Modular Verification of Concurrent Thread Management (Extended)

Yu Guo\textsuperscript{1}, Xinyu Feng\textsuperscript{1}, Zhong Shao\textsuperscript{2}, and Peizhi Shi\textsuperscript{1}

\textsuperscript{1} University of Science and Technology of China
\{guoyu,xyfeng\}@ustc.edu.cn  sea10197@mail.ustc.edu.cn
\textsuperscript{2} Yale University
shao@cs.yale.edu

Abstract. Thread management is an essential functionality in OS kernels. However, verification of thread management remains a challenge, due to two conflicting requirements: on the one hand, a thread manager—operating below the thread abstraction layer—should hide its implementation details and be verified independently from the threads being managed; on the other hand, the thread management code in many real-world systems is concurrent, which might be executed by the threads being managed, so it seems inappropriate to abstract threads away in the verification of thread managers. Previous approaches on kernel verification view thread managers as sequential code, thus cannot be applied to thread management in realistic kernels. In this paper, we propose a novel two-layer framework to verify concurrent thread management. We choose a lower abstraction level than the previous approaches, where we abstract away the context switch routine only, and allow the rest of the thread management code to run concurrently in the upper level. We also treat thread management data as abstract resources so that threads in the environment can be specified in assertions and be reasoned about in a proof system similar to concurrent separation logic.

1 Introduction

Thread scheduling in modern operating systems provides the functionality of virtualizing processors: when a thread is waiting for an event, it gives the control of the processor to another thread to make the illusion that each thread has its own processor.

Inside a kernel, a thread manager supervises all threads in the system by manipulating data structures called thread control blocks (TCBs). A TCB is used to record important information of a thread, such as the machine context (or processor state), the thread identifier, the status description, the location and size of the stack, the priority for scheduling, and the entry point of thread code. The TCBs are often implemented using data structures such as queues for ready and waiting threads. Clearly, modifying thread queues and TCBs would drastically change the behaviors of threads. Therefore, a correct implementation of thread management is crucial for guaranteeing the whole system safety. Unfortunately, modular verification of real-world thread management code remains a big challenge today.

The challenge comes from two apparently conflicting goals which we want to achieve at the same time: abstraction (for modular verification) and efficiency (for real-world
usability). On the one hand, TCBs, thread queues, and the thread scheduler are specifics used to implement threads so they should sit at a lower abstraction layer. It is natural to abstract them away from threads, and to verify threads and the thread scheduler separately at different abstraction layers. Previous work has shown it is extremely difficult to verify them together in one logic system [16]. On the other hand, in many real-world systems such as Linux-2.6.10 [13] and FreeBSD-5.2 [14], the thread scheduler code itself is also concurrent in the sense that there may be multiple threads in the system running the scheduler at the same time. For instance, when a thread invokes a thread scheduler routine (e.g., cleaning up dead threads, load balancing, or thread scheduling) and traverses the thread queue, it may be preempted by other threads who may call the same routine and traverse the queue too. Also, in some systems [13,1] the thread scheduling itself is implemented as a separate thread that runs in concurrent with other threads. In these cases, we need to verify thread schedulers in a “multi-threaded” logic, taking threads into account instead of abstracting them away.

Earlier work on thread scheduling verification fails to achieve the two goals at the same time. Ni et al. [16] verified both the thread switch and the threads in one logic [15], which treats thread return addresses as first-class code pointers. Although their method may support concurrent thread schedulers in real systems, it loses the abstraction of threads completely, and makes the logic and specifications too complex for practical use. Recent work [3,7] adopts two-layer verification frameworks to verify concurrent kernels. Kernel code is divided into two layers: sequential code in the lower layer and concurrent in the upper layer. In their frameworks, they put the code manipulating TCBs (e.g., thread schedulers) in the low layer, and hide the TCBs of threads in the upper layer so that the threads cannot modify them. Then they use sequential program logics to verify thread management code. However, this approach is not usable for many realistic kernels where thread managers themselves are concurrent and the threads are allowed to modify the TCBs. Other work on OS verification [12,10] only supports non-reentrant kernels, i.e., there is only one thread running in the kernel at any time.

In this paper, we propose a more natural framework to verify concurrent thread managers. Our framework follows the two-layer approach, so concurrent code at the upper layer can be verified modularly with thread abstractions. However, the abstraction level of our framework is much lower than previous frameworks [3,7]. Most part of the code manipulating thread queues and TCBs is put in the upper layer and can be verified as concurrent code. Our framework successfully achieves both verification goals: it not only allows abstraction and modular verification, but also supports concurrency in real-world thread management.

Our work is based on previous work on thread scheduler verification, but makes the following new contributions:

- We introduce a fine-grained abstraction in our two-layer verification framework. The abstraction protects only a small part of sensitive data in TCBs, and at the same time allows multiple threads to modify other part of TCBs safely. Our division of the two abstraction layers is consistent with many real systems. It is more natural and can support more realistic thread managers than previous work.

- In the upper layer, we introduce the idea of treating threads as resources. The abstract thread resources can be specified explicitly in the assertion language, and
their use by concurrent programs can be reasoned about modularly following concurrent separation logic (CSL) [17]. By enforcing the invariant that the abstract resource is consistent with the concrete thread meta data, we can ensure the safety of the accesses over TCBs and thread queues inside threads.

– Because of the fine-grained abstraction of our approach, the semantics of thread scheduling do not have to be hardwired in the logic. Therefore, our framework can be used to verify various implementation patterns of thread management. We show how to verify the three common patterns of thread scheduling in realistic OS kernels (while previous two-layer frameworks [3,7] can only verify one of them).

– In our extended TR [8], we also use our framework to verify thread schedulers with hardware interrupts, scheduling over multiprocessor with load-balancing, and a set of other thread management routines such as thread creation, join and termination.

The rest of this paper is organized as follows: we first introduce a simplified abstract machine model for the higher-layer of our framework in Sec. 3 to show our main idea, we propose in Sec. 4 our proof system for concurrent thread scheduling code over the abstract machine. We show how to verify two prototypes of schedulers based on context switch in Sec. 5. We compare with related work in Sec. 9 and conclude in Sec. 10.

2 Challenges and our approach

In this section, we illustrate the challenges of verifying code of thread scheduling by showing three patterns of schedulers and discuss the verification issues. Then we informally explain the basic ideas of our approach.

2.1 Three patterns of thread scheduling

By deciding which thread to run next, the thread scheduler is responsible for best utilizing the system and makes multiple threads run concurrently. The scheduling process consists of the following steps: selecting which thread to run next in a thread queue by modifying TCBs, saving the context data of the current thread, and loading the context data of the next thread. Context data is the state of the processor. By saving and loading context data, the processor can run in multiple control flows, i.e., threads. Usually, context data can be saved on stacks or TCBs (we assume in this paper that context
data is saved in TCBs for the brevity of presentation). There are various ways to implement thread schedulers. In Fig. 1 we show three common implementation patterns, all modeled from real systems.

Pattern (I) is popular among embedded OS kernels (e.g., FreeRTOS) and some micro-kernels (e.g., Minix [9] and Exokernel [2]). The scheduler in this pattern is invoked by function calls or interrupts. Thereafter, the scheduling is done in the following steps: (1) saving the current context data, (2) finding the next thread, and (3) loading the context data of the next thread (and switching to it implicitly through function return).

In pattern (II), the scheduling process is a function with the following steps: (1) finding the next thread firstly, (2) performing context switch (saving the current context data, loading the next one, and jumping to the next thread immediately), (3) and running the remaining code of the function when the control is switched back from other threads. This pattern is modeled from some mainstream monolithic kernels (e.g., Linux [13], and FreeBSD). Some embedded kernels (e.g., RTEMS and uClinux) adopt it too. Note that both the involved threads should be allowed to access the thread queue and TCBs when calling the scheduler.

Pattern (III) uses a separate thread, called scheduler thread, to do scheduling. One thread may perform scheduling by doing context switch to the scheduler thread. The scheduler thread is a big infinite loop: finding the next thread; performing context switch to the next thread; and looping after return. This pattern can be seen in the GNU-pth thread library, MIT-xv6 kernel, L4::Ka, etc. Similar to pattern (II), all involved threads in this pattern should be allowed to access the TCB of the scheduler thread and the thread queue.

2.2 Challenges

As we can see from the patterns in Fig. 1 the control flow in the scheduling process is very complicated. Threads switch back and forth via manipulating the thread queues and TCBs. It is very natural to share TCBs and the thread queue among threads in order to support all these scheduling patterns. On the other hand, it is important to ensure that the TCBs are accessed in the right way. The system would go wrong if, for instance, a thread erased the context data of another by mistake, or put a dead thread back into the ready thread queue.

