; \mathsf{false} \models \{p\}s\{\mathsf{false}\}$.

Definition 4.5 (Well-Formed Internal Functions). $\chi; I \models \eta_i : \Gamma$ holds, iff $dom(\Gamma) = dom(\eta_i)$ and for any f, fp, fq, $\bar{v}$, p and r, if $\Gamma(f) = (fp, fq)$, $p = \mathsf{BuildFunPre}(\eta_i, f, \bar{v}, fp)$ and $r = \mathsf{BuildFunRet}(\eta_i, f, \bar{v}, fq)$ then there exists s, such that $\eta_i(f) = (\_, \_, \_, s)$ and $\Gamma; \chi; I; r; \mathsf{false} \models \{p\}s\{\mathsf{false}\}$.

Soundness Proof. To prove the soundness of our logic (Theorem 4.11), following [22], we need to define the following two simulation relations as bridges:

1. A whole-program simulation relation (defined in Def. 4.6) between concrete and abstract levels, which implies that the observable behaviors of the low level is a subset of the high level (see Lemma 4.8);
2. A task-local simulation relations (defined in Def. 4.7) between the low-level task and the high-level task, which can be composed together to ensure the whole program simulation (see Lemma 4.9). And it could be implied by the method simulation with some side conditions (see Lemma 4.10).

Definition 4.6 (Program Simulation). $(P, W) \preceq (\mathbb{P}, \mathbb{W})$ holds, whenever:

- for any W, W' and P, if $P \vdash W =\!\!=\!\!L\!\!\Rightarrow W'$, then there exist $\mathbb{W}'$, such that the followings hold:
  - $\mathbb{P} \vdash \mathbb{W} =\!\!=\!\!H\!\!\Rightarrow^{*} \mathbb{W}'$ and $W'.\Delta = \mathbb{W}'.\Delta$,
  - $(P, W') \preceq (\mathbb{P}, \mathbb{W}')$.

- for any W, W' and P, if $P \vdash W =\!\!=\!\!\overset{S}{L}\!\!\Rightarrow W'$, then there exist $\mathbb{W}'$, such that the followings hold:
  - $\mathbb{P} \vdash \mathbb{W} =\!\!=\!\!\overset{S}{H}\!\!\Rightarrow^{*} \mathbb{W}'$ and $W'.\Delta = \mathbb{W}'.\Delta$,
  - $(P, W') \preceq (\mathbb{P}, \mathbb{W}')$.

- for any W, W' and P, if $P \vdash W =\!\!=\!\!L\!\!\Rightarrow \mathbf{abort}$, then $\mathbb{P} \vdash \mathbb{W} =\!\!=\!\!H\!\!\Rightarrow^{*} \mathbf{abort}$

Definition 4.7 (Task Simulation). $P; \mathbb{P}; I; p \models (C_l, \sigma) \preceq (C_h, \Sigma)$ holds, whenever:

- **Normal Steps:** for any $C_l', \Delta, \Delta', \sigma_s, \Sigma_s, \sigma_1$ and σ_1', if $(\sigma_s, \Sigma_s, \_) \models [I]$, $\sigma_1 = \sigma \uplus \sigma_s$, $\Sigma \perp \Sigma_s$ and $P \vdash (C, \Delta, \sigma_1) \text{—}\!\!L\!\!\rightarrow (C', \Delta', \sigma_1')$, then there exist $\Sigma_s', C_h', \Sigma', \sigma'$ and σ_s', such that the followings hold:
  - $\sigma_1' = \sigma' \uplus \sigma_s'$, $(\sigma_s', \Sigma_s', \_) \models [I]$,
  - $\mathbb{P} \vdash (C_h, \Delta, \Sigma \uplus \Sigma_s) \text{—}\!\!H\!\!\rightarrow^{*} (C_h', \Delta', \Sigma' \uplus \Sigma_s')$,
  - $P; \mathbb{P}; I; p \models (C_l', \sigma') \preceq (C_h', \Sigma')$.
- **Event Steps:** for any $C_l', \Delta, \Delta', \sigma_s, \Sigma_s, \sigma_1$ and σ_1', if $(\sigma_s, \Sigma_s, \_) \models [I]$, $\sigma_1 = \sigma \uplus \sigma_s$, $\Sigma \perp \Sigma_s$ and $P \vdash (C, \Delta, \sigma_1) \text{—}\!\!\overset{S}{L}\!\!\rightarrow (C', \Delta', \sigma_1')$, then there exist $\Sigma_s', C_h', \Sigma', \sigma'$ and σ_s', such that the followings hold:
  - $\sigma_1' = \sigma' \uplus \sigma_s'$, $(\sigma_s', \Sigma_s', \_) \models [I]$,
  - $\mathbb{P} \vdash (C_h, \Delta, \Sigma \uplus \Sigma_s) \text{—}\!\!\overset{S}{H}\!\!\rightarrow^{*} (C_h', \Delta', \Sigma' \uplus \Sigma_s')$,
  - $P; \mathbb{P}; I; p \models (C_l', \sigma') \preceq (C_h', \Sigma')$.
- **Context Switch:** for any $\Delta, \sigma_s, \kappa_s, x$ and Σ_s, if $\sigma \perp \sigma_s$, $C = (\mathbf{switch}\ x, (\circ, \kappa_s))$, $(\sigma_s, \Sigma_s, \_) \models [I]$ and $\Sigma \perp \Sigma_s$, then there exist $\sigma_1, \sigma', \Sigma', \Sigma_1, \Sigma_s', \chi$, $\mathbb{s}'$ and K, such that the followings hold:
  - $\mathbb{P} = (\_, (\_, \_, \chi))$
  - $\mathbb{P} \vdash (C_h, \Delta, \Sigma \uplus \Sigma_s) \text{—}\!\!H\!\!\rightarrow^{*} ((\mathbf{sched}; \mathbb{s}', K), \Delta, \Sigma_1 \uplus \Sigma' \uplus \Sigma_s')$,
  - $(\sigma_s, \Sigma_s', \_) \models [I]$, $\sigma = \sigma_1 \uplus \sigma'$,

