A Practical Verification Framework for Preemptive OS Kernels

Fengwei Xu[1,2], Ming Fu[1,2](✉), Xinyu Feng[1,2], Xiaoran Zhang[1,2], Hui Zhang[1,2], and Zhaohui Li[1,2]

[1] School of Computer Science and Technology, University of Science and Technology of China, Hefei, China
fuming@ustc.edu.cn
[2] Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou, China

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 [2], including the scheduler, interrupt handlers, message queues, and mutexes *etc.* We also verify the $\underline{\text{p}}$riority-$\underline{\text{i}}$nversion-$\underline{\text{f}}$reedom (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 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 [4,13,16,27] in recent years, but most of them have no or limited support of kernel-level preemption, which allows tasks to be preempted even in kernel mode. This limitation restricts their applicability to real-time systems, where preemptive multitasking is 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 make 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 [2], a commercial preemptive real-time multitasking kernel for microprocessors and microcontrollers. The verification is fully mechanized

This work is supported in part by grants from National Natural Science Foundation of China (NSFC) under Grant Nos. 61103023, 61229201, 61379039 and 91318301.

S. Chaudhuri and A. Farzan (Eds.): CAV 2016, Part II, LNCS 9780, pp. 59–79, 2016.
DOI: 10.1007/978-3-319-41540-6_4

in Coq [1]. To our knowledge, it is the first verification of (key modules of) a *preemptive* OS kernel with machine-checkable proofs. The key contribution of the work is to adapt existing theories on interrupt verification [11] and contextual refinement of concurrent programs [17,19,24,25], and integrate them into a framework for real-world preemptive OS kernel verification. Specifically, our work 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 approaches have been applied in earlier work on OS kernel verification [4,13,16], we believe our work is the first to explicitly specify and prove contextual refinement for APIs of a preemptive OS kernel, following recent progress on refinement verification of *concurrent* programs [17,19,24,25]. As we explain in Sect. 2.2, contextual refinement not only serves as a very strong notion of functional correctness of system APIs, but also allows us to prove properties based on the more abstract API specifications and then carry it down to the level of concrete implementations, which makes the verification much simpler than doing proofs directly at the concrete level.

Second, we provide a simple modeling language for specifying kernel primitives. The language strives for 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. 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 abstract 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 [21] (CSL) with relational assertions that relate program states at the implementation and the specification levels, as in Liang *et al.* [17,19]. It also assigns ownership-transfer semantics to interrupt management operations and verify multi-level hardware interrupts in a realistic setting. Different from traditional Hoare-style program logics, whose soundness ensures the semantic interpretation of Hoare-triples, our logic explicitly establishes contextual refinement, which is more useful for establishing abstractions for system APIs, as explained above.

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 handler (and a pseudo interrupt handler 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 [4,13,16,27] that are focused on kernels 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, we also specify and 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. Moreover, since the specification and verification are done at the level of the abstract model (*i.e.*, specifications) of the kernel, they also help validate our model of system APIs. As we explain above, many important properties cannot be specified if the model is too weak or overly abstract.

Coq proofs and a companion technical report are available at http://staff.ustc.edu.cn/~fuming/research/certiucos.

2 Background and Overview of Our Work

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, with priority-based 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 8259 A 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 8259 A 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 with 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 an interrupt handler, 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 ($\texttt{IF} = 1$), the handler could be interrupted by interrupts at a lower or the same level.

Overview of μC/OS-II. μC/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*" [2]. 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.

2.2 Overview of the Verification Framework

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 [14]. 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 specifications 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 $[\![\_]\!]$ maps a program P to the set of its observable behaviors. It says that, for all applications, executing the concrete code O does not have more observable behaviors than executing the abstract version $\mathbb{O}$. In this paper, observable behaviors are defined as finite prefixes of execution traces consisting of observable events, following Liang *et al.* [17].

Contextual refinement is a very strong notion of functional correctness of system APIs since it quantifies over *all* applications. Moreover, it makes verification of system-wide properties simpler. For instance, if we want to verify certain property Φ about a whole system $A[O]$, *i.e.*, Φ holds over every trace in $[\![A[O]]\!]$, we could prove that it holds over every trace in the superset $[\![A[\mathbb{O}]]\!]$ instead. Proofs at the abstract level could be much simpler than the concrete level.

