J Autom Reasoning (2009) 42:301–347
DOI 10.1007/s10817-009-9118-9

Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads

Xinyu Feng · Zhong Shao · Yu Guo · Yuan Dong

Received: 26 February 2009 / Accepted: 26 February 2009 / Published online: 20 March 2009

Abstract Hardware interrupts are widely used in the world's critical software systems to support preemptive threads, device drivers, operating system kernels, and hypervisors. Handling interrupts properly is an essential component of low-level system programming. Unfortunately, interrupts are also extremely hard to reason about: they dramatically alter the program control flow and complicate the invariants in low-level concurrent code (e.g., implementation of synchronization primitives). Existing formal verification techniques—including Hoare logic, typed assembly language, concurrent separation logic, and the assume-guarantee method—have consistently ignored the issues of interrupts; this severely limits the applicability and power of today's program verification systems. In this paper we present a novel Hoare-logic-

A preliminary version of this paper appeared in the Proceedings of ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI'08), pp. 170–182, ACM Press (2008).

X. Feng (✉)
Toyota Technological Institute at Chicago,
6045 S. Kenwood Avenue, Chicago, IL 60637, USA
e-mail: feng@tti-c.org

Z. Shao
Department of Computer Science, Yale University, 51 Prospect Street,
New Haven, CT 06520-8285, USA
e-mail: shao@cs.yale.edu

Y. Guo
Department of Computer Science and Technology,
University of Science and Technology of China, Hefei,
Anhui, 230026, China
e-mail: guoyu@mail.ustc.edu.cn

Y. Dong
Department of Computer Science and Technology, Tsinghua University,
Beijing, 100084, China
e-mail: dongyuan@tsinghua.edu.cn

like framework for certifying low-level system programs involving both hardware interrupts and preemptive threads. We show that enabling and disabling interrupts can be formalized precisely using simple ownership-transfer semantics, and the same technique also extends to the concurrent setting. By carefully reasoning about the interaction among interrupt handlers, context switching, and synchronization libraries, we are able to—for the first time—successfully certify a preemptive thread implementation and a large number of common synchronization primitives. Our work provides a foundation for reasoning about interrupt-based kernel programs and makes an important advance toward building fully certified operating system kernels and hypervisors.

Keywords Operating system verification · Hardware interrupts · Preemptive threads · Thread libraries · Synchronization primitives · Separation logic · Modularity

1 Introduction

Low-level system programs (*e.g.,* thread implementations, device drivers, operating system kernels, and hypervisors) form the backbone of almost every safety-critical software system in the world. It is thus highly desirable to formally certify the correctness of these programs. Indeed, there have been several new projects launched recently—including Verisoft/XT [14, 32], L4.verified [36], and Singularity [20]—all aiming to build certified OS kernels and/or hypervisors. With formal specifications and provably safe components, certified system software can provide a trustworthy computing platform and enable anticipatory statements about system configurations and behaviors [20].

Unfortunately, system programs—especially those involving both interrupts and concurrency—are extremely hard to reason about. In Fig. 1, we divide programs in a typical preemptible uniprocessor OS kernel into two layers. At the "higher" abstraction level, we have threads that follow the standard concurrent programming model [17]: interrupts are invisible, but the execution of a thread can be preempted by other threads; synchronization operations are treated as primitives.

Below this layer (see the shaded box), we have more subtle "lower-level" code involving both interrupts and concurrency. The implementation of many synchronization primitives and input/output operations requires explicit manipulation of

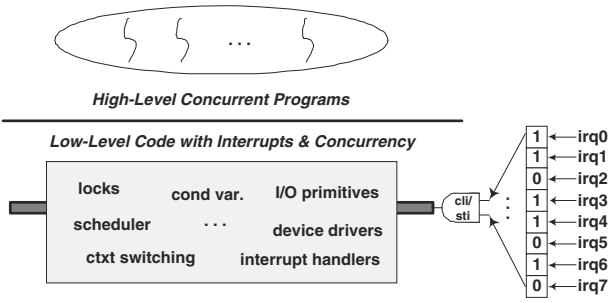


Fig. 1 "High-level" vs. "low-level" system programs

interrupts; they behave concurrently in a preemptive way (if interrupt is enabled) or a non-preemptive way (if interrupt is disabled). When execution of a thread is interrupted, control is transferred to an interrupt handler, which may call the thread scheduler and switch the control to another thread. Some of the code in the shaded box (e.g., the scheduler and context switching routine) may behave sequentially since they are always executed with interrupt disabled.

Existing program verification techniques (including Hoare logic [16], typed assembly language [26], concurrent separation logic [5, 29], and its assume-guarantee variant [8, 37]) can probably handle those high-level concurrent programs, but they have consistently ignored the issues of interrupts thus cannot be used to certify concurrent code in the shaded box. Having both explicit interrupts and threads creates the following new challenges:

- *Asymmetric preemption relations.* Non-handler code may be preempted by an interrupt handler (and low-priority handlers can be preempted by higher-priority ones), but not vice versa. Interrupt handlers cannot be simply treated as threads [33]: verification based on the standard thread semantics is too conservative and may give false positive reports of race conditions when interrupt handlers take advantage of their higher priorities.
- *Subtle intertwining between interrupts and threads.* In Fig. 2, thread A is interrupted by the interrupt request irq0. In the handler, the control is switched to thread B. From thread A's point of view, the behavior of the handler 0 is complex: should the handler be responsible for the behavior of thread B?
- *Asymmetric synchronizations.* Synchronization between handler and non-handler code is achieved simply by enabling and disabling interrupts (via sti and cli instructions in x86). Unlike locks, interrupts can be disabled by one thread and enabled by another. In Fig. 2, thread A disables interrupts and then switches control to thread B (step (5)), which will enable interrupts.
- Handler for higher-priority interrupts might be "interrupted" by lower-priority ones. In Fig. 2, handler 0 switches the control to thread B at step (1); thread B enables interrupts and is interrupted by irq1, which may have a lower-priority than irq0.

In this paper we tackle these challenges directly and present a novel framework for certifying low-level programs involving both interrupts and preemptive threads. We introduce a new abstract interrupt machine (named AIM, see Section 3 and the upper half of Fig. 3) to capture "interrupt-aware" concurrency, and use simple ownership-transfer semantics to reason about the interaction among interrupt handlers,

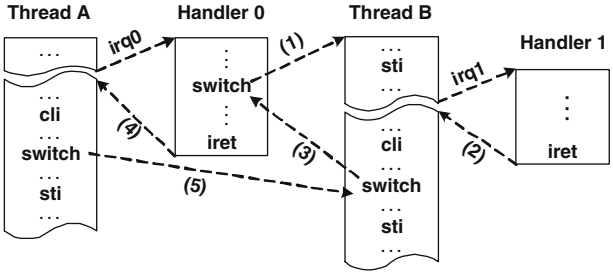


Fig. 2 Interaction between threads and interrupts

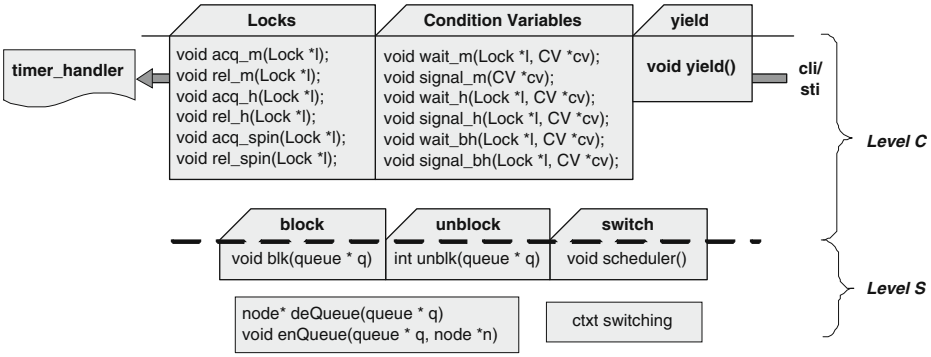


Fig. 3 Structure of our certified preemptive thread implementation

context switching, and synchronization libraries. Our paper makes the following new contributions:

- As far as we know, our work presents the first program logic (see Section 4) that can successfully certify the correctness of low-level programs involving both interrupts and concurrency. Our idea of using ownership-transfer semantics to model interrupts is both novel and general (since it also works in the concurrent setting). Our logic supports modular verification: threads and handlers can be certified in the same way as we certify sequential code without worrying about possible interleaving. Soundness of our logic is formally proved in the Coq proof assistant.
- Following separation logic's local-reasoning idea, our program logic also enforces partitions of resources between different threads and between threads and interrupt handlers. These logical partitions at different program points essentially give an abstract formalization of the semantics of interrupts and the interaction between handlers and threads.
- Our AIM machine (see Section 3) unifies both the preemptive and non-preemptive threading models, and to our best knowledge, is the first to successfully formalize concurrency with explicit interrupt handlers. In AIM, operations that manipulate thread queues are treated as primitives; These operations, together with the scheduler and context-switching code (the low half of Fig. 3), are strictly sequential thus can be certified in a simpler logic. Certified code at different levels is linked together using an OCAP-style framework [9, 12].
- Synchronization operations can be implemented as subroutines in AIM. To demonstrate the power of our framework, we have certified, for the first time, various implementations of locks and condition variables (see Section 5). Our specifications pinpoint precisely the differences between different implementations.

2 Informal Development

Before presenting our formal framework, we first informally explain the design of our abstract machine and the ownership-transfer semantics for reasoning about interrupts.

2.1 Design of the Abstract Machine

In Fig. 3 we outline the structure of a thread implementation taken from a simplified OS kernel. We split all "shaded" code into two layers: the upper level C (for "Concurrent") and the low level S (for "Sequential"). Code at Level C is concurrent; it handles interrupts explicitly and implements interrupt handlers but abstracts away the implementation of threads. Code at Level S is sequential (always executed with interrupts disabled); functions that need to know the concrete representations of thread control blocks (TCBs) and thread queues are implemented at Level S; there are one queue for ready threads and multiple queues for blocked threads.

We implement three primitive thread operations at Level S: switch, block, and unblock. The switch primitive, shown as the `scheduler()` function in Fig. 3, saves the execution context (consisting of the program counter, the stack pointer and other states) of the current thread into its TCB, put the TCB into the ready queue, picks another TCB from the queue, and switches to the execution context of the new thread. The block primitive takes a pointer to a block queue as argument, puts the current thread into the block queue, and switches the control to a thread in the ready queue. The unblock primitive also takes a pointer to a block queue as argument; it moves a thread from the block queue to the ready queue but does not do context switching. Level S also contains code for queue operations and thread context switching, which are called by these thread primitives.

In the abstract machine at Level C, we use instructions sti/cli to enable/disable interrupts (as on x86 processors); the primitives switch, block and unblock are also treated as instructions; thread queues are now abstract algebraic structures outside of the data heap and can only be accessed via the thread primitives.

2.2 Ownership-Transfer Semantics

Concurrent entities, *i.e.,* the handler code and the threads consisting of the non-handler code, all need to access memory. To guarantee the non-interference, we enforce the following invariant, inspired by recent work on Concurrent Separation Logic [5, 29]: *there always exists a partition of memory among the concurrent entities, and each entity can only access its own part of memory*. There are two important points about this invariant:

- the partition is *logical*; we do not need to change our model of the physical machine, which only has one global shared data heap. The logical partition can be enforced following Separation Logic [21, 34], as we will explain below.
- the partition is not static; it can be dynamically adjusted during program execution, which is done by transferring the ownership of memory from one entity to the other.

Instead of using the operational semantics of cli, sti and thread primitives described above to reason about programs, we model their semantics in terms of memory ownership transfers. This semantics completely hides thread queues and thus the complex interleaving between concurrent entities.

We first study the semantics of cli and sti assuming that the non-handler code is single-threaded. Since the interrupt handler can preempt the non-handler code but not vice versa, we reserve the part of memory used by the handler from the global

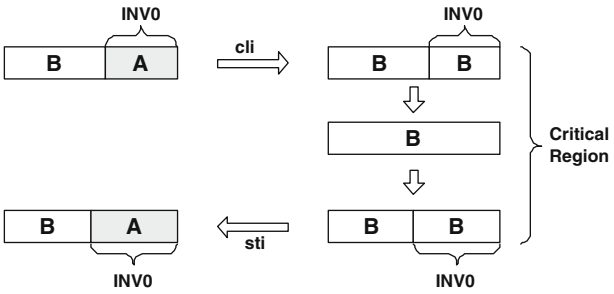


Fig. 4 Memory partition for handler and non-handler

memory, shown as block A in Fig. 4. Block A needs to be well-formed with respect to the precondition of the handler, which ensures safe execution of the handler code. We call the precondition an invariant INV0, since the interrupt may come at any program point (as long as it is enabled) and this precondition needs to always hold. If the interrupt is enabled, the non-handler code can only access the rest part of memory, called block B. If it needs to access block A, it has to first disable the interrupt by cli. Therefore we can model the semantics of cli as a transfer of ownership of the *well-formed* block A, as shown in Fig. 4. The non-handler code does not need to preserve the invariant INV0 if the interrupt is disabled, but it needs to ensure INV0 holds before it enables the interrupt again using sti. The sti instruction returns the well-formed block A to the interrupt handler.

If the non-handler code is multi-threaded, we also need to guarantee non-interference between these threads. Figure 5 refines the memory model. The block A is still dedicated to the interrupt handler. The memory block B is split into three parts (assuming there are only two threads): each thread has its own private memory, and both threads share the block C. When block C is available for sharing, it needs to be well-formed with some specification INV1. However, a thread cannot directly access block C if the interrupt is enabled, even if the handler does not access it. That is because the handler may switch to another thread, as shown in Fig. 2 (step (1)). To access block A and C, the current thread, say T_1, needs to disable the interrupt; so cli grants T_1 the ownership of *well-formed* blocks A and C. If T_1 wants to switch control to T_2, it first makes sure that INV0 and INV1 hold over A and C respectively. The

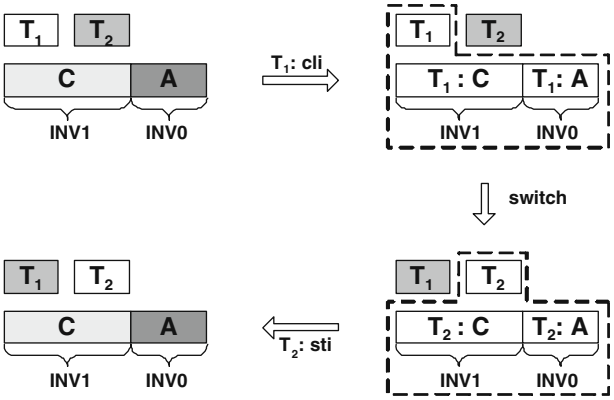


Fig. 5 The memory model for multi-threaded non-handler

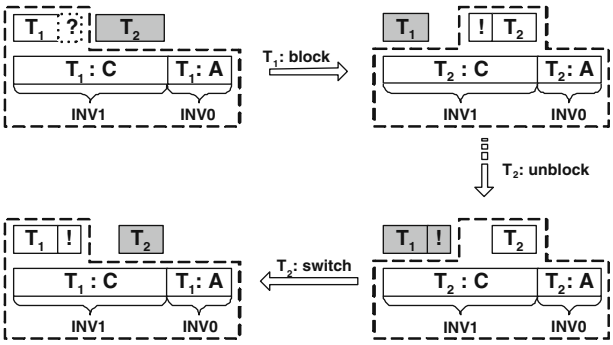


Fig. 6 Block and unblock

switch operation transfers the ownership of A and C from T_1 to T_2, knowing that the interrupt remains disabled. Enabling the interrupt (by T_2) releases the ownership.

Blocking thread queues are used to implement synchronization primitives, such as locks or condition variables. When the lock is not available, or the condition associated with the condition variable does not hold, the current thread is put into the corresponding block queue. We can also model the semantics of block and unblock as resource ownership transfers: a blocked thread is essentially waiting for the availability of some resource, *e.g.,* the lock and the resource protected by the lock, or the resource over which the condition associated with the condition variable holds. As shown in Fig. 6, thread T_1 executes block when it waits for some resource (represented as the dashed box containing "?"). Since block switches control to other threads, T_1 needs to ensure that INV0 and INV1 hold over A and C, which is the same requirement as switch. When T_2 makes the resource available, it executes unblock to release a thread in the corresponding block queue, and transfers the ownership of the resource to the released thread. Note that unblock itself does not do context switching. When T_1 takes control again, it will own the resource. From T_1's point of view, the block operation acquires the resource associated with the corresponding block queue. This view of block and unblock is very flexible: by choosing whether the resource is empty or not, we can certify implementations of Mesa- and Hoare-style condition variables (see Section 5).

3 The Abstract Interrupt Machine (AIM)

We present our Abstract Interrupt Machine (AIM) in two steps. AIM-1 shows the interaction between the handler and sequential non-handler code. AIM-2, the final definition of AIM, extends AIM-1 with multi-threaded non-handler code.

3.1 AIM-1

AIM-1 is defined in Fig. 7. The whole machine configuration $\mathbb{W}$ consists of a code heap $\mathbb{C}$, a mutable program state $\mathbb{S}$, a control stack $\mathbb{K}$, and a program counter pc. The code heap $\mathbb{C}$ is a finite partial mapping from code labels f to commands c (represented as $\{\mathtt{f} \rightsquigarrow \mathtt{c}\}^*$). Each command c is either a sequential or branch instruction ι, or jump or return instructions. The state $\mathbb{S}$ contains a data heap $\mathbb{H}$, a

$$
\begin{array}{rlll}
(\textit{World}) & \mathbb{W} & ::= & (\mathbb{C}, \mathbb{S}, \mathbb{K}, \mathtt{pc}) \\
(\textit{CodeHeap}) & \mathbb{C} & ::= & \{\mathtt{f} \rightsquigarrow \mathtt{c}\}^* \\
(\textit{State}) & \mathbb{S} & ::= & (\mathbb{H}, \mathbb{R}, \mathtt{ie}, \mathtt{is}) \\
(\textit{Heap}) & \mathbb{H} & ::= & \{\mathtt{l} \rightsquigarrow \mathtt{w}\}^* \\
(\textit{RegFile}) & \mathbb{R} & ::= & \{r_0 \rightsquigarrow \mathtt{w}_0, \ldots, r_k \rightsquigarrow \mathtt{w}_k\} \\
(\textit{Stack}) & \mathbb{K} & ::= & \mathsf{nil} \mid \mathtt{f} :: \mathbb{K} \mid (\mathtt{f}, \mathbb{R}) :: \mathbb{K} \\
(\textit{Bit}) & \mathtt{b} & ::= & 0 \mid 1 \\
(\textit{Flags}) & \mathtt{ie}, \mathtt{is} & ::= & \mathtt{b} \\
(\textit{Labels}) & \mathtt{l}, \mathtt{f}, \mathtt{pc} & ::= & n \ \ (\textit{nat nums}) \\
(\textit{Word}) & \mathtt{w} & ::= & i \ \ (\textit{integers}) \\
(\textit{Register}) & r & ::= & \mathtt{r}_0 \mid \mathtt{r}_1 \mid \ldots \\
(\textit{Instr}) & \iota & ::= & \mathsf{mov}\ r_d, r_s \mid \mathsf{movi}\ r_d, \mathtt{w} \mid \mathsf{add}\ r_d, r_s \mid \mathsf{sub}\ r_d, r_s \mid \mathsf{ld}\ r_d, \mathtt{w}(r_s) \mid \mathsf{st}\ \mathtt{w}(r_t), r_s \\
 & & & \mid \mathsf{beq}\ r_s, r_t, \mathtt{f} \mid \mathsf{call}\ \mathtt{f} \mid \mathsf{cli} \mid \mathsf{sti} \\
(\textit{Commd}) & \mathtt{c} & ::= & \iota \mid \mathsf{j}\ \mathtt{f} \mid \mathsf{ret} \mid \mathsf{iret} \\
(\textit{InstrSeq}) & \mathbb{I} & ::= & \iota; \mathbb{I} \mid \mathsf{j}\ \mathtt{f} \mid \mathsf{ret} \mid \mathsf{iret}
\end{array}
$$

Fig. 7 Definition of AIM-1

register file $\mathbb{R}$, and flags `ie` and `is`. The data heap is modeled as a finite partial mapping from labels to integers. The register file is a total function that maps register names to integers. The binary flags `ie` and `is` record whether the interrupt is disabled, and whether it is currently being serviced, respectively. The abstract control stack $\mathbb{K}$ saves the return address of the current function or the interrupt handler. An empty stack is represented as nil. Each stack frame contains either a code label `f` or a pair $(\mathtt{f}, \mathbb{R})$. The latter is pushed onto the stack when the interrupt handler is triggered. More details are shown in the operational semantics below. The program counter `pc` is a code label pointing to the current command in $\mathbb{C}$.

A command `c` can be an *instruction* ι, a "jump", a "return" from functions (ret) or a "return " from interrupt handlers (iret). An instruction can be a sequential operation (*e.g.,* "move", arithmetic operations, or memory "load" and "store"), conditional branch or a function call. The registers r_s, r_t and r_d represent the source, temporary and target registers respectively. For simplicity, here we only show the most common instructions. We also define the instruction sequence $\mathbb{I}$ as a sequence of instructions ending with a jump or return command (*i.e.,* a basic block). $\mathbb{C}[\mathtt{f}]$ extracts an instruction sequence starting from `f` in $\mathbb{C}$, as defined in Fig. 8.

Figure 8 also defines some representations used in this paper. The function update is represented as $F\{a \rightsquigarrow b\}$, which maps a into b and all other arguments x into $F(x)$. We use the dot notation to represent a component in a tuple, *e.g.,* $\mathbb{S}.\mathbb{H}$ means the

Fig. 8 Definition of representations

$$
\mathbb{C}[\mathtt{f}] \stackrel{\text{def}}{=} \begin{cases} \mathtt{c} & \mathtt{c} = \mathbb{C}(\mathtt{f}) \text{ and } \mathtt{c} = \mathsf{j}\ \mathtt{f}', \mathsf{ret}, \text{ or } \mathsf{iret} \\ \iota; \mathbb{I} & \iota = \mathbb{C}(\mathtt{f}) \text{ and } \mathbb{I} = \mathbb{C}[\mathtt{f}+1] \\ \textit{undefined} & \text{otherwise} \end{cases}
$$

$$
(F\{a \rightsquigarrow b\})(x) \stackrel{\text{def}}{=} \begin{cases} b & \text{if } x = a \\ F(x) & \text{otherwise .} \end{cases}
$$

$$
\mathbb{S}|_{\mathbb{H}'} \stackrel{\text{def}}{=} (\mathbb{H}', \mathbb{S}.\mathbb{R}, \mathbb{S}.\mathtt{ie}, \mathbb{S}.\mathtt{is}) \qquad \mathbb{S}|_{\{\mathtt{ie}=\mathtt{b}\}} \stackrel{\text{def}}{=} (\mathbb{S}.\mathbb{H}, \mathbb{S}.\mathbb{R}, \mathtt{b}, \mathbb{S}.\mathtt{is})
$$

$$
\mathbb{S}|_{\mathbb{R}'} \stackrel{\text{def}}{=} (\mathbb{S}.\mathbb{H}, \mathbb{R}', \mathbb{S}.\mathtt{ie}, \mathbb{S}.\mathtt{is}) \qquad \mathbb{S}|_{\{\mathtt{is}=\mathtt{b}\}} \stackrel{\text{def}}{=} (\mathbb{S}.\mathbb{H}, \mathbb{S}.\mathbb{R}, \mathbb{S}.\mathtt{ie}, \mathtt{b})
$$

data heap in state $\mathbb{S}$. $\mathbb{S}|_{\mathbb{H}'}$ is the new state that changes the data heap in $\mathbb{S}$ to $\mathbb{H}'$. $\mathbb{S}|_{\mathbb{R}'}$, $\mathbb{S}|_{\{\mathtt{ie}=b\}}$ and $\mathbb{S}|_{\{\mathtt{is}=b\}}$ are defined similarly.

Operational semantics We use a non-deterministic operational semantics to model the hardware interrupt request. Instead of using an oracle telling us when the interrupts would come, we assume the interrupt request may come at any time. The binary step relation $\Longmapsto$ models single-step transitions of program configurations:

$$\mathbb{W} \Longmapsto \mathbb{W}' \stackrel{\text{def}}{=} (\mathbb{W} \longmapsto \mathbb{W}') \vee (\mathbb{W} \lightning \mathbb{W}') \tag{1}$$

At each step, the machine either executes the next instruction at pc ($\mathbb{W} \longmapsto \mathbb{W}'$) or jumps to the interrupt handler to handle the incoming interrupt request ($\mathbb{W} \lightning \mathbb{W}'$). We use $\mathbb{W} \Longmapsto^n \mathbb{W}'$ to mean $\mathbb{W}'$ is reached from $\mathbb{W}$ in n steps:

$$\frac{}{\mathbb{W} \Longmapsto^0 \mathbb{W}} \qquad \frac{\mathbb{W} \Longmapsto \mathbb{W}'' \quad \mathbb{W}'' \Longmapsto^n \mathbb{W}'}{\mathbb{W} \Longmapsto^{n+1} \mathbb{W}'}$$

$\mathbb{W} \Longmapsto^* \mathbb{W}'$ is defined as $\exists n.\ \mathbb{W} \Longmapsto^n \mathbb{W}'$.

The following IRQ rule defines the transition relation $\mathbb{W} \lightning \mathbb{W}'$.

$$\frac{\mathtt{ie} = 1 \quad \mathtt{is} = 0}{(\mathbb{C}, (\mathbb{H}, \mathbb{R}, \mathtt{ie}, \mathtt{is}), \mathbb{K}, \mathtt{pc}) \lightning (\mathbb{C}, (\mathbb{H}, \mathbb{R}, 0, 1), (\mathtt{pc}, \mathbb{R}) :: \mathbb{K}, \mathtt{h\_entry})} \ (\text{IRQ})$$

An incoming interrupt request is processed only if the ie bit is set, and no interrupt is currently being serviced (*i.e.,* is = 0). The processor saves the execution context (pc, $\mathbb{R}$) of the current program onto the stack $\mathbb{K}$, clears the ie bit, sets is to 1, and sets the new pc to h_entry. To simplify the presentation, the machine supports only one interrupt with a global interrupt handler entry h_entry. It can be extended easily to support multi-level interrupts. We discuss about the extension in Section 7.