- $(\sigma', \Sigma', \_) \models \mathsf{SWINV}(I) \wedge (\chi \triangleright x)$
- for any σ'', σ''', Σ'' and Σ''', if $\sigma''' = \sigma_1 \uplus \sigma''$, $\Sigma''' = \Sigma_1 \uplus \Sigma''$ and $(\sigma'', \Sigma'', \_) \models \mathsf{SWINV}(I)$, then $P; \mathbb{P}; I; p \models ((\mathbf{skip}, (\circ, \kappa_s)), \sigma''') \preceq ((\mathbb{s}', K), \Sigma''')$.

– **Skip:** for any Δ, σ_s and Σ_s, if $C = (\mathbf{skip}, (\circ, \bullet))$, $\sigma \perp \sigma_s$, $(\sigma_s, \Sigma_s, \_) \models [I]$ and $\Sigma \perp \Sigma_s$, then there exist Σ', Σ'_s, such that $(\sigma_s, \Sigma'_s, \_) \models [I]$, $(\sigma, \Sigma', \_) \models p$ and $\mathbb{P} \vdash (C_h, \Delta, \Sigma \uplus \Sigma_s) \xrightarrow{H}{}^{*} ((\mathbf{skip}, (\circ, \bullet)), \Delta, \Sigma' \uplus \Sigma'_s)$, .

– **Abort:** for any Δ, Σ_s, σ_s and σ', if $\sigma' = \sigma \uplus \sigma_s$, $(\sigma_s, \Sigma_s, \_) \models [I]$ and $\Sigma \perp \Sigma_s$ and $P \vdash (C_l, \Delta, \sigma') \xrightarrow{L} \mathbf{abort}$, then $\mathbb{P} \vdash (C_h, \Delta, \Sigma \uplus \Sigma_s) \xrightarrow{H}{}^{*} \mathbf{abort}$.

Lemma 4.8 (ProgSim Implies Event Trace Refinement). For any P, $\mathbb{P}, W$ and $\mathbb{W}$, if $(P, W) \preceq (\mathbb{P}, \mathbb{W})$, then $(P, W) \preccurlyeq (\mathbb{P}, \mathbb{W})$.

Lemma 4.9 (Compositionality). For any η_a, η_i, θ, φ, ε, χ, ψ, T_l, T_h, Δ, t_c and Σ, if the followings holds:

– $P = (A, (\eta_a, \eta_i, \theta))$, $\mathbb{P} = (A, (\varphi, \varepsilon, \chi))$
– $\mathsf{Match}(\psi, (T_l, \Lambda, \Delta, t_c), (T_h, \Sigma, \Delta))$, $\Lambda = ((G, \Pi, M), isr, \pi)$
– $\Gamma; \chi; I \models \theta : \varepsilon, \quad \chi; I \models \eta_i : \Gamma, \quad \chi; I \models \eta_i : \Gamma$
– $T_l = \{t_1 \rightsquigarrow C_{l1}, \ldots, t_n \rightsquigarrow C_{ln}\}$, $T_h = \{t_1 \rightsquigarrow C_{h1}, \ldots, t_n \rightsquigarrow C_{hn}\}$
– $M = M_1 \uplus M_2 \uplus \cdots \uplus M_n \uplus M_s$, $\Sigma = \Sigma_1 \uplus \Sigma_2 \uplus \cdots \uplus \Sigma_n \uplus \Sigma_s$
– $\Lambda|_{t_c} = \sigma_c$, $(\sigma_c \lhd M_s, \Sigma_s, \_) \models [I]$
– $P; \mathbb{P}; I; p \models (C_{lc}, \sigma_c \lhd M_c) \preceq (C_{hc}, \Sigma_c)$
– for any $i, \sigma_i, \sigma_r, \Sigma_r$, $i \neq c$, $\sigma_i = (\Lambda|_{t_i}) \lhd M_i$, $(\sigma_r, \Sigma_r, \_) \models \mathsf{SWINV}(I)$, $\sigma_i \perp \sigma_r$, $\Sigma_i \perp \Sigma_r$, then $P; \mathbb{P}; I; p \models (C_{li}, \sigma_i \uplus \sigma_r) \preceq (C_{hi}, \Sigma_i \uplus \Sigma_r)$

then $(P, (T_l, \Lambda, \Delta, t_c)) \preceq (\mathbb{P}, (T_h, \Sigma, \Delta))$.