The Whole Verification Framework. Figure 1 shows the structure of our verification framework. To model OS kernels and applications, we introduce two languages (in block A), the low-level language for the concrete code implementation and the high-level language for the abstract specification. Above them we have a program logic (in block B) that allows us to prove the low-level kernel implementation contextually refines the high-level specifications. The framework also provides a set of Coq tactics (in block C) to automatically generate and prove verification conditions. The μC/OS-II modules certified in this framework are shown in block D. Below we give details of some of the building blocks.

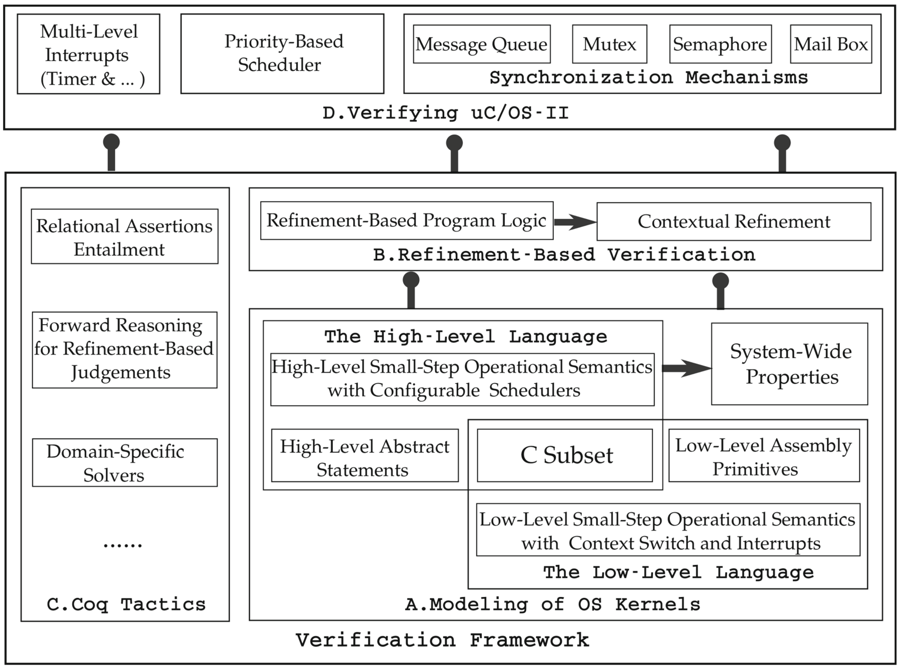
Multi-Level Interrupts (Timer & ...)
Priority-Based Scheduler
Message Queue
Mutex
Semaphore
Mail Box
Synchronization Mechanisms
D.Verifying uC/OS-II
Relational Assertions Entailment
Forward Reasoning for Refinement-Based Judgements
Domain-Specific Solvers
......
C.Coq Tactics
Refinement-Based Program Logic
Contextual Refinement
B.Refinement-Based Verification
The High-Level Language
High-Level Small-Step Operational Semantics with Configurable Schedulers
System-Wide Properties
High-Level Abstract Statements
C Subset
Low-Level Assembly Primitives
Low-Level Small-Step Operational Semantics with Context Switch and Interrupts
The Low-Level Language
A.Modeling of OS Kernels
Verification Framework

Fig. 1. Structure of the verification framework and μC/OS-II verification

3 Modeling of OS Kernels

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 (or modeling) languages for the three entities (see block A in Fig. 1). Due to space limit, we only show the main language features with simplifications for clear presentation. The details are available at TR and the Coq code [26].

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. 2. 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).

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

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

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

Fig. 2. The language for applications and kernel implementation

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. 2 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 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 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. 2, 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.

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 δ in Fig. 2, 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 [22], 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 ι 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. 2. The *global* register *isr* is shared by all tasks. It models the *isr* register in the 8259 A interrupt controller, as explained in Sect. 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 Sect. 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.

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

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

Fig. 3. High-level spec. language and abstract states

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. 3, 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. 2). $\mathbb{O}$ contains the specifications φ for kernel APIs, ε for interrupt handlers, and χ 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. At the high level an incoming level-k event is always handled by executing $\varepsilon(k)$, *i.e.* the k-th handler specified in ε.

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. 3, χ is a binary relation between abstract states and task identifiers. That is, given an abstract state Σ (defined at the bottom of Fig. 3), 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 could be non-deterministic. Since χ is provided as part of the kernel specification, the semantics of **sched** in our language is configurable. 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. It can be instantiated 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 Σ at the high level. As defined at the bottom of Fig. 3, 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.