To guarantee the safety of the scheduling process, we must fulfill two requirements:

(1) No thread can incorrectly modify the context data in TCBs.
(2) The scheduler should know the status of each thread in the thread queues and decide which to run next.

To satisfy the requirement (1), some previous work [3,7] adopts a two-layer-based approach and protects the TCBs through abstraction, where the TCBs are simply hidden from kernel threads and become inaccessible. This approach can be used to verify schedulers of pattern (I), for which we show the abstraction line in Fig. 2 (a). Threads above the line cannot modify TCBs, while the scheduler is below this line and has full accesses to them. The lower-layer scheduler provides an abstract interface to the verification of concurrent thread code at the upper layer. Since it modifies the TCBs in the scheduling time only, we can view the scheduler as a sequential function which does not belong to any thread and can be verified by a conventional Hoare-style logic. However, this approach cannot verify the other two patterns, nor it fulfills the requirement (2) for concurrent schedulers, where the TCBs are manipulated concurrently (not sequentially as in pattern (I)) and should be known by threads. That is, we cannot completely hide the TCBs from the upper-layer concurrent threads for patterns (II) and (III).

2.3 Our approach

If we inspect the TCB data carefully, we can see only a small part of the data is crucial to thread behaviors and cannot be accessed concurrently. It is unnecessary to access it concurrently either. The data includes the machine context data and the stack location. We call them safety-critical values. Some values can be modified concurrently, but the correctness of those is still important to the safety of the kernel, e.g., the pointers used to organize thread queues and the status field belong to this kind of values. Other values of TCBs have nothing to do with the safety of the kernel and can be modified concurrently definitely, e.g., the name of a thread or debug information.

Lowering the abstraction level. To protect the safety critical part of TCBs, we lower the abstraction line, as shown in Fig. 2 (b). In our framework, the safety-critical data of TCBs is under the abstraction line and hidden from threads. The corresponding operations such as context saving, loading and switching are abstracted away from threads too, with only interfaces exposed to the upper layer. The other part of TCBs are lifted above this line, which can be accessed by concurrent threads.
Building abstract threads. We still need to ensure the concurrent accesses of non-safety-critical TCB data are correct. For instance, we cannot allow a dead thread to be put onto a ready thread queue. To address this issue, we build abstract threads to carry information of threads from TCBs to guide modifications by each other. In Fig. 3 we use the notation \([t]\) to specify the running thread, and the notation \(\langle t \rangle\), for a ready thread. Here \(t\) is the identifier of the thread. With the knowledge about the existence of a ready thread \(B\) pointed by \(\text{next}\) (i.e., \(\langle B \rangle\)), we know it is safe to switch to it via the operation \(\text{cswitch}(A, \text{next})\). Since abstract threads can be described in specifications, it allows us to write more intuitive and readable specifications for kernel code.

Treating abstract threads as resources. Like heap resources, abstract thread resources can be either local or shared. We can do ownership transfers on thread resources. When context switches, one thread will transfer some of the abstract thread resources (shared) along with the shared memory to the next thread. As shown in Fig. 3 when thread \(A\) context switches to thread \(B\), the notation \([A]\) will be changed to \(\langle A \rangle\) after context saving; \(\langle A \rangle\) and \(\langle B \rangle\) are transferred to the thread \(B\) along with the shared memory resource \(\text{next}\); then \(\langle B \rangle\) will be changed to \([B]\) after context loading. With transferred thread resources, thread \(B\) will know there is a ready thread \(A\) to switch to. Therefore, by treating abstract threads as resources, we find a simple and natural way to specify and reason about context switches. We design a proof system similar to CSL for modular verification with the support of ownership transfers on thread resources.

Defining concrete thread resources. To establish the soundness of our proof system, we must ensure that the abstract threads can be reified by real concrete threads. The concrete representation of abstract threads, including stack, TCBs etc., can be defined from a global point of view. In Fig. 3 suppose that thread \(A\) is running, we ensure that there are two blocks of resources in the system. One of them is the running thread \(\text{CThrd}_A\) and the other is a ready thread \(\text{RThrd}_B\). They correspond to the abstract threads \([A]\) and \(\langle B \rangle\) in the assertions of thread \(A\). We use the concrete thread resources to specify the global invariant of the machine, which allows us to prove the soundness of our proof system.

In summary, we propose a new verification framework based on the ideas above, which allows us to verify concurrent thread management of kernels in a modular way.

3 Machine model

In this section, we define a two-layer machine model. The physical machine we use is similar to realistic hardware, where no concept of thread exists. Based on it, we define an abstract machine with logical abstract threads, whose meta-data is abstracted into a thread pool. Moreover, the operation of context switch is abstracted as a primitive abstract instruction.

3.1 Meta language

First of all, a mechanized meta-language is required to formalize all the concepts in this paper, such as machine models, programs, specifications, inference rules, theorems
Fig. 4. Physical machine model

and proofs. The meta-language we use is the calculus of inductive constructions (CiC), which is supported by the proof assistant Coq.

\[
\text{Term } A, B ::= \text{Set} \mid \text{Prop} \mid \text{Type} \\
| \quad X \mid \forall X : A \cdot B \mid A \cdot B \mid A \to B \mid A \land B \mid A \lor B \mid \text{True} \mid \text{False} \\
| \quad \forall X : A \cdot B \mid \exists X : A \cdot B \mid \text{inductive def.} \mid \cdots
\]

CiC is a typed lambda calculus, and its syntax follows the convention of common lambda calculi. For example, \(A \to B\) represents function spaces. It also means logical implication when \(A\) and \(B\) have sort \(\text{Prop}\). In addition, \(\text{Prop}\) is the universe of all propositions, \(\text{Set}\) is the universe of all data sets, and \(\text{Type}\) is the (stratified) universe of all terms.

### 3.2 Physical machine

The formal definition of the physical machine is shown in Fig. 4. A physical machine configuration \(W\) consists of a code block \(C\), a mutable memory \(M\), a register file \(R\), a program counter \(pc\). The simplified machine has 6 general registers. The register \(v0\) is used for passing the return-value of functions; \(a0, a1, a2\) are used for passing arguments. The register \(sp\) always points to the top of stack, growing downwards. The register \(ra\) is used for storing the return address of functions. The code block \(C\) is a mapping from code labels \(f\) to instructions \(i\). Some common instructions are defined to write programs in this paper, including arithmetic, memory instructions for loading and storing values, jump and function call/ret instructions. We assume that the length of every instruction is equal to one, \(i.e.,\) if an instruction is at \(f\), the next instruction will be at \(f + 1\). For simplicity, we omit many realistic hardware details, \(e.g.,\) address alignment and bit-arithmetic.

For all kinds of partial mappings, we use \(\sqcup\) to denote the disjoint union of two mappings, and use the notation like \(R[r : v]\) to denote the updating of a mapping.

We use a relation \(W \rightarrow W'\) to specify the operational semantics.
(AbsMach) \( W := (C, S, pc) \)
(State) \( S := (M, R, P) \)
(AbsCode) \( C := \{ f : c \}^* \)
(Mem) \( M := \{ l : w \}^* \)
(RegFile) \( R := \{ r : w \}^* \)
(TID) \( t := w \)
(Pool) \( P := \{ t : T \}^* \)
(Thrd) \( T := \text{run} \mid \text{rdy}, R \)
(AbsInstr) \( c := \text{cswitch} \mid i \)

Fig. 5. Abstract machine model

\[ \frac{C(pc) = i \quad (M, R, pc) \xrightarrow{\delta} (M', R', pc')} {(C, M, R, pc) \rightarrow (C, M', R', pc')} \]

where the state transition relation \((M, R, pc) \xrightarrow{\delta} (M', R', pc')\) is defined in Fig. 6.

