# A Practical Verification Framework for Preemptive OS Kernels

**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 $\mu$C/OS-II [20], including the scheduler, interrupt handlers, message queues, and mutexes *etc.* We also verify the priority-inversion-freedom (PIF) in $\mu$C/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

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 $\mu$C/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 $\mu$C/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 [5, 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 $\mu$C/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

### 2.1 Preemptive OS Kernels and Interrupts

In a preemptive OS kernel, execution of a task inside the kernel can be interrupted at any program point (unless interrupts are disabled). Then the control is switched to the interrupt handler. When the handler finishes, it may invoke

the scheduler and switch the execution context to a different task, instead of returning to the original interrupted task. For instance, in priority scheduling, the interrupt handler always switches to the highest priority task at its end.

*The x86 interrupt mechanism.* Interrupt handling and management are indispensable in preemptive OS kernels. We give an 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.

On the invocations of interrupt handlers, the CPU flags (including `IF`) are saved on the stack, and interrupts are disabled automatically. If interrupts are enabled again inside the handler, the handler could be further interrupted by requests with higher priorities, causing nested interrupts.

The handler returns to the program being interrupted using the **iret** instruction, which also restores the flags (including `IF`). Before the handler returns, it needs to execute **eoi** to send an "end of interrupt" signal to the interrupt controller, which clears the corresponding bit in `isr`. Note that after **eoi** but before **iret**, if interrupts are enabled (`IF` = 1), the handler could be interrupted by interrupts at a lower or the same level.

*Overview of $\mu$C/OS-II.* $\mu$C/OS-II is a commercial preemptive real-time multitasking 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". $\mu$C/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.

### 2.2 Correctness of OS Kernels

An OS kernel hides details of the underlying hardware and provides an abstract programming model for application-level programmers. The implementation of the kernel must ensure that behaviors of user applications in the real machine are consistent with their behaviors under the abstract model [17]. Thus the OS verification can be reduced to verifying refinement between the concrete and abstract programming models.

*Contextual refinement as correctness.* We consider three entities, the application $A$, the abstract specification of the system APIs and interrupt handlers $\mathbb{O}$, and their concrete implementations $O$. When system calls are made or interrupts are handled, routines in $O$ are invoked in the real execution, while in the programmers' mind those in $\mathbb{O}$ are invoked instead at the abstract level. Then the correctness of OS kernels requires $O$ refines $\mathbb{O}$ under *all contexts* $A$:

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

where $[\![P]\!]$ represents the set of the observable behaviors of the program $P$. It says that, for all applications, executing the concrete code $O$ does not have more behaviors than executing the abstract version $\mathbb{O}$.

## 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. Due to space limit, we only show the main language features with simplifications for clear presentation. The details are available in TR [4] and the supplementary Coq code.

### 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 at the top of Fig. 1. 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 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).

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.* The middle of Fig. 1 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 make the semantics overly complex. To avoid this problem, we extend the C statements with assembly primitives $\iota$ 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

$$
\begin{array}{lll}
(\mathit{AExpr}) & e & ::= n \mid x \mid *e \mid \&e \mid e.id \mid e[e] \mid \ldots \\
(\mathit{AppStmts}) & d & ::= e{=}e \mid f(\bar{e}) \mid d; d \mid \textbf{while}\ (e)\ d \mid \textbf{if}\ (e)\ d\ \textbf{else}\ d \mid \textbf{return}\ e \mid \ldots \\
(\mathit{AppCode}) & A & ::= \{f_1 \rightsquigarrow d_1, \ldots, f_n \rightsquigarrow d_n\}
\end{array}
$$

