Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads

Xinyu Feng[†‡] Zhong Shao[†] Yuan Dong[†§] Yu Guo[*]

[†]Yale University, New Haven, CT 06520-8285, U.S.A.
[‡]Toyota Technological Institute at Chicago, Chicago, IL 60637, U.S.A.
[§]Tsinghua University, Beijing 100084, China
[*]University of Science and Technology of China, Hefei, Anhui 230026, China
feng@tti-c.org shao@cs.yale.edu dongyuan@tsinghua.edu.cn guoyu@mail.ustc.edu.cn

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-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.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.2.4 [*Software Engineering*]: Software/Program Verification — Correctness proofs, formal methods, reliability; D.3.1 [*Programming Languages*]: Formal Definitions and Theory — Semantics; D.4.5 [*Operating Systems*]: Reliability — Verification

General Terms Languages, Reliability, Security, Verification

Keywords Hardware Interrupts, Preemptive Threads, Certified System Software, Concurrency, Separation Logic

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.

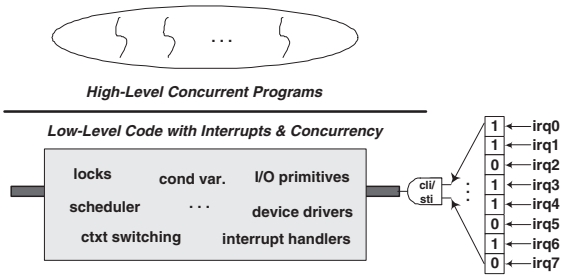


Figure 1. "High-Level" vs. "Low-Level" System Programs

1. Introduction

Low-level system programs (e.g., thread implementations, device drivers, OS 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 (Gargano et al. 2005; Paul et al. 2007), L4.verified (Tuch et al. 2005), and Singularity (Hunt and Larus 2004)—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 (Hunt and Larus 2004).

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 (Hoare 1972): 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 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 (Hoare 1969), typed assembly language (Morrisett et al.

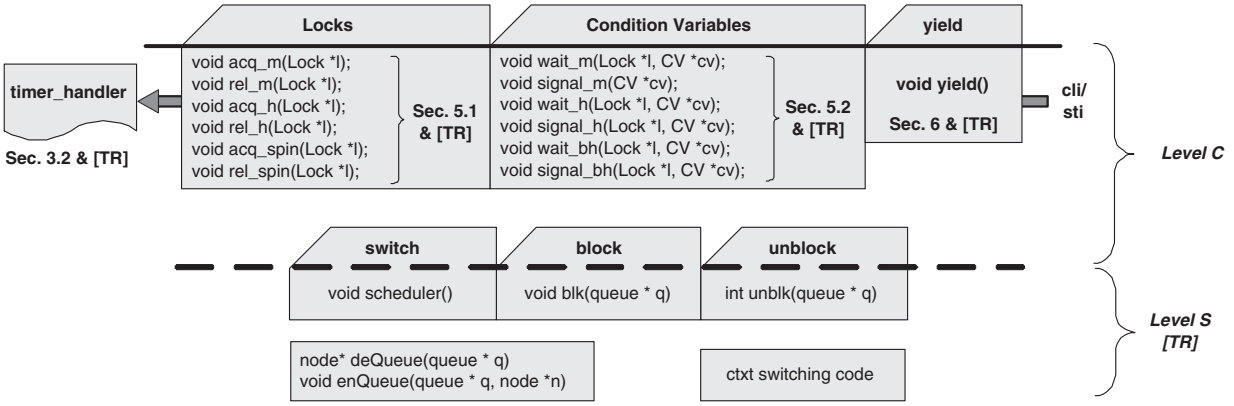


Figure 3. Structure of Our Certified Preemptive Thread Implementation

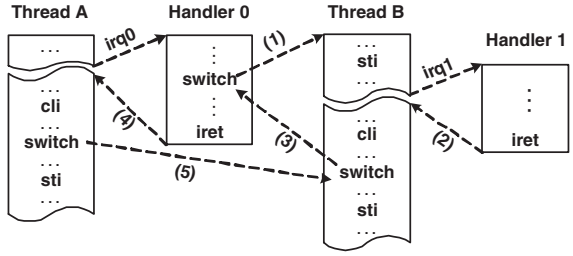


Figure 2. Interaction between Threads and Interrupts

1998), concurrent separation logic (O'Hearn 2004; Brookes 2004), and its assume-guarantee variant (Feng et al. 2007a; Vafeiadis and Parkinson 2007)) 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 (Regehr and Cooprider 2007).
- *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 Sec. 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, 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 Sec. 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 Sec. 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 (Feng et al. 2007b).
- 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 Sec. 5). Our specifications pinpoint precisely the differences between different implementations.

2. Informal Development

Before presenting our formal framework, we first informally explain the key ideas underlying our abstract machine and our 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

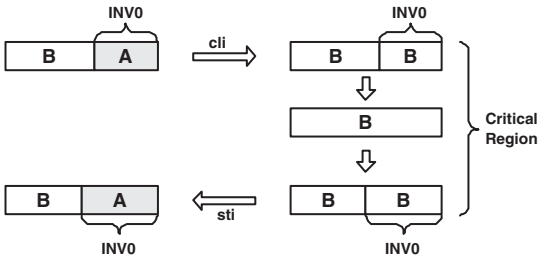


Figure 4. Memory Partition for Handler and Non-Handler

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 of the current thread into the ready queue, picks another one 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 (O'Hearn 2004; Brookes 2004): *there always exists a partition of memory among these 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 (Ishtiaq and O'Hearn 2001; Reynolds 2002), 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 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

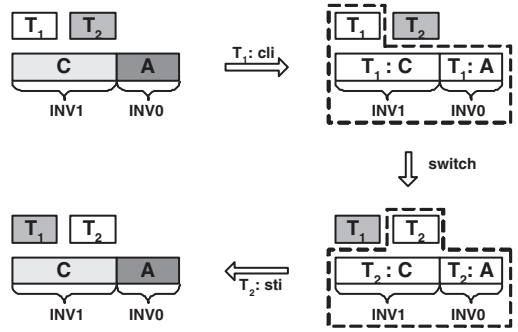


Figure 5. The Memory Model for Multi-Threaded Non-Handler

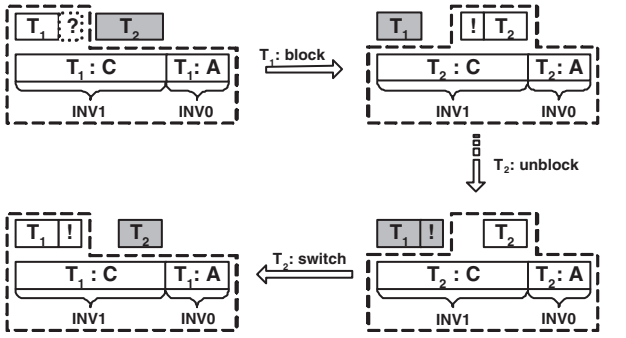


Figure 6. Block and Unblock

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. Fig. 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 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

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

Figure 7. Definition of AIM-1

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

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

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

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

Figure 8. Definition of Representations

switches control to other threads, $\mathtt{T}_1$ needs to ensure that INV0 and INV1 hold over A and C, which is the same requirement as switch. When $\mathtt{T}_2$ 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 $\mathtt{T}_1$ takes control again, it will own the resource. From $\mathtt{T}_1$'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 Sec. 5).

