Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads

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

[†]Yale University [‡]Toyota Technological Institute at Chicago

[§]Tsinghua University [*]University of Science and Technology of China

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.

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 (e.g., Verisoft/XT [8, 19], L4.verified [23], Singularity [12]), 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 statement about system configuration and behavior [12].

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 OS kernel into two layers. At the "higher" abstraction level, we have threads that follow the standard concurrent programming model [10]: 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

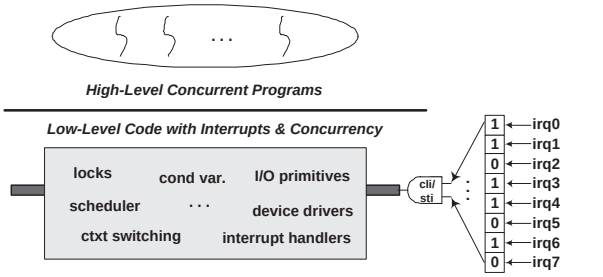


Figure 1. "High-Level" vs. "Low-Level" System Programs

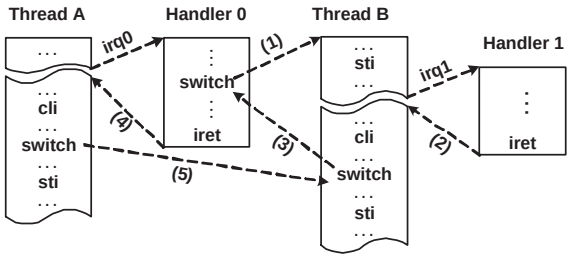


Figure 2. Interaction between Threads and Interrupts

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, typed assembly language [15], concurrent separation logic [17, 3], and its assume-guarantee variant [24]) 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 [20].
- *Subtle intertwining between interrupts and threads.* In Fig 2, thread A is interrupted by irq0 (say, the timer). 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

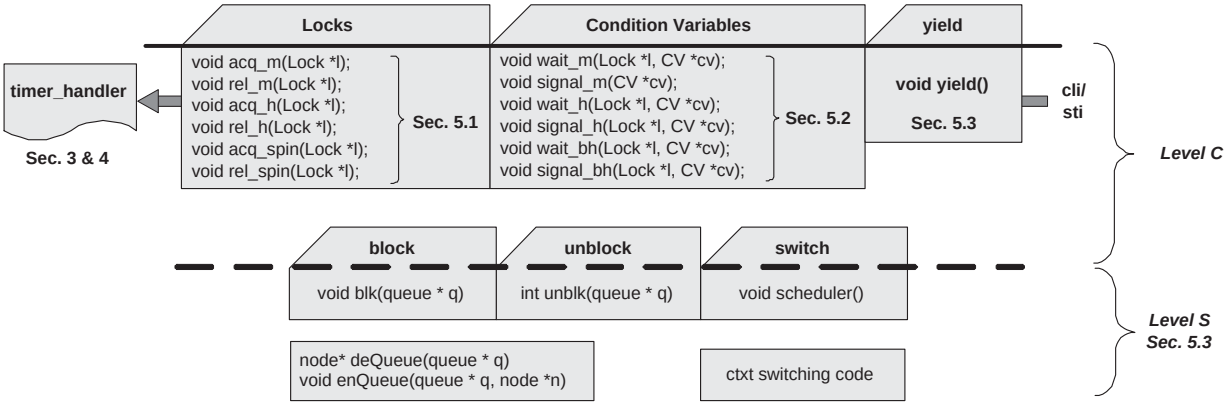


Figure 3. Structure of Our Certified Preemptive Thread Implementation

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 manipulates 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 are linked together using an OCAP-style framework [5].
- 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 Figure 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 interrupt explicitly and implements the interrupt handler but abstracts away the implementation of threads. Code at Level S is sequential (always executed with interrupt disabled); functions that require to know the concrete representations of thread control blocks (TCBs) and thread queues are implemented at Level S; there is 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 cli/sti to enable/disable interrupts (same as on x86 processors); the primitives switch, block and unblock are also treated as instructions; thread queues are now abstract algebraic structures outside of data heap and can only be accessed via the thread primitives.

Although the abstract machine is based on our implementation of the preemptive thread library, the idea of interfacing using Levels C and S is very general, given that the thread primitives we use are very common in most implementations of thread libraries. Note that we are not trying to claim this is the only way or the best way to divide code into separate abstraction levels, but this design does give us a nice abstraction at Level C so that we can focus on the interaction between threads and interrupts.

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 [3, 17]: *there al-*

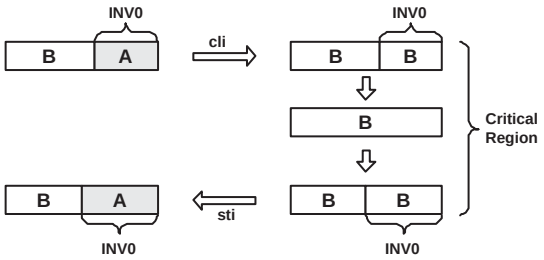


Figure 4. Memory Partition for Handler and Non-Handler

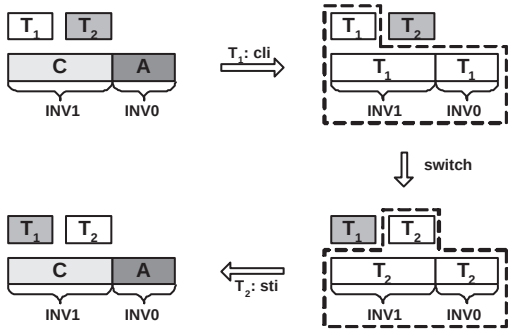


Figure 5. The Memory Model for Multi-Threaded Non-Handler

ways 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 [13, 21], 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 transfer. This semantics completely hides thread queues and thus the complex interleaving between the non-handler threads and the handler code.

We first study the semantics of cli and sti assuming that the non-handler code is always sequential. 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 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

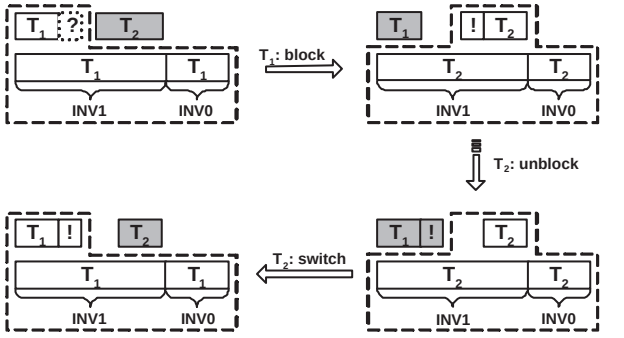


Figure 6. Block and Unblock

are only two threads): each thread has its own private memory, and both threads share the block C. When block C is available for share, 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 block C. 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 $\mathtt{T}_1$, needs to disable the interrupt; so cli grants $\mathtt{T}_1$ the ownership of well-formed blocks A and C. If $\mathtt{T}_1$ wants to switch control to $\mathtt{T}_2$, it first makes sure that INV0 and INV1 hold over A and C respectively. When $\mathtt{T}_2$ takes control, it can either access A and C, or enable the interrupt and release their ownership (knowing they are well-formed).

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 transfer: 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 $\mathtt{T}_1$ executes block when it waits for some resource (represented as the dashed box containing "?"). Since block 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'll 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 Figure 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. Data heap is modeled as a *finite partial* mapping from labels to integers. The register file is a total function

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

Figure 7. Definition of AIM-1

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

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

$$
\mathbb{S}|_{\mathbb{H}'} \triangleq (\mathbb{H}', \mathbb{S}.\mathbb{R}, \mathbb{S}.\mathtt{ie}, \mathbb{S}.\mathtt{is})
$$

$$
\mathbb{S}|_{\mathbb{R}'} \triangleq (\mathbb{S}.\mathbb{H}, \mathbb{R}, \mathbb{S}.\mathtt{ie}, \mathbb{S}.\mathtt{is})
$$

$$
\mathbb{S}|_{\{\mathtt{ie}=\mathtt{b}\}} \triangleq (\mathbb{S}.\mathbb{H}, \mathbb{S}.\mathbb{R}, \mathtt{b}, \mathbb{S}.\mathtt{is})
$$

$$
\mathbb{S}|_{\{\mathtt{is}=\mathtt{b}\}} \triangleq (\mathbb{S}.\mathbb{H}, \mathbb{S}.\mathbb{R}, \mathbb{S}.\mathtt{ie}, \mathtt{b})
$$

Figure 8. Definition of Representations

which maps register names to integers. The binary flags `ie` and `is` record whether the interrupt is disabled, and whether it is currently being serviced, respectively. The abstract control stack $\mathbb{K}$ saves the return address of the current function or the interrupt handler. Each stack frame either contains a code label `f` or a pair $(\mathtt{f},\mathbb{R})$. The frame $(\mathtt{f},\mathbb{R})$ is pushed when the interrupt is processed, which will be explained below. An empty stack is represented as nil. The program counter `pc` points to the current command in $\mathbb{C}$. 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 Figure 8. We use 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 Figure 8.

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 the Section 4.6. An incoming interrupt is processed only if the `ie` bit is set, and no interrupt is currently being serviced (*i.e.,* $\mathtt{is} = 0$). The processor handles the interrupt in the following steps:

- pushes the current `pc` and register file $\mathbb{R}$ onto the stack $\mathbb{K}$;
- clears the `ie` bit and sets the `is` bit;
- sets the `pc` to `h_entry`.

The state transition $(\mathbb{W} \lightning \mathbb{W}')$ is defined in the IRQ rule in Fig 9.