$\mathtt{NextS}_{(\mathtt{c},\mathbb{K})}\ \mathbb{S}\ \mathbb{S}'$ where $\mathbb{S} = (\mathbb{H}, \mathbb{R}, \mathtt{ie}, \mathtt{is})$	
if c =	$\mathbb{S}' =$
mov r_d, r_s	$(\mathbb{H}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{R}(r_s)\}, \mathtt{ie}, \mathtt{is})$
movi r_d, w	$(\mathbb{H}, \mathbb{R}\{r_d \rightsquigarrow \mathtt{w}\}, \mathtt{ie}, \mathtt{is})$
add r_d, r_s	$(\mathbb{H}, \mathbb{R}\{r_d \rightsquigarrow (\mathbb{R}(r_s) + \mathbb{R}(r_d))\}, \mathtt{ie}, \mathtt{is})$
sub r_d, r_s	$(\mathbb{H}, \mathbb{R}\{r_d \rightsquigarrow (\mathbb{R}(r_d) - \mathbb{R}(r_s))\}, \mathtt{ie}, \mathtt{is})$
ld r_d, w(r_s)	$(\mathbb{H}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{H}(\mathbb{R}(r_s) + \mathtt{w})\}, \mathtt{ie}, \mathtt{is})$ if $(\mathbb{R}(r_s) + \mathtt{w}) \in dom(\mathbb{H})$
st w(r_t), r_s	$(\mathbb{H}\{(\mathbb{R}(r_t) + \mathtt{w}) \rightsquigarrow \mathbb{R}(r_s)\}, \mathbb{R}, \mathtt{ie}, \mathtt{is})$ if $(\mathbb{R}(r_t) + \mathtt{w}) \in dom(\mathbb{H})$
cli	$\mathbb{S}|_{\{\mathtt{ie}=0\}}$
sti	$\mathbb{S}|_{\{\mathtt{ie}=1\}}$
iret	$(\mathbb{H}, \mathbb{R}', 1, 0)$ if is = 1, $\mathbb{K} = (\mathtt{f}, \mathbb{R}') :: \mathbb{K}'$ for some f and $\mathbb{K}'$
other cases	$\mathbb{S}$

$\mathtt{NextK}_{(\mathtt{pc},\mathtt{c})}\ \mathbb{K}\ \mathbb{K}'$	
if c =	$\mathbb{K}' =$
call f	$(\mathtt{pc}+1) :: \mathbb{K}$
ret	$\mathbb{K}''$ if $\mathbb{K} = \mathtt{f} :: \mathbb{K}''$ for some f
iret	$\mathbb{K}''$ if $\mathbb{K} = (\mathtt{f}, \mathbb{R}) :: \mathbb{K}''$ for some f and $\mathbb{R}$
other cases	$\mathbb{K}$

$\mathtt{NextPC}_{(\mathtt{c},\mathbb{R},\mathbb{K})}\ \mathtt{pc}\ \mathtt{pc}'$	
if c =	$\mathtt{pc}' =$
beq r_s, r_t, f	f if $\mathbb{R}(r_s) = \mathbb{R}(r_t)$
beq r_s, r_t, f	pc + 1 if $\mathbb{R}(r_s) \neq \mathbb{R}(r_t)$
call f	f
j f	f
ret	f if $\mathbb{K} = \mathtt{f} :: \mathbb{K}'$ for some $\mathbb{K}'$
iret	f if $\mathbb{K} = (\mathtt{f}, \mathbb{R}') :: \mathbb{K}'$ for some $\mathbb{K}'$ and $\mathbb{R}'$
other cases	pc + 1

Fig. 9 Operational semantics of instructions

The program transition $\mathbb{W} \longmapsto \mathbb{W}'$ models the execution of the next instruction at pc. It is defined by the PC rule below:

$$\frac{\begin{array}{l}\mathtt{c} = \mathbb{C}(\mathtt{pc}) \\ \mathtt{NextS}_{(\mathtt{c},\mathbb{K})}\ \mathbb{S}\ \mathbb{S}' \quad \mathtt{NextK}_{(\mathtt{pc},\mathtt{c})}\ \mathbb{K}\ \mathbb{K}' \quad \mathtt{NextPC}_{(\mathtt{c},\mathbb{S},\mathbb{R},\mathbb{K})}\ \mathtt{pc}\ \mathtt{pc}'\end{array}}{(\mathbb{C}, \mathbb{S}, \mathbb{K}, \mathtt{pc}) \longmapsto (\mathbb{C}, \mathbb{S}', \mathbb{K}', \mathtt{pc}')}\ (\textsc{pc})$$

where the auxiliary relations $\mathtt{NextS}_{(\mathtt{c},\mathbb{K})}$, $\mathtt{NextK}_{(\mathtt{pc},\mathtt{c})}$ and $\mathtt{NextPC}_{(\mathtt{c},\mathbb{R},\mathbb{K})}$ are defined in Fig. 9. The relation $\mathtt{NextS}_{(\mathtt{c},\mathbb{K})}$ shows the transition of states by executing c with stack $\mathbb{K}$; $\mathtt{NextK}_{(\mathtt{pc},\mathtt{c})}$ describes the change of stacks made by c at the program counter pc; while $\mathtt{NextPC}_{(\mathtt{c},\mathbb{R},\mathbb{K})}$ shows how pc changes after c is executed with $\mathbb{R}$ and $\mathbb{K}$. Semantics of most instructions are straightforward, except iret which runs at the end of each interrupt handler and does the following:

- pops the stack frame on the top of the stack $\mathbb{K}$; the frame must be in the form of $(\mathtt{f}, \mathbb{R}')$, which is saved when the interrupt is handled (see the IRQ rule);
- restores ie and is with the value when the interrupt occurs, which must be 1 and 0 respectively (otherwise the interrupt cannot have been handled);
- resets the pc and the register file $\mathbb{R}$ with f and $\mathbb{R}'$, respectively.

In AIM, the register file $\mathbb{R}$ is automatically saved and restored at the entry and exit point of the interrupt handler. This is a simplification of the x86 interrupt mechanism for a cleaner presentation. In our implementation (Section 6), the interrupt handler code needs to save and restore the registers.

```
incleft: -{(p0, NoG)}
         movi $r1, RIGHT
         movi $r2, LEFT
l_loop:  -{(p1, NoG)}
         movi $r3, 0
         cli
         -{(p2, NoG)}
         ld   $r4, 0($r1)
         beq  $r3, $r4, l_win
         movi $r3, 1
         sub  $r4, $r3
         st   0($r1), $r4
         ld   $r4, 0($r2)
         add  $r4, $r3
         st   0($r2), $r4
         sti
         -{(p1, NoG)}
         j    l_loop
l_win:   -{(p3, NoG)}
         sti
         j    l_loop
```

```
h_entry: -{(pi, gi)}
         movi $r1, LEFT
         movi $r2, RIGHT
         movi $r3, 0
         ld   $r4, 0($r1)
         beq  $r3, $r4, r_win
         movi $r3, 1
         sub  $r4, $r3
         st   0($r1), $r4
         ld   $r4, 0($r2)
         add  $r4, $r3
         st   0($r2), $r4
         iret
r_win:   -{(p4, gid)}
         iret
```

Fig. 10 Sample AIM-1 program: Teeter-Totter

Note that, given a $\mathbb{W}$, there may not always exist a $\mathbb{W}'$ such that $(\mathbb{W} \longmapsto \mathbb{W}')$ holds. If there is no such $\mathbb{W}'$, we say the program *aborts* at $\mathbb{W}$:

$$\frac{\neg\exists \mathbb{W}'.\ \mathbb{W} \longmapsto \mathbb{W}'}{\mathbb{W} \Longmapsto \mathsf{abort}} \qquad \frac{n > 0 \quad \mathbb{W} \Longmapsto^{n-1} \mathbb{W}' \quad \mathbb{W}' \Longmapsto \mathsf{abort}}{\mathbb{W} \Longmapsto^{n} \mathsf{abort}}$$

$\mathbb{W} \Longmapsto^n \mathsf{abort}$ means the execution starting from $\mathbb{W}$ aborts at the n-th step. One important goal of our program logic is to show that certified programs never abort.

Figure 10 shows a sample AIM-1 program. The non-handler code (on the left) and the interrupt handler (on the right) share two memory cells at locations `LEFT` and `RIGHT`. They initially contain the same value (say, 50). The non-handler increases the value stored at `LEFT` and decrease the value at `RIGHT`. The interrupt handler code does the reverse. One wins if it decreases the value of the other side to 0. Therefore which side wins depends on how frequent the interrupt comes. To avoid races, the non-handler code always disables interrupts before it accesses `LEFT` and `RIGHT`. We will explain the program specifications in shaded boxes and the verification of the program in Section 4.

3.2 AIM-2

Figure 11 defines AIM-2 as an extension over AIM-1. We extend the world $\mathbb{W}$ with an abstract thread queue $\mathbb{T}$, a set of block queues $\mathbb{B}$, and the id `tid` for the current thread. $\mathbb{T}$ maps a thread id to a thread execution context, which contains the register file, the stack, the `is` flag and `pc`. $\mathbb{B}$ maps block queue ids w to block queues $\mathbb{Q}$. These block queues are used to implement synchronization primitives such as locks and condition variables. $\mathbb{Q}$ is a set of thread ids pointing to thread contexts in $\mathbb{T}$. Note here we do not need a separate $\mathbb{Q}$ for ready threads, which are threads in $\mathbb{T}$ but not blocked:

$$\mathtt{readyQ}(\mathbb{T}, \mathbb{B}) \stackrel{\text{def}}{=} \{\mathtt{tid} \mid \mathtt{tid} \in \mathit{dom}(\mathbb{T}) \wedge \neg\exists w.\ \mathtt{tid} \in \mathbb{B}(w)\}\,. \tag{2}$$

We also add three primitive instructions: switch, block and unblock.

The step relation $(\mathbb{W} \longmapsto \mathbb{W}')$ of AIM-2 is defined in Fig. 12. The switch instruction saves the execution context of the current thread into the thread queue $\mathbb{T}$, and picks a thread nondeterministically from $\mathtt{readyQ}(\mathbb{T}, \mathbb{B})$ to run. To let our abstraction fit into the interfaces shown in Fig. 3, we require that the interrupt be disabled before switch. This also explains why `ie` is not saved in the thread context, and why it is set to 0 when a new thread is scheduled from $\mathbb{T}$: the only way to switch control from one thread to the other is to execute switch, which can be executed only if the interrupt is disabled. The "block r_t" instruction puts the current thread id into the block queue $\mathbb{B}(r_t)$, and switches the control to another thread in $\mathtt{readyQ}(\mathbb{T}, \mathbb{B})$. If there are no other threads in `readyQ`, the machine stutters (in our x86 implementation, this would never happen because there is an idle thread and our program logic prohibits it

$$\begin{array}{llll}
(\mathit{World}) & \mathbb{W} ::= (\mathbb{C}, \mathbb{S}, \mathbb{K}, \mathtt{pc}, \mathtt{tid}, \mathbb{T}, \mathbb{B}) & & \\
(\mathit{ThrdSet}) & \mathbb{T} ::= \{\mathtt{tid} \rightsquigarrow (\mathbb{R}, \mathbb{K}, \mathtt{is}, \mathtt{pc})\}^* & (\mathit{ThrdID}) & \mathtt{tid} ::= n \ \ (\mathit{nat\ nums,\ and}\ n > 0) \\
(\mathit{BlkQSet}) & \mathbb{B} ::= \{w \rightsquigarrow \mathbb{Q}\}^* & (\mathit{qID}) & w ::= n \ \ (\mathit{nat\ nums,\ and}\ n > 0) \\
(\mathit{ThrdQ}) & \mathbb{Q} ::= \{\mathtt{tid}_1, \ldots, \mathtt{tid}_n\} & (\mathit{Instr}) & \iota ::= \ldots \mid \mathsf{switch} \mid \mathsf{block}\ r_t \mid \mathsf{unblock}\ r_t, r_d \mid \ldots
\end{array}$$

Fig. 11 AIM-2 defined as an extension of AIM-1

$(\mathbb{C}, \mathbb{S}, \mathbb{K}, \mathtt{pc}, \mathtt{tid}, \mathbb{T}, \mathbb{B}) \longmapsto \mathbb{W}'$ where $\mathbb{S} = (\mathbb{H}, \mathbb{R}, \mathtt{ie}, \mathtt{is})$	
if $\mathbb{C}(\mathtt{pc}) =$	$\mathbb{W}' =$
switch	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}', 0, \mathtt{is}'), \mathbb{K}', \mathtt{pc}', \mathtt{tid}', \mathbb{T}', \mathbb{B})$ if $\mathtt{ie} = 0$, $\mathbb{T}' = \mathbb{T}\{\mathtt{tid} \rightsquigarrow (\mathbb{R}, \mathbb{K}, \mathtt{is}, \mathtt{pc}+1)\}$, $\mathtt{tid}' \in \mathtt{readyQ}(\mathbb{T}, \mathbb{B})$, and $\mathbb{T}'(\mathtt{tid}') = (\mathbb{R}', \mathbb{K}', \mathtt{is}', \mathtt{pc}')$
block r_t	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}', \mathtt{ie}, \mathtt{is}'), \mathbb{K}', \mathtt{pc}', \mathtt{tid}', \mathbb{T}', \mathbb{B}')$ if $\mathtt{ie} = 0$, $w = \mathbb{R}(r_t)$, $\mathbb{B}(w) = \mathbb{Q}$, $\mathbb{B}' = \mathbb{B}\{w \rightsquigarrow (\mathbb{Q} \cup \{\mathtt{tid}\})\}$, $\mathtt{tid}' \in \mathtt{readyQ}(\mathbb{T}, \mathbb{B}')$, $\mathbb{T}(\mathtt{tid}') = (\mathbb{R}', \mathbb{K}', \mathtt{is}', \mathtt{pc}')$ and $\mathbb{T}' = \mathbb{T}\{\mathtt{tid} \rightsquigarrow (\mathbb{R}, \mathbb{K}, \mathtt{is}, \mathtt{pc}+1)\}$
block r_t	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}, \mathtt{ie}, \mathtt{is}), \mathbb{K}, \mathtt{pc}, \mathtt{tid}, \mathbb{T}, \mathbb{B})$ if $\mathtt{ie} = 0$, and $\mathtt{readyQ}(\mathbb{T}, \mathbb{B}) = \{\mathtt{tid}\}$
unblock r_t, r_d	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}', \mathtt{ie}, \mathtt{is}), \mathbb{K}, \mathtt{pc}+1, \mathtt{tid}, \mathbb{T}, \mathbb{B})$ if $\mathtt{ie} = 0$, $w = \mathbb{R}(r_t)$, $\mathbb{B}(w) = \emptyset$, and $\mathbb{R}' = \mathbb{R}\{r_d \rightsquigarrow 0\}$
unblock r_t, r_d	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}', \mathtt{ie}, \mathtt{is}), \mathbb{K}, \mathtt{pc}+1, \mathtt{tid}, \mathbb{T}, \mathbb{B}')$ if $\mathtt{ie} = 0$, $w = \mathbb{R}(r_t)$, $\mathbb{B}(w) = \mathbb{Q} \uplus \{\mathtt{tid}'\}$, $\mathbb{B}' = \mathbb{B}\{w \rightsquigarrow \mathbb{Q}\}$, and $\mathbb{R}' = \mathbb{R}\{r_d \rightsquigarrow \mathtt{tid}'\}$
other c	$(\mathbb{C}, \mathbb{S}', \mathbb{K}', \mathtt{pc}', \mathtt{tid}, \mathbb{T}, \mathbb{B})$ if $\mathtt{NextS}_{(\mathtt{c},\mathbb{K})}\ \mathbb{S}\ \mathbb{S}'$, $\mathtt{NextK}_{(\mathtt{pc},\mathtt{c})}\ \mathbb{K}\ \mathbb{K}'$, and $\mathtt{NextPC}_{(\mathtt{c},\mathbb{R},\mathbb{K})}\ \mathtt{pc}\ \mathtt{pc}'$

Fig. 12 The step relation for AIM-2

from executing block). The "unblock r_t, r_d" instruction removes a thread from $\mathbb{B}(r_t)$ and puts its `tid` into r_d if the queue is not empty; otherwise r_d contains 0. Here $\uplus$ represents the union of two disjoint sets. By the definition of `readyQ`, we know `tid` will be in `readyQ` after being unblocked. unblock does not switch controls. Like switch, block and unblock can be executed only if the interrupt is disabled. The effects of other instructions over $\mathbb{S}$, $\mathbb{K}$ and `pc` are the same as in AIM-1. They do not change $\mathbb{T}$, $\mathbb{B}$ and `tid`. The transition $(\mathbb{W} \not\downarrow \mathbb{W}')$ for AIM-2 is almost the same as the one for AIM-1 defined by the IRQ rule. It does not change $\mathbb{T}$, $\mathbb{B}$ and `tid` either. Note that our threads are at the kernel level. Just as the interrupt handler and the non-handler code share stacks in AIM-1, here we let the interrupt handler share the stack space with the interrupted thread. The definition of $(\mathbb{W} \Longmapsto \mathbb{W}')$ is unchanged.

Fig. 13 A preemptive timer handler

```
h_entry: -{(p_i, g_i)}
         j      h_timer
h_timer: -{(p_i, g_i)}
         movi   $r1, CNT
         ld     $r2, 0($r1)          ; $r2 <- [CNT]
         movi   $r3, 100
         beq    $r2, $r3, schd       ; if ([CNT]=100)
         movi   $r3, 1               ;         goto schd
         add    $r2, $r3
         st     0($r1), $r2          ; [CNT]++
         iret
schd:    -{(p_0, g_0)}
         movi   $r2, 0
         st     0($r1), $r2          ; [CNT] := 0
         switch
         iret
```

$$\mathtt{p}_0 \stackrel{\text{def}}{=} \mathsf{enable}_{\mathsf{iret}} \wedge (r_1 = \mathtt{CNT})$$

$$\mathtt{g}_0 \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathtt{CNT} \mapsto - \\ \mathsf{INV0} \end{array} \right\} \wedge (\mathtt{ie} = \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}')$$

Our AIM machine is designed for uniprocessor systems. A thread cannot be preempted directly by other threads, but it can be preempted by interrupt handers, which may switch the execution to another thread. For higher-level concurrent programs (see Fig. 1), the design of AIM is very interesting in that it supports both preemptive threads (if the interrupt is enabled and the handler does context switching) and non-preemptive ones (if the interrupt is disabled, or if the interrupt is enabled but the handler does no context switching).

A preemptive timer interrupt handler Figure 13 shows the implementation of a preemptive timer interrupt handler. Each time the interrupt comes, the handler tests the value of the counter at memory location `CNT`. If the counter reaches 100, the handler switches control to other threads; otherwise it increases the counter by 1 and returns to the interrupted thread. We will explain the meanings of specifications and show how the timer handler is verified in Section 4.

4 The Program Logic

We propose a Hoare-style program logic to verify the safety and partial correctness of AIM programs. To verify programs, the programmer writes specifications specifying their functionalities, and then applies our logical rules to prove their "well-formedness" with respect to the specifications. The soundness of our logic guarantees that well-formed programs are safe to execute and their behaviors indeed satisfy the specifications.

4.1 Assertions and Specifications

Instead of defining a new logic to write assertions, we use the mechanized *meta-logic* implemented in the Coq proof assistant [6] as our assertion language. The logic corresponds to Higher-Order Logic with inductive definitions. This approach is known as "shallow embedding" of assertions [23]. However, it is important to note that our program logic is independent of any special features of the meta-logic. It is also independent of the use of "shallow embedding".

To specify the behavior of AIM programs, the programmer writes specifications $\mathtt{s}$ at different program points. As shown in Fig. 14, the specification Ψ of a code heap $\mathbb{C}$ is then a set of $(\mathtt{f}, \mathtt{s})$ pairs, where $\mathtt{s}$ is inserted at $\mathtt{f}$ in $\mathbb{C}$. We allow each $\mathtt{f}$ to have more than one $\mathtt{s}$, just as a function may have multiple specified interfaces. The specification $\mathtt{s}$ is a pair $(\mathtt{p}, \mathtt{g})$. The assertion $\mathtt{p}$ is a predicate over a stack $\mathbb{K}$ and a program state $\mathbb{S}$, (its meta-type in Coq is the type of functions that take $\mathbb{K}$ and $\mathbb{S}$ as arguments and return logical propositions; Prop is the universe of logical assertions in Coq), while $\mathtt{g}$

Fig. 14 Specification constructs

(*CdHpSpec*)	Ψ	::=	$\{(\mathtt{f}_1, \mathtt{s}_1), \ldots, (\mathtt{f}_n, \mathtt{s}_n)\}$
(*Spec*)	$\mathtt{s}$	::=	$(\mathtt{p}, \mathtt{g})$
(*Pred*)	$\mathtt{p}$	$\in$	$\mathit{Stack} \to \mathit{State} \to \mathsf{Prop}$
(*Guarantee*)	$\mathtt{g}$	$\in$	$\mathit{State} \to \mathit{State} \to \mathsf{Prop}$
(*MPred*)	$\mathtt{m}, \mathsf{INV0}, \mathsf{INV1}$	$\in$	$\mathit{Heap} \to \mathsf{Prop}$
(*WQSpec*)	Δ	::=	$\{w \rightsquigarrow \mathtt{m}\}^*$

Fig. 15 Definitions of separation logic assertions

$$\mathbb{H}_1 \perp \mathbb{H}_2 \quad \stackrel{\text{def}}{=} \quad dom(\mathbb{H}_1) \cap dom(\mathbb{H}_1) = \varnothing$$

$$\mathbb{H}_1 \uplus \mathbb{H}_2 \quad \stackrel{\text{def}}{=} \quad \begin{cases} \mathbb{H}_1 \cup \mathbb{H}_2 & \text{if } \mathbb{H}_1 \perp \mathbb{H}_2 \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{true} \quad \stackrel{\text{def}}{=} \quad \lambda \mathbb{H}.\ \mathsf{True} \qquad\qquad \mathsf{emp} \quad \stackrel{\text{def}}{=} \quad \lambda \mathbb{H}.\ \mathbb{H} = \varnothing$$

$$\mathtt{l} \mapsto \mathtt{w} \quad \stackrel{\text{def}}{=} \quad \lambda \mathbb{H}.\ \mathbb{H} = \{\mathtt{l} \rightsquigarrow \mathtt{w}\} \qquad\qquad \mathtt{l} \mapsto \_ \quad \stackrel{\text{def}}{=} \quad \lambda \mathbb{H}.\ \exists \mathtt{w}.\ (\mathtt{l} \mapsto \mathtt{w})\ \mathbb{H}$$

$$\mathtt{m}_1 * \mathtt{m}_2 \quad \stackrel{\text{def}}{=} \quad \lambda \mathbb{H}.\ \exists \mathbb{H}_1, \mathbb{H}_2.\ (\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H}) \wedge \mathtt{m}_1\ \mathbb{H}_1 \wedge \mathtt{m}_2\ \mathbb{H}_2$$

$$\mathtt{p} * \mathtt{m} \quad \stackrel{\text{def}}{=} \quad \lambda \mathbb{K}, \mathbb{S}.\ \exists \mathbb{H}_1, \mathbb{H}_2.\ (\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{S}.\mathbb{H}) \wedge \mathtt{p}\ \mathbb{K}\ \mathbb{S}|_{\mathbb{H}_1} \wedge \mathtt{m}\ \mathbb{H}_2$$

$$\mathtt{m} \mathrel{-\!\!*} \mathtt{m}' \quad \stackrel{\text{def}}{=} \quad \lambda \mathbb{H}.\ \forall \mathbb{H}', \mathbb{H}''.\ (\mathbb{H} \uplus \mathbb{H}' = \mathbb{H}'') \wedge \mathtt{m}\ \mathbb{H}' \rightarrow \mathtt{m}'\ \mathbb{H}''$$

$$\mathtt{m} \mathrel{-\!\!*} \mathtt{p} \quad \stackrel{\text{def}}{=} \quad \lambda \mathbb{K}, \mathbb{S}.\ \forall \mathbb{H}', \mathbb{H}''.\ (\mathbb{H}' \uplus \mathbb{S}.\mathbb{H} = \mathbb{H}'') \wedge \mathtt{m}\ \mathbb{H}' \rightarrow \mathtt{p}\ \mathbb{K}\ \mathbb{S}|_{\mathbb{H}''}$$

$$\mathsf{precise}(\mathtt{m}) \quad \stackrel{\text{def}}{=} \quad \forall \mathbb{H}, \mathbb{H}_1, \mathbb{H}_2.\ (\mathbb{H}_1 \subseteq \mathbb{H}) \wedge (\mathbb{H}_2 \subseteq \mathbb{H}) \wedge \mathtt{m}\ \mathbb{H}_1 \wedge \mathtt{m}\ \mathbb{H}_2 \rightarrow (\mathbb{H}_1 = \mathbb{H}_2)$$

is a predicate over two program states. As we can see, the $\mathtt{NextS}_{(\mathtt{C},\mathbb{K})}$ relation defined in Fig. 9 is a special form of g. Following our previous work on reasoning low-level code with stack based control abstractions [13], we use p to specify the precondition over the stack and state at the corresponding program point, and use g to specify the guaranteed behavior from the specified program point to the point where the *current* function returns.