3. The Abstract Interrupt Machine (AIM)

In this section, 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 to commands c. Each command c is either a sequential or branch instruction ι, or jump or return instructions. The state $\mathbb{S}$ contains the data heap $\mathbb{H}$, the register file $\mathbb{R}$, and flags ie and is. 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. Each stack frame either contains a code label f or a pair $(\mathtt{f},\mathbb{R})$. We also define the instruction sequence $\mathbb{I}$ as a sequence of sequential instructions ending with jump or return commands. $\mathbb{C}[\mathtt{f}]$ extracts an instruction sequence starting from f in $\mathbb{C}$, as defined in Fig. 8. We use the dot notation to represent a component in a tuple, *e.g.,* $\mathbb{S}.\mathbb{H}$ means the data heap in state $\mathbb{S}$. More representations are defined in Fig. 8.

$\mathtt{NextS}_{(\mathtt{c},\mathbb{K})}\ \mathbb{S}\ \mathbb{S}'$ 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 $\mathtt{is} = 1, \mathbb{K} = (\mathtt{f},\mathbb{R}') :: \mathbb{K}'$ 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

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

$$
\frac{\mathtt{ie} = 1 \quad \mathtt{is} = 0}{(\mathbb{C},(\mathbb{H},\mathbb{R},\mathtt{ie},\mathtt{is}),\mathbb{K},\mathtt{pc}) \not\downarrow (\mathbb{C},(\mathbb{H},\mathbb{R},0,1),(\mathtt{pc},\mathbb{R}) :: \mathbb{K},\mathtt{h\_entry})} \ (\textsc{irq})
$$

$$
\mathbb{W} \Longmapsto \mathbb{W}' \stackrel{\text{def}}{=} (\mathbb{W} \longmapsto \mathbb{W}') \vee (\mathbb{W} \not\downarrow \mathbb{W}')
$$

Figure 9. Operational Semantics of Instructions

Operational semantics. At each step, the machine either executes the next instruction at pc or jumps to handle the incoming interrupt. To simplify the presentation, the machine supports only one interrupt, with a global interrupt handler entry h_entry. Support of multi-level interrupts is discussed in Sec. 6. An incoming interrupt is processed only if the ie bit is set, and no interrupt is currently being serviced (*i.e.,* is = 0). The processor pushes $(\mathtt{pc},\mathbb{R})$ onto the stack $\mathbb{K}$, clears ie, sets is, and sets the new pc to h_entry. The state transition $(\mathbb{W} \not\downarrow \mathbb{W}')$ is defined in the IRQ rule in Fig. 9.

The operational semantics of each instruction is defined in Fig. 9. We use relations $\mathtt{NextS}_{(\mathtt{c},\mathbb{K})}$, $\mathtt{NextK}_{(\mathtt{pc},\mathtt{c})}$ and $\mathtt{NextPC}_{(\mathtt{c},\mathbb{R},\mathbb{K})}$ to show the change of states, stacks and program counters respectively when c is executed. Semantics of most instructions are straightforward. The command iret pops the stack frame $(\mathtt{f},\mathbb{R}')$ on top of $\mathbb{K}$ and resets pc and the register file $\mathbb{R}$ with f and $\mathbb{R}'$, respectively. It also 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). 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

$(\mathbb{C},\mathbb{S},\mathbb{K},\mathtt{pc},\mathtt{tid},\mathbb{T},\mathbb{B}) \longmapsto \mathbb{W}'$ 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 $\mathtt{ie}=0$, $w=\mathbb{R}(r_t)$, $\mathbb{B}(w)=\varnothing$, 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 $\mathtt{NextPC}_{(\mathtt{c},\mathbb{R},\mathbb{K})}\ \mathtt{pc}\ \mathtt{pc}'$

Figure 12. The Step Relation for AIM-2

```
incleft: -{(p0, NoG)}          h_entry: -{(pi, gi)}
         movi $r1, RIGHT                 movi $r1, LEFT
         movi $r2, LEFT                  movi $r2, RIGHT
l_loop:  -{(p1, NoG)}                    movi $r3, 0
         movi $r3, 0                     ld   $r4, 0($r1)
         cli                             beq  $r3, $r4, r_win
         -{(p2, NoG)}                    movi $r3, 1
         ld   $r4, 0($r1)                sub  $r4, $r3
         beq  $r3, $r4, l_win            st   0($r1), $r4
         movi $r3, 1                     ld   $r4, 0($r2)
         sub  $r4, $r3                   add  $r4, $r3
         st   0($r1), $r4                st   0($r2), $r4
         ld   $r4, 0($r2)                iret
         add  $r4, $r3           r_win:  -{(p4, gid)}
         st   0($r2), $r4                iret
         sti
         -{(p1, NoG)}
         j    l_loop
l_win:   -{(p3, NoG)}
         sti
         j    l_loop
```

Figure 10. Sample AIM-1 Program: Teeter-Totter

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

Figure 11. AIM-2 Defined as an Extension of AIM-1

a cleaner presentation. In our implementation (Feng et al. 2007c), the handler code needs to save and restore the registers.

Fig. 9 also defines $(\mathbb{W} \longmapsto \mathbb{W}')$ for executing the instruction at the current pc; program execution is then modeled as $\mathbb{W} \Longmapsto \mathbb{W}'$.

Fig. 10 shows a sample AIM-1 program. The program specifications in shaded boxes are explained in Sec. 4. Initially LEFT and RIGHT point to memory cells containing 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. 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.

```
h_entry: -{(pi, gi)}
         j      h_timer
h_timer: -{(pi, gi)}
         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:    -{(p0, g0)}
         movi   $r2, 0
         st     0($r1), $r2        ; [CNT] := 0
         switch
         iret
```

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

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

Figure 13. A Preemptive Timer Handler

3.2 AIM-2

Fig. 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, 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:

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

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 "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 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;

$$
\begin{array}{rlll}
(\textit{CdHpSpec}) & \Psi & ::= & \{(\mathtt{f}_1,\mathtt{s}_1),\ldots,(\mathtt{f}_n,\mathtt{s}_n)\} \\
(\textit{Spec}) & \mathtt{s} & ::= & (\mathtt{p},\mathtt{g}) \\
(\textit{Pred}) & \mathtt{p} & \in & \textit{Stack} \to \textit{State} \to \textit{Prop} \\
(\textit{Guarantee}) & \mathtt{g} & \in & \textit{State} \to \textit{State} \to \textit{Prop} \\
(\textit{MPred}) & \mathtt{m},\mathsf{INV0},\mathsf{INV1} & \in & \textit{Heap} \to \textit{Prop} \\
(\textit{WQSpec}) & \Delta & ::= & \{w \rightsquigarrow \mathtt{m}\}^*
\end{array}
$$

Figure 14. Specification Constructs

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. The definition of $(\mathbb{W} \Longmapsto \mathbb{W}')$ is unchanged.

A preemptive timer interrupt handler. 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) for higher-level concurrent programs (see Fig. 1).

Fig. 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 certified in Sec. 4.