$$
\begin{array}{lll}
(\mathit{LPrim}) & \iota & ::= \textbf{switch}\ x \mid \textbf{encrt} \mid \textbf{excrt} \mid \textbf{eoi}\ k \mid \textbf{iext} \mid \ldots \\
(\mathit{LStmts}) & s & ::= d \mid \iota \mid s; s \mid \textbf{while}\ (e)\ s \mid \ldots \\
(\mathit{ProgUnit}) & \eta & ::= \{f_1 \rightsquigarrow s_1, \ldots, f_n \rightsquigarrow s_n\} \\
(\mathit{LProg}) & P & ::= (A, O)
\end{array}
\qquad
\begin{array}{l}
(\mathit{ItrpCode})\ \theta ::= [s_0, \ldots, s_{N-1}] \\
(\mathit{LOSCode})\ O ::= (\eta_a, \eta_i, \theta)
\end{array}
$$

$$
\begin{array}{lll}
(\mathit{BitVal}) & b, ie \in \{0, 1\} & (\mathit{ISRReg})\ isr ::= [b_0, \ldots, b_{N-1}] \\
(\mathit{CrtStk}) & cs ::= \mathsf{nil} \mid ie :: cs & (\mathit{ItrpStk})\ is ::= \mathsf{nil} \mid k :: is \\
(\mathit{ItrpTaskSt}) & \delta ::= (ie, is, cs) & (\mathit{ItrpSt})\ \pi ::= \{t_1 \rightsquigarrow \delta_1, \ldots, t_n \rightsquigarrow \delta_n\}
\end{array}
$$

**Fig. 1.** The Language for Applications and Kernel Implementation

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. 1, 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 $\eta_a$, the internal functions $\eta_i$ and the interrupt handlers $\theta$. The internal functions are called only by code in $\eta_a$ or $\theta$. $\theta$ 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 $O$.

*Operational semantics.* 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 $\delta$ in Fig. 1, 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.

We give small-step operational semantics to the language. For each step, the processor picks the continuation of the current task and executes its current command or expression. To model concurrency and interrupts, 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).

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 other assembly primitives $\iota$ are all related to interrupts management and handling. To model their semantics, we introduce interrupt states in the state model, as shown at the bottom of Fig. 1. 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 $\delta$ 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. $\pi$ records the $\delta$ 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.

### 3.2 The High-Level Specification Language

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. 2, 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. 1). $\mathbb{O}$ contains the specifications $\varphi$ for kernel APIs, $\varepsilon$ for interrupt handlers, and $\chi$ for the scheduler.

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 points. Handlers of the events are also specified as $\varepsilon$ 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 $\chi$. As defined in Fig. 2, $\chi$ is a binary relation between abstract states and task identifiers. That is, given an abstract state $\Sigma$ (defined at the bottom of Fig. 2), $\chi$ finds a related task identifier as the next task to execute. Note that $\chi$ is a relation instead of a function, therefore the abstract scheduler does not have to be

$$
\begin{array}{llll}
(\mathit{HStmts}) & \mathbb{s} ::= \mathbf{sched} \mid \gamma(\bar{v}) \mid \mathbf{assert}\ \mathbb{b} \mid \mathbf{end} \mid \mathbb{s}_1; \mathbb{s}_2 \mid \mathbb{s}_1 + \mathbb{s}_2 & & \\
(\mathit{HAPISet}) & \varphi ::= \{f_1 \rightsquigarrow \mathbb{s}_1, \ldots, f_n \rightsquigarrow \mathbb{s}_n\} & (\mathit{HEvtSet}) & \varepsilon ::= [\mathbb{s}_0, \ldots, \mathbb{s}_{N-1}] \\
(\mathit{HSched}) & \chi \in \mathit{HAbsSt} \to \mathit{TaskId} \to \mathit{Prop} & (\mathit{TaskId}) & t \in \mathit{Nat} \\
(\mathit{HOSCode}) & \mathbb{O} ::= (\varphi, \varepsilon, \chi) & (\mathit{HProg}) & \mathbb{P} ::= (A, \mathbb{O}) \\
\hline
(\mathit{HAbsSt}) & \Sigma ::= \{\mathbb{a}_1 \rightsquigarrow \Omega_1, \ldots, \mathbb{a}_n \rightsquigarrow \Omega_n\} & (\mathit{HDataNm}) & \mathbb{a} ::= \mathsf{tcbls} \mid \mathsf{ctid} \mid \ldots \\
(\mathit{HData}) & \Omega ::= \alpha \mid t \mid \ldots & (\mathit{HStatus}) & ts ::= \mathsf{rdy} \mid \ldots \\
(\mathit{HTCBLs}) & \alpha ::= \{t_1 \rightsquigarrow (pr_1, ts_1, \ldots), \ldots, t_n \rightsquigarrow (pr_n, ts_n, \ldots)\} & &
\end{array}
$$