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

$\mathtt{NextK}_{(\mathtt{pc},\mathtt{c})}\ \mathbb{K}\ \mathbb{K}'$	
if $\mathtt{c} =$	$\mathbb{S}' =$
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 $\mathtt{c} =$	$\mathtt{pc}' =$
$\mathsf{beq}\ r_s, r_t, \mathtt{f}$	`f` if $\mathbb{R}(r_s) = \mathbb{R}(r_t)$
$\mathsf{beq}\ r_s, r_t, \mathtt{f}$	$\mathtt{pc}+1$ if $\mathbb{R}(r_s) \neq \mathbb{R}(r_t)$
call `f`	`f`
j `f`	`f`
$\mathsf{jr}\ r_s$	$\mathbb{R}(r_s)$
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	$\mathtt{pc}+1$

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

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

$$
\mathbb{W} \longmapsto \mathbb{W}' \triangleq (\mathbb{W} \longmapsto \mathbb{W}') \vee (\mathbb{W} \lightning \mathbb{W}')
$$

Figure 9. Operational Semantics

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

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

```
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

In AIM, the register file $\mathbb{R}$ is automatically saved and restored at the entry and exit point of the interrupt handler. This is a simplification of the x86 interrupt mechanism for a cleaner presentation and is not essential for the technical development. In our implementation for x86, 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 $\mathbb{W} \Longmapsto \mathbb{W}'$. Note that our semantics of AIM-1 programs is not deterministic: the state transition may be made either by executing the next instruction or by handling an incoming interrupt. Also, given a $\mathbb{W}$, there may not always exist a $\mathbb{W}'$ such that $(\mathbb{W} \longmapsto \mathbb{W}')$ holds. If there is no such $\mathbb{W}'$, we say the program gets stuck at $\mathbb{W}$. One important goal of our program logic is to show that certified programs never get stuck.

Fig 10 shows a sample AIM-1 program. The program specifications in shadowed boxes are explained in Section 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 interrupt before it accesses `LEFT` and `RIGHT`.

3.2 AIM-2

Fig 11 defines AIM-2 as an extension over AIM-1. We extend 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