We also use the predicate m to specify data heaps. In Fig. 15 we encode Separation Logic connectors [21, 34] in our assertion language. We use $\mathbb{H}_1 \perp \mathbb{H}_2$ to represent that data heaps $\mathbb{H}_1$ and $\mathbb{H}_2$ have disjoint domains. $\mathbb{H}_1 \uplus \mathbb{H}_2$ is the union of the disjoint heaps $\mathbb{H}_1$ and $\mathbb{H}_2$. Assertions in Separation Logic capture ownership of heaps. The assertion "$\mathtt{l} \mapsto n$" holds iff the heap has only one cell at l containing n. It can also be interpreted as the ownership of this memory cell. The separating conjunction of m and $\mathtt{m}'$ ($\mathtt{m} * \mathtt{m}'$) means the heap can be split into two *disjoint* parts, and m and $\mathtt{m}'$ hold over one of them respectively. The separating implication "$\mathtt{m} \mathrel{-\!\!*} \mathtt{m}'$" holds over $\mathbb{H}$ iff, for any disjoint heap $\mathbb{H}'$ satisfying m, $\mathbb{H} \uplus \mathbb{H}'$ satisfies $\mathtt{m}'$. We also lift the separating conjunction and the separating implication to state predicates p. A heap predicate m is precise (*i.e.,* precise(m) holds) if, for all heap, there is at most one sub-heap that satisfies m.

The specification Δ in Fig. 14 maps a block queue identifier w to a heap predicate m specifying the well-formedness of the resource that the threads in the block queue $\mathbb{B}(w)$ are waiting for.

Specifications of the shared resources The heap predicates INV0 and INV1 are part of our program specifications, which specify the well-formedness of the shared sub-heap A and C respectively, as shown in Figs. 5 and 6. The definition of INV0 depends on the functionality of the global interrupt handler; and INV1 depends on the sharing of resources among threads. To simplify the presentation, we treat them as global parameters throughout this paper.[1]

[1]They can also be treated as local parameters threading through judgments in our program logic (as Ψ and Δ in Fig. 16). To avoid the requirement of the global knowledge about shared resources and to have better modularity, frame rules [30, 34] can be supported following the same way they are supported in SCAP [11]. We do not discuss the details in this paper.

$\boxed{\Psi,\Delta \vdash \{\mathtt{s}\}\mathtt{f} : \mathbb{I}}$ (***Well-Formed Instruction Sequence***)

$$\frac{\iota \notin \{\mathsf{call}\ldots,\mathsf{beq}\ldots\} \quad \Psi,\Delta \vdash \{(\mathtt{p}',\mathtt{g}')\}\mathtt{f}+1 : \mathbb{I} \quad \mathsf{enable}(\mathtt{p},\mathtt{g}_\iota) \quad (\mathtt{p} \rhd \mathtt{g}_\iota) \Rightarrow \mathtt{p}' \quad (\mathtt{p} \circ (\mathtt{g}_\iota \circ \mathtt{g}')) \Rightarrow \mathtt{g}}{\Psi,\Delta \vdash \{(\mathtt{p},\mathtt{g})\}\mathtt{f} : \iota;\mathbb{I}} \text{ (SEQ)}$$
$$\text{where} \quad \mathtt{g}_\iota \stackrel{\text{def}}{=} [\![\iota]\!]_\Delta$$

$$\frac{(\mathtt{f}',(\mathtt{p}',\mathtt{g}')) \in \Psi \qquad \Psi,\Delta \vdash \{(\mathtt{p}'',\mathtt{g}'')\}\mathtt{f}+1 : \mathbb{I} \quad \forall \mathbb{K},\mathbb{S},\mathtt{pc}.\ \mathtt{p}\ \mathbb{K}\ \mathbb{S} \to \mathtt{p}'\ (\mathtt{pc}::\mathbb{K})\ \mathbb{S} \qquad (\mathtt{p} \rhd \mathtt{g}') \Rightarrow \mathtt{p}'' \qquad (\mathtt{p} \circ (\mathtt{g}' \circ \mathtt{g}'')) \Rightarrow \mathtt{g}}{\Psi,\Delta \vdash \{(\mathtt{p},\mathtt{g})\}\mathtt{f} : \mathsf{call}\ \mathtt{f}';\mathbb{I}} \text{ (CALL)}$$

$$\frac{\mathtt{p} \Rightarrow \mathsf{enable}_{\mathsf{iret}} \qquad (\mathtt{p} \circ \mathsf{gid}) \Rightarrow \mathtt{g}}{\Psi,\Delta \vdash \{(\mathtt{p},\mathtt{g})\}\mathtt{f} : \mathsf{iret}} \text{ (IRET)} \qquad \frac{\mathtt{p} \Rightarrow \mathsf{enable}_{\mathsf{ret}} \qquad (\mathtt{p} \circ \mathsf{gid}) \Rightarrow \mathtt{g}}{\Psi,\Delta \vdash \{(\mathtt{p},\mathtt{g})\}\mathtt{f} : \mathsf{ret}} \text{ (RET)}$$

$$\text{where } \mathsf{enable}_{\mathsf{iret}} \stackrel{\text{def}}{=} \lambda\mathbb{K},\mathbb{S}.\ \exists \mathtt{f},\mathbb{R},\mathbb{K}'.\ \mathbb{K} = (\mathtt{f},\mathbb{R})::\mathbb{K}' \wedge \mathbb{S}.\mathtt{is} = 1 \qquad \text{where } \mathsf{enable}_{\mathsf{ret}} \stackrel{\text{def}}{=} \lambda\mathbb{K},\mathbb{S}.\ \exists \mathtt{f},\mathbb{K}'.\ \mathbb{K} = \mathtt{f}::\mathbb{K}'$$

$$\frac{\begin{array}{l}(\mathtt{f}',(\mathtt{p}',\mathtt{g}')) \in \Psi \qquad \Psi,\Delta \vdash \{(\mathtt{p}'',\mathtt{g}'')\}\mathtt{f}+1 : \mathbb{I} \\ (\mathtt{p} \rhd \mathsf{gid}_{r_s=r_t}) \Rightarrow \mathtt{p}' \quad (\mathtt{p} \circ (\mathsf{gid}_{r_s=r_t} \circ \mathtt{g}')) \Rightarrow \mathtt{g} \\ (\mathtt{p} \rhd \mathsf{gid}_{r_s\neq r_t}) \Rightarrow \mathtt{p}'' \quad (\mathtt{p} \circ (\mathsf{gid}_{r_s\neq r_t} \circ \mathtt{g}'')) \Rightarrow \mathtt{g}\end{array}}{\Psi,\Delta \vdash \{(\mathtt{p},\mathtt{g})\}\mathtt{f} : \mathsf{beq}\ r_s,r_t,\mathtt{f}';\mathbb{I}} \text{ (BEQ)} \qquad \frac{(\mathtt{f}',(\mathtt{p}',\mathtt{g}')) \in \Psi \quad \mathtt{p} \Rightarrow \mathtt{p}' \quad (\mathtt{p} \circ \mathtt{g}') \Rightarrow \mathtt{g}}{\Psi,\Delta \vdash \{(\mathtt{p},\mathtt{g})\}\mathtt{f} : \mathsf{j}\ \mathtt{f}'} \text{ (J)}$$

$\boxed{\Psi,\Delta \vdash \mathbb{C} : \Psi'}$ (***Well-Formed Code Heap***)

$$\frac{\text{for all } (\mathtt{f},\mathtt{s}) \in \Psi' : \quad \Psi,\Delta \vdash \{\mathtt{s}\}\mathtt{f} : \mathbb{C}[\mathtt{f}]}{\Psi,\Delta \vdash \mathbb{C} : \Psi'} \text{ (CDHP)}$$

$\boxed{\Psi,\Delta \vdash \mathbb{W}}$ (***Well-Formed World***)

$$\frac{\begin{array}{l}\mathbb{T} \setminus \mathtt{tid} = \{\mathtt{tid}_1 \rightsquigarrow (\mathbb{R}_1,\mathbb{K}_1,\mathtt{is}_1,\mathtt{pc}_1),\ldots,\mathtt{tid}^n \rightsquigarrow (\mathbb{R}n,\mathbb{K}n,\mathtt{is}^n,\mathtt{pc}_n)\} \\ \mathbb{S}.\mathbb{H} = \mathbb{H}_0 \uplus \ldots \uplus \mathbb{H}n \qquad \mathbb{S}_i = (\mathbb{H}_i,\mathbb{R}_i,0,\mathtt{is}_i) \quad (0 < i \leq n) \\ \Psi,\Delta \vdash \mathbb{C} : \Psi' \qquad \Psi \subseteq \Psi' \qquad dom(\Delta) = dom(\mathbb{B}) \\ \mathsf{WFCth}(\mathbb{S}_0,\mathbb{K},\mathtt{pc},\Psi') \qquad \text{where } \mathbb{S}_0 = \mathbb{S}|_{\mathbb{H}_0} \\ \text{for all } 0 < k \leq n \text{ such that } \mathtt{tid}k \in \mathtt{readyQ}(\mathbb{T},\mathbb{B}) : \qquad \mathsf{WFRdy}(\mathbb{S}_k,\mathbb{K}_k,\mathtt{pc}_k,\Psi') \\ \text{for all } w \text{ and } 0 < j \leq n \text{ such that } \mathtt{tid}j \in \mathbb{B}(w) : \qquad \mathsf{WFWait}(\mathbb{S}_j,\mathbb{K}_j,\mathtt{pc}_j,\Psi',\Delta(w))\end{array}}{\Psi,\Delta \vdash (\mathbb{C},\mathbb{S},\mathbb{K},\mathtt{pc},\mathtt{tid},\mathbb{T},\mathbb{B})} \text{ (WLD)}$$

Fig. 16 Inference rules

Specification of the interrupt handler We need to give a specification to the interrupt handler to certify the handler code and ensure the non-interference. We let $(\mathtt{h\_entry}, (\mathtt{p_i}, \mathtt{g_i})) \in \Psi$, where $\mathtt{p_i}$ and $\mathtt{g_i}$ are defined as follows:

$$\mathtt{p_i} \stackrel{\text{def}}{=} \lambda\mathbb{K},\mathbb{S}.\ ((\mathsf{INV0} * \mathit{true})\ \mathbb{S}.\mathbb{H}) \wedge (\mathbb{S}.\mathtt{is} = 1) \wedge (\mathbb{S}.\mathtt{ie} = 0) \wedge \exists \mathtt{f},\mathbb{R},\mathbb{K}'.\ \mathbb{K} = (\mathtt{f},\mathbb{R})::\mathbb{K}' \tag{3}$$

$$\mathtt{g_i} \stackrel{\text{def}}{=} \lambda\mathbb{S},\mathbb{S}'.\ \left\{\begin{array}{l}\mathsf{INV0}\\ \mathsf{INV0}\end{array}\right\}\ \mathbb{S}.\mathbb{H}\ \mathbb{S}'.\mathbb{H} \wedge (\mathbb{S}'.\mathtt{ie} = \mathbb{S}.\mathtt{ie}) \wedge (\mathbb{S}'.\mathtt{is} = \mathbb{S}.\mathtt{is}) \tag{4}$$

The precondition $\mathtt{p}_i$ specifies the stack and state at the entry `h_entry`. It requires that the local heap used by the handler (block A in Fig. 5) satisfy INV0. It leaves block C and the local heap of the non-handler code unspecified because the handler does not access them. The precondition also specifies the expectations over `is`, `ie` and the stack, which will be guaranteed by the operational semantics (see the IRQ rule in Section 3.1). The guarantee $\mathtt{g}_i$ specifies the behavior of the handler. The arguments $\mathbb{S}$ and $\mathbb{S}'$ correspond to program states at the beginning and the end of the interrupt handler, respectively. It says the `ie` and `is` bits in $\mathbb{S}'$ have the same value as in $\mathbb{S}$, and the handler's local heap satisfies INV0 in $\mathbb{S}$ and $\mathbb{S}'$, while the rest of the heap remains unchanged. The predicate $\left\{\begin{array}{l}\mathtt{m}_1\\ \mathtt{m}_2\end{array}\right\}$ is defined below.

$$\left\{\begin{array}{l}\mathtt{m}_1\\ \mathtt{m}_2\end{array}\right\} \stackrel{\text{def}}{=} \lambda\mathbb{H}_1, \mathbb{H}_2.\ \exists\mathbb{H}'_1, \mathbb{H}'_2, \mathbb{H}.\ (\mathtt{m}_1\ \mathbb{H}'_1) \wedge (\mathtt{m}_2\ \mathbb{H}'_2) \wedge (\mathbb{H}'_1 \uplus \mathbb{H} = \mathbb{H}_1) \wedge (\mathbb{H}'_2 \uplus \mathbb{H} = \mathbb{H}_2) \tag{5}$$

It means part of the heap in $\mathbb{H}_1$ satisfies $\mathtt{m}_1$ and is transformed into a sub-heap satisfying $\mathtt{m}_2$ in $\mathbb{H}_2$. The rest part of $\mathbb{H}_1$ is preserved in $\mathbb{H}_2$. It has the following nice monotonicity with respect to heap extension:

Proposition 1 *For all* $\mathbb{H}_1$, $\mathbb{H}_2$ *and* $\mathbb{H}'$, *if* $\left\{\begin{array}{l}\mathtt{m}_1\\ \mathtt{m}_2\end{array}\right\}\ \mathbb{H}_1\ \mathbb{H}_2$, $\mathbb{H}_1 \perp \mathbb{H}'$, *and* $\mathbb{H}_2 \perp \mathbb{H}'$, *then* $\left\{\begin{array}{l}\mathtt{m}_1\\ \mathtt{m}_2\end{array}\right\}\ (\mathbb{H}_1 \uplus \mathbb{H}')\ (\mathbb{H}_2 \uplus \mathbb{H}')$.

Specifying heap transitions and ownership transfers The guarantee g in general specifies state transitions. Predicates of the form $\left\{\begin{array}{l}\mathtt{m}_1\\ \mathtt{m}_2\end{array}\right\}$ are very useful to specify transitions of data heaps. Below we show some commonly used patterns of transitions.

$$\mathsf{hid} \stackrel{\text{def}}{=} \left\{\begin{array}{l}\mathsf{emp}\\ \mathsf{emp}\end{array}\right\} \qquad \mathsf{Recv}(\mathtt{m}) \stackrel{\text{def}}{=} \left\{\begin{array}{l}\mathsf{emp}\\ \mathtt{m}\end{array}\right\} \qquad \mathsf{Send}(\mathtt{m}) \stackrel{\text{def}}{=} \left\{\begin{array}{l}\mathtt{m}\\ \mathsf{emp}\end{array}\right\}$$

$$\mathsf{Presv}(\mathtt{m}) \stackrel{\text{def}}{=} \left\{\begin{array}{l}\mathtt{m}\\ \mathtt{m}\end{array}\right\}$$

The transition hid represents an identity transition of heaps. Transitions Recv(m) and Send(m) represent transitions of the ownership of the sub-heap specified by m between a thread and its environment. Recv(m) means the heap at the beginning of the transition is preserved at the end. In addition, the thread gets the extra ownership of the sub-heap m. Send(m) means a sub-heap of the initial heap satisfies m and the ownership of it is lost at the end of the transition. The rest part of the initial heap is preserved at the end. The transition Presv(m) means there are sub-heaps satisfying m at the beginning and the end, and the rest part of the initial heap is preserved at the end.

We can also define separating conjunction for heap transitions, which is similar to the separating conjunction for heap predicates.

$$\left\{\begin{array}{l}\mathtt{m}_1\\ \mathtt{m}_2\end{array}\right\} \circledast \left\{\begin{array}{l}\mathtt{m}'_1\\ \mathtt{m}'_2\end{array}\right\} \stackrel{\text{def}}{=} \lambda\mathbb{H}_1, \mathbb{H}_2.\ \exists\mathbb{H}'_1, \mathbb{H}''_1, \mathbb{H}'_2, \mathbb{H}''_2.\ (\mathbb{H}_1 = \mathbb{H}'_1 \uplus \mathbb{H}''_1) \wedge (\mathbb{H}_2 = \mathbb{H}'_2 \uplus \mathbb{H}''_2)$$
$$\wedge \left\{\begin{array}{l}\mathtt{m}_1\\ \mathtt{m}_2\end{array}\right\}\ \mathbb{H}'_1\ \mathbb{H}'_2 \wedge \left\{\begin{array}{l}\mathtt{m}'_1\\ \mathtt{m}'_2\end{array}\right\}\ \mathbb{H}''_1\ \mathbb{H}''_2 \tag{6}$$

It satisfies the following properties:

Proposition 2 *For all* $\mathbb{H}_1$ *and* $\mathbb{H}_2$, *we have:*

- $\left(\left\{\begin{array}{l}\mathtt{m}_1\\ \mathtt{m}_2\end{array}\right\} \circledast \left\{\begin{array}{l}\mathtt{m}'_1\\ \mathtt{m}'_2\end{array}\right\}\right)\ \mathbb{H}_1\ \mathbb{H}_2 \iff \left\{\begin{array}{l}\mathtt{m}_1 * \mathtt{m}'_1\\ \mathtt{m}_2 * \mathtt{m}'_2\end{array}\right\}\ \mathbb{H}_1\ \mathbb{H}_2$
- $\left\{\begin{array}{l}\mathtt{m}_1\\ \mathtt{m}_2\end{array}\right\} \iff \mathsf{Send}(\mathtt{m}_1) \circledast \mathsf{Recv}(\mathtt{m}_2)$

The proposition above also shows that the transition $\mathsf{Presv}(\mathtt{m})$ does not imply hid because the sub-heaps satisfying $\mathtt{m}$ at the beginning and the end do not have to be the same.

4.2 Inference Rules

Inference rules of the program logic are shown in Fig. 16. The judgment $\Psi, \Delta \vdash \{\mathtt{s}\}\,\mathtt{f} : \mathbb{I}$ defines the well-formedness of the instruction sequence $\mathbb{I}$ starting at the code label $\mathtt{f}$, given the imported interfaces in Ψ, the specification Δ of block queues, and the specification $(\mathtt{p}, \mathtt{g})$. Informally, it says if the state satisfies $\mathtt{p}$ and the code blocks that might be reached from $\mathbb{I}$ through jumps, conditional branch instructions or function calls are also well-formed with respect to their specifications in Ψ, then the execution starting from $\mathtt{f}$ would not abort, and the transition from $\mathtt{f}$ to the end of the current function (not necessarily the end of $\mathbb{I}$) satisfies $\mathtt{g}$. The specification Δ of block queues is used in the rules for block and unblock instructions shown below. It specifies the resources that the threads in the corresponding block queues are waiting for.

The SEQ rule is a *schema* for instruction sequences starting with an instruction ι (excluding the branch and function call instructions). We need to find an intermediate specification $(\mathtt{p}', \mathtt{g}')$, with respect to which the remaining instruction sequence is well-formed. It is also used as a post-condition for the first instruction. We use $\mathtt{g}_\iota$ to represent the state transition $[\![\,\iota\,]\!]_\Delta$ made by the instruction ι, which is defined in Fig. 18 and is explained below. The premise $\mathsf{enable}(\mathtt{p}, \mathtt{g}_\iota)$ is defined in Fig. 17. It means that the state transition $\mathtt{g}_\iota$ would not abort as long as the starting stack and

$$\mathsf{enable}(\mathtt{p},\mathtt{g}) \stackrel{\text{def}}{=} \forall\mathbb{K}, \mathbb{S}.\ \mathtt{p}\ \mathbb{K}\ \mathbb{S} \to \exists\mathbb{S}', \mathtt{g}\ \mathbb{S}\ \mathbb{S}' \qquad \mathtt{p} \rhd \mathtt{g} \stackrel{\text{def}}{=} \lambda\mathbb{K}, \mathbb{S}.\ \exists\mathbb{S}_0, \mathtt{p}\ \mathbb{K}\ \mathbb{S}_0 \wedge \mathtt{g}\ \mathbb{S}_0\ \mathbb{S}$$
$$\mathtt{g} \circ \mathtt{g}' \stackrel{\text{def}}{=} \lambda\mathbb{S}, \mathbb{S}''.\ \exists\mathbb{S}'.\ \mathtt{g}\ \mathbb{S}\ \mathbb{S}' \wedge \mathtt{g}'\ \mathbb{S}'\ \mathbb{S}'' \qquad \mathtt{p} \Rightarrow \mathtt{p}' \stackrel{\text{def}}{=} \forall\mathbb{K}, \mathbb{S}.\ \mathtt{p}\ \mathbb{K}\ \mathbb{S} \to \mathtt{p}'\ \mathbb{K}\ \mathbb{S}$$
$$\mathtt{p} \circ \mathtt{g} \stackrel{\text{def}}{=} \lambda\mathbb{S}, \mathbb{S}'.\ \exists\mathbb{K}.\ \mathtt{p}\ \mathbb{K}\ \mathbb{S} \wedge \mathtt{g}\ \mathbb{S}\ \mathbb{S}' \qquad \mathtt{g} \Rightarrow \mathtt{g}' \stackrel{\text{def}}{=} \forall\mathbb{S}, \mathbb{S}'.\ \mathtt{g}\ \mathbb{S}\ \mathbb{S}' \to \mathtt{g}'\ \mathbb{S}\ \mathbb{S}'$$

Fig. 17 Connectors for $\mathtt{p}$ and $\mathtt{g}$

$$P\,?\,\mathtt{m}:\mathtt{m}' \stackrel{\text{def}}{=} \lambda\mathbb{H}.\,(P \wedge \mathtt{m}\,\mathbb{H}) \vee (\neg P \wedge \mathtt{m}'\,\mathbb{H})$$

$$[\![\,\mathsf{cli}\,]\!]_\Delta \stackrel{\text{def}}{=} \lambda(\mathbb{H},\mathbb{R},\mathtt{ie},\mathtt{is}),(\mathbb{H}',\mathbb{R}',\mathtt{ie}',\mathtt{is}').\ (\mathtt{is}=\mathtt{is}') \wedge (\mathbb{R}=\mathbb{R}') \wedge (\mathtt{ie}'=0) \wedge \mathsf{Recv}((\mathtt{ie}=1 \wedge \mathtt{is}=0)\,?\,(\mathsf{INV0}*\mathsf{INV1}):\mathsf{emp})\,\mathbb{H}\,\mathbb{H}'$$

$$[\![\,\mathsf{sti}\,]\!]_\Delta \stackrel{\text{def}}{=} \lambda(\mathbb{H},\mathbb{R},\mathtt{ie},\mathtt{is}),(\mathbb{H}',\mathbb{R}',\mathtt{ie}',\mathtt{is}').\ (\mathtt{is}=\mathtt{is}') \wedge (\mathbb{R}=\mathbb{R}') \wedge (\mathtt{ie}'=1) \wedge \mathsf{Send}((\mathtt{ie}=0 \wedge \mathtt{is}=0)\,?\,(\mathsf{INV0}*\mathsf{INV1}):\mathsf{emp})\,\mathbb{H}\,\mathbb{H}'$$

$$[\![\,\mathsf{switch}\,]\!]_\Delta \stackrel{\text{def}}{=} \lambda(\mathbb{H},\mathbb{R},\mathtt{ie},\mathtt{is}),(\mathbb{H}',\mathbb{R}',\mathtt{ie}',\mathtt{is}').\ (\mathtt{ie}=0) \wedge (\mathtt{ie}=\mathtt{ie}') \wedge (\mathbb{R}=\mathbb{R}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{Presv}(\mathsf{INV0}*(\mathtt{is}=0\,?\,\mathsf{INV1}:\mathsf{emp}))\,\mathbb{H}\,\mathbb{H}'$$

$$[\![\,\mathsf{block}\ r_s\,]\!]_\Delta \stackrel{\text{def}}{=} \lambda(\mathbb{H},\mathbb{R},\mathtt{ie},\mathtt{is}),(\mathbb{H}',\mathbb{R}',\mathtt{ie}',\mathtt{is}').\ (\mathtt{ie}=0) \wedge (\mathtt{ie}=\mathtt{ie}') \wedge (\mathbb{R}=\mathbb{R}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \exists\mathtt{m}.\,\Delta(\mathbb{R}(r_s))=\mathtt{m} \wedge (\mathsf{Presv}(\mathsf{INV0}*(\mathtt{is}=0\,?\,\mathsf{INV1}:\mathsf{emp})) \circledast \mathsf{Recv}(\mathtt{m}))\,\mathbb{H}\,\mathbb{H}'$$

$$[\![\,\mathsf{unblock}\ r_s, r_d\,]\!]_\Delta \stackrel{\text{def}}{=} \lambda(\mathbb{H},\mathbb{R},\mathtt{ie},\mathtt{is}),(\mathbb{H}',\mathbb{R}',\mathtt{ie}',\mathtt{is}').\ (\mathtt{ie}=0) \wedge (\mathtt{ie}=\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge (\forall r \neq r_d.\,\mathbb{R}(r)=\mathbb{R}'(r)) \wedge \exists\mathtt{m}.\,\Delta(\mathbb{R}(r_s))=\mathtt{m} \wedge (\mathtt{m}*\mathsf{true})\,\mathbb{H} \wedge \mathsf{Send}((\mathbb{R}'(r_d)=0)\,?\,\mathsf{emp}:\mathtt{m})\,\mathbb{H}\,\mathbb{H}'$$

$$[\![\,\iota\,]\!]_\Delta \stackrel{\text{def}}{=} \mathtt{NextS}_{(\iota,\_)} \qquad \text{(for all other } \iota\text{)}$$

Fig. 18 Thread-local state transitions made by ι

state satisfy p. The predicate $\mathtt{p} \rhd \mathtt{g}_\iota$, shown in Fig. 17, specifies the stack and state resulting from the state transition $\mathtt{g}_\iota$, knowing the initial state satisfies p. It is the strongest post condition after $\mathtt{g}_\iota$. The composition of two subsequent transitions g and $\mathtt{g}'$ is represented as $\mathtt{g} \circ \mathtt{g}'$, and $\mathtt{p} \circ \mathtt{g}$ refines g with the extra knowledge that the initial state satisfies p. We also lift the implication relation between p's and g's. The last premise in the SEQ rule requires the composition of $\mathtt{g}_\iota$ and $\mathtt{g}'$ fulfills g, knowing the current state satisfies p.