4. The Program Logic

4.1 Specification Language

We use the mechanized *meta-logic* implemented in the Coq proof assistant (Coq 2006) as our specification language. The logic corresponds to higher-order logic with inductive definitions.

As shown in Fig. 14, the specification Ψ for the code heap $\mathbb{C}$ associates code labels `f` with specifications `s`. We allow each `f` to have more than one `s`, just as a function may have multiple specified interfaces. The specification `s` is a pair $(\mathtt{p},\mathtt{g})$. The assertion `p` is a predicate over a stack $\mathbb{K}$ and a program state $\mathbb{S}$, while `g` 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`. As in SCAP (Feng et al. 2006), we use `p` to specify the precondition over stack and state, 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. We encode in Fig. 15 Separation Logic connectors (Ishtiaq and O'Hearn 2001; Reynolds 2002) in our specification language. 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. "$\mathtt{m} * \mathtt{m}'$" means the heap can be split into two *disjoint* parts, and `m` and $\mathtt{m}'$ hold over one of them respectively. "$\mathtt{m} -\!\!* \mathtt{m}'$" holds over $\mathbb{H}$ iff, for any disjoint heap $\mathbb{H}'$ satisfying `m`, $\mathbb{H} \uplus \mathbb{H}'$ satisfies $\mathtt{m}'$.

The specification Δ maps an 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. 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]

$$
\begin{array}{l}
\mathsf{true} \stackrel{\text{def}}{=} \lambda\mathbb{H}.\ \mathsf{True} \qquad\qquad \mathsf{emp} \stackrel{\text{def}}{=} \lambda\mathbb{H}.\ \mathbb{H} = \varnothing \\
\mathtt{l} \mapsto \mathtt{w} \stackrel{\text{def}}{=} \lambda\mathbb{H}.\ \mathbb{H} = \{\mathtt{l} \rightsquigarrow \mathtt{w}\} \qquad\qquad \mathtt{l} \mapsto \_ \stackrel{\text{def}}{=} \lambda\mathbb{H}.\ \exists \mathtt{w}.\ (\mathtt{l} \mapsto \mathtt{w})\ \mathbb{H} \\
\mathbb{H}_1 \perp \mathbb{H}_2 \stackrel{\text{def}}{=} \mathit{dom}(\mathbb{H}_1) \cap \mathit{dom}(\mathbb{H}_1) = \varnothing \\
\mathbb{H}_1 \uplus \mathbb{H}_2 \stackrel{\text{def}}{=} \begin{cases} \mathbb{H}_1 \cup \mathbb{H}_2 & \text{if } \mathbb{H}_1 \perp \mathbb{H}_2 \\ \textit{undefined} & \text{otherwise} \end{cases} \\
\mathtt{m}_1 * \mathtt{m}_2 \stackrel{\text{def}}{=} \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} \stackrel{\text{def}}{=} \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} -\!\!* \mathtt{m}' \stackrel{\text{def}}{=} \lambda\mathbb{H}.\ \forall \mathbb{H}', \mathbb{H}''.\ (\mathbb{H} \uplus \mathbb{H}' = \mathbb{H}'') \wedge \mathtt{m}\ \mathbb{H}' \to \mathtt{m}'\ \mathbb{H}'' \\
\mathtt{m} -\!\!* \mathtt{p} \stackrel{\text{def}}{=} \lambda\mathbb{K}, \mathbb{S}.\ \forall \mathbb{H}', \mathbb{H}''.\ (\mathbb{H}' \uplus \mathbb{S}.\mathbb{H} = \mathbb{H}'') \wedge \mathtt{m}\ \mathbb{H}' \to \mathtt{p}\ \mathbb{K}\ \mathbb{S}|_{\mathbb{H}''} \\
\mathsf{precise}(\mathtt{m}) \stackrel{\text{def}}{=} \forall \mathbb{H}, \mathbb{H}_1, \mathbb{H}_2. \\
\qquad (\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 \to (\mathbb{H}_1 = \mathbb{H}_2)
\end{array}
$$

Figure 15. Definitions of Separation Logic Assertions

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 $(\mathtt{h\_entry}, (\mathtt{p}_\mathsf{i}, \mathtt{g}_\mathsf{i})) \in \Psi$, where $\mathtt{p}_\mathsf{i}$ and $\mathtt{g}_\mathsf{i}$ are defined as follows:

$$
\begin{array}{rl}
\mathtt{p}_\mathsf{i} \stackrel{\text{def}}{=} & \lambda\mathbb{K}, \mathbb{S}.\ ((\mathsf{INV0} * \mathsf{true})\ \mathbb{S}.\mathbb{H}) \wedge (\mathbb{S}.\mathtt{ie} = 1) \wedge (\mathbb{S}.\mathtt{ie} = 0) \\
& \wedge \exists \mathtt{f}, \mathbb{R}, \mathbb{K}'.\ \mathbb{K} = (\mathtt{f}, \mathbb{R}) :: \mathbb{K}' \qquad (1) \\
\mathtt{g}_\mathsf{i} \stackrel{\text{def}}{=} & \lambda\mathbb{S}, \mathbb{S}'.\ \left\{ \begin{array}{l} \mathsf{INV0} \\ \mathsf{INV0} \end{array} \right\}\ \mathbb{S}.\mathbb{H}\ \mathbb{S}'.\mathbb{H} \\
& \wedge (\mathbb{S}'.\mathtt{ie} = \mathbb{S}.\mathtt{ie}) \wedge (\mathbb{S}'.\mathtt{is} = \mathbb{S}.\mathtt{is}) \qquad (2)
\end{array}
$$

The precondition $\mathtt{p}_\mathsf{i}$ 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) satisfies INV0. The guarantee $\mathtt{g}_\mathsf{i}$ specifies the behavior of the handler. The arguments $\mathbb{S}$ and $\mathbb{S}'$ correspond to program states at the entry and exit points, 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.

$$
\begin{array}{rl}
\left\{ \begin{array}{l} \mathtt{m}_1 \\ \mathtt{m}_2 \end{array} \right\} \stackrel{\text{def}}{=} & \lambda\mathbb{H}_1, \mathbb{H}_2.\ \exists \mathbb{H}_1', \mathbb{H}_2', \mathbb{H}.\ (\mathtt{m}_1\ \mathbb{H}_1') \wedge (\mathtt{m}_2\ \mathbb{H}_2') \wedge \\
& (\mathbb{H}_1' \uplus \mathbb{H} = \mathbb{H}_1) \wedge (\mathbb{H}_2' \uplus \mathbb{H} = \mathbb{H}_2) \qquad (3)
\end{array}
$$

It has the following nice monotonicity: for any $\mathbb{H}_1$, $\mathbb{H}_2$ and $\mathbb{H}'$, if $\mathbb{H}_1$ and $\mathbb{H}_2$ satisfy the predicate, $\mathbb{H}_1 \perp \mathbb{H}'$, and $\mathbb{H}_2 \perp \mathbb{H}'$, then $\mathbb{H}_1 \uplus \mathbb{H}'$ and $\mathbb{H}_2 \uplus \mathbb{H}'$ satisfy the predicate.

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 `f`, given the imported interfaces in Ψ, the specification Δ of block queues, and a precondition $(\mathtt{p}, \mathtt{g})$.