Example of High-Level Specifications. We use $\mathbb{s}_{\mathsf{dly}} \stackrel{\text{def}}{=} (\gamma_{\text{err}}(\mathsf{ticks}) + (\gamma_{\text{dly}}(\mathsf{ticks}); \mathbf{sched}))$ to specify the system API "$\mathsf{void\ OSTimeDly(Int16u\ ticks)}$", which delays the current task for the specified number of system ticks. The atomic operation $\gamma_{\text{err}}(\mathsf{ticks})$ specifies the error case when $\mathsf{ticks} = 0$. $\gamma_{\text{dly}}(\mathsf{ticks})$ defines the atomic behavior of updating the status of the current task from "ready" to "waiting" with the duration set to ticks when $\mathsf{ticks} > 0$, and the following **sched** switches to another ready task, following the scheduling policy specified by the abstract scheduler χ. Note that the exclusive conditions over ticks in $\gamma_{\text{err}}(\mathsf{ticks})$ and $\gamma_{\text{dly}}(\mathsf{ticks})$ make the non-deterministic choice statement deterministic. We omit the definitions of $\gamma_{\text{err}}(\mathsf{ticks})$ and $\gamma_{\text{dly}}(\mathsf{ticks})$ here.

As another example, below we show the abstract scheduler specification $\chi_{\mu\text{C/OS-II}}$ for μC/OS-II. It requires that the selected task be ready and have the highest priority among all the ready tasks.

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

3.3 OS Correctness

As we explain in Sect. 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 $\psi \in \mathit{LOSFullSt} \rightarrow \mathit{HAbsSt} \rightarrow \mathit{Prop}$ and

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

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 [17]).

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

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

```
inc(){
  int done=0, tmp;
  while(!done){
    tmp=cnt;
    done=cas(&cnt,tmp,tmp+1) }
}
```
(a) Implementation of `inc`

$\{\texttt{cnt} = N\}$
`inc();`
$\{\texttt{cnt} = N+1\}$

(b) Wrong spec.

$\{\exists N.\, \texttt{cnt} = N\}$
`inc();`
$\{\exists N.\, \texttt{cnt} = N\}$

(c) Weak spec.

$\{\texttt{cnt} = \texttt{CNT} \wedge [\!|\langle \texttt{CNT++}\rangle|\!]\}$
`inc();`
$\{\texttt{cnt} = \texttt{CNT} \wedge [\!|\mathbf{end}|\!]\}$

(d) Refinement spec.

Fig. 4. Specification of concurrent programs

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.

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. 4(a) increments the counter `cnt`. It may be called simultaneously by concurrent tasks. Figure 4(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. 4(c), which is too weak to capture the functionality.

Figure 4(d) gives a relational specifications to show that `inc` refines an abstract operation $\langle\texttt{CNT++}\rangle$ [19], 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 [19]). 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 (Σ) and the abstract statement ($\mathbb{s}$).

Assertions. Below is the assertion language, and its semantics is given in Fig. 5.

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

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

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

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

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

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

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

Fig. 5. Semantics of 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. 3. σ, as shown in Fig. 5, 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. 2). 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. $\mathsf{ISR}(\mathit{isr})$, $\mathsf{IS}(\mathit{is})$, $\mathsf{IE}(\mathit{ie})$ and $\mathsf{CS}(\mathit{cs})$ specify the value of the corresponding interrupt status (see Fig. 2). $\llcorner k \lrcorner$ 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.

Ownership-Transfer Semantics for Multi-level Interrupts. CSL [21] prevents data races by enforcing disjoint ownership of resources among tasks. Synchronization is modeled in terms of ownership

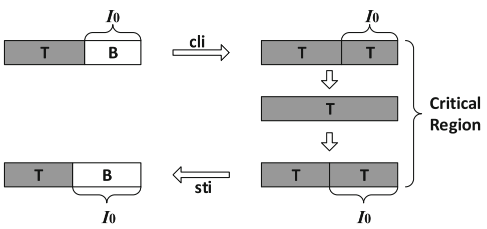

Fig. 6. Memory partition for handler and non-handler (Figure taken from [11])

transfer. Feng *et al.* [11] extend CSL and assign ownership-transfer semantics to interrupt operations. The idea is demonstrated in Fig. 6, which shows the *logical* memory model when there are only one task and single-level interrupt. Since the interrupt handler can preempt the task, we let the handler to reserve its required memory first (represented as block B). B must remain publicly available if the interrupt is enabled. Then the task can only access the remaining part (block T). We use grey boxes to represent *local* resources of the task. Disabling interrupts (**cli**) by the task essentially transfers the ownership of B from public to task-local. Correspondingly, **sti** converts the block from task-local to public, therefore the task *cannot* access it anymore. Similarly, invocation of the interrupt handler (not shown in the figure) automatically transfers B from public to the local resource of the handler, while **iret** transfers it back to public.