**Fig. 2.** High-Level Spec. Language and Abstract States

deterministic. Since $\chi$ 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** represents the end of abstract APIs 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 $\Sigma$ at the high level. As defined at the bottom of Fig. 2, $\Sigma$ is a mapping from names $\mathbb{a}$ to the abstract data $\Omega$. Here $\mathsf{tcbls}$ is the name for the high-level abstract TCB list $\alpha$, 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. $\mathsf{ctid}$ is the name for the current task identifier $t$.

### 3.3 OS Correctness

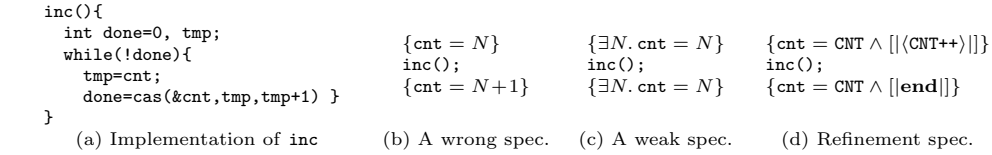
As we explain in Sec.2.2, the correctness of OS kernels can be defined in terms of contextual refinement. Below we give its formal definition.

**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

$$
\begin{array}{l}
\mathsf{Match}(\psi, (T, \Delta, \Lambda, t), (T, \Delta, \Sigma)) \stackrel{\text{def}}{=} \\
\qquad (t \in \mathit{dom}(T)) \wedge (\psi\ \Lambda\ \Sigma) \wedge (t = \Sigma(\mathsf{ctid})) \wedge (\mathit{dom}(T) = \mathit{dom}(\Sigma(\mathsf{tcbls})))
\end{array}
$$

The low-level kernel code $O$ refines its high-level abstract specifications $\mathbb{O}$ with constraints $\psi$ 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. $\psi$), 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]).

Due to space limit, we elide the definitions of $W$ and $\mathbb{W}$ in Sec. 3.1 and 3.2. The low-level whole program state $W$ is in the form of $(T, \Delta, \Lambda, t)$, where the *task pool* $T$ maps task identifiers to their continuations, $\Delta$ is the client state, $\Lambda$ is the low-level kernel state, and $t$ is the identifier of the current task. The




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

**Fig. 3.** Specification of Concurrent Programs

high-level program state $\mathbb{W}$ is in the form of $(T, \Delta, \Sigma)$, where $\Sigma$ is an abstraction of the low-level kernel state $\Lambda$ and the current task id $t$.

The constraint $\mathsf{Match}$ requires that: (1) initially $W$ and $\mathbb{W}$ have the same task pool $T$ and client state $\Delta$; (2) the current task $t$ is in $T$; (3) the low-level kernel state $\Lambda$ and the high-level abstract state satisfy $\psi$; (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.

## 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.

*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. 3(a) increments the counter `cnt`. It may be called simultaneously by concurrent tasks. Figure 3(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. 3(c), which is too weak to capture the functionality.

Figure 3(d) gives a relational specifications to show that `inc` refines an abstract operation $\langle \texttt{CNT++} \rangle$ [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 ($\langle \texttt{CNT++} \rangle$) 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 the abstract operation that `inc` needs to refine is $\langle \texttt{CNT++} \rangle$. The postcondition ensures `cnt` and `CNT` remain consistent and the remaining abstract operation that needs to be refined is **end** (*i.e.*, $\langle \texttt{CNT++} \rangle$ 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 ($\Sigma$) and the API specifications ($\mathbb{s}$).