Lemma 4.10. For any s, P, $\mathbb{P}$, I,

– $P = (A, (\eta_a, \eta_i, \theta))$, $\mathbb{P} = (A, (\varphi, \varepsilon, \chi))$
– $(\sigma, \Sigma, \_) \models \mathsf{OS}[\bar{0}, 1, \mathsf{nil}, \mathsf{nil}]$
– $\Gamma; \chi; I \models \theta : \varepsilon, \quad \chi; I \models \eta_i : \Gamma, \quad \chi; I \models \eta_i : \Gamma$

then $P; \mathbb{P}; I; \mathsf{OS}[\bar{0}, 1, \mathsf{nil}, \mathsf{nil}] \models ((s, (\circ, \bullet)), \sigma) \preceq ((s, (\circ, \bullet)), \Sigma)$.

Here we present the proof sketch of Theorem 4.11, and omit the proofs of other lemmas. The complete proofs can be found in our Coq proof [3]. Note the co-inductive proofs done in Coq for the soundness theorem are non-trivial.

Theorem 4.11 (Soundness).

1. $\Gamma; \chi; I; r; p_i \vdash \{\, p \,\}\, s\, \{\, q \,\} \Longrightarrow \Gamma; \chi; I; r; p_i \models \{p\} s \{q\}$
2. $\Gamma; \chi; I \vdash \theta : \varepsilon \Longrightarrow \Gamma; \chi; I \models \theta : \varepsilon$
3. $\Gamma; \chi; I \vdash \eta_a : \varphi \Longrightarrow \Gamma; \chi; I \models \eta_a : \varphi$
4. $\chi; I \vdash \eta : \Gamma \Longrightarrow \chi; I \models \eta : \Gamma$
5. $\vdash_\psi O : \mathbb{O} \Longrightarrow O \sqsubseteq_\psi \mathbb{O}$

Proof: Theorems 1 - 4 are proved in Lemmas 4.12-4.15. Here we focus on the proof of TopRule. Suppose $O = (\eta_a, \eta_i, \theta)$ and $\mathbb{O} = (A, (\varphi, \varepsilon, \chi)$, then from the definition of $O \sqsubseteq_\psi \mathbb{O}$, we need to prove that:

for any $A,\ \Lambda,\ \Sigma, \Delta,\ t, T$, if $\mathsf{Match}(\psi, (T, \Delta, \Lambda, t), (T, \Delta, \Sigma))$ then

$$((A, O), W) \preccurlyeq ((A, \mathbb{O}), \mathbb{W})$$

then from Lemma 4.8, we need prove that

$$((A, O), W) \preceq ((A, \mathbb{O}), \mathbb{W})$$

from the definition of Match we know there exists $T, \Delta, t, \Lambda, \Sigma$ such that,

$$W = (T, \Delta, \Lambda, t_c) \quad \mathbb{W} = (T, \Delta, \Sigma) \quad (\psi\ \Lambda\ \Sigma)$$

suppose that $\Lambda = ((G, \Pi, M), isr, \pi)$, then from Lemma 4.9, we need to prove that there exists $M_s,\ \Sigma_s,\ M_1,\ \Sigma_1,\ M_2,\ \Sigma_2,\ \ldots,\ M_n,\ \Sigma_n,\ C_{lc},\ C_{hc}$ such that:

(1) $\Gamma; \chi; I \models \theta : \varepsilon \quad \chi; I \models \eta_i : \Gamma \quad \chi; I \models \eta_i : \Gamma$

(2) $M = M_1 \uplus M_2 \uplus \cdots \uplus M_n \uplus M_s,\ \Sigma = \Sigma_1 \uplus \Sigma_2 \uplus \cdots \uplus \Sigma_n \uplus \Sigma_s$

(3) $\Lambda|_{t_c} = \sigma_c,\ (\sigma_c \lhd M_s, \Sigma_s, \_) \models [I]$

(4) $P; \mathbb{P}; I; p \models (C_{lc}, \sigma_c \lhd M_c) \preceq (C_{hc}, \Sigma_c) \quad T_l(t_c) = C_{lc} \quad T_h(t_c) = C_{hc}$

(5) for any $i,\ \sigma_i,\ \sigma_r,\ \Sigma_r,\ C_{li},\ C_{hi},\ i \neq c,\ T_l(t_i) = C_{li},\ T_h(t_i) = C_{hi}\ \sigma_i = (\Lambda|_{t_i}) \lhd M_i,\ (\sigma_r, \Sigma_r, \_) \models \mathsf{SWINV}(I),\ \sigma_i \perp \sigma_r,\ \Sigma_i \perp \Sigma_r$, then $P; \mathbb{P}; I; p \models (C_{li}, \sigma_i \uplus \sigma_r) \preceq (C_{hi}, \Sigma_i \uplus \Sigma_r)$

where $((G, E, M), isr, \delta) \lhd M' \stackrel{\text{def}}{=} ((G, E, M'), isr, \delta)$. From TopRule and Lemma 4.12 - 4.15, we can trivially know that (1) holds, then from TopRule we know that

$$\lfloor \psi \rfloor \Rightarrow I[0, N] * \mathsf{OS}[\bar{0}, 1, \mathsf{nil}, \mathsf{nil}] * \mathsf{empE} \qquad (1)$$

then from the semantics of the assertion we know that exist

$$M_1 = M_2 = \cdots = M_n = \emptyset, \quad \Sigma_1 = \Sigma_2 = \cdots = \Sigma_n = \emptyset$$

and

$$M_s = M, \quad \Sigma_s = \Sigma$$

such that (2) and (3) hold.