Since block B is shared between the interrupt handler and the task, it must be well-formed when it is public. We use the resource invariant I_0 to specify the well-formedness. Then the above ownership transfer semantics of **cli** and **sti** can be formalized in the following (simplified) program logic rules:

$$\frac{}{I_0 \vdash \{p_t\}\,\mathbf{cli}\,\{p_t * I_0\}} \qquad\qquad \frac{}{I_0 \vdash \{p_t * I_0\}\,\mathbf{sti}\,\{p_t\}}$$

Note that the partition between B and T is enforced *logically* using the separating conjunction in separation logic (see Fig. 5). It does not require physical separation in the program state model.

In this paper we extend this idea to support multi-level nested interrupts, where the ownership transfer of interrupt primitives is determined not only by the *ie* flag, but also by the *isr* register. Figure 7 shows the memory model (where the number N of interrupts is set to 6). Interrupt handlers at levels 0 to $N-1$ are assigned with resource blocks $B_0, \ldots, B_{N-1}$ respectively. B_N represents the resource shared only among tasks, *i.e.*, the non-handler code. We omit task-local resources, therefore there are *no* counterparts to block T in Fig. 6. Handlers' priorities to reserve 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 has the lowest priority. Each block B_k is specified by the resource invariant $I(k)$, where I is defined as a sequence of $N+1$ assertions (see the assertion syntax defined above).

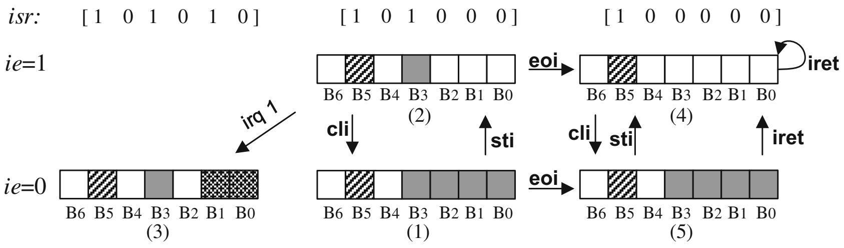

Fig. 7. Ownership-transfer for multi-level interrupts

Figure 7 demonstrates the ownership transfer of resource caused by interrupt operations under different conditions. The grey or dotted blocks represent resources exclusively owned in interrupt handlers, different textures for different interrupts. The white ones represent resources *available* for share. Suppose initially we are at state (1), where the level-3 handler is being executed, as the value of *isr* indicates. 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 $\mathit{isr}(3) = 1$ and requests of the same (or lower) level are not handled. However, if $\mathit{isr}(3) = 0$ instead (as in state (5)), executing **sti** loses B_3 as well. Ownership transfer by **cli** is the dual of **sti**.

Executing **eoi** at state (1) leads to 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 $\mathit{isr}(1)$, 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. 8. 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 Sect. 3.3).

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 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 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 variable environment is empty. $I[n, m]$ defined in Fig. 8 is the separating conjunction of invariants from level n to m. $\mathsf{OS}[\mathit{isr}, \mathit{ie}, \mathit{is}, \mathit{cs}]$ 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 (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