The SEQ rule is a *schema* for instruction sequences starting with an instruction ι (ι cannot be 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 $\mathtt{g}_\iota$

[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 (Reynolds 2002; O'Hearn et al. 2004) can be supported following the same way they are supported in SCAP (Feng and Shao 2008). We do not discuss the details in this paper.

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

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

$$\frac{\begin{array}{ll}(\mathtt{f}{+}1,(\mathtt{p}'',\mathtt{g}''))\in\Psi & \Psi,\Delta\vdash\{(\mathtt{p}'',\mathtt{g}'')\}\,\mathtt{f}{+}1:\mathbb{I}\\ (\mathtt{f}',(\mathtt{p}',\mathtt{g}'))\in\Psi & \forall\mathbb{K},\mathbb{S},\mathtt{pc}.\,\mathtt{p}\,\mathbb{K}\,\mathbb{S}\to\mathtt{p}'\,(\mathtt{pc}::\mathbb{K})\,\mathbb{S}\\ (\mathtt{p}\rhd\mathtt{g}')\Rightarrow\mathtt{p}'' & (\mathtt{p}\circ(\mathtt{g}'\circ\mathtt{g}''))\Rightarrow\mathtt{g}\end{array}}{\Psi,\Delta\vdash\{(\mathtt{p},\mathtt{g})\}\,\mathtt{f}:\ \mathsf{call}\ \mathtt{f}';\mathbb{I}}\ (\textsc{call})$$

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

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

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

where $\mathsf{enable}_{\mathsf{ret}}\stackrel{\text{def}}{=}\lambda\mathbb{K},\mathbb{S}.\ \exists\mathtt{f},\mathbb{K}'.\ \mathbb{K}=\mathtt{f}::\mathbb{K}'$

$$\frac{\begin{array}{ll}(\mathtt{f}',(\mathtt{p}',\mathtt{g}'))\in\Psi & \Psi,\Delta\vdash\{(\mathtt{p}'',\mathtt{g}'')\}\,\mathtt{f}{+}1:\mathbb{I}\\ (\mathtt{p}\rhd\mathsf{gid}_{r_s=r_t})\Rightarrow\mathtt{p}' & (\mathtt{p}\circ(\mathsf{gid}_{r_s=r_t}\circ\mathtt{g}'))\Rightarrow\mathtt{g}\\ (\mathtt{p}\rhd\mathsf{gid}_{r_s\neq r_t})\Rightarrow\mathtt{p}'' & (\mathtt{p}\circ(\mathsf{gid}_{r_s\neq r_t}\circ\mathtt{g}''))\Rightarrow\mathtt{g}\end{array}}{\Psi,\Delta\vdash\{(\mathtt{p},\mathtt{g})\}\,\mathtt{f}:\ \mathsf{beq}\ r_s,r_t,\mathtt{f}';\mathbb{I}}\ (\textsc{beq})$$

$$\frac{(\mathtt{f}',(\mathtt{p}',\mathtt{g}'))\in\Psi\quad\mathtt{p}\Rightarrow\mathtt{p}'\quad(\mathtt{p}\circ\mathtt{g}')\Rightarrow\mathtt{g}}{\Psi,\Delta\vdash\{(\mathtt{p},\mathtt{g})\}\,\mathtt{f}:\ \mathtt{j}\ \mathtt{f}'}\ (\textsc{j})$$

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

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

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

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

Figure 16. Inference Rules

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

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

$\mathsf{enable}(\mathtt{p},\mathtt{g})\stackrel{\text{def}}{=}\forall\mathbb{K},\mathbb{S}.\,\mathtt{p}\,\mathbb{K}\,\mathbb{S}\to\exists\mathbb{S}',\mathtt{g}\,\mathbb{S}\,\mathbb{S}'$

$\mathtt{p}\rhd\mathtt{g}\stackrel{\text{def}}{=}\lambda\mathbb{K},\mathbb{S}.\ \exists\mathbb{S}_0,\mathtt{p}\,\mathbb{K}\,\mathbb{S}_0\wedge\mathtt{g}\,\mathbb{S}_0\,\mathbb{S}$

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

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

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

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

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

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

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

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

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

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

Figure 18. Thread-Local State Transitions Made by ι

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 $\mathtt{p}$ contains 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. The transition $[\![\mathsf{sti}]\!]_\Delta$ is defined similarly. Note that the premise $\mathsf{enable}(\mathtt{p},\mathtt{g}_\iota)$ in the SEQ rule requires that, before executing sti, the precondition $\mathtt{p}$ must contain the ownership $(\mathtt{ie}=0\wedge\mathtt{is}=0)\,?\,(\mathsf{INV1}*\mathsf{INV0}):\mathsf{emp}$.

$[\![\mathsf{switch}]\!]_\Delta$ 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 $\mathsf{enable}(\mathtt{p},\mathtt{g}_\iota)$ requires that $\mathtt{p}$ ensures $\mathtt{ie}=0$ and $\mathsf{INV0}*(\mathtt{is}=0\,?\,\mathsf{INV1}:\mathsf{emp})$ holds over some sub-heap.

$[\![\mathsf{block}\ r_s]\!]_\Delta$ requires $\mathtt{ie}=0$ and r_s contains an identifier of a block queue with 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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 19. Well-Formed Current, Ready and Waiting Threads

contain a sub-heap satisfying 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 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. In the CALL rule in Fig. 16, we treat the state transition $\mathtt{g}'$ made by the callee as the transition of the call instruction. We also require that the precondition p implies the precondition $\mathtt{p}'$ of the callee, which corresponds to the enable premise in the SEQ rule. IRET and RET rules require that the function has 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 beq never gets stuck. The J rule can be viewed as a specialization of BEQ.

Well-formed code heap. 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 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

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

Figure 20. Specifications of the Teeter-Totter Example

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 pc has a specification $(\mathtt{p},\mathtt{g})$ in Ψ, thus $\mathbb{C}[\mathtt{pc}]$ is well-formed with respect to $(\mathtt{p},\mathtt{g})$. The current thread's stack and its local state (containing the sub-heap $\mathbb{H}_0$) need to satisfy $\mathtt{p} * \mathsf{Inv}$. Here p specifies the state accessible by the current thread, while Inv, defined in Fig. 19, specifies the inaccessible part of the shared heap. If the current program point is in the interrupt handler ($\mathtt{is}=1$), p leaves the memory block C (in Fig. 5) unspecified, therefore Inv 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 specifies their well-formedness in this case. The predicate WFST, defined in Fig. 19, says there exists a well-formed stack with some depth k. The definition is similar to the one in SCAP (Feng et al. 2006) and is not explained here.

The definition of well-formed ready threads WFRdy is very straightforward: 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 m becomes a well-formed *ready* thread if it gets m (see Fig. 6). The definitions of WFRdy and WFWait concisely formulate the relationship between current, ready and waiting threads.

The Teeter-Totter example. With our program logic, we can now certify the Teeter-Totter example shown in Fig. 10. 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 (1) and (2) in Sec. 4.1. We let INV1 be emp, since the non-handler code is sequential.

Specifications are shown in Fig. 20. Recall $\mathsf{enable}_{\mathsf{iret}}$ is defined in Fig. 16. To simplify our presentation, we present the predicate p in the form of a proposition with free variables referring to components of the state $\mathbb{S}$. Also, we use 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 LEFT and RIGHT. Since the non-handler never returns, we simply use NoG (see Fig. 20) as the guarantee for the state transition from the specified point to the return point.