### 3.3 Abstract machine

The abstract machine is shown in Fig. 5 (right side), where threads are introduced at this level. It is more intuitive to build a proof system (Sec. 4) to verify concurrent kernel code at this level. A abstract machine configuration \( W \) is a triple of a read-only code block \( C \), a mutable machine state \( S \), and a program counter \( pc \). The code block of the abstract machine is a partial mapping from labels \( f \) to abstract instructions \( c \). A machine state \( S \) consists of a memory block \( M \), a register file \( R \) and a thread pool \( P \). A memory block is a partial mapping from memory addresses \( l \) to machine words \( w \). A thread pool \( P \) is a partial mapping from thread IDs \( t \) to abstract threads \( T \). Each abstract thread has a tag specifying its status, which is either running (run) or ready (rdy). Each ready thread has a copy of saved register file as its machine context data. The abstract instructions include an abstract operation of context switch (csswitch) and other physical machine instructions defined on the left. We model the operational semantics using the step transition relation \( W \rightarrow W' \) defined in Fig. 8. In the physical machine, this abstract instruction can be implemented using normal machine instructions.

\[
\text{csswitch:}
\begin{align*}
\text{sw} & \text{ ra, 0(a0)} \quad \mid \quad \text{lw} & \text{ ra, 0(a1)} \\
\text{sw} & \text{ v0, 4(a0)} \quad \mid \quad \text{lw} & \text{ v0, 4(a1)} \\
\text{sw} & \text{ a0, 8(a0)} \quad \mid \quad \text{lw} & \text{ a0, 8(a1)} \\
\text{sw} & \text{ a1, 12(a0)} \quad \mid \quad \text{lw} & \text{ a1, 16(a1)} \\
\text{sw} & \text{ a2, 16(a0)} \quad \mid \quad \text{lw} & \text{ sp, 20(a1)} \\
\text{sw} & \text{ sp, 20(a0)} \quad \mid \quad \text{lw} & \text{ a1, 12(a1)} \quad \mid \quad \text{ret}
\end{align*}
\]

The machine context, a register file, can be defined as the structure as below:
The abstract instruction `cswitch` requires two thread IDs passed as arguments in `a0` and `a1`, one of which is tagged by `rdy` and the other is tagged by `ady` in the thread pool. After `cswitch`, the two abstract threads exchange tags, and the control of processor is passed from the old thread to the new one. The registers of old thread are saved in the source abstract thread and the registers in the destination thread are loaded into machine state. Except for `cswitch`, the state transitions of other instructions are similar to those of the physical machine.

### 3.4 Machine translation

In our proof system, once a program is proved safe at the abstract machine level, it should be proved safe as well at the physical machine level. We define a relation between abstract machine with physical machine, \( W \downarrow W \) in Fig. 9. A code block of the abstract machine is translated to a code block at the physical level by the replacing of `cswitch` with a function call to the code of implementation of context switch, `c_cs`. A thread pool \( P \) is translated into a block of memory with the context data of all threads, each of which is specified by `mc(l, R)`. The whole memory of the physical machine consists of two parts: the memory translated from \( P \) and the memory translated from \( M \). The translations of \( M \) and \( R \) are straightforward.

We give a formal definition of the safety property:
Fig. 7. Definition of abstract machine

\[
(M,R,P) \xrightarrow{c} (M',R',P') \land P = P'
\]

\[
\begin{align*}
\text{i} & \quad ((M,R),pc) \xrightarrow{c} ((M',R'),pc') \\
\text{cswitch} & \quad \exists R',P'. M = M' \land R' = R[ra:pc+1] \land t = R(a0) \\
& \quad \land t' = R(a1) \land pc' = R'\{ra\} \\
& \quad \land P = \{t : \text{run}, t' : (rdy,R')\} \uplus P' \\
& \quad \land P' = \{t : (rdy,R'\{t\}), t' : \text{run}\} \uplus P'' \\
& \text{R and R' is complete.}
\end{align*}
\]

Fig. 8. Operational semantics of abstract machine (part)
Lemma 1 (Deterministic physical machine). For any physical machine configuration \( \mathbb{W} \), if there exists \( \mathbb{W}' \) and \( \mathbb{W}'' \) such that \( \mathbb{W} \rightarrow \mathbb{W}' \) and \( \mathbb{W} \rightarrow \mathbb{W}'' \), then the two machine configurations are \( \mathbb{W}' = \mathbb{W}'' \).

Proof. By induction on the transition relation of physical machine. \( \Box \)

Here we prove that the implementation of context switch is consistent to the operational semantics of context switch defined in Fig. 8.

Lemma 2 (Mapping of context switch). If \( W = (C, S, pc) \), \( \mathbb{W} \), \( W \downdownarrows \mathbb{W} \), and \( C(pc) = c\text{switch} \), if \( W \) can run one step to \( W' \), then \( \mathbb{W} \) can also run \( n \) steps (\( n > 0 \)) to some \( \mathbb{W}' \) such that \( \mathbb{W} \rightarrow^* \mathbb{W}' \) and \( W' \downdowndownarrows \mathbb{W}' \).

Proof. By the operational semantics of the instruction \( c\text{switch} \) and the implementation of context switch routine. \( \Box \)

Lemma 3 (Mapping of One Step). For any abstract machine configuration \( W \) and physical machine configuration \( \mathbb{W} \), if they satisfies the translation relation \( W \downdowndownarrows \mathbb{W} \), and for any \( W' \) such that \( W \rightarrow \mathbb{W}' \), then there must exists a \( \mathbb{W}' \) that \( \mathbb{W} \rightarrow^* \mathbb{W}' \) and \( W' \downdowndownarrows \mathbb{W}' \).
Proof. By induction on the operational semantics of the abstract machine. 

By the translation, we can prove that if the abstraction machine is safe, then its counter-part of physical machine is also safe.

**Theorem 1 (Safety Preservation).** For any machine configuration $W$ and $\forall W$, if $W \Downarrow W$ and Safe($W$) then Safe($W$).

**Proof.** Induction on step, and by Lemma $3$. 

4 Proof system

In this section, we extend the assertion language of CSL to specify the thread resources, and propose a small proof system supporting verify concurrent code with modification of TCBs at the assembly level.

4.1 Assertion language and code specification

We use $p$ and $q$ as assertion variables, which are predicates over machine states. The assertion constructs, adapted from separation logic\cite{18}, are *shallowly embedded* in the meta language, as shown in Fig. 10. The notation of $\emptyset$ asserts an empty state, i.e. all components are empty, where we use the notation $\{\cdot\}$ to denote an empty mapping. The assertion of separation conjunction $p * q$ specifies the state $S$ which can be split into two disjoint parts, satisfying $p$ and $q$ respectively. The separation implication $p \rightarrow q$ means that: if merged resources specified by $p$, the state will satisfy $q$. In our assertion language, there are two special assertion constructs for abstract threads. One of them is $\langle t \rangle$ specifying resources of a ready thread and the other is $\lbrack t \rbrack$ specifying resources of current running thread. Since threads are explicit resources in the abstract machine, their machine context data (values in registers) are preserved across context switch. Hence the resources of registers shouldn’t be shared. We explicitly mark a pure assertion by $\sharp$, which forbids an assertion specifying resources. An unary notation $(\diamond p)$ mark an assertion $p$ that only specifies shared resources but no thread local resources (e.g., registers). Registers are also treated as resources, and $r \mapsto w$ specifies a register with the value of $w$. The notation $\tau_1, \ldots, \tau_n \mapsto w_1, \ldots, w_n$ is a compact form for multiple registers.

We borrow the idea from SCAP\cite{4} that a $(p,g)$ pair is used to specify instructions at assembly-level. The pre-condition $p$ describes the state before the first instruction of an instruction sequence, while the action $g$ describes the actions done by the whole instruction sequence. In the proof system, each instruction is associated with a $(p,g)$ pair, where $g$ describes the actions from this instruction to a $\text{ret}$ instruction. For all instructions in $C$, all $(p,g)$ pairs are put in a global mapping, $\Psi$, from labels to specifications.

\[
\begin{align*}
\text{(Assert)} & \ p,q \in \text{State} \rightarrow \text{Prop} \\
\text{(Action)} & \ g \in \text{State} \rightarrow \text{State} \rightarrow \text{Prop} \\
\text{(Spec)} & \ \Psi ::= \{f : (p,g)\}^*
\end{align*}
\]

The specification form $(p,g)$ is different from the traditional pre-condition and post-condition, which are both assertions and related by auxiliary variables. We can still use a notation to specify instructions in the traditional style,
emp  ≜  λ(M,R,P).M=\{\} ∧ R=\{\} ∧ P=\{}
true  ≜  λS. True
false  ≜  λS. False
p □ q  ≜  λS. (p S) ∧ (q S)
p □ q  ≜  λS. (p S) ∨ (q S)
∃v, p ≜  λS. ∃v. p S
p + q  ≜  λ(M,R,P).∃M_1, M_2, R_1, R_2, P_1, P_2.
\quad M=M_1 \uplus M_2 ∧ R=R_1 \uplus R_2 ∧ P=P_1 \uplus P_2
\quad ∧ p (M_1, R_1, P_1) ∧ q (M_2, R_2, P_2)
p → q  ≜  λ(M,R,P).∀M_1, R_1, P_1, M', R', P'.
\quad (M'=M_1 \uplus M ∧ R'=R_1 \uplus R ∧ P'=P_1 \uplus P)
\quad → p (M_1, R_1, P_1) → q (M', R', P')