If ι is an arithmetic instruction, move instruction or memory operation, we define $[\![\,\iota\,]\!]_\Delta$ in Fig. 18 as $\mathtt{NextS}_{(\iota,\_)}$. Since NextS does not depend on the stack for these instructions (recall its definition in Fig. 9), we use "_" to represent arbitrary stacks. Also note that the NextS relations for ld or st require the target address to be in the domain of heap, therefore the premise $\mathsf{enable}(\mathtt{p}, \mathtt{g}_\iota)$ requires that p contain the ownership of the target memory cell.

Interrupts and thread primitive instructions One of the major technical contributions of this paper is our formulation of $[\![\,\iota\,]\!]_\Delta$ for cli, sti, switch, block and unblock, which, as shown in Fig. 18, gives them an axiomatic ownership transfer semantics.

The transition $[\![\,\mathsf{cli}\,]\!]_\Delta$ says that, if cli is executed in the non-handler ($\mathtt{is}=0$) and the interrupt is enabled ($\mathtt{ie}=1$), the current thread gets ownership of the well-formed sub-heap A and C satisfying $\mathsf{INV0}*\mathsf{INV1}$, as shown in Fig. 5; otherwise there is no ownership transfer because the interrupt has already been disabled before cli. The transition $[\![\,\mathsf{sti}\,]\!]_\Delta$ is defined similarly. Note that when ι in the SEQ rule is instantiated with sti, the premise $\mathsf{enable}(\mathtt{p}, \mathtt{g}_\iota)$ in the rule requires that the precondition p must contain the ownership of $(\mathtt{ie}=0 \wedge \mathtt{is}=0)\,?\,(\mathsf{INV1}*\mathsf{INV0}):\mathsf{emp}$.

$[\![\,\mathsf{switch}\,]\!]_\Delta$ requires that the sub-heap A and C (in Fig. 5) be well-formed before and after switch. However, if we execute switch in the interrupt handler ($\mathtt{is}=1$), we know INV1 always holds and leave it implicit. Also the premise $\mathsf{enable}(\mathtt{p}, \mathtt{g}_\iota)$ in the SEQ rule requires that p imply $\mathtt{ie}=0$ and $\mathsf{INV0}*(\mathtt{is}=0\,?\,\mathsf{INV1}:\mathsf{emp})$ holds over some sub-heap.

The transitions $[\![\ \mathsf{block}\ r_s\ ]\!]_\Delta$ and $[\![\ \mathsf{unblock}\ r_s, r_d\ ]\!]_\Delta$ refer to the specification Δ. $[\![\ \mathsf{block}\ r_s\ ]\!]_\Delta$ requires $\mathtt{ie} = 0$ and that r_s contain an identifier of a block queue with some specification $\mathtt{m}$ in Δ. It is similar to $[\![\ \mathsf{switch}\ ]\!]_\Delta$, except that the thread gets the ownership of $\mathtt{m}$ after it is released (see Fig. 6). In $[\![\ \mathsf{unblock}\ r_s, r_d\ ]\!]_\Delta$, we require the initial heap must contain a sub-heap satisfying $\mathtt{m}$, because unblock may transfer it to a blocked thread. However, since unblock does not immediately switch controls, we do not need the sub-heap A and C to be well-formed. If r_d contains non-zero value at the end of unblock, some thread has been released from the block queue. The current thread transfers $\mathtt{m}$ to the released thread and has no access to it any more. Otherwise, no thread is released and there is no ownership transfer.

Other instructions The CALL rule in Fig. 16 requires that the callee function $\mathtt{f}'$ be specified in Ψ with some specification $(\mathtt{p}', \mathtt{g}')$. We view the state transition $\mathtt{g}'$ made by the callee as the transition of the call instruction, like $[\![\ \iota\ ]\!]_\Delta$ in the SEQ rule. The rule also requires that the precondition $\mathtt{p}$ imply the precondition $\mathtt{p}'$ of the callee, which corresponds to the enable premise in the SEQ rule. The specification $(\mathtt{p}'', \mathtt{g}'')$, as in the SEQ rule, serves as both the post-condition of the function call and the precondition of the remaining instruction sequence. IRET and RET rules require that the interrupt handler or the function have finished its guaranteed transition at this point. So an identity transition gid should satisfy the remaining transition g. The predicates $\mathsf{enable}_{\mathsf{iret}}$ and $\mathsf{enable}_{\mathsf{ret}}$ specify the requirements over stacks. In the BEQ rule, we use $\mathsf{gid}_{r_s=r_t}$ and $\mathsf{gid}_{r_s\neq r_t}$ to represent identity transitions with extra knowledge about r_s and r_t:

$$\begin{aligned}
\mathsf{gid} &\stackrel{\text{def}}{=} \lambda\mathbb{S}, \mathbb{S}'.\ \mathbb{S} = \mathbb{S}' \\
\mathsf{gid}_{r_s=r_t} &\stackrel{\text{def}}{=} \lambda\mathbb{S}, \mathbb{S}'.\ (\mathsf{gid}\ \mathbb{S}\ \mathbb{S}') \wedge (\mathbb{S}.\mathbb{R}(r_s) = \mathbb{S}.\mathbb{R}(r_t)) \\
\mathsf{gid}_{r_s\neq r_t} &\stackrel{\text{def}}{=} \lambda\mathbb{S}, \mathbb{S}'.\ (\mathsf{gid}\ \mathbb{S}\ \mathbb{S}') \wedge (\mathbb{S}.\mathbb{R}(r_s) \neq \mathbb{S}.\mathbb{R}(r_t))
\end{aligned}$$

We do not have an enable premise because executing beq never aborts. The J rule can be viewed as a specialization of the BEQ rule where $r_s = r_t$ is always true.

Well-formed code heaps The CDHP rule says the code heap is well-formed if and only if each instruction sequence specified in Ψ' is well-formed. Ψ and Ψ' can be viewed as the imported and exported interfaces of $\mathbb{C}$ respectively.

Program invariants The WLD rule defines the well-formedness of the whole program configuration $\mathbb{W}$. It also formulates the program invariant enforced by our program logic. If there are n threads in $\mathbb{T}$ in addition to the current thread, the heap can be split into $n + 1$ blocks. Each block $\mathbb{H}_k$ $(k > 0)$ is for a ready or blocked thread in queues. The block $\mathbb{H}_0$ is assigned to the current thread, which includes both its private heap and the shared part (blocks A and C, as shown in Fig. 5). The code heap $\mathbb{C}$ needs to be well-formed, as defined by the CDHP rule. We require the imported interface Ψ is a subset of the exported interface Ψ', therefore $\mathbb{C}$ is self-contained and each imported specification has been certified. The domain of Δ should be the same with the domain of $\mathbb{B}$, *i.e.,* Δ specifies and only specifies block queues in $\mathbb{B}$. The WLD rule also requires that the local heaps and execution contexts of the current thread, ready threads and blocked threads are all well-formed (see Fig. 19).

WFCth defines the well-formedness of the current thread. It requires that the $\mathtt{pc}$ have a specification $(\mathtt{p}, \mathtt{g})$ in Ψ. By the premise $\Psi, \Delta \vdash \mathbb{C} : \Psi'$ we know $\mathbb{C}[\mathtt{pc}]$

$$\mathsf{Inv}(\mathtt{ie},\mathtt{is}) \stackrel{\text{def}}{=} \begin{cases} \mathsf{INV1} & \mathtt{is}=1 \\ \mathsf{emp} & \mathtt{is}=0 \text{ and } \mathtt{ie}=0 \\ \mathsf{INV}_s & \mathtt{is}=0 \text{ and } \mathtt{ie}=1 \end{cases}$$

where $\mathsf{INV}_s \stackrel{\text{def}}{=} \mathsf{INV0} * \mathsf{INV1}$

$$\mathtt{p} * \mathsf{Inv} \stackrel{\text{def}}{=} \lambda\mathbb{K},\mathbb{S}.\ (\mathtt{p} * \mathsf{Inv}(\mathbb{S}.\mathtt{ie},\mathbb{S}.\mathtt{is}))\ \mathbb{K}\ \mathbb{S}$$

$$\lfloor \mathtt{g} \rfloor \stackrel{\text{def}}{=} \lambda(\mathbb{H},\mathbb{R},\mathtt{ie},\mathtt{is}),(\mathbb{H}',\mathbb{R}',\mathtt{ie}',\mathtt{is}').\ \exists \mathbb{H}_1,\mathbb{H}_2,\mathbb{H}'_1,\mathbb{H}'_2.\ (\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H}) \wedge (\mathbb{H}'_1 \uplus \mathbb{H}'_2 = \mathbb{H}') \wedge \mathtt{g}\ (\mathbb{H}_1,\mathbb{R},\mathtt{ie},\mathtt{is})\ (\mathbb{H}'_1,\mathbb{R}',\mathtt{ie}',\mathtt{is}') \wedge \mathsf{Inv}(\mathtt{ie},\mathtt{is})\ \mathbb{H}_2 \wedge \mathsf{Inv}(\mathtt{ie}',\mathtt{is}')\ \mathbb{H}'_2$$

$$\mathsf{WFST}(\mathtt{g},\mathbb{S},\mathsf{nil},\Psi) \stackrel{\text{def}}{=} \neg\exists\mathbb{S}'.\ \mathtt{g}\ \mathbb{S}\ \mathbb{S}'$$

$$\mathsf{WFST}(\mathtt{g},\mathbb{S},\mathtt{f}::\mathbb{K},\Psi) \stackrel{\text{def}}{=} \exists \mathtt{p_f},\mathtt{g_f}.\ (\mathtt{f},\ (\mathtt{p_f},\mathtt{g_f})) \in \Psi \wedge \forall\mathbb{S}'.\ \mathtt{g}\ \mathbb{S}\ \mathbb{S}' \rightarrow (\mathtt{p_f} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}' \wedge \mathsf{WFST}(\lfloor \mathtt{g_f} \rfloor,\mathbb{S}',\mathbb{K},\Psi)$$

$$\mathsf{WFST}(\mathtt{g},\mathbb{S},(\mathtt{f},\mathbb{R})::\mathbb{K},\Psi) \stackrel{\text{def}}{=} \exists \mathtt{p_f},\mathtt{g_f}.\ (\mathtt{f},\ (\mathtt{p_f},\mathtt{g_f})) \in \Psi \wedge \forall\mathbb{S}'.\ \mathtt{g}\ \mathbb{S}\ \mathbb{S}' \rightarrow (\mathtt{p_f} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}'' \wedge \mathsf{WFST}(\lfloor \mathtt{g_f} \rfloor,\mathbb{S}'',\mathbb{K},\Psi) \text{ where } \mathbb{S}'' = (\mathbb{S}'.\mathbb{H},\mathbb{R},1,0)$$

$$\mathsf{WFCth}(\mathbb{S},\mathbb{K},\mathtt{pc},\Psi) \stackrel{\text{def}}{=} \exists \mathtt{p},\mathtt{g}.\ (\mathtt{pc},(\mathtt{p},\mathtt{g})) \in \Psi \wedge (\mathtt{p} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S} \wedge \mathsf{WFST}(\lfloor \mathtt{g} \rfloor,\mathbb{S},\mathbb{K},\Psi)$$

$$\mathsf{WFCth}(\mathtt{pc},\Psi) \stackrel{\text{def}}{=} \lambda\mathbb{K},\mathbb{S}.\ \mathsf{WFCth}(\mathbb{S},\mathbb{K},\mathtt{pc},\Psi)$$

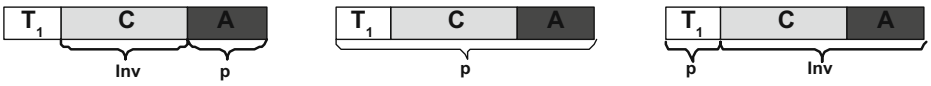
$$\mathsf{WFRdy}(\mathbb{S},\mathbb{K},\mathtt{pc},\Psi) \stackrel{\text{def}}{=} ((\mathsf{INV0} * \mathsf{INV1}) \mathrel{-\!\!*} \mathsf{WFCth}(\mathtt{pc},\Psi))\ \mathbb{K}\ \mathbb{S}$$

$$\mathsf{WFRdy}(\mathtt{pc},\Psi) \stackrel{\text{def}}{=} \lambda\mathbb{K},\mathbb{S}.\ \mathsf{WFRdy}(\mathbb{S},\mathbb{K},\mathtt{pc},\Psi)$$

$$\mathsf{WFWait}(\mathbb{S},\mathbb{K},\mathtt{pc},\Psi,\mathtt{m}) \stackrel{\text{def}}{=} (\mathtt{m} \mathrel{-\!\!*} \mathsf{WFRdy}(\mathtt{pc},\Psi))\ \mathbb{K}\ \mathbb{S}$$

Fig. 19 Well-formed current, ready and waiting threads

is well-formed with respect to (p, g). It also requires that the stack and the local state (containing the sub-heap $\mathbb{H}_0$) of the current thread satisfy $\mathtt{p} * \mathsf{Inv}$, which is defined in Fig. 19. Here p specifies the state accessible by the current thread, while Inv(ie, is) specifies the inaccessible part of the *shared* heap. As shown in Fig. 20, if the current program point is in the interrupt handler ($\mathtt{is} = 1$), p leaves the memory block C unspecified, therefore Inv(ie, is) is defined as INV1 and specifies the well-formedness of C. Otherwise ($\mathtt{is} = 0$), if $\mathtt{ie} = 0$, blocks A and C become the current thread's private memory and the inaccessible part is empty. If $\mathtt{ie} = 1$, A and C are inaccessible; Inv(ie, is) specifies their well-formedness in this case. Similarly, we use $\lfloor \mathtt{g} \rfloor$ to require that the inaccessible part of the shared heap unspecified in g satisfy Inv(ie, is) at the beginning and the end of g. $\lfloor \mathtt{g} \rfloor$ is used in WFST, which specifies the well-formedness of the stack $\mathbb{K}$.



(a) $\mathtt{is} = 1$ (b) $\mathtt{is} = 0 \wedge \mathtt{ie} = 0$ (c) $\mathtt{is} = 0 \wedge \mathtt{ie} = 1$

Fig. 20 The Meaning of p and Inv in WFCth. The blocks *A* and *C* have the same meanings as in Fig. 5. The block T_1 is the private heap of the current thread

The predicate WFST ensures it is always safe to return to the code labels (*i.e.,* return addresses of functions or interrupt handlers) stored on the top of $\mathbb{K}$. If $\mathbb{K}$ is empty, we are executing the topmost level function and cannot return. This is enforced by requiring the remaining guarantee $\mathtt{g}$ be unsatisfiable. If $\mathbb{K}$ is not empty, the return address $\mathtt{f}$ on the top of the stack needs to be specified in Ψ with a specification $(\mathtt{p_f}, \mathtt{g_f})$. Again, by the premise $\Psi, \Delta \vdash \mathbb{C} : \Psi'$ in the WLD rule we know the return continuation $\mathbb{C}[\mathtt{f}]$ is well-formed. When the remaining guarantee $\mathtt{g}$ is fulfilled and thus the current function (or the interrupt handler) can return, the remaining stack and the state after ret (or iret if in the interrupt handler) need to satisfy $\mathtt{p_f} * \mathsf{Inv}$, therefore it is safe to execute $\mathbb{C}[\mathtt{f}]$. Also, the remaining stack needs to be well-formed with respect to $\lfloor \mathtt{g_f} \rfloor$ in the new state. The definition of WFST follows our previous work on SCAP [13] for stack-based control abstractions.

The definition of well-formed ready threads WFRdy is straightforward. We first overload the name WFCth and define WFCth(pc, Ψ) as a predicate over the stack and state. WFRdy says if the *ready* thread gets the extra ownership of shared memory A and C, it becomes a well-formed *current* thread (see Fig. 5). Recall that $\mathtt{m} \mathrel{-\!\!*} \mathtt{p}$ is defined in Fig. 15. Similarly, WFWait says that the *waiting* thread in a block queue waiting for the resource $\mathtt{m}$ becomes a well-formed *ready* thread if it gets $\mathtt{m}$ (see Fig. 6). The definitions of WFRdy and WFWait concisely formulate the relationship between current, ready and waiting threads.

4.3 Examples

Using our program logic, we can either certify a program module $\mathbb{C}$ by proving $\Psi, \Delta \vdash \mathbb{C} : \Psi'$, where Ψ, Δ and Ψ' are specifications provided by the user; or certify the well-formedness of a complete program configuration $\mathbb{W}$ by proving $\Psi, \Delta \vdash \mathbb{W}$ with the user provided specification Ψ and Δ. In the second case, we also need to prove $\Psi, \Delta \vdash \mathbb{W}.\mathbb{C} : \Psi'$ for some Ψ' to discharge the premise in the WLD rule, which is the major task of the verification process. In this section, we show how to specify and certify the Teeter-Totter example in Fig. 10 and the preemptive timer handler in Fig. 13.

The Teeter-Totter example We first instantiate INV0, the interrupt handler's specification for its local memory:

$$\mathsf{INV0} \stackrel{\text{def}}{=} \exists \mathtt{w}_l, \mathtt{w}_r.\ ((\mathtt{LEFT} \mapsto \mathtt{w}_l) * (\mathtt{RIGHT} \mapsto \mathtt{w}_r)) \wedge (\mathtt{w}_l + \mathtt{w}_r = n)\,,$$

where n is an auxiliary logical variable. Then we can get the concrete specification of the interrupt handler, following Formulae (3) and (4) in Section 4.1. We let INV1 be emp, since the non-handler code is sequential.

The specifications, including some important intermediate ones used during verification, are shown in Fig. 10 and defined in Fig. 21. Recall $\mathsf{enable}_{\mathsf{iret}}$ is defined in Fig. 16. To simplify our presentation, we present the predicate $\mathtt{p}$ in the form of a proposition with free variables referring to components of the state $\mathbb{S}$. Also, we use the heap predicate $\mathtt{m}$ as a shorthand for the proposition $\mathtt{m}\,\mathbb{H}$ when there is no confusion.

If we compare $\mathtt{p}_1$ and $\mathtt{p}_2$, we will see that the non-handler code cannot access memory at addresses LEFT and RIGHT without first disabling the interrupt because $\mathtt{p}_1$ does not contain the ownership of memory cells at the locations LEFT and

Fig. 21 Specifications of the Teeter-Totter example

$$\mathtt{p} \stackrel{\text{def}}{=} (\mathtt{ie} = 1) \wedge (\mathtt{is} = 0) \quad \mathtt{p}' \stackrel{\text{def}}{=} (\mathtt{ie} = 0) \wedge (\mathtt{is} = 0) \quad \mathtt{p}_0 \stackrel{\text{def}}{=} \mathtt{p}$$
$$\mathtt{p}_1 \stackrel{\text{def}}{=} \mathtt{p} \wedge (r_1 = \mathtt{RIGHT}) \wedge (r_2 = \mathtt{LEFT})$$
$$\mathtt{p}_2 \stackrel{\text{def}}{=} \mathtt{p}' \wedge (r_1 = \mathtt{RIGHT}) \wedge (r_2 = \mathtt{LEFT}) \wedge (r_3 = 0) \wedge (\mathsf{INV0} * \mathsf{true})$$
$$\mathtt{p}_3 \stackrel{\text{def}}{=} \mathtt{p}' \wedge (r_1 = \mathtt{RIGHT}) \wedge (r_2 = \mathtt{LEFT}) \wedge (\mathsf{INV0} * \mathsf{true})$$
$$\mathtt{p}_4 \stackrel{\text{def}}{=} \mathsf{enable}_{\mathsf{iret}} \qquad \mathsf{NoG} \stackrel{\text{def}}{=} \lambda \mathbb{S}, \mathbb{S}'.\mathsf{False}$$

RIGHT. Since the non-handler never returns, we simply use NoG (see Fig. 21) as the guarantee for the state transition from the specified point to the return point.

The code heap specification Ψ is defined as:

$$\Psi \stackrel{\text{def}}{=} \{\mathtt{incleft} \rightsquigarrow (\mathtt{p}_0, \mathsf{NoG}), \mathtt{l\_loop} \rightsquigarrow (\mathtt{p}_1, \mathsf{NoG}), \mathtt{l\_win} \rightsquigarrow (\mathtt{p}_3, \mathsf{NoG}),$$
$$\mathtt{h\_entry} \rightsquigarrow (\mathtt{p}_i, \mathtt{g}_i), \mathtt{r\_win} \rightsquigarrow (\mathtt{p}_4, \mathsf{gid})\}$$

We define Δ as $\varnothing$. To certify the program, we need to prove $\Psi, \Delta \vdash \{\mathtt{s}\}\,\mathtt{f} : \mathbb{C}[\mathtt{f}]$ for each $(\mathtt{f}, \mathtt{s})$ in Ψ. Here $\mathbb{C}$ represents the whole program shown in Fig. 10. The verification follows the rules in Fig. 16. We do not show the details here. Note that $\mathbb{C}[\mathtt{l\_loop}]$ is a sub-sequence of $\mathbb{C}[\mathtt{incleft}]$. However, we do not need to verify $\mathbb{C}[\mathtt{l\_loop}]$ twice. The verification of $\mathbb{C}[\mathtt{l\_loop}]$ can be reused when $\mathbb{C}[\mathtt{incleft}]$ is verified.

The timer handler We briefly explain the specification for the preemptive timer handler shown in Fig. 13. The handler only accesses the memory cell at the location CNT. We instantiate INV0 below:

$$\mathsf{INV0} \stackrel{\text{def}}{=} \exists \mathtt{w}.\ (\mathtt{CNT} \mapsto \mathtt{w}) \wedge (\mathtt{w} \leq 100)\,.$$

Then we get the specification of the handler $(\mathtt{p}_i, \mathtt{g}_i)$ by Formulae (3) and (4). In $\mathtt{g}_0$ (shown in Fig. 13), we use primed variable (*e.g.,* $\mathtt{ie}'$ and $\mathtt{is}'$) to refer to components in the second argument. Like the use of m as the shorthand for $\mathtt{m}\,\mathbb{H}$, we omit the arguments of heap transitions in $\mathtt{g}_0$ for the clarity of presentations.

4.4 Soundness

Our program logic is sound. The soundness theorem, Theorem 3, says that certified programs never abort, and the behaviors of certified programs satisfy their specifications in the sense that assertions p inserted in code heaps $\mathbb{C}$ are satisfied when the specified program points are reached by jump instructions, branch instructions or function calls. Assertions at these points are of particular interest because they correspond to loop invariants and preconditions of functions in higher-level programs.

Theorem 3 (Soundness) If INV0 and INV1 are precise, $\Psi, \Delta \vdash \mathbb{W}$, and $(\mathtt{h\_entry}, (\mathtt{p}_i, \mathtt{g}_i)) \in \Psi$, then there does *not* exist n such that $\mathbb{W} \Longmapsto^n \mathsf{abort}$; and for all n and $\mathbb{W}'$, if $\mathbb{W} \Longmapsto^n \mathbb{W}'$ and $\mathbb{W}' = (\mathbb{C}, \mathbb{S}, \mathbb{K}, \mathtt{pc}, \mathtt{tid}, \mathbb{T}, \mathbb{B})$, then the following are true:

1. if $\mathbb{C}(\mathtt{pc}) = \mathtt{j\ f}$, then there exists $(\mathtt{p}, \mathtt{g})$ such that $(\mathtt{f}, (\mathtt{p}, \mathtt{g})) \in \Psi$ and $(\mathtt{p} * \mathit{true})\ \mathbb{K}\ \mathbb{S}$;
2. if $\mathbb{C}(\mathtt{pc}) = \mathsf{beq}\ r_s, r_t, \mathtt{f}$ and $\mathbb{S}.\mathbb{R}(r_s) = \mathbb{S}.\mathbb{R}(r_t)$, then there exists $(\mathtt{p}, \mathtt{g})$ such that $(\mathtt{f}, (\mathtt{p}, \mathtt{g})) \in \Psi$ and $(\mathtt{p} * \mathit{true})\ \mathbb{K}\ \mathbb{S}$;

3. if $\mathbb{C}(\mathtt{pc}) = \mathsf{call}\ \mathtt{f}$, then there exists $(\mathtt{p}, \mathtt{g})$ such that $(\mathtt{f}, (\mathtt{p}, \mathtt{g})) \in \Psi$ and $(\mathtt{p} * \mathit{true})\ (\mathtt{pc} :: \mathbb{K})\ \mathbb{S}$.

Recall that precision is defined in Fig. 15; $\mathtt{p}_\mathsf{i}$ and $\mathtt{g}_\mathsf{i}$ are defined by formulae (3) and (4).

Proof By Lemma 4 (shown below) we know $\mathbb{W}$ does not abort. Since $\mathbb{W} \Longmapsto^n \mathbb{W}'$, we also know $\Psi, \Delta \vdash \mathbb{W}'$. By Lemma 22 we know items 1–3 are true. □

The following safety lemma shows certified programs never abort. *More importantly*, we know the invariant formulated by the WLD rule always holds during program execution, from which we can derive rich properties of programs.

Lemma 4 (Safety) *If* INV0 *and* INV1 *are precise,* $\Psi, \Delta \vdash \mathbb{W}$*, and* $(\mathtt{h\_entry}, (\mathtt{p}_\mathsf{i}, \mathtt{g}_\mathsf{i})) \in \Psi$*, then there does* not *exist n such that* $\mathbb{W} \Longmapsto^n$ abort*; and for all n and* $\mathbb{W}'$*, if* $\mathbb{W} \Longmapsto^n \mathbb{W}'$*, then* $\Psi, \Delta \vdash \mathbb{W}'$.

Proof We do induction over n. The proof follows the syntactic approach to proving the type safety [38]. We apply the progress lemmas (Lemmas 5 and 7) and the preservation lemma (Lemma 8). The progress lemmas show a well-formed program configuration $\mathbb{W}$ can always execute one more step. The preservation lemma shows that, starting from a well-formed $\mathbb{W}$, the new program configuration reached after one step of execution is also well-formed. □

Lemma 5 (Progress) *If* $\Psi, \Delta \vdash \mathbb{W}$*, then there exists* $\mathbb{W}'$ *such that* $\mathbb{W} \longmapsto \mathbb{W}'$.