The timer handler. We also 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 $(\mathsf{p_i}, \mathsf{g_i})$ by Formulae (1) and (2). In g_0 (shown in Fig. 13), we use primed variable (*e.g.,* $\mathtt{ie}'$ and $\mathtt{is}'$) to refer to components in the second state.

Soundness. We prove the soundness of the program logic following the syntactic approach. Based on the progress and preservation lemmas, we know the program never gets stuck as long as the initial state satisfies the program invariant defined by the WLD rule. *More importantly*, we know the invariant always holds during execution, from which we can derive rich properties of programs. Here, we only show the soundness theorem formalizing the partial correctness of programs. See the TR (Feng et al. 2007c) for proof details.

Theorem 4.1 (Soundness)
If INV0 and INV1 are precise, $\Psi, \Delta \vdash \mathbb{W}$, and $(\mathtt{h\_entry}, (\mathsf{p_i}, \mathsf{g_i})) \in \Psi$, then, for any n, there exists $\mathbb{W}'$ such that $\mathbb{W} \Longrightarrow^n \mathbb{W}'$; and, if $\mathbb{W}' = (\mathbb{C}, \mathbb{S}, \mathbb{K}, \mathtt{pc}, \mathtt{tid}, \mathbb{T}, \mathbb{B})$, then

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

Recall that preciseness is defined in Fig. 15, and the specification $(\mathsf{p_i}, \mathsf{g_i})$ is defined by Formulae (1) and (2).

5. More Examples

In this section, we show how to use AIM and the program logic to implement and certify common synchronization primitives.

5.1 Implementations of Locks

Threads use locks to achieve exclusive access to shared heap. We use Γ to specify invariants of memory blocks protected by locks.

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

In our implementations, we use memory pointers (label l) as lock ids l. Each l points to a memory cell containing a binary flag that records whether the lock has been acquired (flag is 0) or not. The heap used to implement locks and the heap protected by locks are shared by threads in the non-handler code. The invariant $\mathsf{INV}(\Gamma)$ over this part of heap is defined below. We require $\mathsf{INV}_s \Rightarrow \mathsf{INV}(\Gamma) * \mathsf{true}$ (recall that INV_s is a shorthand for $\mathsf{INV0} * \mathsf{INV1}$).

$$\mathsf{INV}(l, \mathtt{m}) \stackrel{\text{def}}{=} \exists \mathtt{w}.\ (l \mapsto \mathtt{w}) * ((\mathtt{w} = 0) \wedge \mathsf{emp} \vee (\mathtt{w} = 1) \wedge \mathtt{m}) \quad (4)$$

$$\mathsf{INV}(\Gamma) \stackrel{\text{def}}{=} \forall_* l \in \mathit{dom}(\Gamma).\ \mathsf{INV}(l, \Gamma(l)) \quad (5)$$

where $\forall_*$ 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, in which we use the lock id as the identifier of the corresponding block queue in $\mathbb{B}$. Then we show an implementation of spinlocks. More detailed explanations are given in the TR (Feng et al. 2007c).

The Hoare-style implementations. In Hoare style, the thread gets the lock (and the resource protected by the lock) immediately after it is released from the block queue. The implementation and specifications are shown in Figs. 21 and 22. The precondition for ACQ_H is $(\mathsf{p}_{01}, \mathsf{g}_{01})$. The assertion p_{01} requires that r_1 contains a lock id and $\Delta(r_1) = \Gamma(r_1)$. The guarantee g_{01} shows that the

```
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          ;; [l] <> 0:
          ret                         ;;   [l] <- 0
gowait:   -{(p12, g11)}                ;; [l] == 0:
          block  $r1                  ;;   block
          -{(p13, gid)}
          ret

REL_H:    -{(p21, g21)}
          cli
          call    REL_H_a
          sti
          ret
REL_H_a:  -{(p31, g31)}
          unblock $r1, $r2
          -{(p32, g32)}
          movi    $r3, 0
          beq     $r2, $r3, rel_lock
          ret
rel_lock: -{(p33, g33)}
          movi    $r2, 1
          st      0($r1), $r2
          -{(p34, gid)}
          ret
```

Figure 21. Hoare-Style Implementation of Locks

$$\mathsf{p}_0 \stackrel{\text{def}}{=} (\mathtt{is} = 0) \wedge \mathsf{enable}_{\mathsf{ret}} \wedge (r_1 \in \mathit{dom}(\Gamma)) \wedge (\Delta(r_1) = \Gamma(r_1))$$
$$\mathsf{p}_{01} \stackrel{\text{def}}{=} \mathsf{p}_0 \wedge (\mathtt{ie} = 1)$$
$$\mathsf{g}_{01} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathsf{emp} \\ \Gamma(r_1) \end{array} \right\} \wedge (\mathtt{ie} = \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}') \wedge \mathsf{trash}(\{r_2, r_3\})$$
$$\mathsf{p}_{11} \stackrel{\text{def}}{=} \mathsf{p}_0 \wedge (\mathtt{ie} = 0) \wedge (\mathsf{INV}_s * \mathsf{true})$$
$$\mathsf{g}_{11} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathsf{INV}_s \\ \mathsf{INV}_s * \Gamma(r_1) \end{array} \right\} \wedge (\mathtt{ie} = \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}') \wedge \mathsf{trash}(\{r_2, r_3\})$$
$$\mathsf{p}_{12} \stackrel{\text{def}}{=} \mathsf{p}_0 \wedge (\mathtt{ie} = 0) \wedge ([r_1] = 0) \wedge (\mathsf{INV}_s * \mathsf{true})$$
$$\mathsf{p}_{13} \stackrel{\text{def}}{=} \mathsf{p}_0 \wedge (\mathtt{ie} = 0) \qquad \wedge (\mathsf{INV}_s * \mathsf{true} * \Gamma(r_1))$$
$$\mathsf{p}_{21} \stackrel{\text{def}}{=} \mathsf{p}_0 \wedge (\mathtt{ie} = 1) \wedge (\Gamma(r_1) * \mathsf{true})$$
$$\mathsf{g}_{21} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \Gamma(r_1) \\ \mathsf{emp} \end{array} \right\} \wedge (\mathtt{ie} = \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}') \wedge \mathsf{trash}(\{r_2, r_3\})$$
$$\mathsf{p}_{31} \stackrel{\text{def}}{=} \mathsf{p}_0 \wedge (\mathtt{ie} = 0) \wedge (\Gamma(r_1) * \mathsf{INV}_s * \mathsf{true})$$
$$\mathsf{g}_a \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \Gamma(r_1) \\ \mathsf{emp} \end{array} \right\} \qquad \mathsf{g}_b \stackrel{\text{def}}{=} \left\{ \begin{array}{l} r_1 \mapsto \_ \\ r_1 \mapsto 1 \end{array} \right\} \qquad \mathsf{hid} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathsf{emp} \\ \mathsf{emp} \end{array} \right\}$$
$$\mathsf{g}_{31} \stackrel{\text{def}}{=} (\mathsf{g}_a \vee \mathsf{g}_b) \wedge (\mathtt{ie} = \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}') \wedge \mathsf{trash}(\{r_2, r_3\})$$
$$\mathsf{p}_{32} \stackrel{\text{def}}{=} \mathsf{p}_0 \wedge (\mathtt{ie} = 0) \wedge ((r_2 = 0) \wedge (\Gamma(r_1) * \mathsf{INV}_s * \mathsf{true}) \vee (r_2 \neq 0) \wedge (\mathsf{INV}_s * \mathsf{true}))$$
$$\mathsf{g}_{32} \stackrel{\text{def}}{=} ((r_2 = 0 \wedge \mathsf{g}_b) \vee (r_2 \neq 0 \wedge \mathsf{hid})) \wedge (\mathtt{ie} = \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}') \wedge \mathsf{trash}(\{r_2, r_3\})$$
$$\mathsf{p}_{33} \stackrel{\text{def}}{=} \mathsf{p}_0 \wedge (\mathtt{ie} = 0) \wedge (\Gamma(r_1) * \mathsf{INV}_s * \mathsf{true})$$
$$\mathsf{g}_{33} \stackrel{\text{def}}{=} \mathsf{g}_b \wedge (\mathtt{ie} = \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}') \wedge \mathsf{trash}(\{r_2, r_3\})$$
$$\mathsf{p}_{34} \stackrel{\text{def}}{=} \mathsf{p}_0 \wedge (\mathtt{ie} = 0) \wedge (\mathsf{INV}_s * \mathsf{true})$$