\varepsilon p  ≜  λ(M,R,P). p ∧ M=\{\} ∧ R=\{\} ∧ P=\{}
\circ p  ≜  λ(M,R,P). p (M, R, P) ∧ R=\{\}
1 \mapsto w  ≜  λ(M,R,P).M=\{ 1 : w \} ∧ 1 \neq NULL ∧ R=\{\} ∧ P=\{}
x \mapsto w  ≜  λ(M,R,P).R=\{ x : w \} ∧ M=\{\} ∧ P=\{}
x \mapsto _\bot  ≜  λ(M,R,P).∃w. R=\{ x : w \} ∧ M=\{\} ∧ P=\{}
x \leftarrow w  ≜  λ(M,R,P).∃R'. R=\{ x : w \} \uplus R'
1 \mapsto _\bot  ≜  λ(M,R,P).∀w. M=\{ 1 : w \} ∧ 1 \neq NULL ∧ R=\{\} ∧ P=\{}
1 \mapsto w  ≜  λ(M,R,P).∃M'. M=\{ 1 : w \} \uplus M' ∧ 1 \neq NULL
[t]  ≜  λ(M,R,P).P=\{ t : \text{run} \} ∧ M=\{\} ∧ R=\{\}
{t}  ≜  λ(M,R,P).P=\{ t : (\text{rdy}, _\bot) \} ∧ M=\{\} ∧ R=\{\}

\varepsilon_1, \ldots, \varepsilon_n \mapsto w_1, \ldots, w_n  ≜  (\varepsilon_1 \mapsto w_1) * \cdots * (\varepsilon_n \mapsto w_n)
1 \mapsto (\varepsilon)  ≜  (1 \mapsto _\bot) * (1+4 \mapsto _\bot) * \cdots * (1+4(n-1) \mapsto _\bot)

Fig. 10. Definition of assertion constructs
\[
\left\{ \begin{array}{c}
p \\ q \end{array} \right\}^{(v_1, \ldots, v_n)} \triangleq ((\lambda S. \exists v_1, \ldots, v_n. \exists p'. (p(v_1, \ldots, v_n) \ast p') S), \\
(\lambda S, S'. \forall p'. \forall v_1, \ldots, v_n. (p(v_1, \ldots, v_n) \ast p') S \rightarrow (q(v_1, \ldots, v_n) \ast p') S'))
\]

where \( p \) is the pre-condition of instructions, \( q \) is the post-condition, and \( v_1, \ldots, v_n \) are auxiliary variables occurring in the precondition and the postcondition. For example, the action of an addition instruction \text{add}i a_0, 2 can be specified as follows:

\[
\left\{ \begin{array}{c}
a_0 \mapsto v \\ a_0 \mapsto v+2 \\
\end{array} \right\}^{(v)}
\]

where \( v \) is an auxiliary variable to relate the pre-condition and post-condition. Then the \( g \) says that for any \( v \), if the state before the addition instruction satisfies \( a_0 \mapsto v \), the state after addition will satisfy \( a_0 \mapsto v+2 \). We define a binary operator for composing two pairs into one.

\[
(p, g) \triangleright (p', g') \triangleq (\lambda S. p S \land (\forall S'. g S \rightarrow p' S'), \\
\land \lambda S, S'. p S \rightarrow (\exists S'. g S \land g' S' S'))
\]

If an instruction sequence satisfies \((p, g)\) and the following instruction sequence satisfies \((p', g')\), then the composed instruction sequence would satisfy \((p, g) \triangleright (p', g')\). The weakening relation between two pairs is defined as below:

\[
(p, g) \Rightarrow (p', g') \triangleq \forall S. p S \rightarrow p' S \land (\forall S'. g S \rightarrow g S S')
\]

The relation implies that: the precondition \( p \) is stronger than \( p' \) and the action \( g \) is weaker than \( g' \). For example, two actions of additions can be bound into one and satisfy the following weakening relation:

\[
\left\{ \begin{array}{c}
a_0 \mapsto v \\ a_0 \mapsto v+2 \\
\end{array} \right\}^{(v)} \triangleright \left\{ \begin{array}{c}
a_0 \mapsto v' \\ a_0 \mapsto v'+3 \\
\end{array} \right\}^{(v')} \Rightarrow \left\{ \begin{array}{c}
a_0 \mapsto v \\ a_0 \mapsto v+5 \\
\end{array} \right\}^{(v)}
\]

### 4.2 Invariant for shared resources

As mentioned previously, our proof system draws ideas of ownership transfer from CSL. By defining invariants for shared resources, our proof system ensures safe operations of TCBs.

Different from original concurrent separation logic, our machine doesn’t support indeterministic concurrency. And since there is only one single processor in our machine model and no interrupt support either, all threads in the machine run cooperatively. Every running thread will obtain the ownership of shared resources. If one running thread invokes context switch to another, the ownership of shared resources will be transferred automatically.

Unlike the invariant in concurrent separation logic, the invariant of shared resources defined in our proof system is parameterized by two thread IDs: \( I(t_s, t_d) \). Briefly, the invariant describes the shared resources before context switch with the direction from the thread \( t_s \) to \( t_d \). One of the benefits of parameters is that the invariant is thread-specific.
Like the abstract invariant \( I \) in CSL, the invariant \( I(t_s, t_d) \) is abstract and can be instantiated to concrete definitions so as to verify various programs, if it satisfies a requirement of being precise \([18]\).

Precisely, the invariant \( I(t_s, t_d) \) describes the shared resources when the context switch invoked from the thread \( t_s \) to the thread \( t_d \), but excluding the resources of the two threads. Since the control flow from one thread to another is deterministic by context switch, every two threads may negotiate a particular invariant that is different from pairs of other threads. We can define different assertions (of shared resources) which depends on the source and the destination thread related to context switch. This is quite different from concurrent code at user-level, where every switch depends on unknown scheduling algorithm.

### 4.3 Inference rules

The judgements of instruction in our proof system are of the following form: \( \Psi, I \vdash \{ (p, g) \} \text{pc} : c \), where \( \Psi \), \( I \) are implementation-specific. The judgement states that an instruction sequence, started with \( c \) at the label of \( \text{pc} \) and ended with a \text{ret}, satisfies specification \((p, g)\) under \( \Psi \) and \( I \). The inference rules are shown in Fig. 11.

The rule of \((\text{CDHP})\) is for reasoning about the entire kernel code. The premise of the rule says that each instruction at \( f \) in \( C \) should satisfy its corresponding specification in \( \Psi, \Psi(f) \).

The rules of \((\text{ADD})\), \((\text{ADDI})\), \((\text{MOV})\), \((\text{MOVI})\), \((\text{LW})\), and \((\text{SW})\) are for non-jump machine instructions. The premises of them are similar to their operational semantics and easy to understand. For example, in the rule of \((\text{ADD})\), the premise says that the specification \((p, g)\) implies the action of the \text{add} instruction composed with the specification of the next instruction, \( \Psi(\text{pc}+1) \). The action of \text{add} instruction is that if the destination register \( r_d \) contains the value of \( w_1 \), and the source register \( r_s \) contains the value of \( w_2 \), then after the instruction, \( r_d \) will contain the sum of \( w_1 \) and \( w_2 \), while \( r_s \) will keep unchanged.

Functions are reasoned with the rules of \((\text{CALL})\) and \((\text{RET})\). The \((\text{CALL})\) rule says that the specification \((p, g)\) implies the action that is composed by (1) the action of instruction \text{call}, (2) the specification of the \text{function} invoked \( \Psi(f) \), (3) the action of instruction \text{ret}, and (4) the specification of the next instruction \( \Psi(\text{pc}+1) \). The \((\text{RET})\) rule says that the specification \((p, g)\) implies an empty action, which means the actions of the current function should be fulfilled. Although, like SCAP, our proof system hasn’t the frame rule explicitly, the proof system can still support local reasoning.

The most important rule in our proof system is the rule of context switch, \((\text{CSW})\). It states that the precondition of \( \text{cswitch} \) should include the following resources:

- \( [t] \): the current thread, whose thread-ID is stored in the register a0;
- \( \langle t' \rangle \): a ready thread, with ID \( t' \), which is stored in the register a1;
- \( \diamond I(t, t') \): a part of resources without any resource of registers.

To the same thread \( t \), after return from context switch, the thread will regain the control. Thus, from a local point of view from one thread, the postcondition of context switch should include:
– \( \text{[t]} \): the current thread;
– registers a0 and a1 are unchanged;
– \( \langle t'' \rangle \): there exists an unknown thread \( t'' \) which just called context switch before the thread \( t \) re-obtains the control;
– \( \odot I(t', t) \): a part of the shared resources, which implies the direction of the last context switch is from \( t'' \) to \( t \).