Proof By the PC rule and the auxiliary relations defined in Section 3, we know there always exists $\mathbb{W}'$ if the next command at pc is one of move instructions, arithmetic instructions, function calls, conditional branches, jumps, cli or sti. So we only discuss about the rest of instructions.

Suppose $\mathbb{W} = (\mathbb{C}, \mathbb{S}, \mathbb{K}, \mathtt{pc}, \mathtt{tid}, \mathbb{T}, \mathbb{B})$ and $\mathbb{S} = (\mathbb{H}, \mathbb{R}, \mathtt{ie}, \mathtt{is})$. Since $\Psi, \Delta \vdash \mathbb{W}$, by the WLD rule we know there exist Ψ' and $\mathbb{H}_0 \subseteq \mathbb{H}$ such that $\Psi, \Delta \vdash \mathbb{C}$: Ψ' and $\mathsf{WFCth}(\mathbb{S}_0, \mathbb{K}, \mathtt{pc}, \Psi')$ hold, where $\mathbb{S}_0 = \mathbb{S}|_{\mathbb{H}_0}$. Therefore, by the definition of WFCth we know there exist $\mathtt{p}$ and $\mathtt{g}$ such that (1) $(\mathtt{pc}, (\mathtt{p}, \mathtt{g})) \in \Psi'$; (2) $(\mathtt{p} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}_0$; and (3) $\mathsf{WFST}(\lfloor \mathtt{g} \rfloor, \mathbb{S}_0, \mathbb{K}, \Psi')$. By (1) and the CDHP rule we have (4) $\Psi, \Delta \vdash \{(\mathtt{p}, \mathtt{g})\}\ \mathtt{pc} : \mathbb{C}[\mathtt{pc}]$.

If $\mathbb{C}(\mathtt{pc})$ is a load or store, by (4) and the SEQ rule we have $\mathsf{enable}(\mathtt{p}, \mathtt{NextS}_{(\mathbb{C}(\mathtt{pc}),\_)})$. By (2), $\mathbb{H}_0 \subseteq \mathbb{H}$ and Lemma 6 we know there exists $\mathbb{S}'$ such that $\mathtt{NextS}_{(\mathbb{C}(\mathtt{pc}),\_)}\ \mathbb{S}\ \mathbb{S}'$. We let $\mathbb{W}' = (\mathbb{C}, \mathbb{S}', \mathbb{K}, \mathtt{pc}{+}1, \mathtt{tid}, \mathbb{T}, \mathbb{B})$. By the PC rule we know $\mathbb{W} \longmapsto \mathbb{W}'$.

If $\mathbb{C}(\mathtt{pc})$ is ret, by (4) and the RET rule we know $\mathtt{p} \Rightarrow \mathsf{enable}_{\mathsf{ret}}$. By (2) we know there exists $\mathtt{f}$ and $\mathbb{K}'$ such that $\mathbb{K} = \mathtt{f} :: \mathbb{K}'$. Let $\mathbb{W}' = (\mathbb{C}, \mathbb{S}, \mathbb{K}', \mathtt{pc}{+}1, \mathtt{tid}, \mathbb{T}, \mathbb{B})$. We know $\mathbb{W} \longmapsto \mathbb{W}'$. The proof is similar if $\mathbb{C}(\mathtt{pc})$ is iret.

If $\mathbb{C}(\mathtt{pc})$ is switch, block r_s or unblock r_s, r_d, by (4) and the SEQ rule we can prove $\mathsf{enable}(\mathtt{p}, [\![\, \mathbb{C}(\mathtt{pc}) \,]\!]_\Delta)$. Then by (2) we know $\mathtt{ie} = 0$. By the operational semantics shown in Fig. 12 we know there exists $\mathbb{W}'$ such that $\mathbb{W} \longmapsto \mathbb{W}'$. □

Proof of Lemma 5 uses the following monotonicity property of program state transitions. It says the safety of the program is preserved by heap extensions.

Lemma 6 (NextS-Monotonicity) *If* $\texttt{NextS}_{(\texttt{C},\mathbb{K})}\ (\mathbb{H}, \mathbb{R}, \texttt{ie}, \texttt{is})\ \mathbb{S}'$, *and* $\mathbb{H} \perp \mathbb{H}'$, *then there exists* $\mathbb{S}''$ *such that* $\texttt{NextS}_{(\texttt{C},\mathbb{K})}\ (\mathbb{H} \uplus \mathbb{H}', \mathbb{R}, \texttt{ie}, \texttt{is})\ \mathbb{S}''$.

Proof Trivial, by inspection of the definition of NextS in Fig. 9. □

The following lemma says the program can always reach the entry point of the interrupt handler as long as the interrupt is enabled and there is no interrupts being serviced.

Lemma 7 (Progress-IRQ) *If* $\mathbb{W}.\mathbb{S}.\texttt{ie} = 1$ *and* $\mathbb{W}.\mathbb{S}.\texttt{is} = 0$, *there always exists* $\mathbb{W}'$ *such that* $\mathbb{W} \lightning \mathbb{W}'$.

Proof It trivially follows the IRQ rule shown in Section 3.1. □

Lemma 8 (Preservation) *If* INV0 *and* INV1 *are precise,* $(\texttt{h\_entry}, (\texttt{p}_\texttt{i}, \texttt{g}_\texttt{i})) \in \Psi$, $\Psi, \Delta \vdash \mathbb{W}$ *and* $\mathbb{W} \Longrightarrow \mathbb{W}'$, *we have* $\Psi, \Delta \vdash \mathbb{W}'$.

Proof Since $\Psi, \Delta \vdash \mathbb{W}'$, we know there are two possible cases: $\mathbb{W} \lightning \mathbb{W}'$ or $\mathbb{W} \longmapsto \mathbb{W}'$. We apply Lemma 9 and Lemma 10 respectively. □

The following lemma says the well-formedness of program configurations is preserved when an interrupt comes and the control is transferred to the interrupt handler.

Lemma 9 (Preservation-IRQ) *If* INV0 *and* INV1 *are precise,* $(\texttt{h\_entry}, (\texttt{p}_\texttt{i}, \texttt{g}_\texttt{i})) \in \Psi$, $\Psi, \Delta \vdash \mathbb{W}$ *and* $\mathbb{W} \lightning \mathbb{W}'$, *we have* $\Psi, \Delta \vdash \mathbb{W}'$.

Proof Suppose $\mathbb{W} = (\mathbb{C}, \mathbb{S}, \mathbb{K}, \texttt{pc}, \texttt{tid}, \mathbb{T}, \mathbb{B})$ and $\mathbb{S} = (\mathbb{H}, \mathbb{R}, \texttt{ie}, \texttt{is})$. By $\Psi, \Delta \vdash \mathbb{W}$ and the WLD rule we know there exist $\Psi' \supseteq \Psi$ and $\mathbb{H}_0$ such that:

$$\mathbb{H}_0 \subseteq \mathbb{H} \tag{p1}$$

$$\mathsf{WFCth}(\mathbb{S}_0, \mathbb{K}, \texttt{pc}, \Psi'),\ \text{where}\ \mathbb{S}_0 = (\mathbb{H}_0, \mathbb{R}, \texttt{ie}, \texttt{is}) \tag{p2}$$

Since $\mathbb{W} \lightning \mathbb{W}'$, by the IRQ rule (with the extension for multiple threads in AIM-II) we know $\texttt{ie} = 1, \texttt{is} = 0$, and $\mathbb{W}' = (\mathbb{C}, (\mathbb{H}, \mathbb{R}, 0, 1), (\texttt{pc}, \mathbb{R}) :: \mathbb{K}, \texttt{h\_entry}, \texttt{tid}, \mathbb{T}, \mathbb{B})$. Since the transition does not change $\mathbb{C}, \mathbb{H}, \mathbb{T}$ and $\mathbb{B}$, by the WLD rule we know we only need to prove $\mathsf{WFCth}(\mathbb{S}'_0, (\texttt{pc}, \mathbb{R}) :: \mathbb{K}, \texttt{h\_entry}, \Psi')$, where $\mathbb{S}'_0 = (\mathbb{H}_0, \mathbb{R}, 0, 1)$.

Since we know $(\texttt{h\_entry}, (\texttt{p}_\texttt{i}, \texttt{g}_\texttt{i})) \in \Psi$, by the definition of WFCth we need to prove:

$$(\texttt{p}_\texttt{i} * \mathsf{Inv})\ ((\texttt{pc}, \mathbb{R}) :: \mathbb{K})\ \mathbb{S}'_0 \tag{g1}$$

$$\mathsf{WFST}(\lfloor \texttt{g}_\texttt{i} \rfloor, \mathbb{S}_0, (\texttt{pc}, \mathbb{R}) :: \mathbb{K}, \Psi') \tag{g2}$$

By (p2) we know there exist $\mathtt{p}$ and $\mathtt{g}$ such that

$$(\mathtt{pc}, (\mathtt{p}, \mathtt{g})) \in \Psi' \tag{p2.1}$$

$$(\mathtt{p} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}_0 \tag{p2.2}$$

$$\mathsf{WFST}(\lfloor \mathtt{g} \rfloor, \mathbb{S}_0, \mathbb{K}, \Psi') \tag{p2.3}$$

Because $\mathtt{ie} = 1$ and $\mathtt{is} = 0$, by (p2.2) and the definition of $\mathtt{p} * \mathsf{Inv}$ we know $(\mathtt{p} * \mathsf{INV}_s)\ \mathbb{K}\ \mathbb{S}_0$. Therefore $(\mathsf{INV}_s * \mathit{true})\ \mathbb{H}_0$. By the definition of $\mathtt{p}_\mathtt{i}$ (Formula (3)) we can prove (g1).

To prove (g2), by the definition of WFST we need to prove that, for all $\mathbb{S}'$ and $\mathbb{S}''$ such that $\lfloor \mathtt{g}_\mathtt{i} \rfloor\ \mathbb{S}_0 \mathbb{S}'$ and $\mathbb{S}'' = (\mathbb{S}'.\mathbb{H}, \mathbb{R}, 1, 0)$, the following are true:

$$(\mathtt{p} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}'' \tag{g2.1}$$

$$\mathsf{WFST}(\lfloor \mathtt{g} \rfloor, \mathbb{S}'', \mathbb{K}, \Psi') \tag{g2.2}$$

By (p2.2), $\lfloor \mathtt{g}_\mathtt{i} \rfloor\ \mathbb{S}_0 \mathbb{S}'$, the precision of INV0 and INV1, and the definition of $\mathtt{g}_\mathtt{i}$ (Formula (4)), we can prove (g2.1). We can also prove

$$\forall \mathbb{S}.\ \lfloor \mathtt{g} \rfloor\ \mathbb{S}''\ \mathbb{S} \rightarrow \lfloor \mathtt{g} \rfloor\ \mathbb{S}_0\ \mathbb{S}\,.$$

Then we know (g2.2) holds by Lemma 16. □

The lemma below shows that executing the next instruction at pc preserves the well-formedness of program configurations.

Lemma 10 (Preservation-PC) *If* INV0 *and* INV1 *are precise,* $\Psi, \Delta \vdash \mathbb{W}$ *and* $\mathbb{W} \longmapsto \mathbb{W}'$*, we have* $\Psi, \Delta \vdash \mathbb{W}'$.

Proof Suppose $\mathbb{W} = (\mathbb{C}, \mathbb{S}, \mathbb{K}, \mathtt{pc}, \mathtt{tid}, \mathbb{T}, \mathbb{B})$ and $\mathbb{S} = (\mathbb{H}, \mathbb{R}, \mathtt{ie}, \mathtt{is})$. Also suppose $\mathbb{T} \setminus \mathtt{tid} = \{\mathtt{tid}_1 \rightsquigarrow (\mathbb{R}_1, \mathbb{K}_1, \mathtt{is}_1, \mathtt{pc}_1), \ldots, \mathtt{tid}_n \rightsquigarrow (\mathbb{R}_n, \mathbb{K}_n, \mathtt{is}_n, \mathtt{pc}_n)\}$. By $\Psi, \Delta \vdash \mathbb{W}$ and the WLD rule we know there exist $\Psi', \mathbb{H}_0, \mathbb{H}_1, \ldots, \mathbb{H}_n$ such that:

$$\mathbb{H} = \mathbb{H}_0 \uplus \ldots \uplus \mathbb{H}_n \tag{p1}$$

$$\Psi, \Delta \vdash \mathbb{C} : \Psi' \tag{p2}$$

$$\Psi \subseteq \Psi' \tag{p3}$$

$$\mathit{dom}(\Delta) = \mathit{dom}(\mathbb{B}) \tag{p4}$$

$$\mathsf{WFCth}(\mathbb{S}_0, \mathbb{K}, \mathtt{pc}, \Psi'),\ \text{where } \mathbb{S}_0 = (\mathbb{H}_0, \mathbb{R}, \mathtt{ie}, \mathtt{is}) \tag{p5}$$

for all $0 < k \leq n$ such that $\mathtt{tid}_k \in \mathtt{readyQ}(\mathbb{T}, \mathbb{B})$:

$$\mathsf{WFRdy}(\mathbb{S}_k, \mathbb{K}_k, \mathtt{pc}_k, \Psi'),\ \text{where } \mathbb{S}_k = (\mathbb{H}_k, \mathbb{R}_k, 0, \mathtt{is}_k) \tag{p6}$$

for all w and $0 < j \leq n$ such that $\mathtt{tid}_j \in \mathbb{B}(w)$:

$$\mathsf{WFWait}(\mathbb{S}_j, \mathbb{K}_j, \mathtt{pc}_j, \Psi', \Delta(w)) \text{ where } \mathbb{S}_j = (\mathbb{H}_j, \mathbb{R}_j, 0, \mathtt{is}_j) \tag{p7}$$

By (p5) we know there exist $\mathtt{p}$ and $\mathtt{g}$ such that

$$(\mathtt{pc}, (\mathtt{p}, \mathtt{g})) \in \Psi' \tag{p5.1}$$

$$(\mathtt{p} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}_0 \tag{p5.2}$$

$$\mathsf{WFST}(\lfloor \mathtt{g} \rfloor, \mathbb{S}, \mathbb{K}, \Psi') \tag{p5.3}$$

By (p5.1), (p2), and the CDHP rule we know:

$$\Psi, \Delta \vdash \{(\mathtt{p}, \mathtt{g})\}\, \mathtt{pc} : \mathbb{C}[\mathtt{pc}] \tag{p2.1}$$

We analyze different cases of $\mathbb{C}(\mathtt{pc})$.

Case: $\mathbb{C}(\mathtt{pc}) = \mathsf{switch}$ Suppose a thread $\mathtt{tid}'$ in $\mathtt{readyQ}(\mathbb{T}, \mathbb{B})$ is picked to run after switch. There are two cases: $\mathtt{tid}' \neq \mathtt{tid}$ or $\mathtt{tid}' = \mathtt{tid}$. In the first case, we know there exists some $i > 0$ such that $\mathtt{tid}' = \mathtt{tid}_i$. Therefore $\mathbb{W}' = (\mathbb{C}, (\mathbb{H}, \mathbb{R}_i, 0, \mathtt{is}_i), \mathbb{K}_i, \mathtt{pc}_i, \mathtt{tid}_i, \mathbb{T}', \mathbb{B})$, where $\mathbb{T}' = \mathbb{T}\{\mathtt{tid} \rightsquigarrow (\mathbb{R}, \mathbb{K}, \mathtt{is}, \mathtt{pc}{+}1)\}$. By the WLD rule, we need to find a Ψ'', $\mathbb{H}'_0$ and $\mathbb{H}'_i$ such that:

$$\mathbb{H} = \mathbb{H}'_0 \uplus \mathbb{H}_1 \ldots \uplus \mathbb{H}_{i-1} \uplus \mathbb{H}'_i \uplus \mathbb{H}_{i+1} \ldots \mathbb{H}_n \tag{g1}$$

$$\Psi, \Delta \vdash \mathbb{C} : \Psi'' \tag{g2}$$

$$\Psi \subseteq \Psi'' \tag{g3}$$

$$\mathsf{WFCth}(\mathbb{S}'_i, \mathbb{K}_i, \mathtt{pc}_i, \Psi''),\ \text{where } \mathbb{S}'_i = (\mathbb{H}'_i, \mathbb{R}_i, 0, \mathtt{is}_i) \tag{g4}$$

$$\mathsf{WFRdy}(\mathbb{S}'_0, \mathbb{K}, \mathtt{pc}{+}1, \Psi''),\ \text{where } \mathbb{S}'_0 = (\mathbb{H}'_0, \mathbb{R}, \mathtt{ie}, \mathtt{is}) \tag{g5}$$

for all $0 < k \leq n$ such that $k \neq i$ and $\mathtt{tid}_k \in \mathtt{readyQ}(\mathbb{T}, \mathbb{B})$:

$$\mathsf{WFRdy}(\mathbb{S}_k, \mathbb{K}_k, \mathtt{pc}_k, \Psi''),\ \text{where } \mathbb{S}_k = (\mathbb{H}_k, \mathbb{R}_k, 0, \mathtt{is}_k) \tag{g6}$$

for all w and $0 < j \leq n$ such that $\mathtt{tid}_j \in \mathbb{B}(w)$:

$$\mathsf{WFWait}(\mathbb{S}_j, \mathbb{K}_j, \mathtt{pc}_j, \Psi'', \Delta(w)) \text{ where } \mathbb{S}_j = (\mathbb{H}_j, \mathbb{R}_j, 0, \mathtt{is}_j) \tag{g7}$$

By (p2.1) and the SEQ rule we know there exist $\mathtt{p}'$ and $\mathtt{g}'$ such that

$$\Psi, \Delta \vdash \{(\mathtt{p}', \mathtt{g}')\}\, \mathtt{pc}{+}1 : \mathbb{C}[\mathtt{pc}{+}1] \tag{p2.1.1}$$

$$\mathsf{enable}(\mathtt{p}, [\![\, \mathsf{switch} \,]\!]_\Delta) \tag{p2.1.2}$$

$$(\mathtt{p} \rhd [\![\, \mathsf{switch} \,]\!]_\Delta) \Rightarrow \mathtt{p}' \tag{p2.1.3}$$

$$(\mathtt{p} \circ ([\![\, \mathsf{switch} \,]\!]_\Delta \circ \mathtt{g}')) \Rightarrow \mathtt{g} \tag{p2.1.4}$$

We let $\Psi'' = \Psi' \cup \{(\mathtt{pc}{+}1, (\mathtt{p}', \mathtt{g}'))\}$. So (g3) is trivial. The proof of (g2), (g6) and (g7) follows Lemmas 17 and 18. By (p5.2) and Lemma 11 we know there exist $\mathbb{H}_{01}$ and $\mathbb{H}_{02}$ such that $\mathbb{H}_0 = \mathbb{H}_{01} \uplus \mathbb{H}_{02}$ and $\mathsf{INV}_s\ \mathbb{H}_{02}$ (INV_s is defined in Fig. 19). We let

$\mathbb{H}'_0 = \mathbb{H}_{01}$ and $\mathbb{H}'_i = \mathbb{H}_i \uplus \mathbb{H}_{02}$, *i.e.,* the current thread $\mathtt{tid}$ transfers the sub-heap $\mathbb{H}_{02}$ to the thread $\mathtt{tid}_i$. (g1) is trivial. We prove (g4) by applying Lemma 11, and (g5) by Lemma 12.

In the second case ($\mathtt{tid}' = \mathtt{tid}$), the current thread $\mathtt{tid}$ is picked to run again. By a combination of Lemma 11 and 12 we know the current thread $\mathtt{tid}$ is still well-formed at $\mathtt{pc}+1$. The proof is similar to the first case.

Case: $\mathbb{C}(\mathtt{pc}) = \mathsf{block}\ r_s$ If there are no other threads in the ready queue except the current thread, the program stutters and the proof is trivial. Otherwise, $\mathsf{block}\ r_s$ puts the current thread onto the corresponding block queue and picks a thread from the ready queue as the current thread. The proof follows similar structure of the proof above for switch. Lemma 13 shows the current thread becomes a well-formed waiting thread after it transfers a sub-heap satisfying INV_s to the ready thread. Lemma 12 shows the ready thread becomes a well-formed current thread after it receives the sub-heap.

Case: $\mathbb{C}(\mathtt{pc}) = \mathsf{unblock}\ r_t, r_d$ If the corresponding block queue is empty, the only effect of this instruction is to set r_d to 0. The proof is simple and elided here. Otherwise, $\mathsf{unblock}\ r_t, r_d$ moves a waiting thread from the block queue to the ready queue. The proof follows similar structure of the proof above for switch. Lemma 14 shows the current thread is still well-formed at $\mathtt{pc}+1$ after transferring a sub-heap satisfying $\Delta(\mathbb{R}(r_t))$ to the blocked thread. Lemma 15 shows the waiting thread becomes a well-formed ready thread after it receives the resource it is waiting for.

Case: $\mathbb{C}(\mathtt{pc}) = \mathsf{iret}$ Since $\mathbb{W} \longmapsto \mathbb{W}'$, we know there exist $\mathtt{pc}'$, $\mathbb{R}'$ and $\mathbb{K}'$ such that $\mathbb{K} = (\mathtt{pc}', \mathbb{R}') :: \mathbb{K}'$ and $\mathbb{W}' = (\mathbb{C}, (\mathbb{H}, \mathbb{R}', 1, 0), \mathbb{K}', \mathtt{pc}', \mathtt{tid}, \mathbb{T}, \mathbb{B})$. To prove $\Psi, \Delta \vdash \mathbb{W}'$, by the WLD rule we only need to prove:

$$\mathsf{WFCth}(\mathbb{S}'_0, \mathbb{K}', \mathtt{pc}', \Psi'), \text{ where } \mathbb{S}'_0 = (\mathbb{H}_0, \mathbb{R}', 1, 0) \tag{g1}$$

By the definition of WFCth, we need to prove there exist $\mathtt{p}'$ and $\mathtt{g}'$ such that

$$(\mathtt{pc}', (\mathtt{p}', \mathtt{g}')) \in \Psi' \tag{g1.1}$$

$$(\mathtt{p}' * \mathsf{Inv})\ \mathbb{K}'\ \mathbb{S}'_0 \tag{g1.2}$$

$$\mathsf{WFST}(\lfloor \mathtt{g}' \rfloor, \mathbb{S}'_0, \mathbb{K}', \Psi') \tag{g1.3}$$

By (p2.1) and the IRET rule we know $(\mathtt{p} \rhd \mathsf{gid}) \Rightarrow \mathtt{g}$. Then by (p5.2) we know $\lfloor \mathtt{g} \rfloor\ \mathbb{S}_0\ \mathbb{S}_0$. Together with (p5.3) and the definition of WFST, we know g1.1, g1.2 and g1.3 are true.

Case: $\mathbb{C}(\mathtt{pc}) = \mathsf{ret}$ The proof is similar to the above proof for iret.

Case: $\mathbb{C}(\mathtt{pc}) = \mathsf{call}\ \mathtt{f}$ **or** $\mathbb{C}(\mathtt{pc}) = \mathtt{j}\ \mathtt{f}$ The proof for call f is similar to the proof of Lemma 9, since the transfer of control to the interrupt handler can be viewed as a special function call. The jump instruction j f is similar, but it does not change the stack.

Case: $\mathbb{C}(\mathtt{pc})$ is one of mov, movi, add, sub, ld, st, cli or sti instructions. These sequential instructions do not change the stack, the ready queue and block queues. We only need to prove that the current thread is still well-formed at pc+1. The proof is similar to the proof for switch. Since the st instruction updates the heap, its proof applies the frame property shown in Lemma 20. For cli and sti, we use $[\![\ \mathsf{cli}\ ]\!]_\Delta$ and $[\![\ \mathsf{sti}\ ]\!]_\Delta$ instead of $\mathtt{NextS}_{(\mathbb{C}(\mathtt{pc}),\mathbb{K})}$ to model state transitions. Their proofs apply Lemma 21.

Case: $\mathbb{C}(\mathtt{pc}) = \mathsf{beq}\ r_s, r_t, \mathtt{f}$ Depending on the validity of the condition, the branch instruction can be viewed either as a jump or a sequential instruction. The proofs for the two cases are similar to the proofs for j f and sequential instructions, respectively. □

The following lemma says that, after executing switch, the current thread becomes a well-formed ready thread and transfers a sub-heap satisfying INV_s to a ready thread that is scheduled to run. Lemma 12 shows the ready thread becomes a well-formed current thread after receiving the sub-heap.

Lemma 11 (Switch) *Suppose the premises of the* SEQ *rule are satisfied when* ι *is instantiated with* switch, *ie.,*(p1) $\Psi, \Delta \vdash \{(\mathtt{p}', \mathtt{g}')\}\,\mathtt{pc}{+}1 : \mathbb{C}[\mathtt{pc}{+}1]$; (p2) $\mathsf{enable}(\mathtt{p}, \mathtt{g}_\iota)$; (p3) $(\mathtt{p} \rhd \mathtt{g}_\iota) \Rightarrow \mathtt{p}'$; *and* (p4) $(\mathtt{p} \circ (\mathtt{g}_\iota \circ \mathtt{g}')) \Rightarrow \mathtt{g}$; *where* $\mathtt{g}_\iota = [\![\ \mathsf{switch}\ ]\!]_\Delta$. *If* $\mathsf{WFCth}(\mathbb{S}, \mathbb{K}, \mathtt{pc}, \Psi')$ *and* $\mathbb{S} = (\mathbb{H}, \mathbb{R}, \mathtt{ie}, \mathtt{is})$, *then there exist* $\mathbb{H}_1$ *and* $\mathbb{H}_2$ *such that* $\mathbb{H} = \mathbb{H}_1 \uplus \mathbb{H}_2$, $\mathsf{INV}_s\ \mathbb{H}_2$, *and* $\mathsf{WFRdy}((\mathbb{H}_1, \mathbb{R}, \mathtt{ie}, \mathtt{is}), \mathbb{K}, \mathtt{pc}{+}1, \Psi'')$, *where* $\Psi'' = \Psi' \cup \{(\mathtt{pc}{+}1, (\mathtt{p}', \mathtt{g}'))\}$.