```
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
```

$$
\begin{array}{l}
\mathtt{p}_0 \triangleq \mathsf{enable}_{\mathsf{iret}} \wedge (r_1 = \mathtt{CNT}) \\
\mathtt{g}_0 \triangleq \left\{ \begin{array}{l} \mathtt{CNT} \mapsto - \\ \mathsf{INV0} \end{array} \right\} \wedge (\mathtt{ie} = \mathtt{ie}') \wedge (\mathtt{is} = \mathtt{is}')
\end{array}
$$

Figure 13. A Preemptive Timer Handler

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:

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

We also add three primitive instructions: switch, block and unblock. They correspond to system calls to low-level thread implementations in the real machine (see Fig 3).

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 randomly picks a new thread from $\mathtt{readyQ}(\mathbb{T}, \mathbb{B})$. To let our abstraction fit into the interfaces shown in Fig 3, we require that the interrupt be disabled before switch. This also explains why `ie` is not saved in the thread context, and why it is set to 0 when a new thread is scheduled from $\mathbb{T}$: the only way to switch control from one thread to the other is to execute switch, which can be executed only if the interrupt is disabled. The "block r_t" instruction puts the current thread id into the block queue $\mathbb{B}(r_t)$, and switches the control to another thread in $\mathtt{readyQ}(\mathbb{T}, \mathbb{B})$. If there are no other threads in `readyQ`, the machine stutters (in our x86 implementation, this would never happen because there is an idle thread and our program logic prohibits it from executing block). The "unblock r_t, r_d" instruction removes a thread from $\mathbb{B}(r_t)$ and puts its `tid` into r_d if the queue is not empty; otherwise r_d contains 0. 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 $(\mathbb{W} \not\longmapsto \mathbb{W}')$ for AIM-2 is almost the same as the one for AIM-1 defined by the IRQ rule. It does not change $\mathbb{T}$ and `tid` either. The definition of $(\mathbb{W} \Longmapsto \mathbb{W}')$ is unchanged.

A preemptive timer interrupt handlers The design of the low-level AIM machine is very interesting in that it provides different choices of thread models for high-level concurrent programs (Figure 1). If high-level threads assumes the interrupt is always disabled, they work in the non-preemptive model and each thread voluntarily gives up the control by switch. However, threads cannot handle interrupts from I/O devices either in this case. An alternative approach to the non-preemptive model is to install an interrupt handler implemented in AIM that does no context switching. To

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

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

Figure 14. Specification Constructs

get the preemptive model, we install a timer interrupt handler that executes switch.

Figure 13 shows the implementation of a preemptive interrupt handler for the timer. Each time the interrupt comes, the hander test 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 meaning of specifications and how the timer handler is be certified in Section 4.

4. The Program Logic

4.1 Specification Language

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

As shown in Figure 14, the specification Ψ for the code heap $\mathbb{C}$ associates code labels $\mathtt{f}$ with specifications $\mathtt{s}$. We allow each $\mathtt{f}$ to have more than one $\mathtt{s}$, just as a function may have multiple specified interfaces. The specification $\mathtt{s}$ is a pair $(\mathtt{p},\mathtt{g})$. The assertion $\mathtt{p}$ is a predicate over a stack $\mathbb{K}$ and a program state $\mathbb{S}$ (its meta-type in Coq is a function that takes $\mathbb{K}$ and $\mathbb{S}$ as arguments and returns a proposition), while $\mathtt{g}$ is a predicate over two program states. As we can see, the $\mathtt{NextS}_{(\mathtt{c},\mathbb{K})}$ relation defined in Figure 9 is a special form of $\mathtt{g}$. Following our previous work on reasoning low-level code with stack based control abstractions [7], we use $\mathtt{p}$ to specify the precondition over stack and state, and use $\mathtt{g}$ to specify the guaranteed behavior from the specified program point to the point when the *current* function returns.

We also use the predicate $\mathtt{m}$ to specify data heaps. The invariants INV0 and INV1 shown in Figures 5 and 6 are both heap predicates. We encode in Figure 15 Separation Logic connectors [13, 21] in our specification language. Assertions in Separation Logic capture

$\mathsf{true} \triangleq \lambda\mathbb{H}.\ \mathsf{True}$ $\qquad$ $\mathsf{emp} \triangleq \lambda\mathbb{H}.\ \mathbb{H}=\varnothing$

$\mathtt{l}\mapsto\mathtt{w} \triangleq \lambda\mathbb{H}.\ \mathbb{H}=\{\mathtt{l}\rightsquigarrow\mathtt{w}\}$ $\qquad$ $\mathtt{l}\mapsto\_ \triangleq \lambda\mathbb{H}.\ \exists\mathtt{w}.\ (\mathtt{l}\mapsto\mathtt{w})\ \mathbb{H}$

$\mathbb{H}_1\perp\mathbb{H}_2 \triangleq dom(\mathbb{H}_1)\cap dom(\mathbb{H}_1)=\varnothing$

$$\mathbb{H}_1\uplus\mathbb{H}_2 \triangleq \begin{cases}\mathbb{H}_1\cup\mathbb{H}_2 & \text{if } \mathbb{H}_1\perp\mathbb{H}_2\\ \textit{undefined} & \text{otherwise}\end{cases}$$

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

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

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

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

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

Figure 15. Definitions of Separation Logic Assertions

$\mathsf{enable}(\mathtt{p},\mathtt{g}) \triangleq \forall\mathbb{K},\mathbb{S}.\ \mathtt{p}\ \mathbb{K}\ \mathbb{S}\rightarrow\exists\mathbb{S}',\mathtt{g}\ \mathbb{S}\ \mathbb{S}'$

$\mathtt{p}\rhd\mathtt{g} \triangleq \lambda\mathbb{K},\mathbb{S}.\ \exists\mathbb{S}_0,\mathtt{p}\ \mathbb{K}\ \mathbb{S}_0\wedge\mathtt{g}\ \mathbb{S}_0\ \mathbb{S}$

$\mathtt{g}\circ\mathtt{g}' \triangleq \lambda\mathbb{S},\mathbb{S}''.\ \exists\mathbb{S}'.\ \mathtt{g}\ \mathbb{S}\ \mathbb{S}'\wedge\mathtt{g}'\ \mathbb{S}'\ \mathbb{S}''$

$\mathtt{p}\circ\mathtt{g} \triangleq \lambda\mathbb{S},\mathbb{S}'.\ \exists\mathbb{K}.\ \mathtt{p}\ \mathbb{K}\ \mathbb{S}\wedge\mathtt{g}\ \mathbb{S}\ \mathbb{S}'$

$\mathtt{p}\Rightarrow\mathtt{p}' \triangleq \forall\mathbb{K},\mathbb{S}.\ \mathtt{p}\ \mathbb{K}\ \mathbb{S}\rightarrow\mathtt{p}'\ \mathbb{K}\ \mathbb{S}$

$\mathtt{g}\Rightarrow\mathtt{g}' \triangleq \forall\mathbb{S},\mathbb{S}'.\ \mathtt{g}\ \mathbb{S}\ \mathbb{S}'\rightarrow\mathtt{g}'\ \mathbb{S}\ \mathbb{S}'$

Figure 16. Connectors for $\mathtt{p}$ and $\mathtt{g}$

ownership of heaps. The assertion "$\mathtt{l}\mapsto n$" holds only if the heap has only one cell at $\mathtt{l}$ containing value 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 $\mathtt{m}$ and $\mathtt{m}'$ hold over one of them respectively. A heap satisfies $\mathtt{m}\mathrel{-\!\!*}\mathtt{m}'$ if and only if the disjoint union of it with any heap satifying $\mathtt{m}$ would satisfy $\mathtt{m}'$.

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

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

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

$$\mathsf{g}_{\mathsf{cli}} \triangleq \lambda(\mathbb{H},\mathbb{R},\mathtt{ie},\mathtt{is}),(\mathbb{H}',\mathbb{R}',\mathtt{ie}',\mathtt{is}').\ (\mathtt{is}=\mathtt{is}') \wedge (\mathbb{R}=\mathbb{R}') \wedge (\mathtt{ie}'=0) \wedge \left\{ \begin{array}{l} \mathsf{emp} \\ (\mathtt{ie}=1 \wedge \mathtt{is}=0)\ ?\ (\mathsf{INV0}*\mathsf{INV1}){:}\mathsf{emp} \end{array} \right\} \mathbb{H}\,\mathbb{H}'$$

$$\mathsf{g}_{\mathsf{sti}} \triangleq \lambda(\mathbb{H},\mathbb{R},\mathtt{ie},\mathtt{is}),(\mathbb{H}',\mathbb{R}',\mathtt{ie}',\mathtt{is}').\ (\mathtt{is}=\mathtt{is}') \wedge (\mathbb{R}=\mathbb{R}') \wedge (\mathtt{ie}'=1) \wedge \left\{ \begin{array}{l} (\mathtt{ie}=0 \wedge \mathtt{is}=0)\ ?\ (\mathsf{INV0}*\mathsf{INV1}){:}\mathsf{emp} \\ \mathsf{emp} \end{array} \right\} \mathbb{H}\,\mathbb{H}'$$

$$\mathsf{g}_{\mathsf{switch}} \triangleq \lambda(\mathbb{H},\mathbb{R},\mathtt{ie},\mathtt{is}),(\mathbb{H}',\mathbb{R}',\mathtt{ie}',\mathtt{is}').\ (\mathtt{ie}=0) \wedge (\mathtt{ie}=\mathtt{ie}') \wedge (\mathbb{R}=\mathbb{R}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \left\{ \begin{array}{l} \mathsf{INV0}*(\mathtt{is}=0\ ?\ \mathsf{INV1}{:}\mathsf{emp}) \\ \mathsf{INV0}*(\mathtt{is}=0\ ?\ \mathsf{INV1}{:}\mathsf{emp}) \end{array} \right\} \mathbb{H}\,\mathbb{H}'$$

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

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

Figure 17. Semantics for `cli`, `sti` and Thread Primitives

g_{i} are defined as follows:

$$\mathsf{p}_{\mathsf{i}} \triangleq \lambda\mathbb{K},\mathbb{S}.\ ((\mathsf{INV0}*\mathsf{true})\,\mathbb{S}.\mathbb{H}) \wedge (\mathbb{S}.\mathtt{is}=1) \wedge (\mathbb{S}.\mathtt{ie}=0) \wedge \exists\mathtt{f},\mathbb{R},\mathbb{K}'.\ \mathbb{K}=(\mathtt{f},\mathbb{R})::\mathbb{K}' \quad (1)$$

$$\mathsf{g}_{\mathsf{i}} \triangleq \lambda\mathbb{S},\mathbb{S}'.\ \left\{ \begin{array}{l} \mathsf{INV0} \\ \mathsf{INV0} \end{array} \right\} \mathbb{S}.\mathbb{H}\,\mathbb{S}'.\mathbb{H} \wedge (\mathbb{S}'.\mathtt{ie}=\mathbb{S}.\mathtt{ie}) \wedge (\mathbb{S}'.\mathtt{is}=\mathbb{S}.\mathtt{is}) \quad (2)$$

The precondition p_{i} specifies the stack and state at the entry `h_entry`. It requires that the local heap used by the handler (block A in Figure 5) satisfies INV0. INV0 is a global parameter of our system, whose definition depends on the functionality of the interrupt handler. The guarantee g_{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.

$$\left\{ \begin{array}{l} \mathtt{m}_1 \\ \mathtt{m}_2 \end{array} \right\} \triangleq \lambda\mathbb{H}_1,\mathbb{H}_2.\ \exists\mathbb{H}'_1,\mathbb{H}'_2,\mathbb{H}.\ (\mathtt{m}_1\,\mathbb{H}'_1) \wedge (\mathtt{m}_2\,\mathbb{H}'_2) \wedge (\mathbb{H}'_1 \uplus \mathbb{H} = \mathbb{H}_1) \wedge (\mathbb{H}'_2 \uplus \mathbb{H} = \mathbb{H}_2) \quad (3)$$

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 Figs. 18 and 20. The judgment for well-formed instruction sequences says it is safe to execute the instruction sequence given the imported interface in Ψ, the specification of block queues Δ, and a precondition (p,g).

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