4.4 Invariant of global resources and soundness

Each abstract thread corresponds to one part of global resources representing the concrete resources allocated for this thread. For example, to an abstract thread \( \langle t \rangle \), there exist resources of its TCB, stack, and private resources. Therefore, all resources can be divided into parts and each of them is associated to one thread. The global invariant is such a logical expression that describes the partition of all resources globally, defined in Fig. 12. The invariant is the key for proving the soundness theorem of our proof system.

Continuation. First, for each thread, we define a predicate \( \text{Cont} \) to specify its resources and control flow, i.e. the continuation of this thread. The first parameter \( n \) of this predicate specifies the number of functions nested in the thread’s control flow. If \( n \) is equal to zero, it means that the thread is running in the topmost function, which could be an infinite loop and cannot return. If the number \( n \) is greater than zero, the predicate says that there is a specification \( (p, g) \) in \( \Psi \) at \( pc \), such that the resources of the thread satisfies \( p \); and \( g \) guarantees that the thread will continue to satisfy \( \text{Cont} \) recursively after it returns to the address \( \text{retaddr} \).

Running thread. The concrete resources of a running thread are specified by a continuation \( \text{Cont} \) with an additional condition, the running thread owns all registers. The parameter \( pc \) points to the next instruction the thread is going to run. Here we use an abbreviation \( \lfloor R \rfloor \) to denote the resources of all registers, except that the value in \( ra \) is of no interest.

\[
\lfloor R \rfloor \triangleq (ra \mapsto \ _ ) * (v0 \mapsto R(v0)) * (sp \mapsto R(sp)) * (a0 \mapsto R(a0)) * (a1 \mapsto R(a1)) * (a2 \mapsto R(a2))
\]

Ready thread. For a ready thread (or a runnable thread), its concrete resources are defined by separation implication \( \rightarrow * \): if given (1) the resources of saved machine context \( \lfloor R \rfloor \), (2) the abstract resource of itself \( \text{[t]} \), (3) another ready thread \( t' \) and (4) shared resources specified by \( \odot I(t', t) \), the resources of the ready thread can be transformed into the resources of a running thread. It thread ID is specified by the second parameter of \( \text{RThrd} \), and the third parameter is the machine context data saved in its TCB. Please note that the program counter of a ready thread is saved into the register \( ra \).
\[
\forall f \in \text{dom}(C). \quad \Psi, I \vdash \{ \Psi(f) \} f : C(f) \quad \text{(CDHP)}
\]

\[
(p, g) \Rightarrow \begin{cases} (r_d \mapsto w_1) \ast (r_s \mapsto w_2) \\ (r_d \mapsto w_1 + w_2) \ast (r_s \mapsto w_2) \end{cases} \vdash \Psi(pc + 1)
\]

\[
\Psi, I \vdash \{(p, g)\} \text{pc : add } r_d, r_s \quad \text{(ADD)}
\]

\[
(p, g) \Rightarrow \begin{cases} (r_d \mapsto w_1) \\ (r_d \mapsto w_1 + w) \end{cases} \vdash \Psi(pc + 1)
\]

\[
\Psi, I \vdash \{(p, g)\} \text{pc : } \text{add } r_d, w \quad \text{(ADDI)}
\]

\[
(p, g) \Rightarrow \begin{cases} (r_d \mapsto w_1) \ast (r_s \mapsto w_2) \\ (r_d \mapsto w_2) \ast (r_s \mapsto w_2) \end{cases} \vdash \Psi(pc + 1)
\]

\[
\Psi, I \vdash \{(p, g)\} \text{pc : mov } r_d, r_s \quad \text{(MOV)}
\]

\[
(p, g) \Rightarrow \begin{cases} (r_d \mapsto w) \\ (r_d \mapsto w) \end{cases} \vdash \Psi(pc + 1)
\]

\[
\Psi, I \vdash \{(p, g)\} \text{pc : movi } r_d, w \{g\} \quad \text{(MOVI)}
\]

\[
(p, g) \Rightarrow \begin{cases} (r_t \mapsto w) \ast (r_s + w \mapsto w_1) \\ (r_t \mapsto w_1) \ast (r_s + w \mapsto w_1) \end{cases} \vdash \Psi(pc + 1)
\]

\[
\Psi, I \vdash \{(p, g)\} \text{pc : lw } r_t, w(r_s) \quad \text{(LW)}
\]

\[
(p, g) \Rightarrow \begin{cases} (r_t \mapsto w) \ast (r_s + w \mapsto w) \\ (r_t \mapsto w_1) \ast (r_s + w \mapsto w_1) \end{cases} \vdash \Psi(pc + 1)
\]

\[
\Psi, I \vdash \{(p, g)\} \text{pc : sw } r_t, w(r_s) \quad \text{(SW)}
\]

\[
(p, g) \Rightarrow \begin{cases} ra \mapsto w \\ ra \mapsto pc + 1 \\ ra \mapsto \_ \end{cases} \vdash (\Psi(f) \triangleright \{\Psi(f)\} \text{pc : call } f)
\]

\[
\Psi, I \vdash \{(p, g)\} \text{pc : ret} \quad \text{(RET)}
\]

\[
(p, g) \Rightarrow \{\Psi(f)\} \triangleright \Psi(f) \vdash \{(p, g)\} \text{pc : jmp } f \quad \text{(JMP)}
\]

\[
\Psi, I \vdash \{(p, g)\} \text{pc : \text{bz } } r_t, f \quad \text{(BZ)}
\]

\[
(p, g) \Rightarrow \begin{cases} [t] * (a_0, a_1, ra \mapsto t, t', \_ \_ \_ ) \ast (t') \ast \circ I(t, t') \\ [t] * (a_0, a_1, ra \mapsto t, t', \_ \_ \_ ) \ast \exists t''. (g'') \ast \circ I(t'', t) \end{cases} \vdash \Psi(pc + 1)
\]

\[
\Psi, I \vdash \{(p, g)\} \text{pc : cs} \quad \text{(CSW)}
\]

Fig. 11. Inference rules selected}
For any machine configuration Immediate. Proof. will satisfy then after machine run a context switch (R)

resources specified by one first prove that if every machine configuration satisfies GINV one running abstract thread, since the physical machine has only one single processor. Our proof system ensures that the machine state always satisfies the global invariant, (GINV)

The soundness property of our proof system states that any program that is well-formed in our proof system will run safely on the abstract machine. The property can be proved by the global invariant GINV, which always holds through machine execution. We can first prove that if every machine configuration satisfies GINV, it can run forward for one step. And we can also prove that if a machine configuration (satisfying GINV) can proceed, the next machine configuration will also satisfy GINV. Hence by the invariant GINV, the soundness theorem of our proof system can be proved. The proof of the soundness theorem has been formalized in Coq.

\begin{align*}
\text{Cont}(n+1, \Psi, \text{pc}) & \triangleq \lambda S. \Psi((p, g) \land (p S) \\
& \quad \land \left(\forall S'. g S S' \rightarrow (\exists \text{retaddr}. (ra \leftrightarrow \epsilon)) \land \text{Cont}(n, \Psi, \text{retaddr})\right) S') \\
\text{Cont}(0, \Psi, \text{pc}) & \triangleq \lambda S. \Psi((p, g) \land (p S) \land \left(\forall S'. g S S' \rightarrow \text{False}\right)) \\
\text{CThrd}(\Psi, t, \text{pc}) & \triangleq \exists n. \text{Cont}(n, \Psi, \text{pc}) \land (\exists t) \land \exists R. \left(\exists R. \left(\exists t\right) \land \left(\exists R. \right) \land \left(\exists t\right) \land \left(\exists R. \right) \land \left(\exists t\right) \land \left(\exists R. \right) \land \left(\exists t\right) \land \left(\exists R. \right)\right) \\
\text{RThrd}(\Psi, t, \text{R}) & \triangleq [R] \ast [\exists t]. \ast [\exists t]. \ast I(t', t) \rightarrow \text{CThrd}(\Psi, t, \text{R}(\text{ra})) \\
\text{GINV}(\Psi, \text{P}, \text{pc}) & \triangleq \text{CThrd}(\Psi, t, \text{pc}) \ast \text{RThrd}(\Psi, t_0, R_0) \ast \cdots \ast \text{RThrd}(\Psi, t_n, R_n) \\
\text{where } P = \{t : \text{run}, t_0 : (\text{rdy}, R_0), \ldots, t_n : (\text{rdy}, R_n)\}
\end{align*}

Fig. 12. Concrete threads and the global invariant

Invariants of global resources. The whole machine state can be partitioned, and each of parts is owned by one thread, which is either running or ready. Thus, the global invariant GINV is defined in the form of separation conjunction by CThrd and RThrd. The structure of GINV is isomorphic to the thread pool P; one abstract running thread is mapped to resources specified by one CThrd; one abstract ready thread is mapped to resources specified by one RThrd. Note that GINV requires that there be one and only one running abstract thread, since the physical machine has only one single processor. Our proof system ensures that the machine state always satisfies the global invariant, (GINV(\Psi, P, \text{pc}) (M, R, P)).