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

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

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

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

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

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

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

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

$$\begin{array}{lcl}
I[n,m] & \stackrel{\text{def}}{=} & \begin{cases} I(n)*I(n+1)*\ldots*I(m) & \text{if } 0\le n\le m\le N\\ \mathsf{emp} & \text{otherwise}\end{cases}\\
\mathsf{OS}[isr,ie,is,cs] & \stackrel{\text{def}}{=} & \exists k.\,\mathsf{ISR}(isr)*\mathsf{IE}(ie)*\mathsf{IS}(is)*\mathsf{CS}(cs)*\llcorner k\lrcorner*(\forall k'.\,0\le k'<k\rightarrow isr(k')=0)\\
\mathsf{INV}(I,k) & \stackrel{\text{def}}{=} & \exists isr.\,\mathsf{ISR}(isr)*((isr(k)=1\wedge\mathsf{emp})\vee((isr(k)=0\vee k=N)\wedge I(k)))\\
\mathsf{SWINV}(I) & \stackrel{\text{def}}{=} & \mathsf{ISR}(\bar{0})*\mathsf{IE}(0)*(\exists\,k.\,\llcorner k\lrcorner*I[0,k])\\
\mathsf{BldItrpPre}(k,\varepsilon,isr,is,I) & \stackrel{\text{def}}{=} & \mathsf{OS}[isr\{k\rightsquigarrow 1\},0,k::is,\mathsf{nil}]*I[0,k]*[|\varepsilon(k)|]*\mathsf{empE}\\
\mathsf{BldItrpRet}(k,isr,is,I) & \stackrel{\text{def}}{=} & \exists ie.\,\mathsf{OS}[isr\{k\rightsquigarrow 0\},ie,k::is,\mathsf{nil}]*((ie=1\wedge\mathsf{emp})\vee(ie=0\wedge I[0,k]))*[|\mathbf{end}|]
\end{array}$$

Fig. 8. Selected inference rules

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.

We build the pre-/post-conditions of handlers with the auxiliary definitions $\mathsf{BldItrpPre}$ and $\mathsf{BldItrpRet}$ given in Fig. 8. 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. 7). 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. 7); 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. 8. If the bit $isr(k)$ is 0 (or $k = N$), the task also gains the ownership of $I(k)$, otherwise it already owns the k-th block and there is no extra ownership transfer. The two scenarios are also demonstrated by the two **cli** steps in Fig. 7. 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. 7). 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. 7).

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. 8, 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 [19], the ABSCSQ rule looks like a regular consequence rule but allows us to *execute* the abstract code. The implication $p \Rrightarrow p'$ is defined below.

$$\forall \sigma, \Sigma, \mathbb{s}.\ ((\sigma, \Sigma, \mathbb{s}) \models p) \Longrightarrow \exists \Sigma', \mathbb{s}'.\ \Big((\mathbb{s}, \Sigma) \bullet\!\!\xrightarrow{H}{}^{*} (\mathbb{s}', \Sigma')\Big) \wedge ((\sigma, \Sigma', \mathbb{s}') \models p')$$

That is, given a related state $(\sigma, \Sigma, \mathbb{s})$ satisfying p, the abstract code $\mathbb{s}$ could execute zero or multiple steps starting from Σ and reach $(\Sigma', \mathbb{s}')$, so that the resulting related state $(\sigma, \Sigma', \mathbb{s}')$ satisfies p'. This rule allows us to establish simulation between the concrete and the abstract code, which then ensures refinement.

We can look at Fig. 4 to see the use of this rule. Suppose we want to verify `inc()` using the specification in Fig. 4(d). When we reach the `cas` command (see Fig. 4(a)), we have the precondition ($\texttt{tmp} = \texttt{cnt} \wedge \texttt{cnt} = \texttt{CNT} \wedge [|\texttt{<CNT++>}|] \vee \ldots$) (the case for $\texttt{tmp} \neq \texttt{cnt}$ omitted). Right after `cas`, we have ($\texttt{done} \wedge \texttt{cnt} = \texttt{CNT}{+}1 \wedge [|\texttt{<CNT++>}|] \vee \neg\texttt{done} \wedge \ldots$). We have $\texttt{cnt} = \texttt{CNT}{+}1$ because `cnt` increments if `cas` succeeds. To establish the simulation, we apply the ABSCSQ rule to execute the abstract code, because $(\texttt{cnt} = \texttt{CNT}+1 \wedge [|\texttt{<CNT++>}|]) \Rrightarrow (\texttt{cnt} = \texttt{CNT} \wedge [|\textbf{end}|])$, following the above definition of $p \Rrightarrow p'$.

Theorem 4.1 gives the soundness of the framework. The proofs are based on a compositional simulation following [18], and have been formalized in Coq. More details about the logic can be seen in TR [26].

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 [6] 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 [23]. 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). 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 assigned by programmers. $\mathsf{IsWait}(t, \Sigma)$ 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 resource (*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 [23].