$$\frac{\iota \notin \{\mathsf{call}\ldots,\mathsf{beq}\ldots\} \quad \Psi,\Delta \vdash \{(\mathsf{p}',\mathsf{g}')\}\mathtt{f}+1 : \mathbb{I} \quad \mathsf{enable}(\mathsf{p},\mathsf{g}_\iota) \quad (\mathsf{p} \rhd \mathsf{g}_\iota) \Rightarrow \mathsf{p}' \quad (\mathsf{p} \circ (\mathsf{g}_\iota \circ \mathsf{g}')) \Rightarrow \mathsf{g}}{\Psi,\Delta \vdash \{(\mathsf{p},\mathsf{g})\}\mathtt{f} : \iota;\mathbb{I}}\ (\textsc{seq})$$

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

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

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

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

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

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

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

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

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

Figure 18. Inference Rules

The predicate p specifies the current stack and state, and g specifies the state transition from the current program point to the return point of the current function (or the interrupt handler).

The SEQ rule is a schema for instruction sequences starting with instructions except branch and function call instructions. We need to find an intermediate specification $(\mathsf{p}',\mathsf{g}')$, 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 g_ι to specify the state transition made by the instruction ι. The premise $\mathsf{enable}(\mathsf{p},\mathsf{g}_\iota)$ is defined in Figure 16. It means that the state transition g_ι would not get stuck as long as the starting stack and state satisfy p. The predicate $\mathsf{p} \rhd \mathsf{g}_\iota$, shown in Figure 16, specifies the stack and state resulting from the state transition g_ι, knowing the initial state satisfies p. It is the strongest post condition after g_ι. The composition of two subsequent transitions g and g' is represented as $\mathsf{g} \circ \mathsf{g}'$, and $\mathsf{p} \circ \mathsf{g}$ refines g with the extra knowledge that the initial state satisfies p. We also lift the implication relation between p's and g's. The last premise in the SEQ rule requires the composition of g_ι and g' fulfills g, knowing the current state satisfies p.

If ι is an arithmetic instruction, move instruction or memory operation, we define g_ι as $\mathtt{NextS}_{(\iota,\_)}$ (see Figure 9). Since `NextS` does not depend on the stack for these instructions, 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 $\mathsf{enable}(\mathsf{p},\mathsf{g}_\iota)$ requires that p contains the ownership of the target memory cell accessed by ld or st.

Interrupts and thread primitive instructions. One of the major technical contributions of this paper is our formulation of g_ι for cli, sti and switch, block and unblock, which, as shown in Figure 17, gives an axiomatic ownership transfer semantics to them.

The transition $\mathsf{g_{cli}}$ 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 Figure 5; otherwise there is no ownership transfer because the interrupt has already been disabled before cli. The transition $\mathsf{g_{sti}}$ is defined similarly. Note that the premise $\mathsf{enable}(\mathtt{p},\mathsf{g_i})$ in the SEQ requires that, before executing sti, the precondition $\mathtt{p}$ must contain the ownership $(\mathtt{ie}=0\wedge\mathtt{is}=0)\,?\,(\mathsf{INV1}*\mathsf{INV0}):\mathsf{emp}$.

The transition $\mathsf{g_{switch}}$ for switch requires that the sub-heap A and C (in Figure 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 $\mathsf{enable}(\mathtt{p},\mathsf{g_i})$ requires that the precondition $\mathtt{p}$ ensures $\mathtt{ie}=0$ and $\mathsf{INV0}*(\mathtt{is}=0\,?\,\mathsf{INV1}:\mathsf{emp})$ holds over some sub-heap.

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

Function calls and memory polymorphism. In the CALL rule, we treat the state transition g' made by the callee as the transition of the call instruction. We also require that the precondition $\mathtt{p}$ implies the precondition $\mathtt{p}'$ of the callee, which corresponds to the enable premise in the SEQ rule. Note that the CALL rule supports memory polymorphism in a natural way: if g' of the callee is specified following the pattern $\left\{\begin{array}{l}\mathtt{m}\\ \mathtt{m}'\end{array}\right\}$ defined by formula (3), the callee's specification does not need to mention data required by the caller but not accessed in the callee, thus achieving a similar effects to the frame rule in Separation Logic. We first introduce the following definitions.

$$\begin{array}{l}
\mathsf{monotonic}(\mathtt{p}) \triangleq\\
\quad \forall\mathbb{K},\mathbb{S},\mathbb{H},\mathbb{H}'.\ (\mathbb{H}'=\mathbb{H}\uplus\mathbb{S}.\mathbb{H})\wedge \mathtt{p}\ \mathbb{K}\ \mathbb{S}\rightarrow \mathtt{p}\ \mathbb{K}\ \mathbb{S}|_{\mathbb{H}'}\\
\mathsf{monotonic}(\mathsf{g}) \triangleq\\
\quad \forall\mathbb{S},\mathbb{S}',\mathbb{H},\mathbb{H}',\mathbb{H}_0.(\mathbb{H}_0\uplus\mathbb{S}.\mathbb{H}=\mathbb{H})\wedge(\mathbb{H}_0\uplus\mathbb{S}'.\mathbb{H}=\mathbb{H}')\\
\qquad\qquad \rightarrow(\mathsf{g}\ \mathbb{S}\ \mathbb{S}'\rightarrow \mathsf{g}\ \mathbb{S}|_{\mathbb{H}}\ \mathbb{S}'|_{\mathbb{H}'})\\
\mathsf{frame}(\mathsf{g}) \triangleq\\
\quad \forall\mathbb{S},\mathbb{S}',\mathbb{H}_0,\mathbb{H}.\ (\mathbb{H}_0\uplus\mathbb{H}=\mathbb{S}.\mathbb{H})\wedge(\mathsf{g}\ \mathbb{S}\ \mathbb{S}')\wedge(\exists\mathbb{S}''.\ \mathsf{g}\ \mathbb{S}|_{\mathbb{H}}\ \mathbb{S}'')\\
\qquad\qquad \rightarrow\exists\mathbb{H}'.(\mathbb{H}_0\uplus\mathbb{H}'=\mathbb{S}'.\mathbb{H})\wedge\mathsf{g}\ \mathbb{S}|_{\mathbb{H}}\ \mathbb{S}'|_{\mathbb{H}'}\\
\mathsf{wff_spec}(\mathtt{p},\mathsf{g}) \triangleq\\
\quad \mathsf{monotonic}(\mathtt{p})\wedge\mathsf{monotonic}(\mathsf{g})\wedge\mathsf{frame}(\mathsf{g})\wedge\mathsf{enable}(\mathtt{p},\mathsf{g})\\
\mathsf{wff_spec}(\Psi) \triangleq \forall\mathtt{f},\mathtt{s}.\ ((\mathtt{f},\mathtt{s})\in\Psi)\rightarrow\mathsf{wff_spec}(\mathtt{s})
\end{array}$$

Lemma 4.1 (call-frame)
If $\Psi,\Delta\vdash\{(\mathtt{p},\mathsf{g})\}\,\mathtt{f}:\mathsf{call}\ \mathtt{f}'\ \mathbb{I}$, $\mathsf{wff\_spec}(\Psi)$ and $\mathsf{wff\_spec}(\mathtt{p},\mathsf{g})$, then we have $\Psi,\Delta\vdash\{(\mathtt{p}*\mathtt{m},\mathsf{g})\}\,\mathtt{f}:\mathsf{call}\ \mathtt{f}'\ \mathbb{I}$ for any $\mathtt{m}$ such that $\mathsf{precise}(\mathtt{m})$.

Proof sketch. To prove the lemma, we need to prove the following propositions:

- if $\mathsf{monotonic}(\mathtt{p}')$, then $(\mathtt{p}\Rightarrow\mathtt{p}')\rightarrow(\mathtt{p}*\mathtt{m}\Rightarrow\mathtt{p}')$;
- if $\mathtt{p}\Rightarrow\mathtt{p}'$, $\mathsf{enable}(\mathtt{p}',\mathsf{g}')$, $\mathsf{frame}(\mathsf{g}')$ and $\mathsf{monotonic}(\mathtt{p}'')$, then $(\mathtt{p}\rhd\mathsf{g}'\Rightarrow\mathtt{p}'')\rightarrow((\mathtt{p}*\mathtt{m})\rhd\mathsf{g}'\Rightarrow\mathtt{p}'')$;

$$\begin{array}{l}
\mathtt{p}*\mathsf{Inv} \triangleq \lambda\mathbb{K},\mathbb{S}.\ (\mathtt{p}*\mathsf{Inv}(\mathbb{S}.\mathtt{ie},\mathbb{S}.\mathtt{is}))\ \mathbb{K}\ \mathbb{S}\\
\mathsf{Inv}(\mathtt{ie},\mathtt{is}) \triangleq \left\{\begin{array}{ll}\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{array}\right.\\
\text{where } \mathsf{INV}_s \triangleq \mathsf{INV0}*\mathsf{INV1}\\
\lfloor\mathsf{g}\rfloor_{(\mathtt{m},\mathtt{m}')} \triangleq \lambda\mathbb{S},\mathbb{S}'.\ \exists\mathbb{H}_1,\mathbb{H}_2,\mathbb{H}'_1,\mathbb{H}'_2.\\
\qquad (\mathbb{H}_1\uplus\mathbb{H}_2=\mathbb{S}.\mathbb{H})\wedge(\mathbb{H}'_1\uplus\mathbb{H}'_2=\mathbb{S}'.\mathbb{H})\\
\qquad\wedge\mathtt{m}\ \mathbb{H}_2\wedge\mathtt{m}'\ \mathbb{H}'_2\wedge\mathsf{g}\ \mathbb{S}|_{\mathbb{H}_1}\ \mathbb{S}'|_{\mathbb{H}'_1}\\
\lfloor\mathsf{g}\rfloor \triangleq \lambda\mathbb{S},\mathbb{S}'.\ \lfloor\mathsf{g}\rfloor_{(\mathsf{Inv}(\mathbb{S}.\mathtt{ie},\mathbb{S}.\mathtt{is}),\mathsf{Inv}(\mathbb{S}'.\mathtt{ie},\mathbb{S}'.\mathtt{is}))}\\
\mathsf{WFST}(\mathsf{g},\mathbb{S},\mathsf{nil},\Psi) \triangleq \neg\exists\mathbb{S}'.\ \mathsf{g}\ \mathbb{S}\ \mathbb{S}'\\
\mathsf{WFST}(\mathsf{g},\mathbb{S},\mathtt{f}::\mathbb{K},\Psi) \triangleq\\
\quad \exists\mathtt{p}_\mathtt{f},\mathsf{g}_\mathtt{f}.\ (\mathtt{f},\ (\mathtt{p}_\mathtt{f},\mathsf{g}_\mathtt{f}))\in\Psi\\
\quad\ \wedge\forall\mathbb{S}'.\ \mathsf{g}\ \mathbb{S}\ \mathbb{S}'\rightarrow(\mathtt{p}_\mathtt{f}*\mathsf{Inv})\ \mathbb{K}\ \mathbb{S}'\wedge\mathsf{WFST}(\lfloor\mathsf{g}_\mathtt{f}\rfloor,\mathbb{S}',\mathbb{K},\Psi)\\
\mathsf{WFST}(\mathsf{g},\mathbb{S},(\mathtt{f},\mathbb{R})::\mathbb{K},\Psi) \triangleq\\
\quad \exists\mathtt{p}_\mathtt{f},\mathsf{g}_\mathtt{f}.\ (\mathtt{f},\ (\mathtt{p}_\mathtt{f},\mathsf{g}_\mathtt{f}))\in\Psi\\
\quad\ \wedge\forall\mathbb{S}'.\ \mathsf{g}\ \mathbb{S}\ \mathbb{S}'\rightarrow(\mathtt{p}_\mathtt{f}*\mathsf{Inv})\ \mathbb{K}\ \mathbb{S}''\wedge\mathsf{WFST}(\lfloor\mathsf{g}_\mathtt{f}\rfloor,\mathbb{S}'',\mathbb{K},\Psi)\\
\quad \text{where } \mathbb{S}''=(\mathbb{S}'.\mathbb{H},\mathbb{R},1,0)
\end{array}$$

Figure 19. Auxiliary Definitions for Program Invariants

- if $\mathtt{p}\Rightarrow\mathtt{p}'$, $(\mathtt{p}\rhd\mathsf{g}'\Rightarrow\mathtt{p}'')$, $\mathsf{enable}(\mathtt{p}',\mathsf{g}')$, $\mathsf{frame}(\mathsf{g}')$, $\mathsf{enable}(\mathtt{p}'',\mathsf{g}'')$, $\mathsf{frame}(\mathsf{g}'')$, and $\mathsf{monotonic}(\mathsf{g})$, then $((\mathtt{p}\circ(\mathsf{g}'\circ\mathsf{g}''))\Rightarrow\mathsf{g})\rightarrow(((\mathtt{p}*\mathtt{m})\circ(\mathsf{g}'\circ\mathsf{g}''))\Rightarrow\mathsf{g})$.

Each proposition can be proved by above definitions. □

Also, we can prove $\mathsf{wff\_spec}(\mathtt{p},\mathsf{g})$ if:

$$\mathtt{p} \triangleq \lambda\mathbb{K},\mathbb{S}.\ \mathtt{m}*\mathsf{true}\ \mathbb{S}.\mathbb{H}$$
$$\mathsf{g} \triangleq \lambda\mathbb{S},\mathbb{S}'.\ \left\{\begin{array}{l}\mathtt{m}\\ \mathtt{m}'\end{array}\right\}\ \mathbb{S}.\mathbb{H}\ \mathbb{S}'.\mathbb{H},$$

for any $\mathtt{m}$, $\mathtt{m}'$ such that $\mathsf{precise}(\mathtt{m})$.

Other instructions. In the BEQ rule, we use $\mathsf{gid}_{r_s=r_t}$ to represent an identity transition with the extra knowledge that r_s and r_t contain the same value. $\mathsf{gid}_{r_s\neq r_t}$ is defined similarly. We do not have an enable premise because the branch instruction never gets stuck. IRET and RET rules require that the code 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. The J rule can be viewed as a specialization of the BEQ rule.