Then from the definition of Match, we know that

$$\forall i, \exists s_i, C_{li} = C_{hi} = (s_i, (\circ, \bullet))$$

Then from Lemma 4.10 we know that (4) holds. From the definition $\mathsf{SWINV}(I)$ and (I) we know that, $((\Lambda|_{t_i}) \lhd \emptyset, \emptyset, \_) \models \mathsf{SWINV}(I)$ and we know that $\Lambda|_{t_i} = (\Lambda|_{t_i}) \uplus ((\Lambda|_{t_i}) \lhd \emptyset)$, then for (5), we only need to prove that for any i, if $i \neq c$, then

$$P; \mathbb{P}; I; p \models (C_{li}, \Lambda|_{t_i}) \preceq (C_{hi}, \Sigma_i)$$

and it can be easily proved by Lemma 4.10.

Lemma 4.12. For all $\Gamma,\ \chi,\ I,\ r,\ p_i,\ p,\ s$ and q, $\Gamma;\chi;I;r;p_i \vdash \{\, p \,\}\, s\, \{\, q \,\} \Longrightarrow \Gamma;\chi;I;r;p_i \models \{p\}s\{q\}$

Proof: First induction over the inference rules, for the compositional cases, like while, we prove them by co-induction. The proof details can be found in [3].

Lemma 4.13. For all $\Gamma,\ \chi,\ I,\ \theta$ and ε, $\Gamma;\chi;I \vdash \theta : \varepsilon \Longrightarrow \Gamma;\chi;I \models \theta : \varepsilon$

Proof: By Def. 4.3 and Lemma 4.12.

Lemma 4.14. For all $\Gamma,\ \chi,\ I,\ \eta_a$ and φ, $\Gamma;\chi;I \vdash \eta_a : \varphi \Longrightarrow \Gamma;\chi;I \models \eta_a : \varphi$

Proof: By Def. 4.4 and Lemma 4.12.

Lemma 4.15. For all $\Gamma,\ \chi,\ I$ and η, $\chi;I \vdash \eta : \Gamma \Longrightarrow \chi;I \models \eta : \Gamma$

Proof: By Def. 4.5 and Lemma 4.12.

4.5 Coq Tactics

We develop a set of practical tactics in Coq based on the rules of the refinement logic, including "sep auto" for proving relational assertions, "hoare forward" for proving refinement judgments, and some domain-specific tactics for proving the properties of integers and bitmaps. Here we omit the implementation details which can be seen in [3].

To demonstrate the efficiency of our tactics, we make two versions of proofs [3] for the API (OSTimeGet), which has only 4 lines code and uses a critical region to read the global clock and return its value. The one using our tactics only needs 11 lines of proof scripts, while the other one using the primitive tactics provided by Coq needs more than 400 lines.

Another advantage of our tactics is that they can help us to smoothly reason about the spatial parts and extract lemmas that are independent of program contexts for verifying functionality of kernels. Users with little knowledge about our framework can prove the code with our tactics.

Because all the APIs are proved in the similar procedure as shown in Fig. 25, first we initialize specifications with the tactics "init spec", and we obtain the Hoare triple with pre- and post- conditions instantiated, next we apply the "hoare forward" tactic step by step till the point before exiting the critical region. Then the high-level specification code must be executed to reestablish the invariant over the related states and exit. Before that we may need to derive the necessary premises for safely executing the specification code in terms of the states satisfying invariants when entering the critical section. By applying the tactic "hoare abscsq", it allows us to change the precondition by applying a lemma of proving the safe execution of the current specification code with one-or-multiple steps. Note that the initial specification code usually contains many statements of non-deterministic choice, and it is users' responsibility to choose a correct branch for accomplishing the proofs. The tactic "sep auto" is used to solve the generated side conditions about entailment of relational assertions.

```
L1   OS_ENTER_CRITICAL();
L2   ticks = OSTime;
L3   OS_EXIT_CRITICAL();
L4   return (ticks);
```

```
Lemma OSTimeGetRight: forall r p,
Some r = BuildRetA' api TimeGet tmgetspec nil ->
Some p = BuildPreA' api TimeGet tmgetspec nil ->
exists t d1 d2 s, api TimeGet = Some (t, d1, d2, s)/\
{|F, X, I, r, Afalse|} |- {{p}} s {{Afalse}}.
Proof.
init spec.        (*Initialize Specification*)
hoare forward.    (*Forward L1*)
hoare unfold pre. (*Prepare the precondition*)
hoare forward.    (*Forward L2 *)
hoare abscsq.     (*The high-level step*)
eapply OSTimeGet_high_level_step; pauto.
hoare forward.    (*Forward L3*)
unfold AOSTime.
sep auto.         (*Sovle side conditions*)
pauto.
hoare forward.    (*Forward L4*)
Qed.
```

Fig. 25. Verifying OSTimeGet() with Tactics

Then the major proof efforts lie in proving the extracted lemmas, which are related to the functional correctness of kernel APIs. For instance, μC/OS-II supports 64 tasks with priorities form 0 to 63, and the scheduling algorithm calculates the highest priority of ready tasks using bit operations over the ready table, and we need to prove the lemma like "$0 \leq x < 64 \rightarrow 0 \leq (x \gg 3) < 8$", which involves bit operations over integers limited in small finite domains, such as 0-8, 0-64 and 0-256. Most of the mathematical properties required by μC/OS-II are limited with small finite domains like this. We develop a domain-specific tactic called "mauto" to automatically prove these mathematical properties. "mauto" is implemented by brute force iterating all the possible inputs and solving each subgoal with "omega".