\textbf{4.5 Soundness}

The soundness property of our proof system states that any program that is well-formed in our proof system will run safely on the abstract machine. The property can be proved by the global invariant GINV, which always holds through machine execution. We can first prove that if every machine configuration satisfies GINV, it can run forward for one step. And we can also prove that if a machine configuration (satisfying GINV) can proceed, the next machine configuration will also satisfy GINV. Hence by the invariant GINV, the soundness theorem of our proof system can be proved. The proof of the soundness theorem has been formalized in Coq.

\textbf{Lemma 4 (Context switch over shared invariant).} For any machine configuration (C, (M, R, P), \text{pc}), and kernel specification \Psi, \Psi. I, I \vdash C, state satisfies the global invariant (M, R, P) \vdash [R] \ast [\exists t]. \ast [t'] \ast I(t', t) \ast P' then after machine run a context switch (M, R, P) \xrightarrow{\text{csWitch}} (M', R', P'), the machine state will satisfy (M', R', P') \vdash [R'] \ast [t'] \ast I(t', t) \ast p',

where R(a0) = t and R(a1) = t'.

\textbf{Proof.} Immediate. \hfill \square

\textbf{Lemma 5 (Context switch over global invariant).} For any machine configuration (C, (M, R, P), \text{pc}), and kernel specification \Psi, \Psi. I, I \vdash C, state satisfies the global invariant
if machine can run a context switch command, \((M,R,P) \xrightarrow{\text{cswitch}} (M',R',P')\) then the state after the command still satisfies the global invariant:

\[(M',R',P') \models \text{GINV}(\Psi, P', pc').\]

Proof. Immediate. \(\square\)

**Theorem 2 (Progress).** For any machine configuration \((C,S,pc)\), if \(\Psi, I \vdash C\) and \((C,S,pc) \models \text{GINV}(\Psi, P, pc)\), then the machine can go forward for one step: \((C,S,pc) \xmapsto{} (C,S',pc')\), where \(S = (M,R,P)\).

Proof. \(\square\)

**Theorem 3 (Preservation).** For any machine configuration \((C,S,pc)\), whose code is verified \(\Psi, I \vdash C\) and state satisfies the global invariant \((C,S,pc) \models \text{GINV}(\Psi, P, pc)\), if it steps to \((C,S,pc) \xmapsto{} (C,S',pc')\), then the new state also satisfies the global invariant: \((C,S,pc') \models \text{GINV}(\Psi, P', pc')\), where \(S = (M,R,P)\) and \(S' = (M',R',P')\).

Proof. \(\square\)

**Theorem 4 (Soundness).** For any machine configuration \((C,(M,R,P),pc)\), if its code is verified, \(\Psi, I \vdash C\), and its state satisfies the global invariant, \((M,R,P) \models \text{GINV}(\Psi, P, pc)\), then it is safe to run.

Proof. By Lemma 2 and Lemma 3.

## 5 Verification cases

In this section, we show how to use the proof system to verify two schedulers of pattern (II) and (III) shown in Fig. 1. The examples are small and simple to help readers understand our framework easily. Still, they can be extended to realistic ones if we improve our verification framework on a more realistic machine model and enrich inference rules as well.

### 5.1 Scheduler as function

We explain the code in C style here, but verify it at assembly-level. This pattern of scheduling is modeled from code scheduler of many realistic OS kernels (FreeBSD, Linux, RTEMS etc.). Since we only implement thread scheduling, we define the structure of TCBs as simple as possible. It only contains machine context and two pointers for organizing the thread queue. Please note that thread IDs are the memory addresses of TCBs.

The scheduler function follows the process discussed in Sec. 2. The functions `deq()` and `enq()` are used to remove and insert nodes in thread queues. The main task of the scheduler is choosing a candidate from the thread queue and doing context switch from the current thread to the candidate. There are two global variables, `cur` and `rq`. The
variable *cur* points the TCB of the running thread; while the variable *rq* points to the thread queue containing TCBs of all other threads. Suppose *schedule_p2()* is invoked by one thread *t_A*, it firstly records the value of variable *cur* to a local variable *old* before modifying *cur*. Then it picks a new thread control block from the ready thread queue by calling function *deq()*; If the queue is empty, this function will return immediately. Otherwise, it puts the TCB of the running thread into the thread queue by calling *enq()*, and modifies *cur* to the new thread (*t_B*). Next, the function does context switch from the current running thread *t_A* to the new thread *t_B*. From then on, the thread *t_A* becomes ready and then waits for the next cycle, and *t_B* becomes a running thread. After a while, if another thread (*t_C*) happens to choose the thread *t_A* as the next thread to context switch, the thread *t_A* will re-obtain the control from *t_C*. Finally, the function returns back to the caller. It’s possible that an invocation of context switch in *t_A* won’t return if *t_A* is never chosen.

```c
struct tcb {
    struct context ctxt;
    struct tcb *prev;
    struct tcb *next;
    struct tcb *rq;
    struct tcb *cur;
}

void schedule_p2()
{
    struct tcb *old, *new;
    old = cur;
    new = deq(&rq);
    if (new == NULL) return;
    enq(&rq, old);
    cur = new;
    cswitch(old,new);
    return;
}
```

We define the following notations to specify the structure of TCB:

- RQseg(*t*, *t*) = (RQseg(*t*, *t*) * *t*)
- RQseg(*t*, *t*) = *t* = NULL
- RQ(*rq*) = RQseg(NULL, *t*)

The specification of *schedule_p2()* is shown below:

```
RQseg(t', t) \triangleq (t' \xrightarrow{\text{prev}} t) * \exists t''.(t \xrightarrow{\text{next}} t'') * RQseg(t, t'')
RQseg(t', t) \triangleq \#(t = \text{NULL})
RQ(qr) \triangleq \exists t. (qr \xrightarrow{t}) * RQseg(NULL, t)
```
\begin{align*}
\{ & [t] \cdot \text{ptcb}(t) \cdot (\text{cur} \mapsto t) \cdot \text{RQ}(\text{rq}) \cdot (\text{ra} \mapsto \text{ret}) \cdot \text{K}(bp, 20) \cdot (v_0, a_0, a_1 \mapsto \_). \}\quad \text{\scriptsize \((t, \text{ret}, bp)\)} \\
\{ & [t] \cdot \text{ptcb}(t) \cdot (\text{cur} \mapsto t) \cdot \text{RQ}(\text{rq}) \cdot (\text{ra} \mapsto \text{ret}) \cdot \text{K}(bp, 20) \cdot (v_0, a_0, a_1 \mapsto \_). \}\}
\end{align*}

The precondition states the following resources:

- \([t]\): the current thread resource with thread ID \(t\);
- \(\text{ptcb}(t)\): the two pointers in TCB of current thread \(t\);
- \((\text{cur} \mapsto t)\): the global variable \(\text{cur}\) pointing to current thread;
- \(\text{RQ}(\text{rq})\): a ready thread which includes ready thread resources;
- \((\text{ra} \mapsto \text{ret})\): the register \(\text{ra}\) containing the return address;
- \(\text{K}(bp, 20)\): the stack register \(sp\) and a block of memory as an unused stack, with base address of \(bp\) and the size of 20;
- \((v_0, a_0, a_1, a_2 \mapsto \_): all of other registers used for the function.

The postcondition of the schedule function is same with the precondition. Here we use a notation \(K(bp, n, w :: w' :: \cdots)\) to describe a stack frame. The first parameter \(bp\) is the base address of a stack frame. The second parameter \(n\) is the size of unused space (number of words). Please remember that all stacks grow downwards in our machine model. And the third parameter is a list of words, representing the values on stack top down, that is, the leftmost value in the list is the topmost value in the stack frame. If the stack frame is empty, we omit the third parameter. The definition of \(K\) is given below:

\[
K(bp, n, w_0 :: w_1 :: \cdots :: w_m) \triangleq \exists sp. (\text{sp} \mapsto \text{sp}) \star \{sp = bp + 4n\} \\
\star (bp \mapsto (n) \star (\text{sp} \mapsto w_0) \star (sp + 4 \mapsto w_1) \star \cdots \star (sp + 4m \mapsto w_m) \\
K(bp, n, \cdot)
\]

Note that the stack register \(sp\) is included in the \(K(\cdot)\).