Well-formed code heaps. The CDHP rule says the code heap is well-formed if and only if each instruction sequence specified in Ψ' is well-formed.

Program Invariants. The program invariant enforced by our program logic is defined by the PROG rule in Figure 20, which is another major technical contribution of this work. It says that, if there are n threads in the thread queue in addition to the current thread, the heap can be split into $n+1$ blocks, each for one thread. Each block $\mathbb{H}_k$ ($k>0$) is for a ready or blocked thread in queues. The block $\mathbb{H}_0$ is assigned to the current thread, which includes both its private memory and the shared memory A and C shown in Figure 5. The code heap needs to be well-formed, defined by the CDHP rule. The domain of Δ should be the same with the domain of $\mathbb{B}$, *i.e.,* Δ only specifies block queues in $\mathbb{B}$.

The PROG rule also requires that the current thread, ready threads and blocked threads are all well-formed. The CTH rule defines the well-formedness of the current thread. It requires that the pc have a specification $(\mathtt{p},\mathsf{g})$ in Ψ'. By the well-formedness of the code

$$\frac{\begin{array}{l}\mathbb{T}\setminus \mathtt{tid} = \{\mathtt{tid}_1 \rightsquigarrow (\mathbb{R}_1,\mathbb{K}_1,\mathtt{is}_1,\mathtt{pc}_1),\ldots,\\ \qquad \mathtt{tid}_n \rightsquigarrow (\mathbb{R}_n,\mathbb{K}_n,\mathtt{is}_n,\mathtt{pc}_n)\}\\ \mathbb{S}.\mathbb{H} = \mathbb{H}_0 \uplus \ldots \uplus \mathbb{H}_n \quad \mathbb{S}_0 = \mathbb{S}|_{\mathbb{H}_0}\\ \Psi,\Delta \vdash \mathbb{C}:\Psi' \quad \Psi \subseteq \Psi' \quad dom(\Delta) = dom(\mathbb{B}) \quad \Psi' \vdash^{\mathrm{C}} (\mathbb{S}_0,\mathbb{K},\mathtt{pc})\\ \text{for all } 0<k\le n \text{ such that } \mathtt{tid}_k \in \mathtt{readyQ}(\mathbb{T},\mathbb{B}):\\ \qquad \Psi' \vdash^{\mathrm{R}} (\mathbb{H}_k,\mathbb{R}_k,\mathbb{K}_k,\mathtt{is}_k,\mathtt{pc}_k)\\ \text{for all } w,\mathtt{tid}_j \text{ such that } \mathtt{tid}_j \in \mathbb{B}(w):\\ \qquad \Psi',\Delta,w \vdash^{\mathrm{W}} (\mathbb{H}_j,\mathbb{R}_j,\mathbb{K}_j,\mathtt{is}_j,\mathtt{pc}_j)\end{array}}{\Psi,\Delta \vdash (\mathbb{C},\mathbb{S},\mathbb{K},\mathtt{pc},\mathtt{tid},\mathbb{T},\mathbb{B})}\ (\textsc{prog})$$

$$\frac{(\mathtt{pc},(\mathtt{p},\mathtt{g})) \in \Psi' \quad (\mathtt{p} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S} \quad \mathsf{WFST}(\lfloor \mathtt{g} \rfloor,\mathbb{S},\mathbb{K},\Psi')}{\Psi' \vdash^{\mathrm{C}} (\mathbb{S},\mathbb{K},\mathtt{pc})}\ (\textsc{cth})$$

$$\frac{\mathbb{S}_k = (\mathbb{H}_k,\mathbb{R}_k,0,\mathtt{is}_k) \quad (\mathsf{INV}_s \mathrel{-\!\!*} (\Psi' \vdash^{\mathrm{C}} (\_,\_,\mathtt{pc}_k)))\ \mathbb{K}_k\ \mathbb{S}_k}{\Psi' \vdash^{\mathrm{R}} (\mathbb{H}_k,\mathbb{R}_k,\mathbb{K}_k,\mathtt{is}_k,\mathtt{pc}_k)}\ (\textsc{rdy})$$

$$\frac{\Delta(w) = \mathtt{m} \quad (\mathtt{m} \mathrel{-\!\!*} (\Psi' \vdash^{\mathrm{R}} (\_,\mathbb{R}_j,\mathbb{K}_j,\mathtt{is}_j,\mathtt{pc}_j)))\ \mathbb{H}_j}{\Psi',\Delta,w \vdash^{\mathrm{W}} (\mathbb{H}_j,\mathbb{R}_j,\mathbb{K}_j,\mathtt{is}_j,\mathtt{pc}_j)}\ (\textsc{wait})$$

Figure 20. Inference Rules (cont'd)

heap we know PC points to a well-formed instruction sequence. Also $\mathtt{p} * \mathsf{Inv}$ holds over the stack and state with the sub-heap $\mathbb{H}_0$. The definition of $\mathtt{p} * \mathsf{Inv}$ is shown in Figure 19. It specifies the shared memory inaccessible from p. If the current program point is in the interrupt handler ($\mathtt{is} = 1$), p leaves the memory block C (in Figure 5) unspecified, therefore Inv requires it to satisfy INV1. Otherwise ($\mathtt{is} = 0$), if $\mathtt{ie} = 0$, memory C and A become the current thread's private memory and there is no memory to share. If $\mathtt{ie} = 1$, memory C and A are not accessible from p, therefore Inv requires them to be well-formed. The inductively defined predicate WFST is shown in Figure 19. It says there exists a well-formed stack with some depth k. At the end of the current function, when g is fulfilled, there must be a stack frame on top of the stack with a pc pointing to well-formed instruction sequences. Also there is a well-formed stack with depth $k-1$ at the returning state.

Since a judgment is represented as a proposition in our meta-logic, and we reuse the meta-logic as our specification language, we can define $\Psi' \vdash^{\mathrm{C}} (\_,\_,\mathtt{pc})$ as a special predicate p: $\lambda\mathbb{K},\mathbb{S}.\ \Psi' \vdash^{\mathrm{C}} (\mathbb{S},\mathbb{K},\mathtt{pc})$. Then the definition of well-formed ready threads in the RDY rule becomes very straightforward: if the *ready* thread gets the extra ownership of shared memory C and A, it becomes a well-formed *current* thread (see Fig. 5). Recall that $\mathtt{m} \mathrel{-\!\!*} \mathtt{p}$ is defined in Fig. 15.

Similarly, we define $\Psi' \vdash^{\mathrm{R}} (\_,\mathbb{R},\mathbb{K},\mathtt{is},\mathtt{pc})$ as the memory predicate $\lambda\mathbb{H}.\ \Psi' \vdash^{\mathrm{R}} (\_,\mathbb{R},\mathbb{K},\mathtt{is},\mathtt{pc})$. The WAIT rule says that the *waiting* thread in a block queue with identifier w becomes a well-formed *ready* thread if it gets extra ownership of memory $\Delta(w)$. This is also illustrated in Figure 6. The memory predicate $\mathtt{m} \mathrel{-\!\!*} \mathtt{m}'$ is defined in Figure 15.

4.3 Soundness

We prove the soundness of the program logic following the syntactic approach. The progress lemma shows that the program invariant defined by the PROG rule ensures the next instruction does not get stuck. The preservation lemma says the new state after executing the next instruction satisfies the invariant. Therefore, the program never gets stuck as long as the initial state satisfies the invariant. *More importantly*, the invariant always holds during execution, from which we can derive rich properties of programs. For instance, we know there is always a partition of heap for all threads, based on which we can derive the well-formedness of the current thread, ready threads and waiting threads. This can be viewed as a guarantee of non-interference. Here, we only show a soundness theorem formalizing the partial correctness of programs.

Lemma 4.2 (Progress)
If $\Psi,\Delta \vdash \mathbb{W}$, then there exists $\mathbb{W}'$ such that $(\mathbb{W} \longmapsto \mathbb{W}')$.

Proof sketch. Suppose $\mathbb{W} = (\mathbb{C},\mathbb{S},\mathbb{K},\mathtt{pc},\mathtt{tid},\mathbb{T},\mathbb{B})$. By the definition of the operational semantics, execution of $\mathbb{C}(\mathtt{pc})$ would always succeeds unless $\mathbb{C}(\mathtt{pc})$ is one of ld, st, switch, block, unblock, ret or iret instructions. By $\Psi,\Delta \vdash \mathbb{W}$ we know there exist Ψ' and $(\mathtt{p},\mathtt{g})$ such that $(\mathtt{pc},(\mathtt{p},\mathtt{g})) \in \Psi'$, $\mathtt{p} * \mathsf{Inv}\ \mathbb{K}\ \mathbb{S}$, and $\Psi,\Delta \vdash \{\mathtt{s}\}\,\mathbb{C}[\mathtt{pc}]$. Then by the premise $\mathsf{enable}(\mathtt{p},\mathtt{g}_{\mathsf{i}})$ in the SEQ rule we know the first 5 instructions would not get stuck, and, by $\mathsf{enable}_{\mathsf{ret}}$ and the RET rule or $\mathsf{enable}_{\mathsf{iret}}$ and the IRET rule, we know the last two instructions would not get stuck either. □

Lemma 4.3 says the program can always reach the entry point of the interrupt handler as long as the interrupt is enabled and there is no interrupts being serviced.

Lemma 4.3 (IRQ-Progress)
If $\mathbb{W}.\mathbb{S}.\mathtt{ie} = 1$ and $\mathbb{W}.\mathbb{S}.\mathtt{is} = 0$, there always exists $\mathbb{W}'$ such that $\mathbb{W} \not\downarrow \mathbb{W}'$.

Proof sketch. Obvious by the IRQ rule shown in Section 3.1. □

The following two lemmas show that the interrupt handler does not interfere with the non-handler code.

Lemma 4.4 (presv-p)
For all p, $\mathbb{K}$, $\mathbb{S}$ and $\mathbb{S}'$, if $\mathbb{S} = (\mathbb{H},\mathbb{R},1,0)$, $(\mathtt{p} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}$, precise(INV0), precise(INV1), and $\lfloor \mathtt{g}_{\mathsf{i}} \rfloor\ (\mathbb{H},\mathbb{R},0,1)\ \mathbb{S}'$, then $(\mathtt{p} * \mathsf{Inv})\ \mathbb{K}\ (\mathbb{S}'.\mathbb{H},\mathbb{R},1,0)$.

Lemma 4.5 (presv-g)
For all g and $\mathbb{S}$, if $\mathbb{S} = (\mathbb{H},\mathbb{R},1,0)$ for certain $\mathbb{H}$ and $\mathbb{R}$, $(\mathsf{INV}_s * \mathsf{true})\ \mathbb{H}$, precise(INV0) and precise(INV1), then,

$$\forall \mathbb{S}',\mathbb{S}''.\ (\lfloor \mathtt{g}_{\mathsf{i}} \rfloor\ \mathbb{S}|_{\{\mathtt{ie}=0,\mathtt{is}=1\}}\ \mathbb{S}') \wedge (\lfloor \mathtt{g} \rfloor\ (\mathbb{S}'.\mathbb{H},\mathbb{R},1,0)\ \mathbb{S}'') \rightarrow (\lfloor \mathtt{g} \rfloor\ \mathbb{S}\ \mathbb{S}'').$$

The following lemmas illustrate how the program invariant is preserved by cli, sti and thread primitives, based on our definition of their ownership transfer semantics.

Lemma 4.6 (cli)
If $\mathtt{NextS}_{(\mathsf{cli},\mathbb{K})}\mathbb{S}\ \mathbb{S}'$, and $(\mathtt{p} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}$, then $(\mathsf{sp}(\mathtt{g}_{\mathsf{cli}},\mathtt{p}) * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}'$.

Lemma 4.7 (sti)
If $\mathtt{NextS}_{(\mathsf{sti},\mathbb{K})}\mathbb{S}\ \mathbb{S}'$, $(\mathtt{p} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}$, and $\mathtt{p} \Rightarrow \mathsf{Inv}(1-\mathbb{S}.\mathtt{ie},\mathbb{S}.\mathtt{is}) * \mathsf{true}$, then $(\mathsf{sp}(\mathtt{g}_{\mathsf{sti}},\mathtt{p}) * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}'$.

Lemma 4.8 (switch)
If $\mathsf{enable}(\mathtt{p},\mathtt{g}_{\mathsf{switch}})$, $(\mathtt{p} \rhd \mathtt{g}_{\mathsf{switch}}) \Rightarrow \mathtt{p}'$, and $(\mathtt{p} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}$, then

- there exist $\mathbb{H}_1$ and $\mathbb{H}_2$ such that $\mathsf{INV}_s\ \mathbb{H}_1$ and $\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{S}.\mathbb{H}$;
- $\lfloor \mathtt{g}_{\mathsf{sw}} \rfloor\ \mathbb{S}\ \mathbb{S}$;
- for all $\mathbb{H}'_1$, if $\mathsf{INV}_s\ \mathbb{H}'_1$, $\mathbb{H}'_1 \perp \mathbb{H}_2$, and $\mathbb{S}' = \mathbb{S}|_{\mathbb{H}'_1 \cup \mathbb{H}_2}$, then $(\mathtt{p}' * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}'$ and $\lfloor \mathtt{g}_{\mathsf{sw}} \rfloor\ \mathbb{S}\ \mathbb{S}'$.