5 Proving Priority-Inversion-Freedom

Definition of priority-inversion-freedom (PIF). PIF is an important property in real-time systems, but there is no standard formal definition for it. Although there have been efforts trying to formalize and verify it, their definitions all have serious problems. For instance, Def. 5.1 is formal definition of priority inversion given in earlier work [6]. It says priority inversion occurs if there is a higher priority task t waiting directly or indirectly for a lower priority task t'.

Definition 5.1 (Priority Inversion). PI(Σ) holds, iff there exist t and t' such that $t \xrightarrow{\Sigma}{}^{+} t'$ and $\mathsf{CurPr}(t', \Sigma) \prec \mathsf{CurPr}(t, \Sigma)$.

Here the waiting chain $t \xrightarrow{\Sigma}{}^{+} t'$ is the transitive closure of the waiting relation $t \xrightarrow{\Sigma} t'$, saying t waits for the resource owned by t'. $\mathsf{CurPr}(t, \Sigma)$ returns the current priority of t. Then PIF can be simply defined as the negation of PI(Σ) over any state Σ on the execution sequence.

Although it looks simple and intuitive, it cannot be applied to classic real-world algorithms for PIF, *e.g.*, priority ceiling and priority inheritance [28], because they need to dynamically change the priority of tasks. Since the definition refers to the current priority of tasks, its meaning crucially depends on the algorithms, which becomes difficult to understand.

A reasonable definition must be based on the original priorities assigned by the programmer, reflecting the actual degree of urgency. Below we give a new definition for PIF.

Definition 5.2 (Priority Inversion Freedom). PIF(Σ) holds, iff for any t, t_c, pr and pr_c, if $t \neq t_c$, $t_c = \mathsf{CurTask}(\Sigma)$, $pr = \mathsf{OrgPr}(t, \Sigma)$, $pr_c = \mathsf{OrgPr}(t_c, \Sigma)$, $\mathsf{IsWait}(t, \Sigma)$ and $\neg\mathsf{IsOwner}(t_c, \Sigma)$, then $pr \preceq pr_c$.

It says, if the current task t_c does not own any shared resources, then its *original* priority should be higher than (or equal to) any other waiting tasks t. Here $\mathsf{OrgPr}(t, \Sigma)$ represents t's original priority configured by users. $\mathsf{IsWait}(t, \Sigma)$ is defined as $\exists t'.t \xrightarrow{\Sigma}{}^{+} t'$, and $\neg\mathsf{IsOwner}(t_c, \Sigma)$ means that the task t_c does not own any shared resources (*e.g.*, mutexes).

The definition essentially says that the current task can have a lower priority than waiting ones only if it holds resources that might be needed by others (so we want to run it first to make the resource available as soon as possible). Note that if each task eventually releases its shared resource (*i.e.*, there is no deadlock), the waiting task with higher priority will be eventually released and executed. Therefore the definition prevents unbounded priority inversion [28].

We demonstrate the difference between the two notions by showing in Fig. 26 an example violating PIF. It is constructed based on the mutex implementation in μC/OS-II, which is implemented with a simplified priority ceiling protocol [28] and cannot prevent priority inversions when there is nested use of mutexes. The counterexample justifies this limitation of nested mutex PIF failure of μC/OS-II.

Before going through the example, we first explain the protocol used for μC/OS-II mutex. It provides two APIs to acquire ($\mathsf{OSMutexPend}(S)$) and release ($\mathsf{OSMutexPost}(S)$) the mutex, $P(S)$ and $V(S)$ for short. Each mutex S has a unique priority, and it saves its owner task's identifier and current priority when S is acquired. When a task t executes $P(S)$, it executes the following steps: (1) t's current priority must be lower than that of S; (2) if S is available, t successfully acquires S, sets S's owner to t, and saves t's current priority in S for the future recovery; and (3) t is blocked if S is already owned by a another task t'. If t's current priority is higher than that of t', we lift the current priority of t' to S's priority. We omit the behavior of $V(S)$ here, which is not needed to understand the counterexample.

In Fig. 26, we show four tasks (A, B, C and D) and two mutexes (S_1 and S_2). They all have their original priorities configured by users, following the order of $Pr_D < Pr_C < Pr_{S_2} < Pr_B < Pr_A < Pr_{S_1}$. Suppose A, B and C are blocked and waiting for the timer interrupt to wake up, and D is the only running task. First D acquires S_2, then C wakes up and preempts D. C acquires S_1, and then attempts to acquire S_2 owned by D, thus gets blocked. Since $Pr_D < Pr_C$, following the aforementioned protocol we lift D's priority to that of S_2 (Pr_{S2}). Then A wakes up and becomes the highest priority task to run. It tries to acquire S_1 (now owned by C) and gets blocked. Then we lift C's priority to that of S_1 (Pr_{S_1}). Finally B becomes ready and has the highest priority to run. In each step we also show the pairs of the current priority and the task status for corresponding tasks. Now we have:

- **Violation of the old definition:** C waits for D, but C's current priority Pr_{S_1} is higher than that of D (Pr_{S_2}), shown in the dashed box in Fig.26;
- **Violation of our new definition:** B does not own any mutexes, and the waiting task A's original priority Pr_A is higher than that of B (Pr_B), shown in the solid box.