Figure 22. Specifications of Hoare-Style Locks

$p_0 \stackrel{\text{def}}{=} (\texttt{is} = 0) \wedge \mathsf{enable}_{\mathsf{ret}} \wedge (r_1 \in dom(\Gamma)) \wedge (\Delta(r_1) = \mathsf{emp})$

$p_{11} \stackrel{\text{def}}{=} p_0 \wedge (\texttt{ie} = 1)$ $\quad$ $p_{12} \stackrel{\text{def}}{=} p_{11} \wedge (r_3 = 0)$

$g_{11} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathsf{emp} \\ \Gamma(r_1) \end{array} \right\} \wedge (\texttt{ie} = \texttt{ie}') \wedge (\texttt{is} = \texttt{is}') \wedge \mathsf{trash}(\{r_2, r_3\})$

$p_{13} \stackrel{\text{def}}{=} p_0 \wedge (\texttt{ie} = 0) \wedge (\mathsf{INV}_s * \mathsf{true})$

$g_{13} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathsf{INV}_s \\ \Gamma(r_1) \end{array} \right\} \wedge (\texttt{ie} = 1 - \texttt{ie}') \wedge (\texttt{is} = \texttt{is}') \wedge \mathsf{trash}(\{r_2, r_3\})$

$p_{21} \stackrel{\text{def}}{=} p_0 \wedge (\texttt{ie} = 1) \wedge (\Gamma(r_1) * \mathsf{true})$

$g_{21} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \Gamma(r_1) \\ \mathsf{emp} \end{array} \right\} \wedge (\texttt{ie} = \texttt{ie}') \wedge (\texttt{is} = \texttt{is}') \wedge \mathsf{trash}(\{r_2\})$

```
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
           sti
           ret
gowait:    -{(p13, g13)}
           block   $r1
           sti
           j       acq_loop

REL_M:     -{(p21, g21)}
           cli
           unblock $r1, $r2
           movi    $r2, 1
           st      0($r1), $r2
           sti
           ret
```

Figure 23. Mesa-Style Locks

function obtains the ownership of $\Gamma(r_1)$ when it returns. Here we use primed variables (*e.g.,* $\texttt{ie}'$ and $\texttt{is}'$) 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.

We also show some intermediate specifications used during verification. Comparing (p_{01}, g_{01}) and (p_{11}, g_{11}), we can see that (p_{01}, g_{01}) hides INV_s and the implementation details of the lock from the client code. Readers can also compare p_{12} and p_{13} and see how the BLK rule is applied.

Functions REL_H and REL_H_a are specified by (p_{21}, g_{21}) and (p_{31}, g_{31}), respectively. 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 or simply set the lock to be available, as specified in g_{31}, but these details are hidden in g_{21}.

The Mesa-style implementation. Fig. 23 shows the Mesa-style implementation of locks. In the ACQ_M function, the thread needs to start another round of loop to test the availability of the lock after block. The REL_M function always sets the lock to be available, even if it releases a waiting thread. Specifications are the same with Hoare style except that the assertion p_0 requires $\Delta(r_1) = \mathsf{emp}$, which implies the Mesa-style semantics of block and unblock. More intermediate assertions are given in the technical report.

Spinlocks. An implementation of spinlocks for uniprocessor systems and its specifications are shown in Fig. 24. The specifications (p_{11}, g_{11}) and (p_{21}, g_{21}) describes the interface of lock acquire/release. They look very similar to specifications for block-based implementations: "acquire" gets the ownership of the extra resource $\Gamma(r_1)$ protected by the lock in r_1, while "release" loses the ownership so that the client can no longer use the resource af-

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

$p_{11} \stackrel{\text{def}}{=} p \wedge (\texttt{ie} = 1)$ $\quad$ $p_{12} \stackrel{\text{def}}{=} p_{11} \wedge (r_2 = 1)$

$g_{11} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathsf{emp} \\ \Gamma(r_1) \end{array} \right\} \wedge (\texttt{ie} = \texttt{ie}') \wedge (\texttt{is} = \texttt{is}') \wedge \mathsf{trash}(\{r_2, r_3\})$

$p_{13} \stackrel{\text{def}}{=} p \wedge (\texttt{ie} = 0) \wedge ([r_1] = 1) \wedge (\mathsf{INV}_s * \mathsf{true})$

$g_{13} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathsf{INV}_s \\ \Gamma(r_1) \end{array} \right\} \wedge (\texttt{ie} = 1 - \texttt{ie}') \wedge (\texttt{is} = \texttt{is}') \wedge \mathsf{trash}(\{r_2\})$

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

$g_{21} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \Gamma(r_1) \\ \mathsf{emp} \end{array} \right\} \wedge (\texttt{ie} = \texttt{ie}') \wedge (\texttt{is} = \texttt{is}') \wedge \mathsf{trash}(\{r_2\})$

```
;; acquire(l): $r1 contains l
spin_acq:  -{(p11, g11)}
           movi $r2, 1
spin_loop: -{(p12, g11)}
           cli
           ld   $r3, 0($r1)
           beq  $r2, $r3, spin_set
           sti
           j    spin_loop
spin_set:  -{(p13, g13)}
           movi $r2, 0
           st   0($r1), $r2
           sti
           ret
;; release(l): $r1 contains l
spin_rel:  -{(p21, g21)}
           movi $r2, 1
           cli
           st   0($r1), $r2
           sti
           ret
```

Figure 24. A Spinlock

```
WAIT_H:    -{(p11, g11)}            ;; wait(l, cv)
           cli
           mov     $r4, $r2
           call    REL_H_a
           block   $r4
           sti
           ret
SIGNAL_H:  -{(p21, g21)}            ;; signal(l, cv)
           cli
           unblock $r2, $r3
           movi    $r4, 0
           beq     $r3, $r4, sig_done
           block   $r1
sig_done:  -{(p22, g22)}
           sti
           ret

SIGNAL_BH: -{(p31, g31)}            ;; signal(l, cv)
           cli
           unblock $r2, $r3         ;; $r2 contains cv
           movi    $r4, 0
           beq     $r3, $r4, sig_cont
           sti
           ret
sig_cont:  -{(p32, g32)}
           call    REL_H_a          ;; $r1 contains l
           sti
           ret
```