Lemma 4.9 (Ready2Running)
If $\Psi \vdash^{\mathrm{R}} (\mathbb{H},\mathbb{R},\mathbb{K},\mathtt{is},\mathtt{pc})$, then

- pc is a valid code pointer specified in Ψ with some specification $(\mathtt{p},\mathtt{g})$;
- for all $\mathbb{H}'$, if $\mathsf{INV}_s\ \mathbb{H}'$, $\mathbb{H}' \perp \mathbb{H}$, and $\mathbb{S} = (\mathbb{H} \cup \mathbb{H}',\mathbb{R},0,\mathtt{is})$, then $(\mathtt{p} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}$.

Lemma 4.10 (block)
If $\mathsf{enable}(\mathtt{p}, \mathsf{g}^{\Delta}_{\mathsf{block}\ r_s})$, $(\mathtt{p} \rhd \mathsf{g}^{\Delta}_{\mathsf{block}\ r_s}) \Rightarrow \mathtt{p}'$, and $(\mathtt{p} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}$, then

- there exist $\mathbb{H}_1$ and $\mathbb{H}_2$ such that $\mathsf{INV}_s\ \mathbb{H}_1$ and $\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{S}.\mathbb{H}$;
- there exist $\mathtt{m}$ such that $\Delta(\mathbb{S}.\mathbb{R}(r_s)) = \mathtt{m}$;
- for all $\mathbb{H}'_1$, $\mathbb{H}'_3$ and $\mathbb{H}$, if $\mathsf{INV}_s\ \mathbb{H}'_1$, $\mathtt{m}'\ \mathbb{H}'_3$, $\mathbb{H} = \mathbb{H}'_1 \uplus \mathbb{H}_2 \uplus \mathbb{H}'_3$, and $\mathbb{S}' = \mathbb{S}|_{\mathbb{H}}$, then $(\mathtt{p}' * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}'$ and $\lfloor \mathsf{g}^{\Delta}_{\mathsf{block}\ r_s} \rfloor\ \mathbb{S}\ \mathbb{S}'$.

Lemma 4.11 (unblock)
If $\mathsf{enable}(\mathtt{p}, \mathsf{g}^{\Delta}_{\mathsf{unblock}\ r_s, r_d})$, $(\mathtt{p} \rhd \mathsf{g}^{\Delta}_{\mathsf{block}\ r_s}) \Rightarrow \mathtt{p}'$, and $(\mathtt{p} * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}$ (where $\mathbb{S} = (\mathbb{H}, \mathbb{R}, \mathtt{ie}, \mathtt{is})$), then

- there exists $\mathtt{m}$ such that $\Delta(\mathbb{R}(r_s)) = \mathtt{m}$;
- there exist $\mathbb{H}_1$ and $\mathbb{H}_2$ such that $\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H}$ and $\mathtt{m}\ \mathbb{H}_1$;
- let $\mathbb{S}' = (\mathbb{H}, \mathbb{R}\{r_d \rightsquigarrow 0\}, \mathtt{ie}, \mathtt{is})$, then $\lfloor \mathsf{g}^{\Delta}_{\mathsf{block}\ r_s} \rfloor\ \mathbb{S}\ \mathbb{S}'$ and $(\mathtt{p}' * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}'$;
- for all $n > 0$, let $\mathbb{S}' = (\mathbb{H}_2, \mathbb{R}\{r_d \rightsquigarrow n\}, \mathtt{ie}, \mathtt{is})$, then $\lfloor \mathsf{g}^{\Delta}_{\mathsf{block}\ r_s} \rfloor\ \mathbb{S}\ \mathbb{S}'$ and $(\mathtt{p}' * \mathsf{Inv})\ \mathbb{K}\ \mathbb{S}'$.

Lemma 4.12 (Waiting2Ready)
If $\Psi, \Delta, w \vdash^{\mathrm{W}} (\mathbb{H}, \mathbb{R}, \mathbb{K}, \mathtt{is}, \mathtt{pc})$, then

- there exists $\mathtt{m}$ such that $\Delta(w) = \mathtt{m}$;
- for any $\mathbb{H}'$, if $\mathbb{H} \perp \mathbb{H}'$ and $\mathtt{m}\ \mathbb{H}'$, then $\Psi \vdash^{\mathrm{R}} (\mathbb{H} \uplus \mathbb{H}',\ \mathbb{R}, \mathbb{K}, \mathtt{is}, \mathtt{pc})$.

The following lemmas are also used to prove the preservation lemma.

Lemma 4.13 (Stack-Strengthen)
For all $\mathsf{g}, \mathsf{g}', \mathbb{S}$ and $\mathbb{S}'$, if $\forall \mathbb{S}''.\ \mathsf{g}\ \mathbb{S}\ \mathbb{S}'' \rightarrow \mathsf{g}'\ \mathbb{S}'\ \mathbb{S}''$, and $\mathsf{WFST}(\mathsf{g}', \mathbb{S}', \mathbb{K}, \Psi)$, then $\mathsf{WFST}(\mathsf{g}, \mathbb{S}, \mathbb{K}, \Psi)$.

Lemma 4.14 (Code Heap)
If $\Psi, \Delta \vdash \mathbb{C} : \Psi'$, then, for all $\mathtt{f}$ and $\mathtt{s}$, if $(\mathtt{f}\,\mathtt{s}) \in \Psi'$, we know $\mathtt{f} \in dom(\mathbb{C})$.

Lemma 4.15 (Spec. Extension)
If $\Psi, \Delta \vdash \mathbb{C} : \Psi'$ and $\Psi, \Delta \vdash \{\mathtt{s}\}\,\mathtt{f} : \mathbb{C}[\mathtt{f}]$, then we have $\Psi, \Delta \vdash \mathbb{C} : \Psi''$, where $\Psi'' = \Psi' \cup \{(\mathtt{f}, \mathtt{s})\}$.

The preservation lemma is formalized below.

Lemma 4.16 (Preservation)
If $\mathsf{INV0}$ and $\mathsf{INV1}$ are precise (preciseness is defined in Figure 15), $\Psi, \Delta \vdash \mathbb{W}$ and $(\mathbb{W} \Longmapsto \mathbb{W}')$, we have $\Psi, \Delta \vdash \mathbb{W}'$.

Below we show the soundness theorem. Recall that preciseness is defined in Figure 15, and the specification $(\mathsf{p}_\mathsf{i}, \mathsf{g}_\mathsf{i})$ for the interrupt handler is defined by Formulae (1) and (2). The soundness theorem captures the partial correctness of programs in the sense that, when we reach the target address of jump or branch instructions, we know the assertion specified in Ψ holds. These assertions corresponds to loop-invariants and pre- and post-conditions for functions at high-level.

Theorem 4.17 (Soundness)
If $\mathsf{INV0}$ and $\mathsf{INV1}$ are precise, $\Psi, \Delta \vdash \mathbb{W}$, and $(\mathtt{h\_entry}, (\mathsf{p}_\mathsf{i}, \mathsf{g}_\mathsf{i})) \in \Psi$, then, for any n, there exists $\mathbb{W}'$ such that $\mathbb{W} \Longmapsto^n \mathbb{W}'$; 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 $(\mathtt{p}, \mathsf{g})$ such that $(\mathtt{f}, (\mathtt{p}, \mathsf{g})) \in \Psi$ and $\mathtt{p}\ \mathbb{K}\ \mathbb{S}$ 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 $(\mathtt{p}, \mathsf{g})$ such that $(\mathtt{f}, (\mathtt{p}, \mathsf{g})) \in \Psi$ and $\mathtt{p}\ \mathbb{K}\ \mathbb{S}$ holds;
3. if $\mathbb{C}(\mathtt{pc}) = \mathsf{call}\ \mathtt{f}$, then there exists $(\mathtt{p}, \mathsf{g})$ such that $(\mathtt{f}, (\mathtt{p}, \mathsf{g})) \in \Psi$ and $\mathtt{p}\ (\mathtt{pc} :: \mathbb{K})\ \mathbb{S}$ holds;
4. if $\mathbb{C}(\mathtt{pc}) = \mathsf{ret}$, then there exist $\mathtt{pc}'$, $\mathbb{K}'$, and $(\mathtt{p}, \mathsf{g})$ such that $\mathbb{K} = \mathtt{pc}' :: \mathbb{K}'$, $(\mathtt{pc}', (\mathtt{p}, \mathsf{g})) \in \Psi$, and $\mathtt{p}\ \mathbb{K}'\ \mathbb{S}$ holds.

Proof sketch. The theorem describes both non-stuckness and partial correctness. For non-stuckness, we can simply do induction over n and apply Progress (4.2 and 4.3) and Preservation (4.16) lemmas. We actually prove a stronger version than non-stuckness: $(\Psi, \Delta \vdash \_)$ holds over every intermediate world configuration. Therefore, we know $\Psi, \Delta \vdash \mathbb{W}'$ holds. Then the next four bullets about partial correctness can be easily derived by an inversion of the PROG rule. □

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

Figure 21. Specifications for the Teeter-Totter Example

4.4 The Teeter-Totter Example

With our program logic, we can now certify the Teeter-Totter example shown in Fig. 10. We first define $\mathsf{INV0}$, the interrupt handler's specification for its local memory.

$$\mathsf{INV0} \triangleq \exists \mathtt{w}_l, \mathtt{w}_r.\ ((\mathtt{LEFT} \mapsto \mathtt{w}_l) * (\mathtt{RIGHT} \mapsto \mathtt{w}_r)) \wedge (\mathtt{w}_l + \mathtt{w}_r = 100)$$

Then we can get the concrete specification of the interrupt handler, following Formulae (1) and (2). We let $\mathsf{INV1}$ be emp, since the non-handler code is sequential.

Specifications are shown in Figure 21. Recall $\mathsf{enable}_{\mathsf{iret}}$ is defined in Figure 18. To simplify our presentation, we present the predicate $\mathtt{p}$ in the form of a proposition with free variables referring to components of the state $\mathbb{S}$. Also, we use $\mathtt{m}$ as a shorthand for the proposition $\mathtt{m}\ \mathbb{H}$ when there is no confusion.

If we compare $\mathtt{p}_1$ and $\mathtt{p}_2$, we will see that the non-handler code cannot access memory at addresses `LEFT` and `RIGHT` without first disabling the interrupt because, from $\mathtt{p}_1$, we cannot prove that the target addresses `LEFT` and `RIGHT` are in the domain of memory, as required in the instantiation of the SEQ rule for ld and st. Since the non-handler never returns, we do not really care about the guarantee g for the state transition from the specified point to the return point of the function. Here we simply use NoG (see Figure 21) as guarantees.

Certifying the code with respect to the specifications is left to the reader and not shown here.

4.5 The Timer Handler

Here we also briefly explain the specification for the preemptive timer handler shown in Figure 13. The only memory the hander uses is the memory cell at location `CNT`. We define $\mathsf{INV0}$ below.

$$\mathsf{INV0} \triangleq \exists \mathtt{w}.\ (\mathtt{CNT} \mapsto \mathtt{w}) \wedge (\mathtt{w} \leq 100)$$

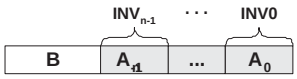
Then we get the specification of the handler $(\mathsf{p}_\mathsf{i}, \mathsf{g}_\mathsf{i})$ by Equations (1) and (2). In g_0, we use primed variable (*e.g.,* $\mathtt{ie}'$ and $\mathtt{is}'$) to refer to components in the second state.

4.6 Futher Extensions

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 partition. 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 $\mathsf{Inv}(\mathtt{ie},\mathtt{is})$ into $\mathsf{Inv}(\mathtt{ie},\mathsf{ivec})$, which models the switch of memory ownership at the points of cli, sti and boundaries of interrupt handers.

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 runtime 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" intruction to enforce that the new handler's interface is compatible to the specification $(\mathtt{p}_i,\mathtt{g}_i)$.

Also, we do not consider dynamic thread creation in this paper. In our previous work [6], 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.* [9] recently showed an extension of concurrent separation logic with dynamic threads and locks. Their technique might be applied here as well to support dynamic creation of block queues.

We will not show the details of supporting multi-level interrupts, dynamic installation of handers, and dynamic creation of threads and block queues, which are extensions orthogonal to the focus of this paper, *i.e.,* interaction between threads and interrupts.

5. More Examples and Implementations

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, a partial mapping from lock ids to heap predicates, 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 $\mathsf{INV0} * \mathsf{INV1}$).

$$\mathsf{INV}(l,\mathtt{m}) \triangleq \exists \mathtt{w}.\ (l \mapsto \mathtt{w}) * ((\mathtt{w}=0) \wedge \mathsf{emp} \vee (\mathtt{w}=1) \wedge \mathtt{m}) \quad (4)$$
$$\mathsf{INV}(\Gamma) \triangleq \forall_* l \in \mathit{dom}(\Gamma).\ \mathsf{INV}(l,\Gamma(l)) \quad (5)$$