Proof By $\mathsf{WFCth}(\mathbb{S}, \mathbb{K}, \mathtt{pc}, \Psi')$ and $\mathbb{S} = (\mathbb{H}, \mathbb{R}, \mathtt{ie}, \mathtt{is})$ we know

$$(\mathtt{pc}, (\mathtt{p}, \mathtt{g})) \in \Psi' \tag{p5}$$

$$(\mathtt{p} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S} \tag{p6}$$

$$\mathsf{WFST}(\lfloor \mathtt{g} \rfloor, \mathbb{S}, \mathbb{K}, \Psi') \tag{p7}$$

By (p2) and (p6) we can prove that there exist $\mathbb{H}_1$ and $\mathbb{H}_2$ such that $\mathbb{H} = \mathbb{H}_1 \uplus \mathbb{H}_2$ and $\mathsf{INV}_s\ \mathbb{H}_2$. To prove $\mathsf{WFRdy}((\mathbb{H}_1, \mathbb{R}, \mathtt{ie}, \mathtt{is}), \mathbb{K}, \mathtt{pc}{+}1, \Psi'')$, we need to prove, by the definition of WFRdy, that for all $\mathbb{H}_2'$ and $\mathbb{H}'$, if $\mathsf{INV}_s\ \mathbb{H}_2'$ and $\mathbb{H}' = \mathbb{H}_1 \uplus \mathbb{H}_2'$, then $\mathsf{WFCth}(\mathbb{S}', \mathbb{K}, \mathtt{pc}{+}1, \Psi'')$, where $\mathbb{S}' = (\mathbb{H}', \mathbb{R}, \mathtt{ie}, \mathtt{is})$. We need to prove

$$(\mathtt{pc}{+}1, (\mathtt{p}', \mathtt{g}')) \in \Psi'' \tag{g1}$$

$$(\mathtt{p}' * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}' \tag{g2}$$

$$\mathsf{WFST}(\lfloor \mathtt{g}' \rfloor, \mathbb{S}', \mathbb{K}, \Psi'') \tag{g3}$$

(g1) trivially follows our assumption. We can prove (g2) by (p2), (p3) and (p6). By (p2), (p4) and (p6) we know $\forall \mathbb{S}''. \lfloor \mathtt{g} \rfloor\ \mathbb{S}\ \mathbb{S}'' \rightarrow \lfloor \mathtt{g}' \rfloor\ \mathbb{S}'\ \mathbb{S}''$. To prove (g3), we apply Lemma 16 and prove $\mathsf{WFST}(\lfloor \mathtt{g} \rfloor, \mathbb{S}, \mathbb{K}, \Psi'')$, which trivially follows Lemma 18 and (p7). □

Lemma 12 (Rdy-to-Run) *If* $\mathsf{WFRdy}(\mathbb{S}, \mathbb{K}, \mathtt{pc}, \Psi)$, $\mathsf{INV}_s\ \mathbb{H}'$ *and* $\mathbb{H}'' = \mathbb{S}.\mathbb{H} \uplus \mathbb{H}'$, *then* $\mathsf{WFCth}(\mathbb{S}|_{\mathbb{H}''}, \mathbb{K}, \mathtt{pc}, \Psi)$.

Proof Trivial, by the definition of WFRdy. □

The following lemma is similar to Lemma 11. It says that, after executing the block instruction, the current thread becomes a well-formed waiting thread and transfers a sub-heap satisfying INV_s to a ready thread that is scheduled to run.

Lemma 13 (Block) *Suppose the premises of the* SEQ *rule are satisfied when* ι *is instantiated with* block r_s, i.e., (p1) $\Psi, \Delta \vdash \{(\mathtt{p}', \mathtt{g}')\}\,\mathtt{pc}{+}1 : \mathbb{C}[\mathtt{pc}{+}1]$; (p2) $\mathsf{enable}(\mathtt{p}, \mathtt{g}_\iota)$; (p3) $(\mathtt{p} \rhd \mathtt{g}_\iota) \Rightarrow \mathtt{p}'$; *and* (p4) $(\mathtt{p} \circ (\mathtt{g}_\iota \circ \mathtt{g}')) \Rightarrow \mathtt{g}$; *where* $\mathtt{g}_\iota = [\![\ \mathsf{block}\ r_s\ ]\!]_\Delta$. *If* $\mathsf{WFCth}(\mathbb{S}, \mathbb{K}, \mathtt{pc}, \Psi')$ *and* $\mathbb{S} = (\mathbb{H}, \mathbb{R}, \mathtt{ie}, \mathtt{is})$, *then there exist* $\mathtt{m}$, $\mathbb{H}_1$ *and* $\mathbb{H}_2$ *such that* $\mathtt{m} = \Delta(\mathbb{R}(r_s))$, $\mathbb{H} = \mathbb{H}_1 \uplus \mathbb{H}_2$, $\mathsf{INV}_s\ \mathbb{H}_2$, *and* $\mathsf{WFWait}((\mathbb{H}_1, \mathbb{R}, \mathtt{ie}, \mathtt{is}), \mathbb{K}, \mathtt{pc}{+}1, \Psi'', \mathtt{m})$, where $\Psi'' = \Psi' \cup \{(\mathtt{pc}{+}1, (\mathtt{p}', \mathtt{g}'))\}$.

Proof Similar to the proof of Lemma 11. □

The following lemma says the current thread is still well-formed after it releases a blocked thread by transferring the resource that the blocked thread is waiting for. Lemma 15 says the waiting thread becomes a well-formed ready thread after receiving the resource.

Lemma 14 (Unblock) *Suppose the premises of the* SEQ *rule are satisfied when* ι *is instantiated with* unblock r_t, r_d, *i.e.,* (p1) $\Psi, \Delta \vdash \{(\mathtt{p}', \mathtt{g}')\}\,\mathtt{pc}{+}1 : \mathbb{C}[\mathtt{pc}{+}1]$; (p2) $\mathsf{enable}(\mathtt{p}, \mathtt{g}_\iota)$; (p3) $(\mathtt{p} \rhd \mathtt{g}_\iota) \Rightarrow \mathtt{p}'$; *and* (p4) $(\mathtt{p} \circ (\mathtt{g}_\iota \circ \mathtt{g}')) \Rightarrow \mathtt{g}$; *where* $\mathtt{g}_\iota = [\![\ \mathsf{unblock}\ r_t, r_d\ ]\!]_\Delta$. If $\mathsf{WFCth}(\mathbb{S}, \mathbb{K}, \mathtt{pc}, \Psi')$ *and* $\mathbb{S} = (\mathbb{H}, \mathbb{R}, \mathtt{ie}, \mathtt{is})$, *then there exist* $\mathtt{m}$, $\mathbb{H}_1$ *and* $\mathbb{H}_2$ *such that* $\mathtt{m} = \Delta(\mathbb{R}(r_t))$, $\mathbb{H} = \mathbb{H}_1 \uplus \mathbb{H}_2$, $\mathtt{m}\ \mathbb{H}_2$, *and for all* $n > 0$, $\mathsf{WFCth}((\mathbb{H}_1, \mathbb{R}\{r_d \rightsquigarrow n\}, \mathtt{ie}, \mathtt{is}), \mathbb{K}, \mathtt{pc}{+}1, \Psi'')$, *where* $\Psi'' = \Psi' \cup \{(\mathtt{pc}{+}1, (\mathtt{p}', \mathtt{g}'))\}$.

Proof Similar to the proof of Lemma 11. □

Lemma 15 (Released) *If* $\mathsf{WFWait}(\mathbb{S}, \mathbb{K}, \mathtt{pc}, \Psi, \mathtt{m})$, $\mathtt{m}\ \mathbb{H}'$ *and* $\mathbb{H}'' = \mathbb{S}.\mathbb{H} \uplus \mathbb{H}'$, *then* $\mathsf{WFRdy}(\mathbb{S}|_{\mathbb{H}''}, \mathbb{K}, \mathtt{pc}, \Psi)$.

Proof Trivial, by the definition of WFWait. □

The following lemma says, if a stack is well-formed at the state $\mathbb{S}$ where the current function has the remaining behavior of $\mathtt{g}$ to fulfill, it is still well-formed after the function reaches a new state $\mathbb{S}'$, as long as the new guaranteed behavior $\mathtt{g}'$ fulfills $\mathtt{g}$. The lemma is used to prove Lemmas 9 and 10.

Lemma 16 (WFST-Strengthen) *If* $\mathsf{WFST}(\mathtt{g}, \mathbb{S}, \mathbb{K}, \Psi)$ *and* $\forall \mathbb{S}''.\ \mathtt{g}'\ \mathbb{S}'\ \mathbb{S}'' \rightarrow \mathtt{g}\ \mathbb{S}\ \mathbb{S}''$, *then* $\mathsf{WFST}(\mathtt{g}', \mathbb{S}', \mathbb{K}, \Psi)$.

Proof Trivial, by the definition of WFST in Fig. 19. □

This lemma says we can extend the exported interface as long as the corresponding code block is well-formed with respect to the newly added specification.

Lemma 17 (Spec-Extension-I) *If* $\Psi, \Delta \vdash \mathbb{C} : \Psi'$, $\Psi, \Delta \vdash \{\mathtt{s}\}\,\mathtt{f} : \mathbb{C}[\mathtt{f}]$ *and* $\Psi'' = \Psi' \cup \{(\mathtt{f}, \mathtt{s})\}$, *then* $\Psi, \Delta \vdash \mathbb{C} : \Psi''$.

Proof It trivially follows the CDHP rule. □

The lemma below says an extension of the specification Ψ preserves the well-formedness of stacks, the current thread, ready threads and waiting threads.

Lemma 18 (Spec-Extension-II) *If* $\Psi \subseteq \Psi'$, *the following are true:*

1. *If* $\mathsf{WFST}(\mathtt{g}, \mathbb{S}, \mathbb{K}, \Psi)$, *then* $\mathsf{WFST}(\mathtt{g}, \mathbb{S}, \mathbb{K}, \Psi')$.
2. *If* $\mathsf{WFCth}(\mathbb{S}, \mathbb{K}, \mathtt{pc}, \Psi)$, *then* $\mathsf{WFCth}(\mathbb{S}, \mathbb{K}, \mathtt{pc}, \Psi')$.
3. *If* $\mathsf{WFRdy}(\mathbb{S}, \mathbb{K}, \mathtt{pc}, \Psi)$, *then* $\mathsf{WFRdy}(\mathbb{S}, \mathbb{K}, \mathtt{pc}, \Psi')$.
4. *If* $\mathsf{WFWait}(\mathbb{S}, \mathbb{K}, \mathtt{pc}, \Psi, \mathtt{m})$, *then* $\mathsf{WFWait}(\mathbb{S}, \mathbb{K}, \mathtt{pc}, \Psi', \mathtt{m})$.

Proof Trivial by inspecting the definitions. □

The next two lemmas show the standard frame properties [39] of the `NextS` relation. They are used to prove the preservation lemma.

Lemma 19 (`NextS`-Frame-I) *If* $\mathtt{NextS}_{(\mathtt{C},\mathbb{K})}\ (\mathbb{H}, \mathbb{R}, \mathtt{ie}, \mathtt{is})\ (\mathbb{H}', \mathbb{R}', \mathtt{ie}', \mathtt{is}')$, $\mathbb{H} = \mathbb{H}_1 \uplus \mathbb{H}_2$, *and there exists a state* $\mathbb{S}'$ *such that* $\mathtt{NextS}_{(\mathtt{C},\mathbb{K})}\ (\mathbb{H}_1, \mathbb{R}, \mathtt{ie}, \mathtt{is})\ \mathbb{S}'$, *then there exists a sub-heap* $\mathbb{H}'_1$ *such that* $\mathbb{H}' = \mathbb{H}'_1 \uplus \mathbb{H}_2$, *and* $\mathtt{NextS}_{(\mathtt{C},\mathbb{K})}\ (\mathbb{H}_1, \mathbb{R}, \mathtt{ie}, \mathtt{is})\ (\mathbb{H}'_1, \mathbb{R}', \mathtt{ie}', \mathtt{is}')$.

Proof By inspection of the definition of the `NextS` relation in Fig. 9. □

Lemma 20 (`NextS`-Frame-II) *If* $\mathtt{NextS}_{(\mathtt{C},\mathbb{K})}\ \mathbb{S}\ \mathbb{S}'$, $(\mathtt{p} * \mathtt{m})\ \mathbb{K}\ \mathbb{S}$, $(\mathtt{p} \rhd \mathtt{NextS}_{(\mathtt{C},\mathbb{K})}) \Rightarrow \mathtt{p}'$, *and* $\mathsf{enable}(\mathtt{p}, \mathtt{NextS}_{(\mathtt{C},\mathbb{K})})$, *then* $(\mathtt{p}' * \mathtt{m})\ \mathbb{K}\ \mathbb{S}'$.

Proof This lemma follows Lemma 19. □

The lemma below is an auxiliary lemma to prove that the well-formed current thread is still well-formed after the state transition of $[\![\,\mathsf{cli}\,]\!]_\Delta$ or $[\![\,\mathsf{sti}\,]\!]_\Delta$. It is used in Lemma 10 to prove the preservation of cli and sti.

Lemma 21 (CLI & STI) *If* $(\mathtt{p} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}$, $\mathbb{S}' = \mathbb{S}|_{\mathtt{ie}=0}$, *and* $\mathtt{g}_\iota = [\![\,\mathsf{cli}\,]\!]_\Delta$ (*or* $\mathtt{g}_\iota = [\![\,\mathsf{sti}\,]\!]_\Delta$), *then*

1. If $(\mathtt{p} \rhd \mathtt{g}_\iota) \Rightarrow \mathtt{p}'$, then $(\mathtt{p}' * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}'$.
2. If $(\mathtt{p} \circ (\mathtt{g}_\iota \circ \mathtt{g}')) \Rightarrow \mathtt{g}$, $\mathsf{WFST}(\lfloor \mathtt{g} \rfloor, \mathbb{S}, \mathbb{K}, \Psi)$, and INV0 and INV1 are precise, then $\mathsf{WFST}(\lfloor \mathtt{g}' \rfloor, \mathbb{S}', \mathbb{K}, \Psi)$.

Proof The proof of 1 simply follows the definition of $\mathtt{p} * \mathsf{Inv}$ and $[\![\,\mathsf{cli}\,]\!]_\Delta$ (or $[\![\,\mathsf{sti}\,]\!]_\Delta$). To prove 2, we apply Lemma 16 and prove $\forall \mathbb{S}''.\ \lfloor \mathtt{g}' \rfloor\ \mathbb{S}'\ \mathbb{S}'' \rightarrow \lfloor \mathtt{g} \rfloor\ \mathbb{S}\ \mathbb{S}''$, which can be proved by the definition of $\lfloor \mathtt{g} \rfloor$ and $[\![\,\mathsf{cli}\,]\!]_\Delta$ (or $[\![\,\mathsf{sti}\,]\!]_\Delta$). □

This lemma says if the current program configuration is well-formed and the instruction at pc is a jump, a conditional branch or a function call, then the specification for the target address is satisfied when it is reached.

Lemma 22 *If* $\Psi, \Delta \vdash \mathbb{W}$ *and* $\mathbb{W} = (\mathbb{C}, \mathbb{S}, \mathbb{K}, \mathtt{pc}, \mathtt{tid}, \mathbb{T}, \mathbb{B})$, *then the following are true:*

1. *if* $\mathbb{C}(\mathtt{pc}) = \mathsf{j}\ \mathtt{f}$, *then there exists* $(\mathtt{p}, \mathtt{g})$ *such that* $(\mathtt{f}, (\mathtt{p}, \mathtt{g})) \in \Psi$ *and* $(\mathtt{p} * \mathit{true})\ \mathbb{K}\ \mathbb{S}$;
2. if $\mathbb{C}(\mathtt{pc}) = \mathsf{beq}\ r_s, r_t, \mathtt{f}$ *and* $\mathbb{S}.\mathbb{R}(r_s) = \mathbb{S}.\mathbb{R}(r_t)$, *then there exists* $(\mathtt{p}, \mathtt{g})$ *such that* $(\mathtt{f}, (\mathtt{p}, \mathtt{g})) \in \Psi$ *and* $(\mathtt{p} * \mathit{true})\ \mathbb{K}\ \mathbb{S}$;
3. *if* $\mathbb{C}(\mathtt{pc}) = \mathsf{call}\ \mathtt{f}$, *then there exists* $(\mathtt{p}, \mathtt{g})$ *such that* $(\mathtt{f}, (\mathtt{p}, \mathtt{g})) \in \Psi$ *and* $(\mathtt{p} * \mathit{true})\ (\mathtt{pc} :: \mathbb{K})\ \mathbb{S}$.

Proof Since $\Psi, \Delta \vdash \mathbb{W}$, as the proof of Lemma 10 shows, there exist $\mathbb{H}_0$, $\mathtt{p}_0$ and $\mathtt{g}_0$ such that $\mathbb{H}_0 \subseteq \mathbb{S}.\mathbb{H}$ (this is (p1) in the proof of Lemma 10), $\Psi, \Delta \vdash \{(\mathtt{p}_0, \mathtt{g}_0)\}\,\mathtt{pc} : \mathbb{C}[\mathtt{pc}]$ ((p2.1) in Lemma 10) and $(\mathtt{p}_0 * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}|_{\mathbb{H}_0}$ ((p5.2) in Lemma 10).

To prove 1, by $\mathbb{C}(\mathtt{pc}) = \mathsf{j}\ \mathtt{f}$, $\Psi, \Delta \vdash \{(\mathtt{p}_0, \mathtt{g}_0)\}\,\mathtt{pc} : \mathbb{C}[\mathtt{pc}]$ and the J rule we know there exist p and g such that $(\mathtt{f}, (\mathtt{p}, \mathtt{g})) \in \Psi$, and $\mathtt{p}_0 \Rightarrow \mathtt{p}$. Since $(\mathtt{p}_0 * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}|_{\mathbb{H}_0}$, we know $(\mathtt{p} * \mathit{true})\ \mathbb{K}\ \mathbb{S}$ holds. The proof of 2 and 3 is similar and is elided here. □

5 Certifying Implementations of Synchronization Primitives

In this section, we show how to implement common synchronization primitives in AIM and certify them using our program logic.

5.1 Certifying Implementations of Locks

Threads use locks to achieve exclusive access to shared heaps. Following concurrent separation logic, each lock is used to protect a region of the heap (a.k.a. a sub-heap). Threads cannot access the sub-heap without first acquiring the corresponding lock. To be shared by multiple threads, the sub-heap must be "well-formed" when it is not exclusively owned by any threads (*i.e.,* the corresponding lock has not been acquired by any threads). The well-formedness can be viewed as the protocal between threads sharing resources.

$$(\mathit{LockID})\quad l \ ::= \ \mathtt{l}$$
$$(\mathit{LockSpec})\quad \Gamma \ ::= \ \{l \rightsquigarrow \mathtt{m}\}^*$$

We use memory pointers (label l) as lock IDs l. The pointer l points to a memory cell containing a binary flag that records whether the lock has been acquired (flag is 0) or not. The well-formedness of the sub-heap protected by a lock is specified using a heap predicate m. The specification Γ maps lock IDs to the corresponding heap predicates.

The heap used to implement locks and the heap protected by locks are shared by threads in the non-handler code, therefore they are part of the block C in Fig. 5. The

well-formedness of this part of heap is specified by $\mathsf{INV}(\Gamma)$ defined below. We require $\mathsf{INV}_s \Rightarrow \mathsf{INV}(\Gamma) * \mathit{true}$ (recall that INV_s is a shorthand for $\mathsf{INV}0 * \mathsf{INV}1$).

$$\mathsf{INV}(l, \mathtt{m}) \stackrel{\text{def}}{=} \exists \mathtt{w}.\ (l \mapsto \mathtt{w}) * ((\mathtt{w} = 0) \wedge \mathsf{emp} \vee (\mathtt{w} = 1) \wedge \mathtt{m}) \tag{7}$$

$$\mathsf{INV}(\Gamma) \stackrel{\text{def}}{=} \forall_* l \in \mathit{dom}(\Gamma).\ \mathsf{INV}(l, \Gamma(l)) \tag{8}$$

$\mathsf{INV}(l, \mathtt{m})$ says there is a binary flag stored at the location l. If the flag is 0, the lock has been acquired by some thread and the sub-heap protected by the lock is not available for sharing (specified by emp). Otherwise the lock is available and the corresponding sub-heap is also available and well-formed (specified by $\mathtt{m}$). $\forall_*$ is an indexed, finitely iterated separating conjunction, which is defined as:

$$\forall_* x \in S.\ P(x) \stackrel{\text{def}}{=} \begin{cases} \mathsf{emp} & \text{if } S = \varnothing \\ P(x_i) * (\forall_* x \in S'.\ P(x)) & \text{if } S = S' \uplus \{x_i\} \end{cases}$$

We first show two block-based implementations of locks, in which threads waiting for the availability of locks are put onto block queues in $\mathbb{B}$. We use the lock ID as the identifier of the corresponding block queue. We also show an implementation of spinlocks for uniprocessor systems.

The Hoare-style implementation In Hoare style, when a thread waiting for the lock is released from the block queue, it immediately owns the lock (and the resource protected by the lock). The acquire and release functions are implemented as `ACQ_H` and `REL_H` respectively in Fig. 22. Each function takes a lock ID as argument, which is passed from the caller through the register r_1.

Specifications are inserted into the code in Fig. 22 and are defined in Fig. 23. The precondition for `ACQ_H` is $(\mathtt{p}_{01}, \mathtt{g}_{01})$. The assertion $\mathtt{p}_{01}$ requires that r_1 contain a lock ID and that $\Delta(r_1) = \Gamma(r_1)$, *i.e.,* threads on the block queue $\mathbb{B}(r_1)$ are waiting for the well-formed resource protected by the lock r_1. The guarantee $\mathtt{g}_{01}$ shows that the function obtains the ownership of $\Gamma(r_1)$ when it returns. Here we use primed variables (*e.g.,* $\mathtt{ie}'$ and $\mathtt{is}'$) to refer to components in the return state, and use $\mathsf{trash}(\{r_2, r_3\})$ to mean that values of all registers other than r_2 and r_3 are preserved:

$$\mathsf{trash}(S) \stackrel{\text{def}}{=} \lambda \mathbb{S}, \mathbb{S}'.\ \forall r.\ r \notin S \rightarrow \mathbb{S}.\mathbb{R}(r) = \mathbb{S}'.\mathbb{R}(r)\,.$$

`ACQ_H` calls `ACQ_H_a` after it disables the interrupt. `ACQ_H_a` is specified by $(\mathtt{p}_{11}, \mathtt{g}_{11})$. Comparing $(\mathtt{p}_{01}, \mathtt{g}_{01})$ and $(\mathtt{p}_{11}, \mathtt{g}_{11})$, we can see that $(\mathtt{p}_{01}, \mathtt{g}_{01})$ hides INV_s and the implementation details of the lock (*e.g.,* the lock name l is a pointer pointing to a binary value) from the client code. We also show some intermediate specifications used during verification. Readers can also compare $\mathtt{p}_{12}$ and $\mathtt{p}_{13}$ and see how the BLK rule is applied.

The functions `REL_H` and `REL_H_a` are specified by $(\mathtt{p}_{21}, \mathtt{g}_{21})$ and $(\mathtt{p}_{31}, \mathtt{g}_{31})$, respectively. The precondition $\mathtt{p}_{21}$ requires that the releasing thread must own the resource $\Gamma(r_1)$ (thus it must be the owner of the lock r_1). The guarantee $\mathtt{g}_{21}$ shows $\Gamma(r_1)$ is released at the end. Depending on whether there are threads waiting for the lock, the current thread may either transfer the ownership of $\Gamma(r_1)$ to a waiting thread $(\mathtt{g}_a)$

```
;; acquire(l): $r1 contains l
ACQ_H:     -{(p01, g01)}
           cli
           call   ACQ_H_a
           sti
           ret
ACQ_H_a:   -{(p11, g11)}
           ld     $r2, 0($r1)          ;; $r2 <- [l]
           movi   $r3, 0
           beq    $r2, $r3, gowait     ;; ([l] == 0)?
           st     0($r1), $r3          ;; No (lock available):
           ret                         ;;   [l] <- 0
gowait:    -{(p12, g11)}               ;; Yes (unavailable):
           block  $r1                  ;;   block
           -{(p13, gid)}               ;;   get the lock after being released
           ret
;; release(l): $r1 contains l
REL_H:     -{(p21, g21)}
           cli
           call    REL_H_a
           sti
           ret
REL_H_a:   -{(p31, g31)}
           unblock $r1, $r2            ;; release a waiting thread if any
           -{(p32, g32)}
           movi    $r3, 0
           beq     $r2, $r3, rel_lock   ;; Is any thread released?
           ret                          ;; Yes: return
rel_lock:  -{(p33, g33)}
           movi    $r2, 1               ;; No (no waiting thread):
           st      0($r1), $r2          ;;   [l] <- 1
           -{(p34, gid)}
           ret
```

Fig. 22 Hoare-style implementation of locks

or simply set the lock to be available (g_b), as specified in g_{31}. Either way, the current thread loses the ownership of $\Gamma(r_1)$:

$$(\mathsf{INV}(\Gamma) * \Gamma(r_1)) \rhd (g_a \vee g_b) \Rightarrow \mathsf{INV}(\Gamma)$$

that is, with an initial state containing the resources $\mathsf{INV}(\Gamma)$ and $\Gamma(r_1)$, we can prove that the new state after the transition g_a or g_b has only $\mathsf{INV}(\Gamma)$. Like the specification (p_{01}, g_{01}) for ACQ_H, (p_{21}, g_{21}) here also hides the implementation details of the lock.