*Assertions.* Below is the assertion language. Its semantics is given in Fig. 4.

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

$$(\mathit{RelState})\ \Theta ::= (\sigma, \Sigma, \mathbb{s}) \quad (\mathit{LTaskCfg})\ \sigma ::= (m, \mathit{isr}, \delta) \quad (\mathit{LTaskSt})\ m ::= (G, E, M)$$

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

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

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

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

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

**Fig. 4.** Relational Assertions and their Semantics

As explained above, the assertions are interpreted over relational states $\Theta$, which consist of the low-level task-local states $\sigma$, the high-level abstract states $\Sigma$, and the abstract statements $\mathbb{s}$ that the low-level code needs to refine. $\Sigma$ and $\mathbb{s}$ are defined in Fig. 2. $\sigma$, as shown in Fig. 4, consists of a task-local view $m$ of program variables and memory, and also the global *isr* register and the task-local interrupt states $\delta$ (see Fig. 1). Here $m$ contains the global and local variables ($G$ and $E$ respectively) and the memory $M$, whose definitions are omitted.

Assertion $\mathsf{emp}$ says the low-level memory and the high-level abstract state are both empty. $\mathsf{empE}$ 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 $x$. $\mathsf{ISR}(\mathit{isr})$, $\mathsf{IS}(\mathit{is})$, $\mathsf{IE}(\mathit{ie})$ and $\mathsf{CS}(\mathit{cs})$ specify the value of the corresponding interrupt status (see Fig. 1). $\llcorner k \lrcorner$ 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 $\chi$ 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 $\Omega$. $[|\mathbb{s}|]$ 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.

*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 5 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. Handlers' priorities to pick their required resources are consistent with their interrupt priority levels. That is, $B_0$ satisfies all the need

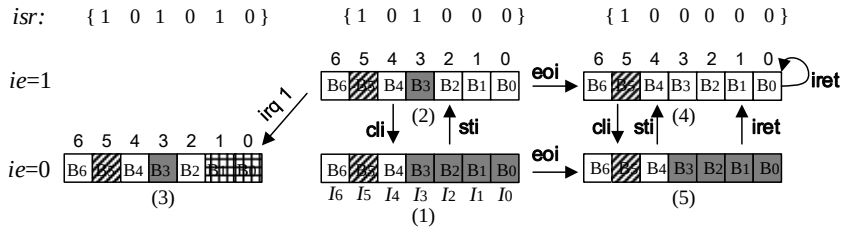



**Fig. 5.** Ownership-Transfer for Multi-level Interrupts

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 has the lowest priority.

Figure 5 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).

*The top rule.* We show some selected program logic rules in Fig. 6. 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 $\psi$ (explained in Def. 3.3).

To verify the kernel, we need to come up with a specification $\Gamma$ for the internal functions $\eta_i$ in the low-level code, and a sequence of invariants $I$ for kernel states. $\Gamma$ assigns a pair of pre-/post-conditions to each internal function. We omit the formal definition here.

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 $\chi$ and the invariant $I$.

The rule also requires that $\psi$ ensures the initial states satisfy the invariant $I[0, N]$, the interrupt-related states are properly initialized, and the initial local variable environment is empty. $I[n, m]$ defined in Fig. 6 is the separating conjunction of invariants from level $n$ to $m$. $\mathsf{OS}[isr, ie, is, cs]$ specifies the status of

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

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

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

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

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

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

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

$$\frac{p\Rightarrow p_i}{\Gamma;\chi;I;\mathsf{false};p_i\vdash\{\,p\,\}\ \mathbf{iext}\ \{\,\mathsf{false}\,\}}\ (\textsc{iext})\qquad \frac{p\Rrightarrow p'\quad \Gamma;\chi;I;r;p_i\vdash\{\,p'\,\}\,s\,\{\,q'\,\}\quad q'\Rrightarrow q}{\Gamma;\chi;I;r;p_i\vdash\{\,p\,\}\,s\,\{\,q\,\}}\ (\textsc{abscsq})$$