The two auxiliary functions \((\text{enq}()\) and \(\text{deq}()\) are used in the schedule function for manipulating thread queues. Their specifications are defined below:

\[
\{ \begin{align*}
\text{RQ}(q) \cdot \text{ptcb}(t) \cdot (a_0, a_1, a_2, v_0, \text{ra} \mapsto q, t, w, \_, \text{ret}) \cdot \text{K}(bp, 10) \\
(\langle t \rangle \mapsto \text{RQ}(q)) \cdot (a_0, a_1, a_2, v_0 \mapsto q, t, w, \text{ret}) \cdot (v_0 \mapsto \_0) \cdot \text{K}(bp, 10)
\end{align*} \} \quad \text{\scriptsize \((q, t, w, \text{ret})\)}
\]

\[
\{ \begin{align*}
\text{RQ}(q) \cdot (a_0, a_1, a_2, \text{ra} \mapsto q, w_1, w_2, \text{ret}) \cdot \text{K}(bp, 10) \cdot (v_0 \mapsto \_) \\
\text{RQ}(q) \cdot (a_0, a_1, a_2, \text{ra} \mapsto q, w_1, w_2, \text{ret}) \cdot \text{K}(bp, 10) \\
\quad \star (\exists t. (v_0 \mapsto t) \mapsto \langle t \rangle \mapsto \text{ptcb}(t)) \star (v_0 \mapsto \_0)
\end{align*} \} \quad \text{\scriptsize \((q, w_1, w_2, \text{ret})\)}
\]

The abstract invariant \(I\) is instantiated to a concrete definition specifying the shared resources before and after context switch for this implementation of scheduler.

\[
I(t, t') \triangleq \text{ptcb}(t') \cdot (\text{cur} \mapsto t') \cdot (\langle t \rangle \mapsto \text{RQ}(\text{rq}))
\]

The invariant \(I(t, t')\) is to specify shared resources excluding resources of \(t\) and \(t'\), but including the following resources:

- \(\text{ptcb}(t')\): the two pointers \((\text{prev})\) and \((\text{next})\) of TCB of the new thread \(t'\);
- \(\text{cur} \mapsto t'\): the global variable \(\text{cur}\), pointing to \(t'\);
- \(\langle t \rangle \mapsto \text{RQ}(\text{rq})\): the thread queue, being excluded the resource of \(\langle t \rangle\), because the current thread is still not a ready thread before context switch.

21
schedule_p2:

\[
\{[t] \ast ptcb(t) \ast (cur \mapsto t) \ast RQ(rq) \ast (a0, a1, v0, ra \mapsto \ldots \ast ret) \ast K(bp, 20)\}
\]

- subi sp, 12
- sw ra, 8(sp)
- movi a0, cur
- lw v0, 0(a0)
- sw v0, 0(sp)

\[
\{[t] \ast ptcb(t) \ast (cur \mapsto t) \ast RQ(rq) \ast K(bp, 17, t \mapsto \ldots \ast ret)
\ast (a0, a1, v0, ra \mapsto cur, t, t)\}
\]

- movi a0, rq
- call deq
- bz v0, Ls_ret

\[
\{[t] \ast ptcb(t) \ast (t') \ast ptcb(t') \ast RQ(rq) \ast K(bp, 17, t' \mapsto \ldots \ast ret)
\ast (cur \mapsto t)
\ast (a0, a1, v0, ra \mapsto rq, t, t')\}
\]

- sw v0, 4(sp)
- lw a1, 0(sp)
- call enq

\[
\{[t] \ast (t') \ast ptcb(t') \ast ((t' \mapsto RQ(rq)) \ast K(bp, 17, t' \mapsto \ldots \ast ret)
\ast (cur \mapsto t)
\ast (a0, a1, v0, ra \mapsto t', t, t, 0)\}
\]

- lw a1, 4(sp)
- movi a0, cur
- sw a1, 0(a0)
- lw a0, 0(sp)

\[
\{[t] \ast (t') \ast (t' \mapsto RQ(rq)) \ast ptcb(t') \ast K(bp, 17, t' \mapsto \ldots \ast ret)
\ast (cur \mapsto t')
\ast (a0, a1, v0, ra \mapsto t', t', 0, 0)\}
\]

- cswitch

\[
\{[t] \ast ptcb(t) \ast \exists t''. (t'') \ast ((t'' \mapsto RQ(rq)) \ast K(bp, 17, t'' \mapsto \ldots \ast ret)
\ast (cur \mapsto t)
\ast (a0, a1, v0, ra \mapsto t', t', 0, 0)\}
\]

Ls_ret:

- lw ra, 8(sp)
- addi sp, 12

\[
\{[t] \ast ptcb(t) \ast (cur \mapsto t) \ast RQ(rq) \ast (a0, a1, v0, ra \mapsto \ldots \ast ret) \ast K(bp, 20)\}
\]

- ret

**Fig. 13.** Verification of schedule_p2()
5.2 Scheduler as a separated thread

A scheduler in the pattern (III) does scheduling jobs in a particular thread. This thread is sometimes called processor-layer thread [19], which means that there will be one processor-layer thread for each processor in a multi-core architecture. The advantage of adding this scheduler thread is that this special thread can easily clean up dead threads who cannot deallocate its own stack since it can not call a function on a stack that is has been released [19], or implement multiple schedulers in one system [5]. A global variable sched is added to represent the TCB of the separate thread. In this pattern of scheduler, separate thread do the scheduling job in an infinite loop. A stub function is needed schedule_p3() for being invoked by other threads.

```c
struct tcb sched; | schedth()
struct tcb *cur, *rq; |
{ |
    while(1){
        schedule_p3() |
            cur = deq(&rq);
        |
            cswitch(&sched, cur);
        |
            cswitch(cur,&sched); |
            enq(&rq, cur);
        |
            return;
|
} |
}
```

We define the specification of schedule_p3() function below:

\[
\begin{align*}
[t] & \ast \text{ptcb}(t) \ast (\text{cur} \mapsto t) \ast (\text{sched} \mapsto (a_0, a_1, \text{ra} \mapsto \ldots, \text{ret}) \ast K(bp, 10))^{(t, bp, \text{ret})} \\
[t] & \ast \text{ptcb}(t) \ast (\text{cur} \mapsto t) \ast (\text{sched} \mapsto (a_0, a_1, \text{ra} \mapsto \ldots, \text{ret}) \ast K(bp, 10)}
\end{align*}
\]

Both the precondition and the postcondition of the schedule function schedule_p3() state the following resources:

- \([t]\): the current thread with thread ID \(t\);
- \(\text{ptcb}(t)\): the two pointers of TCB of current thread \(t\);
- \((\text{cur} \mapsto t)\): the global variable \(\text{cur}\) is pointing to current thread;
- \((\text{ra} \mapsto \text{ret})\): the register containing the return address;
- \(K(bp, 10)\): stack register and a block of memory as an empty stack, with base address of \(bp\) and size of 10;
- \((a_0, a_1 \mapsto \ldots)\): two registers;
- \((\text{sched})\): the thread resources of the separate thread for scheduling.

Differing from the specification of schedule_p2(), the schedule function in this implementation doesn’t own the thread queue. However, since the operations over the thread queue are put into a separated thread, the ownership of the thread queue is owned by the scheduler thread \(\text{sched}\) only. The specification of schedth() function is shown below:

\[
\begin{align*}
[\text{sched}] & \ast (\text{cur} \mapsto \ldots) \ast RQ(rq) \ast (a_0, a_1, a_2, v_0, \text{ra} \mapsto \ldots) \ast K(bp, 10) \quad (bp) \\
\text{false}
\end{align*}
\]