The Mesa-style implementation Figure 24 shows the Mesa-style implementation of locks. In the acquire function ACQ_M, the thread needs another round of loop to test the availability of the lock after it is released from the block queue—the lock is not immediately passed to it. The release function REL_M always sets the lock to be available. It does not pass the lock to the thread released from the block queue. Specifications for ACQ_M and REL_M are the same as Hoare style locks except that the

$$\mathrm{p}_0 \stackrel{\text{def}}{=} (\mathtt{is} = 0) \wedge \mathsf{enable}_{\mathsf{ret}} \wedge (r_1 \in \mathit{dom}(\Gamma)) \wedge (\Delta(r_1) = \Gamma(r_1))$$

$$\mathrm{p}_{01} \stackrel{\text{def}}{=} \mathrm{p}_0 \wedge (\mathtt{ie} = 1)$$

$$\mathrm{g}_{01} \stackrel{\text{def}}{=} \mathsf{Recv}(\Gamma(r_1)) \wedge (\mathtt{ie} = \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}') \wedge \mathsf{trash}(\{r_2, r_3\})$$

$$\mathrm{p}_{11} \stackrel{\text{def}}{=} \mathrm{p}_0 \wedge (\mathtt{ie} = 0) \wedge (\mathsf{INV}_s * \mathsf{true})$$

$$\mathrm{g}_{11} \stackrel{\text{def}}{=} (\mathsf{Presv}(\mathsf{INV}s) \circledast \mathsf{Recv}(\Gamma(r_1))) \wedge (\mathtt{ie} = \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}') \wedge \mathsf{trash}(\{r_2, r_3\})$$

$$\mathrm{p}_{12} \stackrel{\text{def}}{=} \mathrm{p}_0 \wedge (\mathtt{ie} = 0) \wedge ([r_1] = 0) \wedge (\mathsf{INV}_s * \mathsf{true})$$

$$\mathrm{p}_{13} \stackrel{\text{def}}{=} \mathrm{p}_0 \wedge (\mathtt{ie} = 0) \qquad \wedge (\mathsf{INV}_s * \mathsf{true} * \Gamma(r_1))$$

$$\mathrm{p}_{21} \stackrel{\text{def}}{=} \mathrm{p}_0 \wedge (\mathtt{ie} = 1) \wedge (\Gamma(r_1) * \mathsf{true})$$

$$\mathrm{g}_{21} \stackrel{\text{def}}{=} \mathsf{Send}(\Gamma(r_1)) \wedge (\mathtt{ie} = \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}') \wedge \mathsf{trash}(\{r_2, r_3\})$$

$$\mathrm{p}_{31} \stackrel{\text{def}}{=} \mathrm{p}_0 \wedge (\mathtt{ie} = 0) \wedge (\Gamma(r_1) * \mathsf{INV}_s * \mathsf{true})$$

$$\mathrm{g}_a \stackrel{\text{def}}{=} \mathsf{Send}(\Gamma(r_1)) \qquad \mathrm{g}_b \stackrel{\text{def}}{=} \left\{ \begin{array}{l} r_1 \mapsto \_ \\ r_1 \mapsto 1 \end{array} \right\}$$

$$\mathrm{g}_{31} \stackrel{\text{def}}{=} (\mathrm{g}_a \vee \mathrm{g}_b) \wedge (\mathtt{ie} = \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}') \wedge \mathsf{trash}(\{r_2, r_3\})$$

$$\mathrm{p}_{32} \stackrel{\text{def}}{=} \mathrm{p}_0 \wedge (\mathtt{ie} = 0) \wedge ((r_2 = 0) \wedge (\Gamma(r_1) * \mathsf{INV}_s * \mathsf{true}) \vee (r_2 \neq 0) \wedge (\mathsf{INV}_s * \mathsf{true}))$$

$$\mathrm{g}_{32} \stackrel{\text{def}}{=} ((r_2 = 0 \wedge \mathrm{g}_b) \vee (r_2 \neq 0 \wedge \mathsf{hid})) \wedge (\mathtt{ie} = \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}') \wedge \mathsf{trash}(\{r_2, r_3\})$$

$$\mathrm{p}_{33} \stackrel{\text{def}}{=} \mathrm{p}_0 \wedge (\mathtt{ie} = 0) \wedge (\Gamma(r_1) * \mathsf{INV}_s * \mathsf{true})$$

$$\mathrm{g}_{33} \stackrel{\text{def}}{=} \mathrm{g}_b \wedge (\mathtt{ie} = \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}') \wedge \mathsf{trash}(\{r_2\})$$

$$\mathrm{p}_{34} \stackrel{\text{def}}{=} \mathrm{p}_0 \wedge (\mathtt{ie} = 0) \wedge (\mathsf{INV}_s * \mathsf{true})$$

Fig. 23 Specifications of Hoare-style locks

assertion p_0, which is part of the preconditions, requires $\Delta(r_1) = \mathsf{emp}$. This implies the Mesa-style semantics of block and unblock: threads waiting on the block queue do not get any resource when they are released.

Spinlocks An implementation of spinlocks for uniprocessor systems and its specifications are shown in Fig. 25. The acquire function `ACQ_S` and the release function `REL_S` are specified by $(\mathrm{p}_{11}, \mathrm{g}_{11})$ and $(\mathrm{p}_{21}, \mathrm{g}_{21})$ respectively. The specifications look very similar to specifications for block-based implementations: `ACQ_S` gets the ownership of the extra resource $\Gamma(r_1)$ protected by the lock in r_1, while `REL_S` loses the ownership so that the client can no longer use the resource afterwards. The preconditions p_{11} and p_{21} also requires $r_1 \notin \mathit{dom}(\Delta)$, that is, the lock is not associated with a block queue. These specifications also hide the implementation details (the binary flag in heap) from the client code.

Preventing mismatches of acquire and release functions Each acquire function of locks should be paired with the release function of the same style. Mismatches of them would cause incorrect code. Our specifications for different styles effectively prevent the mismatches. We require $\Gamma(l) = \Delta(l)$ in Hoare-style and $\Delta(l) = \mathsf{emp}$ in

$$\mathtt{p}_0 \stackrel{\text{def}}{=} (\mathtt{is}=0)\wedge \mathsf{enable}_{\mathsf{ret}}\wedge(r_1\in dom(\Gamma))\wedge(\Delta(r_1)=\mathsf{emp})$$

$$\mathtt{p}_{11} \stackrel{\text{def}}{=} \mathtt{p}_0\wedge(\mathtt{ie}=1) \qquad \mathtt{p}_{12} \stackrel{\text{def}}{=} \mathtt{p}_{11}\wedge(r_3=0)$$

$$\mathtt{g}_{11} \stackrel{\text{def}}{=} \mathsf{Recv}(\Gamma(r_1))\wedge(\mathtt{ie}=\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2,r_3\})$$

$$\mathtt{p}_{13} \stackrel{\text{def}}{=} \mathtt{p}_0\wedge(\mathtt{ie}=0)\wedge(\mathsf{INV}_s * \mathsf{true})$$

$$\mathtt{g}_{13} \stackrel{\text{def}}{=} (\mathsf{Recv}(\Gamma(r_1))\circledast\mathsf{Send}(\mathsf{INV}_s))\wedge(\mathtt{ie}=1-\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2,r_3\})$$

$$\mathtt{p}_{14} \stackrel{\text{def}}{=} \mathtt{p}_0\wedge(\mathtt{ie}=0)\wedge(\Gamma(r_1)*\mathsf{INV}_s*\mathsf{true})$$

$$\mathtt{g}_{14} \stackrel{\text{def}}{=} \mathsf{Send}(\mathsf{INV}_s)\wedge(\mathtt{ie}=1-\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2,r_3\})$$

$$\mathtt{p}_{21} \stackrel{\text{def}}{=} \mathtt{p}_0\wedge(\mathtt{ie}=1)\wedge(\Gamma(r_1)*\mathsf{true})$$

$$\mathtt{g}_{21} \stackrel{\text{def}}{=} \mathsf{Send}(\Gamma(r_1))\wedge(\mathtt{ie}=\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2\})$$

$$\mathtt{p}_{22} \stackrel{\text{def}}{=} \mathtt{p}_0\wedge(\mathtt{ie}=0)\wedge(\Gamma(r_1)*\mathsf{INV}_s*\mathsf{true})$$

$$\mathtt{g}_{22} \stackrel{\text{def}}{=} \mathsf{Send}(\Gamma(r_1)*\mathsf{INV}_s)\wedge(\mathtt{ie}=1-\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2\})$$

```
;; acquire(l): $r1 contains l
ACQ_M:     -{(p11, g11)}
           movi    $r3, 0
acq_loop:  -{(p12, g11)}
           cli
           ld      $r2, 0($r1)          ;; $r2 <- [l]
           beq     $r2, $r3, gowait     ;; ([l] == 0)?
           st      0($r1), $r3          ;; No: [l] <- 0
           -{(p14, g14)}
           sti
           ret
gowait:    -{(p13, g13)}
           block   $r1                  ;; Yes: wait
           -{(p13, g13)}
           sti
           j       acq_loop             ;;      try again
;; release(l): $r1 contains l
REL_M:     -{(p21, g21)}
           cli
           -{(p22, g22)}
           unblock $r1, $r2             ;; release a waiting thread
           -{(p22, g22)}
           movi    $r2, 1
           st      0($r1), $r2          ;; [l] <- 1
           sti
           ret
```

Fig. 24 Mesa-style locks

Mesa style (a mismatch between Hoare-style and Mesa style is harmless if $\Gamma(l) = \mathsf{emp}$). A spinlock l is not in $dom(\Delta)$.

$$\mathtt{p} \stackrel{\text{def}}{=} (\mathtt{is} = 0) \wedge \mathsf{enable}_{\mathsf{ret}} \wedge (r_1 \in dom(\Gamma)) \wedge (r_1 \notin dom(\Delta))$$

$$\mathtt{p}_{11} \stackrel{\text{def}}{=} \mathtt{p} \wedge (\mathtt{ie} = 1) \qquad \mathtt{p}_{12} \stackrel{\text{def}}{=} \mathtt{p}_{11} \wedge (r_2 = 1)$$

$$\mathtt{g}_{11} \stackrel{\text{def}}{=} \mathsf{Recv}(\Gamma(r_1)) \wedge (\mathtt{ie} = \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}') \wedge \mathsf{trash}(\{r_2, r_3\})$$

$$\mathtt{p}_{13} \stackrel{\text{def}}{=} \mathtt{p} \wedge (\mathtt{ie} = 0) \wedge ((r_1 \mapsto 1) * \mathsf{true}) \wedge (\mathsf{INV}_s * \mathsf{true})$$

$$\mathtt{g}_{13} \stackrel{\text{def}}{=} (\mathsf{Recv}(\Gamma(r_1)) \circledast \mathsf{Send}(\mathsf{INV}_s)) \wedge (\mathtt{ie} = 1 - \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}') \wedge \mathsf{trash}(\{r_2\})$$

$$\mathtt{p}_{21} \stackrel{\text{def}}{=} \mathtt{p} \wedge (\mathtt{ie} = 1) \wedge (\Gamma(r_1) * \mathsf{true})$$

$$\mathtt{g}_{21} \stackrel{\text{def}}{=} \mathsf{Send}(\Gamma(r_1)) \wedge (\mathtt{ie} = \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}') \wedge \mathsf{trash}(\{r_2\})$$

```
;; acquire(l): $r1 contains l
ACQ_S:      -{(p11, g11)}
            movi $r2, 1
spin_loop:  -{(p12, g11)}
            cli
            ld   $r3, 0($r1)          ;; $r3 <- [l]
            beq  $r2, $r3, spin_set   ;; ([l] == 1)?
            sti                       ;; No: enable interrupt
            j    spin_loop            ;;     try again
spin_set:   -{(p13, g13)}
            movi $r2, 0               ;; Yes:
            st   0($r1), $r2                 [l] <- 0
            sti
            ret
;; release(l): $r1 contains l
REL_S:      -{(p21, g21)}
            movi $r2, 1
            cli
            st   0($r1), $r2          ;; [l] <- 1
            sti
            ret
```

Fig. 25 A spinlock

5.2 Certifying Implementations of Condition Variables

Now we show implementations of Hoare style [18], Brinch Hansen style [4], and Mesa style [24] condition variables. Condition variables are used together with locks to implement monitors. Each condition variable corresponds to certain condition over the shared resource protected by the corresponding lock. We use *cv* to represent identifiers of condition variables. Υ maps condition variables to the corresponding conditions $\mathtt{m}$.

$$(\mathit{CondVar})\ \ cv ::= n\ \ (\mathit{nat\ nums})$$

$$(\mathit{CVSpec})\ \ \Upsilon ::= \{cv \rightsquigarrow \mathtt{m}\}^*$$

In our implementation, we let *cv* be an identifier pointing to a block queue in $\mathbb{B}$. A lock *l* needs to be associated with *cv* to guarantee exclusive access of the shared

```
WAIT_H:     -{(p11, g11)}            ;; wait(l, cv)
            cli                      ;; $r1 contains l
            mov     $r4, $r2         ;; $r2 contains cv, save $r2 in $r4
            -{(p12, g12)}
            call    REL_H_a          ;; release the lock
            -{(p13, g13)}
            block   $r4              ;; wait for the condition to come true
            -{(p14, g14)}
            sti
            ret
SIGNAL_H:   -{(p21, g21)}            ;; signal(l, cv)
            cli
            -{(p22, g22)}            ;; $r2 contains cv
            unblock $r2, $r3         ;; release a waiting thread
            -{(p23, g23)}
            movi    $r4, 0
            beq     $r3, $r4, sig_done
            -{(p24, g24)}            ;; lock is passed to the released thread
            block   $r1              ;; wait for the lock ($r1 contains l)
sig_done:   -{(p25, g25)}
            sti
            ret

SIGNAL_BH:  -{(p21, g31)}            ;; signal(l, cv)
            cli
            -{(p22, g32)}
            unblock $r2, $r3         ;; $r2 contains cv
            -{(p23, g33)}
            movi    $r4, 0
            beq     $r3, $r4, sig_cont
            -{(p24, g34)}            ;; lock is passed to the released thread
            sti                      ;; do not wait for the lock again
            ret
sig_cont:   -{(p25, g35)}
            call    REL_H_a          ;; $r1 contains l
            -{(p34, g24)}
            sti
            ret
```

Fig. 26 Impl. of CV - Hoare style and Brinch Hansen style

resource. The difference between $\Gamma(l)$ and $\Upsilon(cv)$ is that $\Gamma(l)$ specifies the basic well-formedness of the resource (*e.g.,* a well-formed queue), while $\Upsilon(cv)$ specifies an extra condition (*e.g.,* the queue is not empty).

Hoare style and Brinch Hansen style implementations The implementations are shown in Fig. 26. The Hoare style is implemented by functions `WAIT_H` and

$$\mathsf{Cond}(l,cv) \stackrel{\text{def}}{=} \Gamma(l) \wedge (\Upsilon(cv) * \mathsf{true}) \qquad \overline{\mathsf{Cond}}(l,cv) \stackrel{\text{def}}{=} \Gamma(l) \wedge \neg(\Upsilon(cv) * \mathsf{true})$$

$$\mathrm{p}(l,cv) \stackrel{\text{def}}{=} (\mathtt{is}=0) \wedge \mathsf{enable}_{\mathsf{ret}} \wedge \exists \mathtt{m},\mathtt{m}'.\ (\Gamma(l)=\mathtt{m}) \wedge (\Delta(l)=\mathtt{m}) \wedge (\Upsilon(cv)=\mathtt{m}') \wedge (\Delta(cv)=\mathsf{Cond}(l,cv))$$

$$\mathrm{p}_{11} \stackrel{\text{def}}{=} \mathrm{p}(r_1,r_2) \wedge (\mathtt{ie}=1) \wedge (\overline{\mathsf{Cond}}(r_1,r_2) * \mathsf{true})$$

$$\mathrm{g}_{11} \stackrel{\text{def}}{=} (\mathsf{Send}(\overline{\mathsf{Cond}}(r_1,r_2)) \circledast \mathsf{Recv}(\mathsf{Cond}(r_1,r_2))) \wedge (\mathtt{ie}=\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

$$\mathrm{p}_{12} \stackrel{\text{def}}{=} \mathrm{p}(r_1,r_4) \wedge (\mathtt{ie}=0) \wedge (\overline{\mathsf{Cond}}(r_1,r_2) * \mathsf{INV}_s * \mathsf{true})$$

$$\mathrm{g}_{12} \stackrel{\text{def}}{=} (\mathsf{Send}(\overline{\mathsf{Cond}}(r_1,r_2) * \mathsf{INV}_s) \circledast \mathsf{Recv}(\mathsf{Cond}(r_1,r_2))) \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3\})$$

$$\mathrm{p}_{13} \stackrel{\text{def}}{=} \mathrm{p}(r_1,r_4) \wedge (\mathtt{ie}=0) \wedge (\mathsf{INV}_s * \mathsf{true})$$

$$\mathrm{g}_{13} \stackrel{\text{def}}{=} (\mathsf{Recv}(\mathsf{Cond}(r_1,r_4)) \circledast \mathsf{Send}(\mathsf{INV}_s)) \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\emptyset)$$

$$\mathrm{p}_{14} \stackrel{\text{def}}{=} \mathrm{p}(r_1,r_4) \wedge (\mathtt{ie}=0) \wedge (\mathsf{Cond}(r_1,r_4) * \mathsf{INV}_s * \mathsf{true})$$

$$\mathrm{g}_{14} \stackrel{\text{def}}{=} \mathsf{Send}(\mathsf{INV}_s) \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\emptyset)$$

$$\mathrm{p}_{21} \stackrel{\text{def}}{=} \mathrm{p}(r_1,r_2) \wedge (\mathtt{ie}=1) \wedge (\mathsf{Cond}(r_1,r_2) * \mathsf{true})$$

$$\mathrm{g}_{21} \stackrel{\text{def}}{=} (\mathsf{Send}(\mathsf{Cond}(r_1,r_2)) \circledast \mathsf{Recv}(\Gamma(r_1))) \wedge (\mathtt{ie}=\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_3,r_4\})$$

$$\mathrm{p}_{22} \stackrel{\text{def}}{=} \mathrm{p}(r_1,r_2) \wedge (\mathtt{ie}=0) \wedge (\mathsf{Cond}(r_1,r_2) * \mathsf{INV}_s * \mathsf{true})$$

$$\mathrm{g}_a \stackrel{\text{def}}{=} \mathsf{Send}(\mathsf{Cond}(r_1,r_2) * \mathsf{INV}_s) \circledast \mathsf{Recv}(\Gamma(r_1)) \qquad \mathrm{g}_b \stackrel{\text{def}}{=} \mathsf{Recv}(\Gamma(r_1)) \circledast \mathsf{Send}(\mathsf{INV}_s)$$

$$\mathrm{g}_{22} \stackrel{\text{def}}{=} \mathrm{g}_a \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_3,r_4\})$$

$$\mathrm{p}_{23} \stackrel{\text{def}}{=} \mathrm{p}(r_1,r_2) \wedge (\mathtt{ie}=0) \wedge ((r_3=0) \wedge (\mathsf{Cond}(r_1,r_2) * \mathsf{INV}_s * \mathsf{true}) \vee (r_3 \neq 0) \wedge (\mathsf{INV}_s * \mathsf{true}))$$

$$\mathrm{g}_{23} \stackrel{\text{def}}{=} (r_3=0 \wedge \mathrm{g}_a \vee r_3 \neq 0 \wedge \mathrm{g}_b) \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_4\})$$

$$\mathrm{p}_{24} \stackrel{\text{def}}{=} \mathrm{p}(r_1,r_2) \wedge (\mathtt{ie}=0) \wedge (\mathsf{INV}_s * \mathsf{true}))$$

$$\mathrm{g}_{24} \stackrel{\text{def}}{=} \mathrm{g}_b \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\emptyset)$$

$$\mathrm{p}_{25} \stackrel{\text{def}}{=} \mathrm{p}(r_1,r_2) \wedge (\mathtt{ie}=0) \wedge (\mathsf{INV}_s * \Gamma(r_1) * \mathsf{true})$$

$$\mathrm{g}_{25} \stackrel{\text{def}}{=} \mathsf{Send}(\mathsf{INV}_s) \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\emptyset)$$

$$\mathrm{g}_{31} \stackrel{\text{def}}{=} \mathsf{Send}(\mathsf{Cond}(r_1,r_2)) \wedge (\mathtt{ie}=\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

$$\mathrm{g}_a \stackrel{\text{def}}{=} \mathsf{Send}(\mathsf{Cond}(r_1,r_2) * \mathsf{INV}_s) \qquad \mathrm{g}_b \stackrel{\text{def}}{=} \mathsf{Send}(\mathsf{INV}s)$$

$$\mathrm{g}_{32} \stackrel{\text{def}}{=} \mathrm{g}_a \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

$$\mathrm{g}_{33} \stackrel{\text{def}}{=} (r_3=0 \wedge \mathrm{g}_a \vee r_3 \neq 0 \wedge \mathrm{g}_b) \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

$$\mathrm{p}_{34} \stackrel{\text{def}}{=} \mathsf{enable}_{\mathsf{ret}} \wedge (\mathtt{is}=0) \wedge (\mathtt{ie}=0) \wedge (\mathsf{INV}_s * \mathsf{true}))$$

$$\mathrm{g}_{34} \stackrel{\text{def}}{=} \mathrm{g}_b \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\emptyset)$$

$$\mathrm{g}_{35} \stackrel{\text{def}}{=} \mathrm{g}_a \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3\})$$

Fig. 27 Spec. of CV - Hoare style and Brinch Hansen style

SIGNAL_H. The wait function for the Brinch Hansen style is the same as WAIT_H. The signal function is shown as SIGNAL_BH. All three functions take two arguments: a lock ID associated with the condition variable and the condition variable itself. They

are passed through registers r_1 and r_2 respectively. Here we use the Hoare-style locks shown in Fig. 22.

Specifications of the functions are defined in Fig. 27. `WAIT_H` is specified by $(\mathtt{p}_{11}, \mathtt{g}_{11})$. As $\mathtt{p}_{11}$ shows, r_1 contains a Hoare-style lock in the sense that $\Delta(r_1) = \Gamma(r_1)$. The register r_2 contains the condition variable with specification $\Upsilon(r_2)$. For Hoare style, we require $\Delta(r_2) = \mathsf{Cond}(r_1, r_2)$ (defined as $\Gamma(r_1) \wedge (\Upsilon(r_2) * \mathit{true})$ in Fig. 27). Therefore, when the blocked thread is released, it gets the resource $(\Gamma(r_1))$ protected by the lock with the extra knowledge $(\Upsilon(r_2) * \mathit{true})$ that the condition associated with the condition variable holds. Here the condition $\Upsilon(r_2)$ does not have to specify the whole resource protected by the lock, therefore we use $\Upsilon(r_2) * \mathit{true}$. Before calling `WAIT_H`, $\mathtt{p}_{11}$ requires that the lock must have been acquired, thus we have the ownership $\Gamma(r_1)$. The condition $\Upsilon(r_2)$ needs to be false (as required in $\overline{\mathsf{Cond}}(r_1, r_2)$). It is not an essential requirement, but we use it to prevent waiting without testing the condition. The guarantee $\mathtt{g}_{11}$ says that, when `WAIT_H` returns, the current thread still owns the lock (and $\Gamma(r_1)$) and it also knows the condition specified in Υ holds. `SIGNAL_H` is specified by $(\mathtt{p}_{21}, \mathtt{g}_{21})$. It requires that the thread own the lock and that the condition $\Upsilon(r_2)$ hold at the beginning. When it releases a thread waiting for the condition, it passes the ownership of the lock and the knowledge that the condition holds to the released thread. Then it blocks itself to wait for the ownership of the lock. When it returns, the thread still owns the lock, but the condition may no longer hold. Intermediate specifications are inserted into the code to show the proof sketch. We do not explain the details here.

The Brinch Hansen style signal function `SIGNAL_BH` is specified by $(\mathtt{p}_{21}, \mathtt{g}_{31})$ defined in Fig. 27. The precondition is the same as `SIGNAL_H`. We simply reuse $\mathtt{p}_{21}$ as the precondition. The definition of $\mathtt{g}_{31}$ shows the difference between Hoare style and Brinch Hansen style: the thread no longer owns the lock when `SIGNAL_BH` returns. Therefore, calling the signal function must be the last command in the critical region.

Mesa style Figure 28 shows the Mesa-style condition variables. `WAIT_M` is specified by $(\mathtt{p}_{11}, \mathtt{g}_{11})$. Similar to the Hoare style wait function, the precondition $\mathtt{p}_{11}$ also requires that the thread owns the lock and that the condition is false. The difference is that we require $\Delta(cv) = \mathsf{emp}$, where r_2 contains the condition variable. Therefore, as $\mathtt{g}_{11}$ shows, the current thread has no idea about the validity of the condition when it returns. Requiring $\Delta(cv) = \mathsf{emp}$ also prevents the mismatch between the Hoare style (Brinch Hansen style) primitives and the Mesa style primitives.

`SIGNAL_M` is specified by $(\mathtt{p}_{21}, \mathtt{g}_{21})$. Unlike `SIGNAL_H`, it does not take the lock as argument. The current thread does not need to own the lock to call `SIGNAL_M`. It simply wakes up a waiting thread without passing the ownership of the lock and the validity of the condition. From $\mathtt{g}_{21}$ we can see that, if we hide the details of releasing a blocked thread, the signal function in Mesa style is just like a skip command that has no effects over states.