Figure 25. Impl. of CV - Hoare Style and Brinch Hansen Style

$$\mathsf{Cond}(r,r') \stackrel{\text{def}}{=} \Gamma(r) \wedge (\Upsilon(r') * \mathsf{true}) \quad \overline{\mathsf{Cond}}(r,r') \stackrel{\text{def}}{=} \Gamma(r) \wedge \neg(\Upsilon(r') * \mathsf{true})$$

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

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

$$\mathtt{g}_{11} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \overline{\mathsf{Cond}}(r_1,r_2) \\ \mathsf{Cond}(r_1,r_2) \end{array} \right\} \wedge (\mathtt{ie}=\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

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

$$\mathtt{g}_{21} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathsf{Cond}(r_1,r_2) \\ \Gamma(r_1) \end{array} \right\} \wedge (\mathtt{ie}=\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

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

$$\mathtt{g}_{22} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathsf{INV}_s \\ \mathsf{emp} \end{array} \right\} \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

$$\mathtt{p}_{31} \stackrel{\text{def}}{=} \mathtt{p}_{21} \qquad \mathtt{p}_{32} \stackrel{\text{def}}{=} \mathtt{p}_{22}$$

$$\mathtt{g}_{31} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathsf{Cond}(r_1,r_2) \\ \mathsf{emp} \end{array} \right\} \wedge (\mathtt{ie}=\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

$$\mathtt{g}_{32} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \Gamma(r_1) * \mathsf{INV}_s \\ \mathsf{emp} \end{array} \right\} \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

Figure 26. Spec. of CV - Hoare Style and Brinch Hansen Style

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

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

$$\mathtt{g}_{11} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \overline{\mathsf{Cond}}(r_1,r_2) \\ \Gamma(r_1) \end{array} \right\} \wedge (\mathtt{ie}=\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

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

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

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

```
WAIT_M:    -{(p11, g11)}              ;; wait(l, cv)
           cli
           mov     $r4, $r2
           call    REL_H_a
           block   $r4
           sti
           call    ACQ_H
           ret
SIGNAL_M:  -{(p21, g21)}              ;; signal(cv)
           cli
           unblock $r1, $r2
           sti
           ret
```

Figure 27. Impl. and Spec. of CV - Mesa Style

terwards. These specifications also hide the implementation details (*e.g.,* the lock name l is a pointer pointing to a binary value) from the client code.

5.2 Implementations of Condition Variables

Now we show implementations of Hoare style (Hoare 1974), Brinch Hansen style (Brinch Hansen 1975), and Mesa style (Lampson and Redell 1980) condition variables. Below we use Υ to specify the conditions associated with condition variables.

$$(\textit{CondVar})\ \ cv ::= n\ \ (\textit{nat nums})$$
$$(\textit{CVSpec})\ \ \Upsilon ::= \{cv \rightsquigarrow \mathtt{m}\}^*$$

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 resource specified by $\Gamma(l)$. 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. The implementations and specifications are shown in Figs. 25 and 26. The precondition for `WAIT_H` is $(\mathtt{p}_{11}, \mathtt{g}_{11})$. 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 $\Delta(r_2) = \Gamma(r_1) \wedge (\Upsilon(r_2) * \mathsf{true})$. Therefore, when the blocked thread is released, it gets the resource protected by the lock with the extra knowledge 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 $\Upsilon(r_2) * \mathsf{true}$. 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. This 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. The precondition for `SIGNAL_H` is $(\mathtt{p}_{21}, \mathtt{g}_{21})$. `SIGNAL_H` requires the thread owns the lock and the condition $\Upsilon(r_2)$ holds at the beginning. When it returns, the thread still owns the lock, but the condition may no longer hold.

Brinch Hansen style condition variables are similar to Hoare-style. The wait function is the same as `WAIT_H`. The signal function `SIGNAL_BH` is specified by $(\mathtt{p}_{31}, \mathtt{g}_{31})$ defined in Fig. 26. Here $\mathtt{p}_{31}$ is the same as $\mathtt{p}_{21}$. The definition of $\mathtt{g}_{31}$ shows the difference: the lock is released when signal returns. Therefore, calling the signal function must be the last command in the critical region.

Mesa-style. Fig. 27 shows Mesa-style condition variables. `WAIT_M` is specified by $(\mathtt{p}_{11}, \mathtt{g}_{11})$. The assertion $\mathtt{p}_{11}$ is similar to the precondition for Hoare-style, except that we require $\Delta(r_2) = \mathsf{emp}$. Therefore, as $\mathtt{g}_{11}$ shows, the current thread has no idea about the validity of the condition when it returns.

`SIGNAL_M` is specified by $(\mathtt{p}_{21}, \mathtt{g}_{21})$. The assertion hid is defined in Fig. 22, which means the function has no effects over data heap. 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. We do not require the current thread to own the lock l before it calls `SIGNAL_M`, since it has no effects over data heap. Intermediate assertions for these examples are given in the TR.

6. Implementations and Further Extensions

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 (Coq 2006). In our implementation, we assume that all interrupts except the timer have been masked. Soundness of the program logic is proved in an OCAP-like (Feng et al. 2007b) framework: inference rules are proved as lemmas in the foundational framework; the soundness of the framework itself is then proved following the syntactic approach. The proof is also formalized in Coq and is machine-checkable.

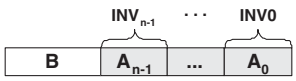
Our preemptive thread libraries (shown in Fig. 3) are also implemented in the x86 assembly code and works in real-mode. Synchronization primitives at Level C are certified using the AIM program logic. The timer handler calls the scheduler implemented at the low-level, which corresponds to the switch 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 in Fig. 3 are

certified as sequential code. Linking of the certified low-level code with the middle-level libraries and the timer handler is done in the OCAP-like framework. Linking of the thread library code at the middle level with the high-level concurrent programs (see Fig. 1) can be done in a similar way and is left as future work.

The Coq implementation has taken many man-months, out of which a significant amount of efforts has been put on the implementation of basic facilities, including lemmas and tactics for partial mappings, heaps, queues, and Separation Logic assertions. These common facilities are independent of the task of certifying thread libraries and can be reused in future projects. The size of the proof scripts, in terms of the number of lines of Coq tactics, is huge compared with the size of the x86 code. For instance, the proof for the Mesa-style condition variables (26 lines of x86 code) is around 5400 lines, including comments and white spaces. However, as observed by McCreight et al. (2007), the length of proof is probably a poor metric of complexity. There is a lot of redundancy in the proof: when an instruction is seen a second time in the code, we simply copy and paste the previous proof, and do some minor changes. We hope the length of the proof can be greatly reduced given better abstractions and tactics. Also, the 5400 line of proof was finished only in two days by one of the authors, who is an experienced Coq user. We believe this is a very reasonable price to pay for fully certified subroutines with machine checkable proofs. The whole Coq implementation is released as part of the TR (Feng et al. 2007c).