$$I[n,m]\ \stackrel{\text{def}}{=}\ \begin{cases} I(n)*I(n+1)*\ldots*I(m) & \text{if } 0\le n\le m\le N\\ \mathsf{emp} & \text{otherwise}\end{cases}$$

$$\mathsf{OS}[isr,ie,is,cs]\ \stackrel{\text{def}}{=}\ \exists k.\,\mathsf{ISR}(isr)*\mathsf{IE}(ie)*\mathsf{IS}(is)*\mathsf{CS}(cs)*\llcorner k\lrcorner*(\forall k'.\,0\le k'<k\rightarrow isr(k')=0)$$

$$\mathsf{INV}(I,k)\ \stackrel{\text{def}}{=}\ \exists isr.\,\mathsf{ISR}(isr)*((isr(k)=1\wedge\mathsf{emp})\vee((isr(k)=0\vee k=N)\wedge I(k)))$$

$$\mathsf{SWINV}(I)\ \stackrel{\text{def}}{=}\ \mathsf{ISR}(\bar{0})*\mathsf{IE}(0)*(\exists\ k.\,\llcorner k\lrcorner*I[0,k])$$

$$\mathsf{BldItrpPre}(k,\varepsilon,isr,is,I)\ \stackrel{\text{def}}{=}\ \mathsf{OS}[isr\{k\rightsquigarrow 1\},0,k::is,\mathsf{nil}]*I[0,k]*[|\varepsilon(k)|]*\mathsf{empE}$$

$$\mathsf{BldItrpRet}(k,isr,is,I)\ \stackrel{\text{def}}{=}\ \exists ie.\,\mathsf{OS}[isr\{k\rightsquigarrow 0\},ie,k::is,\mathsf{nil}]*((ie=1\wedge\mathsf{emp})\vee(ie=0\wedge I[0,k]))*[|\mathbf{end}|]$$

**Fig. 6.** Selected Inference Rules

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 $\psi$ to relational assertions (definition omitted). We also omit some more detailed side conditions about the initial states in the rule.

*Verifying interrupt handlers.* We omit the rules of proving $\chi;I\vdash\eta_i:\Gamma$ and $\Gamma;\chi;I\vdash\eta_a:\varphi$ for internal functions and APIs respectively, which are similar to the rules for 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 $\mathsf{false}$.

We build the pre-/post-conditions of handlers with the auxiliary definitions $\mathsf{BldItrpPre}$ and $\mathsf{BldItrpRet}$ given in Fig. 6. 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. 5). It also needs to refine the high-level specification code $\varepsilon(k)$. $\mathsf{empE}$ 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. 5); 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).

*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 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. 6. 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. 5. 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. 5). Correspondingly there is a EXCRT-0 rule, which is omitted here. 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. 5).

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. 6, 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 $\chi$ 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.

$$\forall \sigma, \Sigma, \mathbb{s}.(\sigma, \Sigma, \mathbb{s}) \models p \longrightarrow \exists \Sigma', \mathbb{s}'.\ \Big((\mathbb{s}, \Sigma) \bullet\!\!-\!\!\!-_H\!\!\!\rightarrow^{*} (\mathbb{s}', \Sigma')\Big) \wedge (\sigma, \Sigma', \mathbb{s}') \models p'$$

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 4.1 gives the soundness of the framework. The proofs are based on a compositional simulation following [23], and have been formalized in Coq. More details about the logic can be seen in TR [4].

**Theorem 4.1 (Soundness).** $\vdash_\psi O : \mathbb{O} \Longrightarrow O \sqsubseteq_\psi \mathbb{O}$.

## 5 Proving Priority-Inversion-Freedom

*Formalization of PIF.* Earlier work [7] defines priority inversions in terms of whether there is a higher priority task waiting directly or indirectly for a lower priority task. Since the definition refers to the *current* priority of tasks, its meaning is affected by algorithms that dynamically change the priority of tasks, such as the classic priority ceiling and priority inheritance algorithms [28]. We give a new formalization of PIF, which is based on the *original* priorities assigned by the programmers, reflecting the actual degree of urgency.