```
ACQ_H:    -{(p01, g01)}
          cli
          -{(p02, g02)}
          call   ACQ_H_a
          -{(p03, g03)}
          sti
          -{(p04, gid)}
          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 22. Hoare-Style Implementation of Lock

where $\forall_*$ is an indexed, finitely iterated separating conjunction, which is defined as:

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

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 spin locks.

The Hoare-style implementation. 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. 22 and 23. The precondition for `ACQ_H` is $(\mathtt{p}_{01},\mathtt{g}_{01})$. The assertion $\mathtt{p}_{01}$ requires that r_1 contains a lock id and $\Delta(r_1) = \Gamma(r_1)$. The guarantee $\mathtt{g}_{01}$ shows that the function obtains the ownership of $\Gamma(r_1)$ when it returns. Here we use primed variables (*e.g.,* $\mathtt{ie}'$ and $\mathtt{is}'$) to refer to components in the return state, and use $\mathsf{trash}(\{r_2,r_3\})$ to mean that values of all registers other than r_2 and r_3 are preserved.

Although the meaning of $\mathtt{p}_{01}$ and $\mathtt{g}_{01}$ is obvious, they can be further simplified when exported to the high-level concurrent programs shown in Figure 3. The high-level specification does not need to refer to `ie` and `is` since they are always 1 and 0 respectively. We do not need to specify stack and trashing of registers,

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

$$\mathtt{p}_{01} \triangleq \mathtt{p}_0\wedge(\mathtt{ie}=1)$$

$$\mathtt{g}_{01} \triangleq \left\{\begin{array}{l}\mathsf{emp}\\ \Gamma(r_1)\end{array}\right\}\wedge(\mathtt{ie}=\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2,r_3\})$$

$$\mathtt{p}_{02} \triangleq \mathtt{p}_0\wedge(\mathtt{ie}=0)\wedge(\mathsf{INV}_s * \mathsf{true})$$

$$\mathtt{g}_{02} \triangleq \left\{\begin{array}{l}\mathsf{INV}_s\\ \Gamma(r_1)\end{array}\right\}\wedge(\mathtt{ie}=1-\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2,r_3\})$$

$$\mathtt{p}_{03} \triangleq \mathtt{p}_0\wedge(\mathtt{ie}=0)\wedge(\Gamma(r_1) * \mathsf{INV}_s * \mathsf{true})$$

$$\mathtt{g}_{03} \triangleq \left\{\begin{array}{l}\mathsf{INV}_s\\ \mathsf{emp}\end{array}\right\}\wedge(\mathtt{ie}=1-\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2,r_3\})$$

$$\mathtt{p}_{04} \triangleq \mathtt{p}_0\wedge(\mathtt{ie}=1)\wedge(\Gamma(r_1) * \mathsf{true})$$

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

$$\mathtt{g}_{11} \triangleq \left\{\begin{array}{l}\mathsf{INV}_s\\ \mathsf{INV}_s * \Gamma(r_1)\end{array}\right\}\wedge(\mathtt{ie}=\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2,r_3\})$$

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

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

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

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

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

$$\mathtt{g}_a \triangleq \left\{\begin{array}{l}\Gamma(r_1)\\ \mathsf{emp}\end{array}\right\}\qquad \mathtt{g}_b \triangleq \left\{\begin{array}{l}r_1\mapsto \_\\ r_1\mapsto 1\end{array}\right\}\qquad \mathsf{hid} \triangleq \left\{\begin{array}{l}\mathsf{emp}\\ \mathsf{emp}\end{array}\right\}$$

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

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

$$\begin{array}{l}\mathtt{g}_{32} \triangleq ((r_2=0\wedge\mathtt{g}_b)\vee(r_2\neq 0\wedge\mathsf{hid}))\\ \qquad\wedge(\mathtt{ie}=\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2,r_3\})\end{array}$$

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

$$\mathtt{g}_{33} \triangleq \mathtt{g}_b\wedge(\mathtt{ie}=\mathtt{ie}')\wedge(\mathtt{is}=\mathtt{is}')\wedge\mathsf{trash}(\{r_2,r_3\})$$

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

Figure 23. Specifications of Hoare-Style Lock

```
ACQ_M:    -{(p11, g11)}
          movi    $r3, 0
acq_loop: -{(p12, g12)}
          cli
          ld      $r2, 0($r1)          ;; $r2 <- [l]
          beq     $r2, $r3, gowait     ;; ([l] == 0)?
          st      0($r1), $r3          ;; [l] <> 0:
          j       acq_done             ;;   [l] <- 0
gowait:   -{(p13, g13)}
          block   $r1
          -{(p13, g13)}
          sti
          j       acq_loop
acq_done: -{(p14, g14)}
          sti
          ret

REL_M:    -{(p21, g21)}
          cli
          -{(p22, g22)}
          unblock $r1, $r2
          -{(p23, g22)}
          -{(p22, g22)}
          movi    $r2, 1
          st      0($r1), $r2
          sti
          ret
```

Figure 24. Mesa-Style Implementation of Locks

which can be inferred from the calling convention. We can also hide Δ, since block and unblock are not visible from the high level. So the specification exported to the high level would be:

$$\begin{array}{rcl}\mathtt{p}_{01} & \triangleq & r_1\in dom(\Gamma)\\ \mathtt{g}_{01} & \triangleq & \left\{\begin{array}{l}\mathsf{emp}\\ \Gamma(r_1)\end{array}\right\}\end{array}$$

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

Functions REL_h_a and REL_h releases the lock with the interrupt disabled and enabled, respectively. They are specified by $(\mathtt{p}_{21},\mathtt{g}_{21})$ and $(\mathtt{p}_{31},\mathtt{g}_{31})$. 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 $\mathtt{g}_{31}$, but these details are hidden in $\mathtt{g}_{21}$.

The Mesa-style implementation. Figure 24 shows the Mesa-style implementation of locks. The specifications are shown in Figure 25. 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 $\mathtt{p}_0$ requires $\Delta(r_1)=\mathsf{emp}$, which implies the Mesa-style semantics of block and unblock.

Spin Locks An implementation of spin locks and its specifications are shown in Figure 26. The specifications $(\mathtt{p}_{11},\mathtt{g}_{11})$ and $(\mathtt{p}_{21},\mathtt{g}_{21})$ 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 after calling "release".

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

$$\mathtt{p}_{11} \triangleq \mathtt{p}_0 \wedge (\mathtt{ie}=1)$$

$$\mathtt{g}_{11} \triangleq \left\{ \begin{array}{l} \mathsf{emp} \\ \Gamma(r_1) \end{array} \right\} \wedge (\mathtt{ie}=\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3\})$$

$$\mathtt{p}_{12} \triangleq \mathtt{p}_{11} \wedge (r_3=0)$$

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

$$\mathtt{g}_{13} \triangleq \left\{ \begin{array}{l} \mathsf{INV}_s \\ \Gamma(r_1) \end{array} \right\} \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3\})$$

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

$$\mathtt{g}_{14} \triangleq \left\{ \begin{array}{l} \mathsf{INV}_s \\ \mathsf{emp} \end{array} \right\} \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3\})$$

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

$$\mathtt{g}_{21} \triangleq \left\{ \begin{array}{l} \Gamma(r_1) \\ \mathsf{emp} \end{array} \right\} \wedge (\mathtt{ie}=\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2\})$$

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

$$\mathtt{g}_{22} \triangleq \left\{ \begin{array}{l} \Gamma(r_1) * \mathsf{INV}_s \\ \mathsf{emp} \end{array} \right\} \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2\})$$

$$\mathtt{p}_{23} \triangleq \mathtt{p}_0 \wedge (\mathtt{ie}=0) \wedge (((\![r_1]\!]=0) \wedge (\Gamma(r_1) * \mathsf{INV}_s) \vee ([\![r_1]\!] \neq 0) \wedge (\Gamma(r_1) * \mathsf{INV}_s)) * \mathsf{true})$$

Figure 25. Specification of Mesa-Style Implementation of Locks

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 Mesa style [14], Hoare style [11] and Brinch Hansen style [2] condition variables. Below we use Υ, a partial mapping from condition variables cv to heap predicates $\mathtt{m}$, to specify the conditions associated with condition variables.

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

In our implementation, we let cv be an identifier pointing to a block queue in $\mathbb{B}$. A lock l needs to be associated with cv to guarantee exclusive access of the 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 implementation and specifications of Hoare-style and Brinch Hansen style are are shown in Figs. 27, 28 and 29. 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

```
;; 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
```

$$\mathtt{p} \triangleq (\mathtt{is}=0)\wedge \mathsf{enable}_{\mathsf{ret}} \wedge (r_1 \in dom(\Gamma))$$

$$\mathtt{p}_{11} \triangleq \mathtt{p} \wedge (\mathtt{ie}=1)$$

$$\mathtt{g}_{11} \triangleq \left\{ \begin{array}{l} \mathsf{emp} \\ \Gamma(r_1) \end{array} \right\} \wedge (\mathtt{ie}=\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3\})$$

$$\mathtt{p}_{12} \triangleq \mathtt{p}_{11} \wedge (r_2=1)$$

$$\mathtt{p}_{13} \triangleq \mathtt{p} \wedge (\mathtt{ie}=0) \wedge (\mathsf{INV}_s * \mathsf{true})$$

$$\mathtt{g}_{13} \triangleq \left\{ \begin{array}{l} \mathsf{INV}_s \\ \Gamma(r_1) \end{array} \right\} \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2\})$$

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

$$\mathtt{g}_{21} \triangleq \left\{ \begin{array}{l} \Gamma(r_1) \\ \mathsf{emp} \end{array} \right\} \wedge (\mathtt{ie}=\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2\})$$

Figure 26. A Spin Lock

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. Figure 28 also shows important intermediate specifications we use during verification.

Brinch Hansen style condition variables is similar to Hoare-style. The wait function and its specifications are the same as `WAIT_H`. The signal function `SIGNAL_BH`, which is omitted here, has specification $(\mathtt{p}_{31},\mathtt{g}_{31})$ defined in Figure 28. Here $\mathtt{p}_{31}$ is the same as $\mathtt{p}_{21}$ for `SIGNAL_H`. The definition of $\mathtt{g}_{31}$ shows the difference between Hoare style and Brinch Hansen style: the lock is released when `SIGNAL_BH` returns. Therefore, calling the `SIGNAL_BH` function must be the last command in the critical region.

Mesa-style. Figure 30 shows Mesa-style condition variables, with specifications shown in Figure 31. `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.

```
WAIT_H:   -{(p11, g11)}            ;; wait(l, cv)
          cli
          mov     $r4, $r2
          -{(p12, g12)}
          call    REL_H_a
          -{(p13, g13)}
          block   $r4
          -{(p14, g14)}
          sti
          ret

SIGNAL_H: -{(p21, g21)}            ;; signal(l, cv)
          cli
          -{(p22, g22)}
          unblock $r2, $r3
          -{(p23, g23)}
          movi    $r4, 0
          beq     $r3, $r4, sig_done
          -{(p24, g24)}
          block   $r1
sig_done: -{(p25, g25)}
          sti
          ret

SIGNAL_BH:-{(p31, g31)}            ;; signal(l, cv)
          cli
          -{(p32, g32)}
          unblock $r2, $r3         ;; $r2 contains cv
          -{(p33, g33)}
          movi    $r4, 0
          beq     $r3, $r4, sig_cont
          -{(p34, g34)}
          j       sig_done
sig_cont: -{(p35, g35)}
          -{(p36, g36)}
          call    REL_0            ;; $r1 contains l
sig_done: -{(p34, g34)}
          sti
          ret
```

Figure 27. Implementation of CV - Hoare Style

SIGNAL_M is specified by $(\mathtt{p}_{21}, \mathtt{g}_{21})$. The assertion hid is defined in Figure 23, 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.

5.3 Other Implementation Details

As shown in Figure 3, we have certified the preemptive thread implementations and libraries extracted from our simplified OS kernel, which is implemented in 16-bit x86 assembly code and works in real-mode. At the lowest level, we have concrete implementations of the scheduler and block/unblock primitives. Thread queues are implemented as a doubly linked list containing thread control blocks. The synchronization primitives in the middle level also have been implemented in x86, which call the underlying primitives. The timer handler simply saves the context and calls the scheduler. The yield function just wraps the scheduler by disabling the interrupt at the beginning and enabling it at the end.