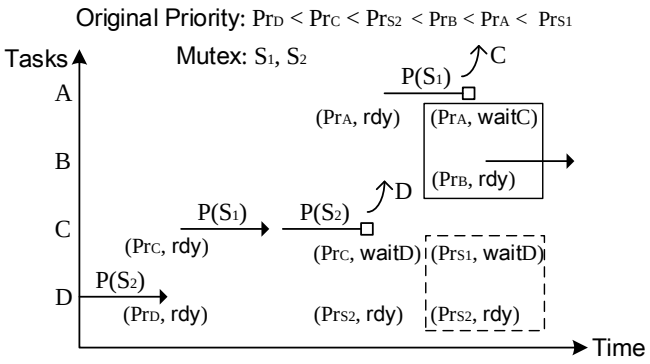


Fig. 26. Violating PIF with Mutex in μC/OS-II

In this example, unlike C and D that are going to release mutexes in bounded number of steps, B does not own any mutexes and may run forever, therefore the most urgent task A may never get a chance to run. This situation is called unbounded priority inversion [28] and should be forbidden in real-time systems. On the other hand, if we only have A, C and D, A only needs to wait for D and C to release their mutexes before it is unblocked. There is no unbounded priority inversion. This case is allowed in our new definition, but disallowed in the old one (as shown in the dashed box).

The above counterexample happens because the priority lifting implemented in μC/OS-II fails to consider nested use of mutexes. Below we show it could satisfy PIF if we disallow nested use of mutexes.

Proving PIF of Mutex in μC/OS-II. We prove that the mutex in μC/OS-II ensures both the old and our new PIF definitions if there are no nested use of mutexes. Some auxiliary definitions are given in Fig.27 based on the mutex

$$\mathsf{CurTask}(\Sigma) \stackrel{\text{def}}{=} \Sigma(\mathsf{tcbls})(\mathsf{ctid}) \qquad \mathsf{CurPr}(t,\Sigma) \stackrel{\text{def}}{=} \begin{cases} pr \text{ if } \Sigma(\mathsf{tcbls})(t) = (pr, \_) \\ \bot \quad \text{otherwise} \end{cases}$$

$$\mathsf{Own}(t, eid, \Sigma) \stackrel{\text{def}}{=} t \in dom(\Sigma(\mathsf{tcbls})) \wedge \Sigma(\mathsf{ecbls})(eid) = (\_, (t, \_))$$

$$\mathsf{IsOwner}(t,\Sigma) \stackrel{\text{def}}{=} \exists eid.\mathsf{Own}(t, eid, \Sigma) \quad \mathsf{OrgPr}(t,\Sigma) \stackrel{\text{def}}{=} \begin{cases} pr \text{ if } \exists eid.\mathsf{Own}(t, eid, \Sigma) \wedge \\ \qquad \Sigma(\mathsf{ecbls})(eid) = (\_, (\_, pr)) \\ pr \text{ if } (\neg \mathsf{IsOwner}(t,\Sigma)) \wedge \\ \qquad \Sigma(\mathsf{tcbls})(t) = (pr, \_) \\ \bot \quad \text{otherwise} \end{cases}$$

$$\mathsf{Wait}(t, eid, \Sigma) \stackrel{\text{def}}{=} \Sigma(\mathsf{tcbls})(t) = (\_, \mathsf{wait}(\mathsf{mtx}(eid), \_))$$

$$t \xrightarrow{\Sigma} t' \qquad \stackrel{\text{def}}{=} \exists eid.\mathsf{Wait}(t, eid, \Sigma) \wedge \mathsf{Own}(t', eid, \Sigma) \wedge t \neq t'$$

$$\mathsf{IsWait}(t,\Sigma) \stackrel{\text{def}}{=} \exists t'.t \xrightarrow{\Sigma} t' \qquad \mathsf{IsRdy}(t,\Sigma) \stackrel{\text{def}}{=} \Sigma(\mathsf{tcbls})(t) = (\_, \mathsf{rdy})$$

Fig. 27. Auxiliary Definitions for PIF

implementation of μC/OS-II. Here we present Theorem 5.3 based on our PIF definition.

Theorem 5.3 (PIF without Nested Use of Mutexes). If $\mathsf{Init}(\Sigma)$, $(A, \mathbb{O}_{\mu\text{C/OS-II}}) \vdash (T, \Delta, \Sigma) =\!\!=\!\!\Rightarrow^{*} (T', \Delta', \Sigma')$, $\mathsf{NoNCR}(A, \Sigma, T, \Delta)$, and $\mathsf{SchedProp}(\Sigma')$, then $\mathrm{PIF}(\Sigma')$. Where $\mathsf{Init}(\Sigma) \stackrel{\text{def}}{=} \forall t \in dom(\Sigma(\mathsf{tcbls})).\neg\mathsf{IsWait}(t,\Sigma)$.

It says, for any application code A, client state Δ and kernel abstract state Σ, if initially there are no tasks waiting for mutexes ($\mathsf{Init}(\Sigma)$), there is no nested use of mutexes in the program ($\mathsf{NoNCR}(A, \Sigma, T, \Delta)$), then for any T', Δ' and Σ' generated during the execution, if Σ' is consistent with the priority-based scheduling ($\mathsf{SchedProp}(\Sigma')$), then it must satisfy PIF.

The following definition $\mathsf{NoNCR}(\eta_c, \Sigma, T, \Delta)$ is semantically defined by excluding program states that the execution of nested use of mutexes might get to.