**Definition 5.1 (Priority Inversion Freedom).** $\mathrm{PIF}(\Sigma)$ 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 assigned by users. $\mathsf{IsWait}(t, \Sigma)$ means that $t$ is blocked, waiting for certain shared resource, and $\neg\mathsf{IsOwner}(t_c, \Sigma)$ means that the task $t_c$ does not own any shared resources (*e.g.*, mutexes).

If each task eventually releases its shared resource (*i.e.*, there is no deadlock), the definition ensures that the waiting task with higher priority will be eventually released and executed. Therefore it prevents unbounded priority inversion [28].

*PIF of μC/OS-II.* The mutex of $\mu$C/OS-II is implemented with a simplified priority ceiling protocol [28]. When proving it satisfies PIF, we find a counterexample (given in the TR [4]) showing that PIF cannot be guaranteed unless there is no nested use of mutexes. By adding the assumption of no nested mutexes, we prove that the mutex in $\mu$C/OS-II ensures our PIF definition.

**Theorem 5.2 (PIF without Nested Use of Mutexes).**
If $\mathsf{Init}(\Sigma)$, $(A, \mathbb{O}_{\mu\mathrm{C/OS\text{-}II}}) \vdash (T, \Delta, \Sigma) =\!\!H\!\!\Rightarrow^* (T', \Delta', \Sigma')$, $\mathsf{NoNCR}(A, \Sigma, T, \Delta)$, and $\mathsf{SchedProp}(\Sigma')$, then $\mathrm{PIF}(\Sigma')$.

It says, for any application code $A$, task pool $T$, client state $\Delta$ and abstract kernel state $\Sigma$, if initially there are no tasks waiting for mutexes ($\mathsf{Init}(\Sigma)$), and there is no nested use of mutexes ($\mathsf{NoNCR}(A, \Sigma, T, \Delta)$), then for any $T'$, $\Delta'$ and $\Sigma'$ generated during the execution, if $\Sigma'$ is consistent with the priority-based scheduling (*i.e.* the current running task always has the highest priority among all the ready tasks, represented as $\mathsf{SchedProp}(\Sigma')$), then it must satisfy PIF. Here we use a simplified $\mathbb{O}_{\mu\mathrm{C/OS\text{-}II}}$ that contains the PIF mutex as the only APIs. The proof is formalized in Coq.

## 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 $\mu$C/OS-II V2.52, including

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

| Certified $\mu$C/OS-II | Coq lines |
|---|---|
| C Code Definitions | 1823 |
| Specifications | 6332 |
| Priority Inversion Freedom | 9467 |
| Libraries for $\mu$C/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 1.** The Verification Package

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.

*Proof efforts.* To formalize the work in this paper, we develop around 215,000 lines of proofs in Coq8.4pl6. Table 1 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 $\mu$C/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 $\mu$C/OS-II the major difficulty comes from the gap between the low-level concrete data structure and the high-level abstract representation. For instance, $\mu$C/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.

*Coq tactics.* Proof automation is essential to improve the productivity. We develop tactics for automatically proving relational separation logic assertions and generating verification conditions based on existing techniques [9, 25, 6]. They do forward reasoning for statements, including function calls and primitives entering and exiting critical regions, *etc.* Also some domain-specific tactics are implemented for individual data structures used in $\mu$C/OS-II, including ones for the arithmetic properties of *Int32* and bitmaps. Thanks to these tactics, the ratio of Coq script to source code is around 26:1. Another advantage of the tactics is that they can extract lemmas independent of program contexts for verifying functionality of code. Users can verify code using the tactics without knowing much about the underlying framework.

## 7 Related Work and Conclusion

There have been a number of OS verification projects, including Spin [8], VeriSoft [5], 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 $\mu$C/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.

*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 $\mu$C/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: $\mu$C/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] A practical verification framework for preemptive OS kernels (technical report), January 2016. `https://sites.google.com/site/certiucos/paper.pdf`.

[5] 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.

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

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

[8] 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.

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

[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.