Since the code are at different abstraction levels, we certify them using different program logics, following the technique proposed

$$\mathsf{Cond}(r,r') \triangleq \Gamma(r) \wedge (\Upsilon(r') * \mathsf{true})$$

$$\overline{\mathsf{Cond}}(r,r') \triangleq \Gamma(r) \wedge \neg(\Upsilon(r') * \mathsf{true})$$

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

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

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

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

$$\mathtt{g}_{12} \triangleq \left\{ \begin{array}{l} \overline{\mathsf{Cond}}(r_1,r_4) * \mathsf{INV}_s \\ \mathsf{Cond}(r_1,r_4) \end{array} \right\} \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

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

$$\mathtt{g}_{13} \triangleq \left\{ \begin{array}{l} \mathsf{INV}_s \\ \mathsf{Cond}(r_1,r_4) \end{array} \right\} \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

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

$$\mathtt{g}_{14} \triangleq \left\{ \begin{array}{l} \mathsf{INV}_s \\ \mathsf{emp} \end{array} \right\} \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

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

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

$$\mathtt{p}_{22} \triangleq \mathtt{p}(r_1,r_2) \wedge (\mathtt{ie}=0) \wedge ((\Gamma(r_1) \wedge \mathsf{Cond}) * \mathsf{INV}_s)$$

$$\mathtt{g}_a \triangleq \left\{ \begin{array}{l} \mathsf{Cond}(r_1,r_2) * \mathsf{INV}_s \\ \Gamma(r_1) \end{array} \right\} \qquad \mathtt{g}_b \triangleq \left\{ \begin{array}{l} \mathsf{INV}_s \\ \Gamma(r_1) \end{array} \right\}$$

$$\mathtt{g}_{22} \triangleq \mathtt{g}_a \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

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

$$\mathtt{g}_{23} \triangleq (r_3=0 \wedge \mathtt{g}_a \vee r_3 \neq 0 \wedge \mathtt{g}_b) \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

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

$$\mathtt{g}_{24} \triangleq \mathtt{g}_b \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

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

$$\mathtt{g}_{25} \triangleq \left\{ \begin{array}{l} \mathsf{INV}_s \\ \mathsf{emp} \end{array} \right\} \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

Figure 28. Spec. of CV - Hoare Style

$$\mathtt{p}_{31} \triangleq \mathtt{p}(r_1,r_2) \wedge (\mathtt{ie}=1) \wedge \Gamma(r_1) \wedge (\mathsf{Cond}(r_1,r_2) * \mathsf{true})$$

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

$$\mathtt{p}_{32} \triangleq \mathtt{p}(r_1,r_2) \wedge (\mathtt{ie}=0) \wedge (\mathsf{Cond}(r_1,r_2) * \mathsf{INV}_s * \mathsf{true})$$

$$\mathtt{g}_a \triangleq \left\{ \begin{array}{l} \mathsf{Cond}(r_1,r_2) * \mathsf{INV}_s \\ \mathsf{emp} \end{array} \right\} \qquad \mathtt{g}_b \triangleq \left\{ \begin{array}{l} \mathsf{INV}_s \\ \mathsf{emp} \end{array} \right\}$$

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

$$\begin{array}{l} \mathtt{p}_{33} \triangleq \mathtt{p}(r_1,r_2) \wedge (\mathtt{ie}=0) \wedge \\ \qquad (((r_3=0) \wedge (\mathsf{Cond}(r_1,r_2) * \mathsf{INV}_s) \vee (r_3 \neq 0) \wedge \mathsf{INV}_s) * \mathsf{true}) \end{array}$$

$$\begin{array}{l} \mathtt{g}_{33} \triangleq (r_3=0 \wedge \mathtt{g}_a \vee r_3 \neq 0 \wedge \mathtt{g}_b) \\ \qquad \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\}) \end{array}$$

$$\mathtt{p}'(r) \triangleq (\mathtt{is}=0) \wedge \exists l,\mathtt{m}.\ (r=l) \wedge (\Gamma(l)=\mathtt{m}) \wedge (\Delta(l)=\mathtt{m})$$

$$\mathtt{p}_{34} \triangleq \mathtt{p}'(r_1) \wedge (\mathtt{ie}=0) \wedge (\mathsf{INV}_s * \mathsf{true})$$

$$\mathtt{g}_{34} \triangleq \mathtt{g}_b \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

$$\mathtt{p}_{35} \triangleq \mathtt{p}(r_1,r_2) \wedge (\mathtt{ie}=0) \wedge (\mathsf{Cond}(r_1,r_2) * \mathsf{INV}_s * \mathsf{true})$$

$$\mathtt{g}_{35} \triangleq \mathtt{g}_a \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

$$\mathtt{p}_{36} \triangleq \mathtt{p}'(r_1) \wedge (\mathtt{ie}=0) \wedge (\Gamma(r_1) * \mathsf{INV}_s * \mathsf{true})$$

$$\begin{array}{l} \mathtt{g}_{36} \triangleq \left\{ \begin{array}{l} \Gamma(r_1) * \mathsf{INV}_s \\ \mathsf{emp} \end{array} \right\} \\ \qquad \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\}) \end{array}$$

Figure 29. Spec. of CV - Brinch Hansen Style

```
WAIT_M:   -{(p11, g11)}              ;; wait(l, cv)
          cli
          mov     $r4, $r2
          -{(p12, g12)}
          call    REL_H_a
          -{(p13, g13)}
          block   $r4
          -{(p14, g13)}
          sti
          -{(p15, g15)}
          call    ACQ_H
          -{(p16, gid)}
          ret

SIGNAL_M: -{(p21, g21)}              ;; signal(cv)
          cli
          -{(p22, g22)}
          unblock $r1, $r2
          -{(p22, g22)}
          sti
          ret
```

Figure 30. Implementation of Condition Variable - Mesa Style

$$\begin{array}{l} \mathtt{p}(r,r') \triangleq (\mathtt{is}=0) \wedge \mathsf{enable}_{\mathsf{ret}} \wedge \\ \qquad \exists l,cv,\mathtt{m},\mathtt{m}'.\ (r=l) \wedge (r'=cv) \wedge (\Gamma(l)=\mathtt{m}) \wedge (\Delta(l)=(\mathtt{m},1)) \\ \qquad\qquad \wedge (\Upsilon(cv)=\mathtt{m}') \wedge (\Delta(cv)=\mathsf{emp}) \end{array}$$

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

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

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

$$\begin{array}{l} \mathtt{g}_{12} \triangleq \left\{ \begin{array}{l} \overline{\mathsf{Cond}}(r_1,r_4) * \mathsf{INV}_s \\ \Gamma(r_1) \end{array} \right\} \\ \qquad \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\}) \end{array}$$

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

$$\begin{array}{l} \mathtt{g}_{13} \triangleq \left\{ \begin{array}{l} \mathsf{INV}_s \\ \Gamma(r_1) \end{array} \right\} \\ \qquad \wedge (\mathtt{ie}=1-\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\}) \end{array}$$

$$\mathtt{p}_{14} \triangleq \mathtt{p}(r_1,r_4) \wedge (\mathtt{ie}=0) \wedge (\mathsf{INV}_s * \mathsf{true})$$

$$\mathtt{p}_{15} \triangleq \mathtt{p}(r_1,r_4) \wedge (\mathtt{ie}=1)$$

$$\mathtt{g}_{15} \triangleq \left\{ \begin{array}{l} \mathsf{emp} \\ \Gamma(r_1) \end{array} \right\} \wedge (\mathtt{ie}=\mathtt{ie}') \wedge (\mathtt{is}=\mathtt{is}') \wedge \mathsf{trash}(\{r_2,r_3,r_4\})$$

$$\mathtt{p}_{16} \triangleq \mathtt{p}(r_1,r_4) \wedge (\mathtt{ie}=1) \wedge \Gamma(r_1)$$

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

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

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

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

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

Figure 31. Specifications of Condition Variable - Mesa Style

by Feng *et al.* [5]. We use SCAP [7] to certify the low-level implementations of thread primitives, including de-queue/en-queue functions and context-switching code, which are all treated as sequential code. SCAP can be viewed as a specialization of our logic for AIM, assuming `ie` is always 0 and prohibiting the execution of cli and sti. The operational semantics for switch, block and unblock instructions in AIM are used as specifications for the concrete implementations at the low level, which also refers to a concrete specification about the date structure of thread control blocks and thread queues.

To certify the library code in the middle level, we adapted our program logic to x86, and proved its soundness in Coq. Instead of implementing an abstract machine like AIM and then compiling AIM code to real x86 code, we simply replace these primitives with function calls to the low-level implementations. However, we still need to define a mapping from the abstract thread queues to their concrete representation in memory. Since code at this level does not touch thread queues, this mapping is always preserved

We link the certified code at different levels in an OCAP-like framework [5]. The basic idea is based on the observation that the low-level code only manipulates TCBs and queues and does not touch data used at high-level, while the high-level code does not

access queues. Therefore, the invariant at each level is preserved by the other side, and the safety property certified at each level still holds when all the code are linked together.

Linking of the thread library code at the middle level with the high-level concurrent programs can be done in a similar way. If we assume $\texttt{ie} = 1$ and $\texttt{is} = 0$, and prohibit the code from executing cli and sti, we can derive a specialized logic from the logic for AIM. Such a logic can be applied to certify high-level concurrent code. We can also expose our specifications of synchronization libraries to the high level code, which would treat them as primitive instructions. We will leave this as future work.

6. Related Work and Conclusions

Regehr and Cooprider [20] 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 [22] 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 [18] 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 [4] 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 [1] showed how to formally certify Kit, an OS kernel implemented in machine code. Gargano *et al.* [8] showed a framework for a certified OS kernel in the Verisoft project. Ni *et al.* [16] 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 have presented a new Hoare-style framework for certifying low-level programs involving both interrupts and concurrency. Following Separation Logic, we formalized the interaction among threads and interrupt handlers in terms of memory ownership transfer. Instead of using the operational semantics of cli, sti and thread primitives, our program logic formulates their local effects over the current thread, as shown in Figure 17, 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).

References

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

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

[3] S. Brookes. A semantics for concurrent separation logic. In *CONCUR'04*, volume 3170 of *LNCS*, pages 16–34, 2004.

[4] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI'01*, pages 59–69, 2001.

[5] X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *TLDI'07*, pages 67–78, 2007.

[6] X. Feng and Z. Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proc. ICFP'05*, pages 254–267, 2005.

[7] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *PLDI'06*, pages 401–414, June 2006.

[8] M. Gargano, M. A. Hillebrand, D. Leinenbach, and W. J. Paul. On the correctness of operating system kernels. In *TPHOLs'05*, 2005.

[9] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *Proceedings of The Fifth ASIAN Symposium on Programming Languages and Systems (APLAS'07)*, page to appear, Nov. 2007.

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

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

[12] G. C. Hunt and J. R. Larus. Singularity design motivation. Technical Report MSR-TR-2004-105, Microsoft Corporation, December 2004.

[13] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL'01*, pages 14–26, 2001.

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

[15] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *POPL'98*, pages 85–97, 1998.

[16] Z. Ni, D. Yu, and Z. Shao. Using XCAP to certify realistic systems code: Machine context management. In *TPHOLs'07*, 2007.

[17] P. W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR'04*, volume 3170 of *LNCS*, pages 49–67, 2004.

[18] J. Palsberg and D. Ma. A typed interrupt calculus. In *FTRTFT'02*, pages 291–310, London, UK, 2002. Springer-Verlag.

[19] W. Paul, M. Broy, and T. In der Rieden. The verisoft xt project. URL: `http://www.verisoft.de`, 2007.

[20] J. Regehr and N. Cooprider. Interrupt verification via thread verification. *Electron. Notes Theor. Comput. Sci.*, 174(9), 2007.

[21] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS'02*, pages 55–74, July 2002.

[22] K. Suenaga and N. Kobayashi. Type based analysis of deadlock for a concurrent calculus with interrupts. In *ESOP'07*, March 2007.

[23] H. Tuch, G. Klein, and G. Heiser. OS verification — now! In *Proc. HotOS-X*, June 2005.

[24] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR'07*, page to appear, 2007.