Extensions and future work. 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 $\mathsf{Inv}(\mathtt{ie},\mathtt{is})$ in Fig. 19 into $\mathsf{Inv}(\mathtt{ie},\mathsf{ivec})$, 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" 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 $(\mathsf{p}_i,\mathsf{g}_i)$.

Also, we do not consider dynamic creation of threads and block queues in this paper. In our previous work (Feng and Shao 2005), 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. Gotsman et al. (2007) and Hobor et al. (2008) extended concurrent separation logic with dynamic creation of 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.

There are other possible extensions of the program logic to increase its expressiveness. Bornat et al. (2005) showed refinements of Separation Logic assertions to distinguish read-only access and read/write access of memory cells. The refinements can be applied to our program logic to support verification of reader/writer locks. Also, we can change the current invariant-based specifications for the well-formedness of shared memory into rely-guarantee style specifications, where assertions specify transitions of states and are more expressive than invariants (Feng et al. 2007a; Vafeiadis and Parkinson 2007).

7. Related Work and Conclusion

Regehr and Cooprider (2007) showed how to translate interrupt-driven programs to thread-based programs. However, their technique cannot be directly applied for our goal to build certified OS kernel. 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 a formal semantics of interrupts. Our work actually provides such formal semantics. Second, their translation requires higher-level language constructs such as locks, while we certify the implementation of locks based on our AIM.

Suenaga and Kobayashi (2007) 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 interrupts. Our AIM is at a lower abstraction level than theirs with no built-in locks. Also, 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 (2002) 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 space. DeLine and Fähndrich (2001) showed how to enforce protocols with regard to interrupts levels as an application of Vault's type system. However, it is not clear how to use the type system for general properties of interrupts.

Bevier (1989) certified Kit, an OS kernel implemented in machine code. Gargano et al. (2005) showed a framework for a certified OS kernel in the Verisoft project. Ni et al. (2007) certified a non-preemptive thread implementation. In all these cases, implementations of kernels or thread libraries are all sequential. They cannot be interrupted and there is no preemptive concurrency.

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 ef-

fects 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.

Acknowledgments

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 grant CCR-0524545. Yuan Dong is supported in part by National Natural Science Foundation of China (under grant No. 60573017), China Scholarship Council, and Basic Research Foundation of Tsinghua National Laboratory for Information Science and Technology (TNList). Yu Guo is supported in part by grants from National Natural Science Foundation of China (under grants No. 60673126 and No. 90718026) and Intel China Research Center. Any opinions, findings, and contributions contained in this document are those of the authors and do not reflect the views of these agencies.

References

William R. Bevier. Kit: A study in operating system verification. *IEEE Trans. Softw. Eng.*, 15(11):1382–1396, 1989.

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proc. 32nd ACM Symp. on Principles of Prog. Lang.*, pages 259–270, January 2005.

Per Brinch Hansen. The programming language concurrent pascal. *IEEE Trans. Software Eng.*, 1(2):199–207, 1975.

Stephen Brookes. A semantics for concurrent separation logic. In *Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 16–34. Springer, September 2004.

Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.1, 2006.

Rober DeLine and Manual Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. 2001 ACM Conf. on Prog. Lang. Design and Impl.*, pages 59–69. ACM Press, June 2001.

Xinyu Feng and Zhong Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proc. 2005 ACM Int'l Conf. on Functional Prog.*, pages 254–267, September 2005.

Xinyu Feng and Zhong Shao. Local reasoning and information hiding in SCAP. Technical Report YALEU/DCS/TR-1398, Dept. of Computer Science, Yale University, New Haven, CT, February 2008. `http://flint.cs.yale.edu/publications/SCAPFrame.html`.

Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular verification of assembly code with stack-based control abstractions. In *Proc. 2006 ACM Conf. on Prog. Lang. Design and Impl.*, pages 401–414. ACM Press, June 2006.

Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Proc. 16th European Symp. on Prog. (ESOP'07)*, volume 4421 of *LNCS*, pages 173–188. Springer, March 2007a.

Xinyu Feng, Zhaozhong Ni, Zhong Shao, and Yu Guo. An open framework for foundational proof-carrying code. In *Proc. 2007 ACM Workshop on Types in Lang. Design and Impl.*, pages 67–78, January 2007b.

Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. Technical Report YALEU/DCS/TR-1396 and Coq Implementations, Dept. of Computer Science, Yale University, New Haven, CT, November 2007c. `http://flint.cs.yale.edu/publications/aim.html`.

Mauro Gargano, Mark A. Hillebrand, Dirk Leinenbach, and Wolfgang J. Paul. On the correctness of operating system kernels. In *Proc. 18th Int'l Conf. on Theorem Proving in Higher Order Logics*, volume 3603 of *LNCS*, pages 1–16, August 2005.

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *Proc. 5th ASIAN Symp. on Prog. Lang. and Sys. (APLAS'07)*, pages 19–37, 2007.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 26(1):53–56, October 1969.

C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

C. A. R. Hoare. Towards a theory of parallel programming. In *Operating Systems Techniques*, pages 61–71. Academic Press, 1972.

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proc. 17th European Symp. on Prog. (ESOP'08)*, page to appear, 2008.

Galen C. Hunt and James R. Larus. Singularity design motivation. Technical Report MSR-TR-2004-105, Microsoft Corporation, December 2004.

Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM Symp. on Principles of Prog. Lang.*, pages 14–26. ACM Press, January 2001.

Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2):105–117, 1980.

Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. In *Proc. 2007 ACM Conf. on Prog. Lang. Design and Impl.*, pages 468–479. ACM Press, June 2007.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Proc. 25th ACM Symp. on Principles of Prog. Lang.*, pages 85–97. ACM Press, January 1998.

Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic systems code: Machine context management. In *Proc. 20th Int'l Conf. on Theorem Proving in Higher Order Logics*, volume 4421 of *LNCS*, pages 189–206. Springer, September 2007.

Peter W. O'Hearn. Resources, concurrency and local reasoning. In *Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 49–67. Springer, September 2004.

Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proc. 31th ACM Symp. on Principles of Prog. Lang.*, pages 268–280. ACM Press, January 2004.

Jens Palsberg and Di Ma. A typed interrupt calculus. In *Proc. 7th Int'l Symp. on Formal Tech. in Real-Time and Fault-Tolerant Sys. (FTRTFT'02)*, volume 2469 of *LNCS*, pages 291–310, September 2002.

Wolfgang Paul, Manfred Broy, and Thomas In der Rieden. The Verisoft XT project. URL: `http://www.verisoft.de`, 2007.

John Regehr and Nathan Cooprider. Interrupt verification via thread verification. *Electron. Notes Theor. Comput. Sci.*, 174(9), 2007.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th Annual IEEE Symp. on Logic in Comp. Sci. (LICS'02)*, pages 55–74. IEEE Computer Society, July 2002.

Kohei Suenaga and Naoki Kobayashi. Type based analysis of deadlock for a concurrent calculus with interrupts. In *Proc. 16th European Symp. on Prog. (ESOP'07)*, volume 4421 of *LNCS*, pages 490–504, March 2007.

Harvey Tuch, Gerwin Klein, and Gernot Heiser. OS verification — now! In *Proc. 10th Workshop on Hot Topics in Operating Systems*, June 2005.

Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *Proc. 18th Int'l Conf. on Concurrency Theory (CONCUR'07)*, volume 4703 of *LNCS*, pages 256–271, September 2007.