Definition 5.4 (No Nested). $\mathsf{NoNCR}(\eta_c, \Sigma, T, \Delta)$ holds, iff for any Σ', T' and Δ', if $(A, \mathbb{O}_{\mu\text{C/OS-II}}) \vdash (T, \Delta, \Sigma) =\!\!=\!\!\Rightarrow^{*} (T', \Delta', \Sigma')$, then $\mathsf{NNest}(\Sigma')$ defined as below.

$$\mathsf{NNest}(\Sigma) \stackrel{\text{def}}{=} \forall t, eid.(t \in dom(\Sigma(\mathsf{tcbls})) \wedge \mathsf{Own}(t, eid, \Sigma)) \rightarrow \neg(\mathsf{IsWait}(t,\Sigma) \vee \exists eid'.eid' \neq eid \wedge \mathsf{Own}(t, eid', \Sigma))$$

$\mathsf{SchedProp}(\Sigma')$ given in Def.5.5 requires that the current running task always has the highest priority among all the ready tasks. It can be guaranteed by the highest-priority-based scheduling strategy of μC/OS-II. Here we use a simplified $\mathbb{O}_{\mu\text{C/OS-II}}$ that contains the PIF mutex as the only APIs. $\mathsf{IsRdy}(t,\Sigma)$ means that the task t is ready in Σ. The proof is formalized in Coq.

Definition 5.5 (Highest-Priority-Based Scheduling).
$\mathsf{SchedProp}(\Sigma)$ holds, iff for any t_c and pr_c, if $t_c = \mathsf{CurTask}(\Sigma)$ and $pr_c = \mathsf{CurPr}(t_c, \Sigma)$, then $\mathsf{IsRdy}(t_c, \Sigma)$ and for any t and pr, if $t \neq t_c$, $pr = \mathsf{CurPr}(t, \Sigma)$ and $\mathsf{IsRdy}(t, \Sigma)$, then $pr \preceq pr_c$.

6 Verifying μC/OS-II

We have applied our framework to verify key modules (around 1300 lines of C code without counting comments and empty lines) of μC/OS-II V2.52, including the scheduler, the timer interrupt handler, mutexes, message queues, mail boxes, semaphores, and the time management. The modules verified in our framework cover 65% of the frequently used APIs [2]. We ignore some synchronization APIs which have similar functionality as the verified ones. Verification of task creation/deletion is still ongoing work based on the presented framework.

Modifications to the original code. Our verification is based on the original code with some minor modifications. For instance, the API OSQPend(S) is used to receive a message from a queue, and its original code does not check if the input pointer S points to a valid data structure, because they assume that users always get S by calling OSQCreate() (S should already be valid there). We drop this assumption about client code. Correspondingly we add some code checking S to ensure we get a valid pointer. If S is invalid we return a new error code. Similar modifications are made to some other modules too.

Framework	**Coq lines**
Basic Libraries	12235
Machine & Logic	42903
Automated Tactics	21050
Total	76188

Certified μC/OS-II	**Coq lines**
C Code Definitions	1823
Specifications	6332
Priority Inversion Freedom	9467
Libraries for μC/OS-II	62490
Auto. Generated Code	25357
Total	105469

Verified Modules	**lines of C**	**Coq lines**
Global Declarations	187	
Message Queue	240	4537
Semaphore	166	2441
Mailbox	171	3325
Mutex	301	17331
Time Management	39	612
Timer Interrupt	17	443
Internal Functions	195	5447
Total	1316	34172

Table 2. The Verification Package

Proof efforts. To formalize the work in this paper, we develop around 215,000 lines of proofs in Coq8.4pl6. Table 2 gives a break down of the number of lines for various components. Compiling the entire Coq package takes around 16 hours on a machine with 3.6GHz cpu and 32G memory. It takes us around 5.5 person years in total, out of which 4 person years are for the framework, and 1 person year for verifying the first module (Message Queue) of the kernel. With the facilities (tactics, libraries and invariants *etc.*) being stabilized, verifying the remaining modules (around 900 lines of C code) only takes us around 6 person months.

The most challenging part is to verify the timer interrupt handler, which traverses the entire TCB list and updates task status in each TCB block. It needs to access all the shared data structures in μC/OS-II. Several different updates to shared data structures make the loop invariant quite complicated.

Also verifying an existing OS kernel is more difficult than verifying a new one written for verification purpose. When verifying μC/OS-II the major difficulty comes from the gap between the low-level concrete data structure and the high-level abstract representation. For instance, μC/OS-II uses a smart bitmap algorithm to record whether a task is in the waiting queue. The implementation requires us to establish a subtle consistency relation between the low-level bitmap and the high-level abstract waiting queue. The verification would have been much simpler if the waiting queue is simply implemented as a linked list.

7 Related Work and Conclusion

OS kernel verification. There have been a number of OS verification projects, including Spin [7], VeriSoft [4], seL4 [19, 18], Verve [32], and CertiKOS [15]. As far as we know, none of them except Verve [32] has verified preemptive kernels with multi-level interrupts.

seL4 [19, 18] is one of the milestone OS kernel verification projects. The kernel of seL4 is sequential, which cannot be interrupted. In addition, in their high-level model, their scheduler is unspecified, and it is allowed to pick any ready task to run, therefore their high-level model cannot be used to specify and verify scheduling properties as we do. On the other hand, the kernel of μC/OS-II does not have virtual memory, which has been certified in seL4. Gu *et al.* [15] verify the CertiKOS hypervisor. Their kernel is sequential too. It has been compiled with a certified compiler CompCertX, an extension of CompCert [21]. It might be possible to reuse and extend CompCertX or Compositional CompCert [29] to compile our kernel too, which we leave as future work.