PIF of μC/OS-II. The mutex of μC/OS-II is implemented with a simplified priority ceiling protocol [23]. When proving it satisfies PIF, we find a counterexample (given in TR [26]) 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 μC/OS-II ensures our PIF definition.

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

It says, for any application code A, task pool T, client state Δ and abstract kernel state Σ, 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', Δ' and Σ' generated during the execution, if Σ' is consistent with the priority-based scheduling (*i.e.*, the currently 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 μC/OS-II V2.52, including the scheduler, the timer interrupt handler, mutexes, message queues, mail boxes, semaphores, and the time management. These 1300 lines of C code verified in our framework correspond to around 3250 lines of code in their original format (with comments and empty lines) in the source files of μC/OS-II, including "ucos_ii.h", "os_q.c", "os_sem.c", "os_mbox.c", "os_mutex.c", "os_time.c", "os_core.c" and "os_cpu_a.c". The verified modules cover 63 % of the frequently used APIs and internal functions [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 event control block, because it assumes that the client code always gets S by calling OSQCreate() (thus S should already be valid). We drop this assumption about the client code. Correspondingly we insert code that checks whether S is a valid pointer. If S is invalid a new error code is returned. Similar modifications are made to some other modules too. The reason for doing above modifications is that the contextual refinement proved in our verification framework assumes arbitrary client code, while kernels are usually implemented with assumptions over client code for efficiency.

Table 1. The Verification Package

Framework	**Coq lines**
Basic Libraries	32061
Machine & Logic	23095
Automated Tactics	21050
Total	76206
Certified μC/OS-II	**Coq lines**
C Code Definitions	1824
Specifications	6012
Priority Inversion Freedom	9570
Libraries for μC/OS-II	62085
Auto. Generated Code	25357
Total	104848

Verified Modules	**lines of C**	**Coq lines**
Global Declarations	187	-
Message Queue	240	4537
Semaphore	166	2441
Mailbox	171	3326
Mutex	301	17331
Time Management	39	861
Timer Interrupt	17	443
Internal Functions	195	5447
Final Theorems	-	501
Total	1316	34887

Proof Efforts. The Coq implementation consists of around 216,000 lines of code and 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 h on a machine with 3.6 GHz cpu and 32G memory. The work takes us around 5.5 person years in total, including 4 person years for the framework and 1 person year for verifying the first μC/OS-II module (Message Queue). 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.

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 [5,7,20]. 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 μC/OS-II, including ones for the arithmetic properties of *Int32* and bitmaps. Thanks to these tactics, the ratio of Coq proof scripts to the verified C 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 seL4 [15,16], Verisoft [4], VCC/VeriSoftXT [3,9], Verve [27], and CertiKOS [8,13]. Most of them have no or limited support of preemption and multi-level interrupts.

seL4 [15,16] is one of the milestone OS kernel verification projects. The verification is fully mechanized in Isabelle/HOL. The kernel of seL4 does not support general preemption. Instead, tasks are preemptible only at specific points. Therefore the code verified is mostly sequential. On the other hand, the seL4 project has verified rich features and properties such as virtual memory, real-time properties and security properties, which are not done in our work.

The Verisoft project also verifies OS microkernels [4] in Isabelle/HOL, but the CVM model used there does not permit interrupts inside the kernel. Its successor project, Verisoft XT [3], uses VCC [9] to verify the commercial Hyper-V hypervisor. VCC supports verification of concurrent C code by inserting auxiliary code and ghost states. The proofs have a refinement flavor, but VCC does not establish contextual refinement as what we do. Also it is unclear how VCC is applied to verify multi-level nested interrupts in hypervisors.

Verve [27] combines a type-safe kernel with a minimal hardware abstraction layer. The kernel is concurrent, but the properties verified are mostly about type safety, much weaker than our contextual refinement property. Also Verve simply squashes multiple interrupt levels into a single level and does not really handle multi-level interrupts. VCC/VerisoftXT and Verve use the Z3 SMT solver [10] for better automation, while we use Coq which generates machine-checkable proofs. Also the soundness of our program logic is proved in Coq. Therefore the trusted computing base (TCB) of our approach is smaller.

Gu *et al.* [13] verify the mCertiKOS hypervisor. Their kernel is sequential. Recently, Chen *et al.* [8] propose a framework for building certified interruptible OS kernels (based on mCertiKOS) with device drivers. Their framework does not support preemptive concurrency as ours, and it requires that interrupt handlers for device drivers and non-handler kernel code should not share any state.