6 Certifying X86 Primitives

The program logic presented in this paper has been adapted for the 16-bit, real-mode x86 architecture. We have formalized a subset of the x86 assembly language, its operational semantics, and the program logic in the Coq proof assistant [6].

$$\mathrm{p}(l,cv) \stackrel{\text{def}}{=} (\mathtt{is}=0)\wedge \mathsf{enable}_{\mathsf{ret}}\wedge\exists \mathtt{m},\mathtt{m}'.\ (\Gamma(l)=\mathtt{m})\wedge(\Delta(l)=\mathtt{m})\wedge(\Upsilon(cv)=\mathtt{m}')\wedge(\Delta(cv)=\mathsf{emp})$$

$$\mathrm{p}_{11} \stackrel{\text{def}}{=} \mathrm{p}(r_1,r_2)\wedge(\mathtt{ie}=1)\wedge(\overline{\mathsf{Cond}}(r_1,r_2)*\mathsf{true})$$

$$\mathrm{g}_{11} \stackrel{\text{def}}{=} (\mathsf{Send}(\overline{\mathsf{Cond}}(r_1,r_2))\circledast\mathsf{Recv}(\Gamma(r_1)))\wedge(\mathtt{ie}=\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2,r_3,r_4\})$$

$$\mathrm{p}_{12} \stackrel{\text{def}}{=} \mathrm{p}(r_1,r_4)\wedge(\mathtt{ie}=0)\wedge(\overline{\mathsf{Cond}}(r_1,r_2)*\mathsf{INV}_s*\mathsf{true})$$

$$\mathrm{g}_{12} \stackrel{\text{def}}{=} (\mathsf{Send}(\overline{\mathsf{Cond}}(r_1,r_2)*\mathsf{INV}_s)\circledast\mathsf{Recv}(\Gamma(r_1)))\wedge(\mathtt{ie}=1-\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2,r_3,r_4\})$$

$$\mathrm{p}_{13} \stackrel{\text{def}}{=} \mathrm{p}(r_1,r_4)\wedge(\mathtt{ie}=0)\wedge(\mathsf{INV}_s*\mathsf{true})$$

$$\mathrm{g}_{13} \stackrel{\text{def}}{=} (\mathsf{Send}(\mathsf{INV}_s)\circledast\mathsf{Recv}(\Gamma(r_1)))\wedge(\mathtt{ie}=1-\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2,r_3\})$$

$$\mathrm{p}_{14} \stackrel{\text{def}}{=} \mathrm{p}(r_1,r_4)\wedge(\mathtt{ie}=0)$$

$$\mathrm{g}_{14} \stackrel{\text{def}}{=} \mathsf{Recv}(\Gamma(r_1))\wedge(\mathtt{ie}=\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2,r_3\})$$

$$\mathrm{p}_{15} \stackrel{\text{def}}{=} \mathrm{p}(r_1,r_4)\wedge(\mathtt{ie}=0)\wedge(\Gamma(r_1)*\mathsf{true})$$

$$\mathrm{p}'(cv) \stackrel{\text{def}}{=} (\mathtt{is}=0)\wedge\exists\mathtt{m}.\ (\Upsilon(cv)=\mathtt{m})\wedge(\Delta(cv)=\mathsf{emp})$$

$$\mathrm{p}_{21} \stackrel{\text{def}}{=} \mathrm{p}'(r_1)\wedge(\mathtt{ie}=1)$$

$$\mathrm{g}_{21} \stackrel{\text{def}}{=} \mathsf{hid}\wedge(\mathtt{ie}=\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2\})$$

$$\mathrm{p}_{22} \stackrel{\text{def}}{=} \mathrm{p}'(r_1)\wedge(\mathtt{ie}=0)\wedge(\mathsf{INV}_s*\mathsf{true})$$

$$\mathrm{g}_{21} \stackrel{\text{def}}{=} \mathsf{Send}(\mathsf{INV}_s)\wedge(\mathtt{ie}=1-\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2\})$$

```
WAIT_M:   -{(p11, g11)}            ;; wait(l, cv)
          cli                      ;; $r1 contains l
          mov     $r4, $r2         ;; $r2 contains cv, save $r2 in $r4
          -{(p12, g12)}
          call    REL_H_a          ;; release lock
          -{(p13, g13)}
          block   $r4              ;; sleep for a while
          -{(p13, g13)}
          sti
          -{(p14, g14)}
          call    ACQ_H            ;; try to acquire the lock again
          -{(p15, gid)}
          ret
SIGNAL_M: -{(p21, g21)}            ;; signal(cv)
          cli                      ;; $r1 contains cv
          -{(p22, g22)}
          unblock $r1, $r2         ;; release a thread, no resource transfer
          -{(p22, g22)}
          sti
          ret
```

Fig. 28 Impl. and spec. of CV - Mesa style

In our implementation, we assume that all interrupts except the timer have been masked. The three abstract instructions switch, block and unblock are replaced with function calls to the concrete implementations of primitives `scheduler`, `blk` and

Component	# of lines
Basic Utility Definitions & Lemmas	2,766
Machine, Opr. Semantics & Lemmas	3,269
Separation Logic, Lemmas & Tactics	6,340
OCAP-x86 & Soundness	1,711
SCAP-x86 & Soundness	1,357
Thread Queues & Lemmas	1,199
AIM-Logic & Soundness	26,347

Component	# of lines
Assembly Code at Level S	292*
enQueue/deQueue	4,838
scheduler, block, unblock	7,107
Assembly Code at Level C	411*
Timer Handler	2,344
`yield` & `sleep`	7,364
locks: `acq_h` & `rel_h`	10,838
cond. var.: `wait_m` & `signal_m`	5,440

* They are the Coq source files containing the encoding of the assembly code. The real assembly code in these two files is around 300 lines.

Fig. 29 The verified package in Coq

`unblk` at Level S in Fig. 3. The soundness of the program logic is proved in the OCAP-x86 framework, which adapts the foundational OCAP framework [9] for x86. The inference rules of our program logic are proved as lemmas in OCAP-x86. The soundness of OCAP-x86 itself is then proved following the syntactic approach [38]. The proofs are also formalized in Coq and are machine-checkable.

Our preemptive thread libraries (shown in Fig. 3) are also implemented in the x86 assembly code and run in real-mode. Primitives at Level C and Level S are certified using different program logics. Synchronization primitives at Level C are certified using the AIM program logic. The timer interrupt handler calls the scheduler implemented at the low-level, which corresponds to the switch instruction in AIM. The yield function simply wraps the scheduler by disabling the interrupt at the beginning and enabling it at the end. They are also certified using this logic. Thread primitives at Level S are executed with interrupts disabled. They are certified as sequential code using SCAP-x86, an adaptation of the SCAP logic [13] for x86, which is like a subset of our program logic for AIM without rules for cli, sti, iret, switch, block and unblock. We also link Level C programs with Level S programs to get a fully certified package. The linking is done in the OCAP-x86 framework. It is based on the observation that code at Level S only accesses thread queues and it does not touch the sub-heap accessed by threads at higher level. On the other hand, code at Level C does not touch the thread queues unless it calls Level S primitives. Therefore, the certified code at one abstraction level can work together with code at the other level, knowing that its program invariant would be preserved. More details about the methodology of using "domain-specific" program logics to certify modules at different abstraction levels and then linking them in a foundational logical framework are presented in one of our recent papers [12]. The following implementation details are also taken from that paper.

The whole Coq implementation has around 82,000 lines, including 1165 definitions and 1848 lemmas and theorems. The package is available online.[2] Figure 29 gives a break down of the number of lines for various components. The implementation has taken many man-months, including the implementation of basic facilities such as lemmas and tactics for partial mappings, queues, and separation logic assertions. One of the lessons we learn is that writing proofs is very similar to developing large-scale

[2] http://flint.cs.yale.edu/flint/publications/aim.coq.tar.gz.

software systems—many software-engineering principles would be equally helpful for proof development; especially a careful design and a proper infrastructure is crucial to the success. We started to prove the main theorems without first developing a proper set of lemmas and tactics. The proofs done at the beginning used only the most primitive tactics in Coq. Auxiliary lemmas were proved on the fly and some were proved multiple times by different team members. The early stage was very painful and some of our members were so frustrated by the daunting amount of work. After one of us ported a set of lemmas and tactics for separation logic from a different project [25], we were surprised to see how the proof was expedited. The tactics manipulating separation logic assertions, especially the ones that reorder sub-terms of separating conjunctions, have greatly simplified the reasoning about memory.

Another observation is that certifying both the library (*e.g.,* Level S primitives) and its clients (*e.g.,* Level C code) is an effective way to validate specifications. We have found some of our initial specifications for Level S code are too strong or too weak to be used by Level C. Also, to avoid redoing the whole proof after fixing the specifications, it is helpful to decompose big lemmas into smaller ones with clear interfaces.

The size of our proof scripts is huge, comparing with the size of the assembly code. This is caused in part by the duplicate proof of the same lemmas by different team members. Another reason is the lack of proper design and abstraction: when an instruction is seen a second time in the code, we simply copy and paste the previous proof and do some minor changes. The proof is actually developed very quickly after introducing the tactics for separation logic. For instance, the 5440 lines Coq code certifying condition variables is done by one of the authors in two days. We believe the size of proof scripts can be greatly reduced with more careful abstractions and more reuse of lemmas.

We implement the primitives in 16-bit real-mode x86 mainly for its simplicity, which allows us to quickly adapt our program logic to this hardware-implemented instruction set architecture (ISA). Although this ISA is rarely used in real-world systems these days, the complexities of more popular architectures (*e.g.,* the protected mode x86) are mostly orthogonal to the technical problems addressed in this paper. Therefore our verification still gives us confidence on the expressiveness and the applicability of the logic. Also, our logic does not prevent the use of automated verification techniques, although we do the proof manually in Coq. Our inference rules for instruction sequences shown in Fig. 16 are syntax directed, which can be easily integrated in an algorithmic process to automatically generate verification conditions. Recent efforts on SMT solvers [27] and theorem provers [1] also enable us to discharge the verification conditions automatically. We would like to explore these possibilities in our future work.

7 Related and Future Work

7.1 Reasoning about Interrupts

Regehr and Cooprider [33] also observed that programs with interrupts cannot be directly viewed as a special class of multi-threaded programs. They proposed a non-trivial translation to convert interrupt-driven programs to thread-based programs.

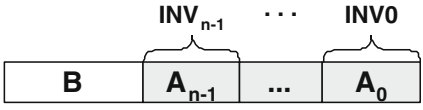
However, their approach cannot be directly applied for our goal to build certified OS kernels. First, proof of the correctness of the translation is non-trivial and has not been formalized. As Regehr and Cooprider pointed out, the proof requires formal semantics of interrupts. Our work actually provides such semantics. Second, their translation requires higher-level language constructs such as locks. Our AIM is at a lower abstraction level and does not have built-in locks. As we have shown, locks can be implemented in AIM and certified using our program logic.

Suenaga and Kobayashi [35] presented a type system to guarantee deadlock-freedom in a concurrent calculus with interrupts. Their calculus is an ML-style language with built-in support of threads, locks and multi-level interrupts. Our AIM is at a lower abstraction level than theirs in that context switching is explicitly done by the programmer in threads or interrupt handlers. Also, AIM does not have built-in locks. Their type system is designed mainly for preventing deadlocks with automatic type inference, while our program logic supports verification of general safety properties, including partial correctness.

Palsberg and Ma [31] proposed a calculus of interrupt driven systems, which has multi-level interrupts but no threads. Instead of a general program logic like ours, they proposed a type system to guarantee an upper bound of stack spaces needed by interrupts. DeLine and Fähndrich [7] showed how to enforce protocols with regard to interrupts levels as an application of Vault's type system, but it is unclear how to use the type system to verify general properties of interrupts.

In AIM, we only support one interrupt in the system, which cannot be interrupted again. It is actually easy to extend the machine to support multi-level interrupts: we change the `is` bit into a vector of bits ivec corresponding to interrupts in service. An interrupt can only be interrupted by other interrupts with higher priorities, which can also be disabled by clearing the `ie` bit. At the end of each interrupt handler, the corresponding in-service bit will be cleared so that interrupts at the same or lower level can be served.

Extension of the program logic to support multi-level interrupts is also straightforward, following the same idea of memory partitions. Suppose there are n interrupts in the system, the memory will be partitioned into $n+1$ blocks, as shown below:



where block A_k will be used by the interrupt handler k. To take care of the preemption relations with multiple handlers, we need to change our definition of Inv(`ie`, `is`) in Fig. 19 into Inv(`ie`, ivec), which models the switch of memory ownership at the points of cli, sti and boundaries of interrupt handlers.

Another simplification in our work is the assumption of a global interrupt handler entry. It is easy to extend our machine and program logic to support run-time installation of interrupt handlers. In our machine, we can add a special register and an "install" instruction to update this register. When interrupt comes, we look up the entry point from this register. This extension has almost no effects over our program logic, thanks to our support of modular reasoning. We only need to add a command rule for the "install" instruction to enforce that the new handler's interface is compatible to the specification (p_i, g_i).

7.2 Verification of OS Kernels and Thread Implementations

In his pioneer work, Bevier [2] showed how to formally certify Kit, an OS kernel implemented in machine code. The kernel supports hardware interrupts. However, interrupt handlers are part of the kernel code, and the kernel code is sequential and cannot be interrupted. Gargano et al. [14] showed a framework to construct a certified OS kernel in the Verisoft project. Similar to our layering of system code shown in Fig. 3, they split the code into two levels: abstract communicating virtual machines (CVM) and a concrete kernel. Like Kit, their kernel is also sequential. User processes (virtual machines) can be interrupted, but they run in different virtual address spaces and do not share memory. Ni et al. [28] certified a non-preemptive thread implementation. Their code is purely sequential and they did not support interrupts and preemption. In all these cases, the certified code is like our code at Level S (but with other features that are not supported here, such as memory management [14]), while we try to certify code at Level C, which involves both hardware interrupts and preemptive concurrency. We also certified code at Level S and linked the two levels to get a fully certified package.

Like Kit [2] and the Verisoft project [14], we only have fixed number of threads in the AIM machine. In our previous work [10], we have shown how to support dynamic thread creation following a similar technique to support dynamic memory allocation in type systems. The technique is fairly orthogonal and can be easily incorporated into this work. Another missing feature is dynamic creation of locks and block queues. Gotsman et al. [15] and Hobor et al. [19] extended concurrent separation logic with dynamic creation of storable locks. Their techniques might be applied here as well to support dynamic block queues.

It is also interesting to extend our logic to support multi-processor machines in the future. The general idea of memory partitions and ownership transfers used here would still apply in a multi-processor setting, except that we need to know which interrupt interrupts which processor. The implementation of kernel-level threads at the Level S in Fig. 3 becomes more complicated because it is no longer sequential, but it still prohibits interrupts at this level and can be certified based on existing work on concurrency verification. Disabling interrupts plays a less important role to bootstrap the implementation of synchronization primitives. To implement spinlocks, we need to use atomic instructions provided by the hardware, *e.g.,* the compare and swap instruction (cas). Also, we would like to see how relaxed memory models affect the reasoning about concurrent programs.

7.3 Concurrency Verification

O'Hearn proposed concurrent separation logic (CSL), which applies separation logic to certify concurrent programs [29]. Brookes [5] gave a trace semantics to CSL. The basic idea of CSL is to ensure that resources accessible by different concurrent entities are disjoint with each other. Accessing shared resources is protected by critical regions. The semantics of entering and exiting critical regions is modeled as resource ownership transfers. The concurrent programming language supported by CSL is a higher-level languages with built-in critical regions and implicit thread context switching. It does not have interrupts. The development of our program logic is inspired by CSL. We assign ownership-transfer semantics to cli, sti and low-level

thread primitives that are not supported in CSL. Similar to CSL, certifying threads and interrupt handlers in our logic is almost the same as certifying sequential programs in the sequential separation logic. We even unify the reasoning of concurrent primitives (cli, sti and thread primitives) with normal sequential instructions in the SEQ rule.

Rely-Guarantee reasoning by Jones [22] is another well-studied methodology to certify concurrent programs. The basic idea is to let each thread to specify its expectations over its environment (the rely condition) and its guarantees to its environment (the guarantee condition). The behavior of a certified thread fulfills its guarantee if the behavior of the environment satisfies its rely condition. There is no interference between threads as long as each thread's rely condition is implied by all other's guarantees. Rely-Guarantee reasoning is more expressive than CSL in that it uses actions (like our g) to specify transitions of shared resources, while CSL uses program invariant (like our INV0 and INV1) as specifications. On the other hand, CSL is more modular than Rely-Guarantee reasoning because of the support of local reasoning. Recent efforts [8, 37] have tried to combine the merits of both sides.

Like CSL, we only use invariants (*e.g.,* INV0 and INV1) to specify shared resources, which cannot be accessed by threads unless interrupts are disabled. This restricts the support of fine-grained concurrency. To lift the restriction, it is possible to use the more expressive rely-guarantee style specifications, following the approaches developed in recent work [8, 37]. Atomic operations can directly access shared resources even if interrupts are enabled, as long as the transitions satisfy the rely/guarantee conditions. Bornat et al. [3] proposed refinements of separation logic assertions to distinguish read-only accesses and read/write accesses of heap. The refinements can be incorporated in our program logic to support verification of reader/writer locks. Another limitation of our logic is that it only supports specifications of safety properties (including partial correctness). We would like to extend it to reason about liveness properties in our future work.

8 Conclusion

In this paper we present a new Hoare-style framework for certifying low-level programs involving both interrupts and concurrency. Following separation logic, we formalize the interaction among threads and interrupt handlers in terms of memory ownership transfers. Instead of using the operational semantics of cli, sti and thread primitives, our program logic formulates their local effects over the current thread, as shown in Fig. 18, which is the key for our logic to achieve modular verification. We have also certified various lock and condition-variable primitives; our specifications are both abstract (hiding implementation details) and precise (capturing the semantic difference among these variations).

Practitioners doing informal proofs can also benefit from our logic by learning how to do informal reasoning in a systematic way for general concurrent programs, whose correctness is usually not obvious. Although the primitives shown in this paper are similar to standard routines in many OS textbooks, we are not aware of any (even informal) proofs for code that involves both hardware interrupts and preemptive concurrency. Saying that the code should work is one thing (it often still requires leap-of-faith in our experience)—knowing why it works (which this paper does) is

another thing. The idea of memory partitions and ownership transfers shown in this paper (and inspired by separation logic) gives general guidelines even for informal proofs.

Acknowledgements We thank anonymous referees for suggestions and comments on an earlier version of this paper. Wei Wang, Haibo Wang, and Xi Wang helped prove some of the lemmas in our Coq implementation. Xinyu Feng and Zhong Shao are supported in part by gift from Microsoft and NSF grants CCR-0524545, CCR-0716540, and CCR-0811665. Yu Guo is supported in part by grants from National Natural Science Foundation of China (under grants No. 60673126 and No. 90718026), China Postdoctoral Science Foundation (No. 20080430770) and Natural Science Foundation of Jiangsu Province, China (No. BK2008181). Yuan Dong is supported in part by National Natural Science Foundation of China (under grants No. 60573017 and No. 90818019), Hi-Tech Research and Development Program Of China (under grant No. 2008AA01Z102), China Scholarship Council, and Basic Research Foundation of Tsinghua National Laboratory for Information Science and Technology (TNList). Any opinions, findings, and contributions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: modular automatic assertion checking with separation logic. In: Proc. 4th International Symposium on Formal Methods for Components and Objects (FMCO'05). LNCS, vol. 4111, pp. 115–137. Springer, New York (2005)
2. Bevier, W.R.: Kit: a study in operating system verification. IEEE Trans. Softw. Eng. **15**(11), 1382–1396 (1989)
3. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: Proc. 32nd ACM Symp. on Principles of Prog. Lang., pp. 259–270. ACM, New York (2005)
4. Brinch Hansen, P.: The programming language concurrent pascal. IEEE Trans. Software Eng. **1**(2), 199–207 (1975)
5. Brookes, S.: A semantics for concurrent separation logic. In: Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04). LNCS, vol. 3170, pp. 16–34. Springer, New York (2004)
6. Coq Development Team: The Coq proof assistant reference manual. The Coq release v8.1 (2006)
7. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: Proc. 2001 ACM Conf. on Prog. Lang. Design and Impl., pp. 59–69. ACM, New York (2001)
8. Feng, X., Ferreira, R., Shao, Z.: On the relationship between concurrent separation logic and assume-guarantee reasoning. In: Proc. 16th European Symp. on Prog. (ESOP'07). LNCS, vol. 4421, pp. 173–188. Springer, New York (2007)
9. Feng, X., Ni, Z., Shao, Z., Guo, Y.: An open framework for foundational proof-carrying code. In: Proc. 2007 ACM Workshop on Types in Lang. Design and Impl., pp. 67–78. ACM, New York (2007)
10. Feng, X., Shao, Z.: Modular verification of concurrent assembly code with dynamic thread creation and termination. In: Proc. 2005 ACM Int'l Conf. on Functional Prog., pp. 254–267. ACM, New York (2005)
11. Feng, X., Shao, Z.: Local reasoning and information hiding in SCAP. Tech. rep. YALEU/DCS/TR-1398, Dept. of Computer Science, Yale University, New Haven, CT (2008). http://flint.cs.yale.edu/publications/SCAPFrame.html
12. Feng, X., Shao, Z., Guo, Y., Dong, Y.: Combining domain-specific and foundational logics to verify complete software systems. In: Proc. Second IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08). LNCS, vol. 5295, pp. 54–69. Springer, New York (2008)
13. Feng, X., Shao, Z., Vaynberg, A., Xiang, S., Ni, Z.: Modular verification of assembly code with stack-based control abstractions. In: Proc. 2006 ACM Conf. on Prog. Lang. Design and Impl., pp. 401–414. ACM, New York (2006)
14. Gargano, M., Hillebrand, M.A., Leinenbach, D., Paul, W.J.: On the correctness of operating system kernels. In: Proc. 18th Int'l Conf. on Theorem Proving in Higher Order Logics. LNCS, vol. 3603, pp. 1–16. Springer, New York (2005)

15. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local reasoning for storable locks and threads. In: Proc. Fifth ASIAN Symp. on Prog. Lang. and Sys. (APLAS'07). LNCS, vol. 4807, pp. 19–37. Springer, New York (2007)
16. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **26**(1), 53–56 (1969)
17. Hoare, C.A.R.: Towards a theory of parallel programming. In: Operating Systems Techniques, pp. 61–71. Academic, London (1972)
18. Hoare, C.A.R.: Monitors: An operating system structuring concept. Commun. ACM **17**(10), 549–557 (1974)
19. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: Proc. 17th European Symp. on Prog. (ESOP'08). LNCS, vol. 4960, pp. 353–367. Springer, New York (2008)
20. Hunt, G.C., Larus, J.R.: Singularity design motivation. Tech. rep. MSR-TR-2004-105, Microsoft Corporation (2004)
21. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: Proc. 28th ACM Symp. on Principles of Prog. Lang., pp. 14–26. ACM, New York (2001)
22. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983)
23. Kleymann, T.: Metatheory of verification calculi in LEGO—to what extent does syntax matter? In: Proc. International Workshop on Types for Proofs and Programs (TYPES'98). LNCS, vol. 1657, pp. 133–148. Springer, New York (1998)
24. Lampson, B.W., Redell, D.D.: Experience with processes and monitors in Mesa. Commun. ACM **23**(2), 105–117 (1980)
25. McCreight, A., Shao, Z., Lin, C., Li, L.: A general framework for certifying garbage collectors and their mutators. In: Proc. 2007 ACM Conf. on Prog. Lang. Design and Impl., pp. 468–479. ACM, New York (2007)
26. Morrisett, G., Walker, D., Crary, K., Glew, N.: From system F to typed assembly language. In: Proc. 25th ACM Symp. on Principles of Prog. Lang., pp. 85–97. ACM, New York (1998)
27. de Moura, L.M., Dutertre, B., Shankar, N.: A tutorial on satisfiability modulo theories. In: Proc. 19th International Conference on Computer Aided Verification (CAV'07). LNCS, vol. 4590, pp. 20–36. Springer, New York (2007)
28. Ni, Z., Yu, D., Shao, Z.: Using XCAP to certify realistic systems code: machine context management. In: Proc. 20th Int'l Conf. on Theorem Proving in Higher Order Logics. LNCS, vol. 4421, pp. 189–206. Springer, New York (2007)
29. O'Hearn, P.W.: Resources, concurrency and local reasoning. In: Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04). LNCS, vol. 3170, pp. 49–67. Springer, New York (2004)
30. O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. In: Proc. 31th ACM Symp. on Principles of Prog. Lang., pp. 268–280. ACM, New York (2004)
31. Palsberg, J., Ma, D.: A typed interrupt calculus. In: Proc. 7th Int'l Symp. on Formal Tech. in Real-Time and Fault-Tolerant Sys. (FTRTFT'02). LNCS, vol. 2469, pp. 291–310. Springer, New York (2002)
32. Paul, W., Broy, M., In der Rieden, T.: The Verisoft XT project. http://www.verisoft.de (2007)
33. Regehr, J., Cooprider, N.: Interrupt verification via thread verification. Electron. Notes Theor. Comput. Sci. **174**(9) (2007)
34. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proc. 17th Annual IEEE Symp. on Logic in Comp. Sci. (LICS'02), pp. 55–74. IEEE Computer Society, Los Alamitos (2002)
35. Suenaga, K., Kobayashi, N.: Type based analysis of deadlock for a concurrent calculus with interrupts. In: Proc. 16th European Symp. on Prog. (ESOP'07). LNCS, vol. 4421, pp. 490–504. Springer, New York (2007)
36. Tuch, H., Klein, G., Heiser, G.: OS verification—now! In: Proc. 10th Workshop on Hot Topics in Operating Systems, Santa Fe, 12–15 June 2005
37. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: Proc. 18th Int'l Conf. on Concurrency Theory (CONCUR'07). LNCS, vol. 4703, pp. 256–271. Springer, New York (2007)
38. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. **115**(1), 38–94 (1994)
39. Yang, H., O'Hearn, P.W.: A semantic basis for local reasoning. In: Proc. 5th Int'l Conf. on Foundations of Software Science and Computation Structures (FoSSaCS'02). LNCS, vol. 2303, pp. 402–416. Springer, New York (2002)