Verve [32] combines a type-safe kernel with a minimal hardware abstraction layer verified using the Z3 SMT solver. The kernel is concurrent, but the properties verified are mostly about type safety, much weaker than our contextual refinement property. Feng *et al.* [11, 12] verified a small thread library with hardware interrupts and preemption using a variant of CSL. We follow their ownership transfer semantics and extend it for multi-level interrupts, but the contextual refinement property we establish is stronger than their notion of correctness. Gotsman *et al.* [13, 14] developed a program logic based on C-SL, which decomposes the verification of preemptive kernels into verifying the scheduler and the tasks. Their proofs are on-paper only and not mechanized. The machine model does not support multi-level interrupts, also their program logic is used to prove partial correctness, not contextual refinement as ours.

Verifying priority-inversion-freedom. There exist two classic protocols (priority inheritance and priority ceiling) [28] that are used to guarantee priority-inversion-freedom of real-time systems. Dutertre [9] gives a formal specification and analysis for the priority ceiling protocol in PVS, and Zhang *et al.* [33] use Isabelle/HOL to verify the correctness of priority inheritance protocol. All the properties they proved are defined based on current priorities, suffering from the same problem with Def. 5.1. We give a new formalization of PIF based on the original priority, therefore it is possible to verify the two different protocols in a

uniform way. Also their proofs are done at the model level only, while our refinement proofs can ensure the low-level C code guarantees PIF when it is verified at the high level.

Limitations of our framework. The major limitations include: (1) Our framework does not support higher-order functions; (2) the assertion language may not be expressive enough to write global invariants for specifying complex concurrent protocols. For the first limitation, we may incorporate the refinement logic for higher-order concurrency proposed in [30] into the framework. For the second one, we can introduce fractional permissions as a remedy, or use rely/guarantee style specifications.

Conclusion. We have developed a practical verification framework for general verification purpose of preemptive OS kernels with multi-level interrupts. Correctness of the OS kernel is formalized as a contextual refinement between the low-level concrete implementations and the high-level specifications. As far as we know, our work is the first to establish contextual refinement for primitives of a preemptive OS kernel. We have applied the framework to verify key modules and PIF of μC/OS-II, a commercial embedded real-time OS.

References

[1] The coq development team: The coq proof assistant. http://coq.inria.fr.

[2] The real-time kernel: μC/OS-II. `http://micrium.com/rtos/ucosii/overview/`.

[3] A practical verification framework for preemptive OS kernels (Coq implementations), January 2016. `https://sites.google.com/site/certiucos/source.zip`.

[4] E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban. Balancing the load: Leveraging semantics stack for systems verification. *J. Autom. Reason.*, 42, Numbers 2-4:389–454, 2009.

[5] A. W. Appel. Tactics for separation logic, 2006. `http://www.cs.princeton.edu/~appel/papers/septacs.pdf`.

[6] O. Babaoglu, K. Marzullo, and F. B. Schneider. A Formalization of Priority Inversion. *Real-time Systems*, 5:285–303, 1993.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the spin operating system. In *Proc. SOSP'95*, pages 267–284, 1995.

[8] J. Cao, M. Fu, and X. Feng. Practical tactics for verifying C programs in coq. In *Proc. CPP'15*, pages 97–108, 2015.

[9] B. Dutertre. The Priority Ceiling Protocol: Formalization and Analysis Using PVS. In *Proc. RTSS'00*, pages 151–160, 2000.

[10] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity os. In *Proc. EuroSys '06*, pages 177–190, 2006.

[11] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI*, pages 170–182, New York, NY, USA, 2008. ACM.

[12] X. Feng, Z. Shao, Y. Guo, and Y. Dong. Certifying low-level programs with hardware interrupts and preemptive threads. *J. Autom. Reasoning*, 42(2-4):301–347, 2009.

[13] A. Gotsman and H. Yang. Modular verification of preemptive os kernels. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 404–417, 2011.

[14] A. Gotsman and H. Yang. Modular verification of preemptive OS kernels. *J. Funct. Program.*, 23(4):452–514, 2013.

[15] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *POPL*, pages 595–608, New York, NY, USA, 2015. ACM.

[16] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650, 2015.

[17] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, June 2010.

[18] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2, 2014.

[19] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *SOSP*, pages 207–220, 2009.

[20] J. J. Labrosse. *MicroC/OS-II: The Real Time Kernel*. CMP Books, 2002.

[21] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[22] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470, 2013.

[23] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, 2012.

[24] H. Liang, X. Feng, and Z. Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *CSL-LICS*, 2014.

[25] A. McCreight. Practical tactics for separation logic. In *Proceedings of Theorem Proving in Higher Order Logics (TPHOLs'09)*, pages 343–358, 2009.

[26] P. W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, pages 49–67, 2004.

[27] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013.

[28] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39:1175–1185, 1990.

[29] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional compcert. In *POPL 2015*, pages 275–287, 2015.

[30] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, 2013.

[31] A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *POPL*, 2013.

[32] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *PLDI 2010*, pages 99–110, 2010.

[33] X. Zhang, C. Urban, and C. Wu. Priority inheritance protocol proved correct. In *ITP'12*, volume 7406 of *Lecture Notes in Computer Science*, pages 217–232, 2012.