Gotsman and Yang [12] developed a program logic based on CSL, 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 we do.

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 system APIs 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.

It is worth noting that although our verification framework is developed to verify μC/OS-II, it is a general verification framework and most of its building blocks can be reused to verify other OS kernels. As shown in Fig. 1, the small-step semantics for the C subset, the program logic and the tactics are all general and mostly independent of the μC/OS-II verification project. A potential limitation is that the interrupt mechanism in our operational semantics is modeled specifically based on the Intel 8259 A interrupt controller, and the program logic rules for interrupts are designed accordingly. However, the logic rules follow the general ownership transfer idea from CSL. With a different processor and interrupt mechanism, even though we may need to change the current inference rules for interrupt primitives, we can apply the same ownership transfer idea, and the required change should be superficial. Another limitation is that our C subset is chosen based on the μC/OS-II code. In particular, it does not allow function pointers, which requires the support of higher-order functions in the logic.

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. The Verisoft XT Project (2007). http://www.verisoftxt.de
4. Alkassar, E., Paul, W.J., Starostin, A., Tsyban, A.: Pervasive verification of an OS microkernel. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 71–85. Springer, Heidelberg (2010)
5. Appel, A.W.: Tactics for separation logic (2006). http://www.cs.princeton.edu/~appel/papers/septacs.pdf
6. Babaoglu, O., Marzullo, K., Schneider, F.B.: A formalization of priority inversion. Real-Time Syst. **5**, 285–303 (1993)
7. Cao, J., Fu, M., Feng, X.: Practical tactics for verifying C programs in coq. In: CPP, pp. 97–108 (2015)
8. Chen, H., Wu, N., Shao, Z., Lockerman, J., Gu, R.: Toward compositional verification of interruptible os kernels and device drivers. In: PLDI (2016, to appear)
9. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
10. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
11. Feng, X., Shao, Z., Dong, Y., Guo, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. In: PLDI, pp. 170–182 (2008)
12. Gotsman, A., Yang, H.: Modular verification of preemptive OS kernels. J. Funct. Program. **23**(4), 452–514 (2013)
13. Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X.N., Weng, S.-C., Zhang, H., Guo, Y.: Deep specifications and certified abstraction layers. In: POPL, pp. 595–608 (2015)
14. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an operating-system kernel. Commun. ACM **53**(6), 107–115 (2010)
15. Klein, G., Andronick, J., Elphinstone, K., Murray, T.C., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst. **32**(1), 2 (2014)
16. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: sel4: Formal verification of an os kernel. In: SOSP, pp. 207–220 (2009)
17. Liang, H., Feng, X.: Modular verification of linearizability with non-fixed linearization points. In: PLDI, pp. 459–470 (2013)
18. Liang, H., Feng, X., Fu, M.: A rely-guarantee-based simulation for verifying concurrent program transformations. In: POPL, pp. 455–468 (2012)
19. Liang, H., Feng, X., Shao, Z.: Compositional verification of termination-preserving refinement of concurrent programs. In: CSL-LICS, pp. 65: 1–65: 10 (2014)
20. McCreight, A.: Practical tactics for separation logic. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 343–358. Springer, Heidelberg (2009)

21. O'Hearn, P.W.: Resources, concurrency and local reasoning. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 49–67. Springer, Heidelberg (2004)
22. Sevcík, J., Vafeiadis, V., Nardelli, F.Z., Jagannathan, S., Sewell, P.: Compcerttso: a verified compiler for relaxed-memory concurrency. J. ACM **60**(3), 22 (2013)
23. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: an approach to real-time synchronization. IEEE Trans. Comput. **39**, 1175–1185 (1990)
24. Turon, A., Dreyer, D., Birkedal, L.: Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In: ICFP, pp. 377–390 (2013)
25. Turon, A., Thamsborg, J., Ahmed, A., Birkedal, L., Dreyer, D.: Logical relations for fine-grained concurrency. In: POPL, pp. 343–356 (2013)
26. Xu, F., Fu, M., Feng, X., Zhang, X., Zhang, H., Li, Z.: A practical verification framework for preemptive OS kernels (technical report and coq implementations), May 2016. http://staff.ustc.edu.cn/~fuming/research/certiucos
27. Yang, J., Hawblitzel, C.: Safe to the last instruction: automated verification of a type-safe operating system. In: PLDI, pp. 99–110 (2010)