The precondition of the function schedth running in the separate thread states the following resources:
schedth:

\[
\{ \text{sched} \ast (\text{cur} \mapsto \_ \ast RQ(rq) \ast K(bp, 10) \ast (a0, a1, v0, ra \mapsto \_,
\_)) \}
\]

movi a0, rq

call deq

bz v0, schedth

\[
\{ \text{sched} \ast (\text{cur} \mapsto \_ \ast \langle t' \rangle \ast ptcb(t') \ast RQ(rq) \ast K(bp, 10) \ast (a0, a1, v0, ra \mapsto \_, t', \_), \_, \_)) \}
\]

movi a2, cur

sw v0, 0(a2)

mov a1, v0

lw a0, sched

\[
\{ \text{sched} \ast \exists t''. \langle t'' \rangle \ast ptcb(t'') \ast (\text{cur} \mapsto t'') \ast K(bp, 10) \ast RQ(rq) \ast (a0, a1, v0, ra \mapsto \_, t', \_, \_, \_)) \}
\]

cswitch

\[
\{ \text{sched} \ast (\text{cur} \mapsto \_) \ast RQ(rq) \ast K(bp, 10) \ast (v0 \mapsto 0) \ast (a0 \mapsto rq) \ast (a1 \mapsto t') \ast (ra \mapsto \_)) \}
\]

jmp schedth

schedule_p3:

\[
\{ \langle t \rangle \ast ptcb(t) \ast (\text{sched} \ast (\text{cur} \mapsto t) \ast K(bp, 10) \ast (a0, a1, ra \mapsto \_, \_, \_, \_)) \}
\]

subi sp, 4

sw ra, 0(sp)

movi a1, cur

lw a0, 0(a1)

movi a1, sched

\[
\{ \langle t \rangle \ast ptcb(t) \ast (\text{sched} \ast (\text{cur} \mapsto t) \ast K(bp, 9, ret) \ast (a0, a1, ra \mapsto t, sched, ret)) \}
\]

cswitch

\[
\{ \langle t \rangle \ast ptcb(t) \ast (\text{sched} \ast (\text{cur} \mapsto t) \ast K(bp, 9, ret) \ast (a0, a1, ra \mapsto \_, \_, \_, \_)) \}
\]

lw ra, 0(sp)

addi sp, 4

\[
\{ \langle t \rangle \ast ptcb(t) \ast (\text{sched} \ast (\text{cur} \mapsto t) \ast K(bp, 10) \ast (a0, a1, ra \mapsto \_, \_, \_, \_)) \}
\]

ret

Fig. 14. Verification of schedule_p3()
– sched: the current thread with thread ID t;
– (cur \rightarrow \_): the global variable cur pointing to current thread;
– (ra \rightarrow ret): the register containing the return address;
– K(bp, 10): the stack register sp and a block of memory as empty stack frame, with base address of bp and the size of 20;
– (v0, a0, a1, a2, ra \rightarrow \_): other registers;

In this version, since the ready thread queue is only operated by the scheduler thread, it needn’t to be shared by other threads. We define the invariant I as shown below:

\[ I(t, t') \triangleq (\sharp(t' = \text{sched}) \ast (\text{cur} \rightarrow t) \ast \text{ptcb}(t)) \lor (\sharp(t = \text{sched}) \ast (\text{cur} \rightarrow t') \ast \text{ptcb}(t')) \]

The definition of \( I(t, t') \) is defined as cases analysis on the direction of context switch: if the destination thread is the scheduler thread, \( I(t, t') \) requires that the value in cur be equal to the ID of the source thread, \( t \); or if the source thread is the scheduler thread, \( I(t, t') \) requires that the value in cur be equal to the ID of the destination thread.

### 6 Extension 1: interrupt and preemptive scheduling

In this section, we extend our proof system with interrupt support so as to verify the implementation of preemptive scheduling, which was able to be verified by previous methods [3,7,20]. Comparing with them, however, our verification is simpler and easier to follow.

We first give a piece of code which implements the scheduling pattern (I) as presented in Figure 1.

```c
struct tcb {
    struct context ctxt;
    int pc;
    struct tcb *prev;
    struct tcb *next;
};

struct tcb *cur;  
void intr_handler() {
    struct tcb *new;
    savectxt(cur);
    new = deq(rq);
    if (new != NULL) {
        cli();
        enq(rq, cur);
        t = cur;
        cur = new;
        sti();
    }
    return;
    loadctxt(cur);
}
```

In code above, there are two global variables, cur and rq, pointing to the TCB of current thread and the ready thread queue. If an interrupt signal is triggered by the timer, the processor will jump to run the interrupt service routine (ISR). Here the function intr_handler(), which does the job of scheduling, is setup to serve as an ISR. The differences between this function with schedulers verified in the last section are that:
Fig. 15. Extension 1: abstract machine

- the context of the current thread is saved right after interrupt occurs;
- the context of the next thread is not loaded until the processor is going to return from the ISR;
- the operations of the thread queue are manipulated inside the ISR.

Abstract machine. To support interrupt, we extend the machine model in Figure 15. The state of machine is added with a flag register \( F \) which is a singleton mapping from the interrupt flag if to a word, which could be (0 or 1). The flag register indicates whether the interrupt is disabled (0 for disabled interrupt). We add a new type of abstract thread to the thread pool, intr, which specifies the processor is running in the ISR. We give a special thread ID isr to intr. Some new instructions are introduced for interrupt support. The abstract instruction iret is used for returning from interrupt handler. The instructions sti and cli are for turning-on or turning-off interrupt by modifying the flag register.

Operational semantics. The extended operational semantics are shown in Figure 16. Interrupts may occur when the flag register is equal to 1, otherwise the interrupt is disabled. If an interrupt occurs, the program counter will be saved in TCB pointed by a2, since processor will re-execute the instruction pointed by pc after return from ISR. At the same time, the flag register is modified to zero to prevent the reentrance of interrupts. The tag of the current thread is changed from run to rdy with saved register file and the program counter. Moreover, an abstract thread isr is added to the thread pool to indicate that the processor is running in the ISR.

If the ISR returns by the instruction iret, a new abstract thread \( t \) will be pointed by a2, the register file and the program counter will be loaded from TCB, the flag register will be turned on, and the tag of the new thread will be changed from rdy to run. And the abstract thread isr will be dismissed.

The binary relation \( W \hookrightarrow W' \) specifies the one-step operational semantics of the machine. The relation \( (S, pc) \rightarrow^\text{inf} (S', pc') \) specifies the state transition when an interrupt occurs; while the relation \( (S, pc) \rightarrow^\text{iret} (S', pc') \) specifies the state transition when the processor returns from the ISR. Note that these operational semantics are a little different from ones of realistic hardware. In fact, the abstract operations include both operations done by hardware and operations done by some built-in code. Readers may see the part of machine translation in this section for more information.
\[
\begin{align*}
C(pc) &= c \quad (S, pc) \xrightarrow{c} (S', pc') \\
(C, S, pc) &\longrightarrow (C, S', pc') \\
S.F(if) &= 1 \quad (S, pc) \xrightarrow{intr} (S', pc') \\
(C, S, pc) &\longrightarrow (C, S', pc') \\
\end{align*}
\]

\[
\begin{array}{l}
(M, R, F, P), pc \xrightarrow{intr} ((M', R', F', P'), pc') \\
M = M' \land R = R' \land t = R(a2) \land pc' = 1^\text{intr} \land F(if) = 1 \land F'(if) = 0 \\
\land P = \{t : \text{run}\} \uplus P'' \\
\land P' = \{t : (\text{rdy}, R, pc)\} \uplus \{\text{isr : intr}\} \uplus P'' \\
((M, R, F, P), pc) \xrightarrow{i} ((M', R', F', P'), pc') \\
i = ((M, R), pc) \xrightarrow{i} ((M', R'), pc') \land P = P' \land F = F' \\
\text{iret} M = M' \land t = R(a2) \land pc' = pc, F(if) = 0 \land F'(if) = 1 \\
\land P = \{t : (\text{rdy}, R, pc)\} \uplus \{\text{isr : intr}\} \uplus P'' \\
\land P' = \{t : \text{run}\} \uplus P'' \\
\text{sti} M = M' \land R = R' \land F(if) = 0 \land F'(if) = 1 \land P = P' \\
cl_i M = M' \land R = R' \land F(if) = 0 \land F'(if) = 0 \land P = P'
\end{array}
\]

Fig. 16. Extension 1: operational semantics

**Assertion language.** Some new assertion constructs are needed. One of these is for specifying the abstract thread of ISR, \([\cdot]\). Its definition is straightforward and shown below. The notation if \(\mapsto b\) specifies the flag register, and if \(\mapsto b\) specifies the flag register and other uninteresting resources. We change the definition of \(\triangleright p\) here, which states that a shared assertion \(p\) doesn’t specify any resource of general register or flag register.

\[
\begin{align*}
[\cdot] & \triangleq \lambda(M, R, F, P). P = \{\text{isr : intr}\} \land M = F = R = \{\cdot\} \\
\text{if} \mapsto b & \triangleq \lambda(M, R, F, P). F = \{\text{if} : b\} \land M = R = P = \{\cdot\} \\
\text{if} \mapsto b & \triangleq \lambda(M, R, F, P). F = \{\text{if} : b\} \\
\triangleright p & \triangleq \lambda(M, R, F, P). p \ (M, R, F, P) \land R = F = \{\cdot\}
\end{align*}
\]

### 6.1 Inference rules

We extend our proof system with the following four rules for new instruction in Figure[7] The invariant of context switch \(I(t)\) is different from the invariant \(I(t, t')\) in Section[4] It has only one parameter \(t\) indicating which thread the current ISR is going to switch to.

The rule of \((\text{INTR})\) states that if interrupt is enabled before processor runs the instruction \(i\), the thread should contain the following resources:

- \([\cdot]\): the resource of the current thread,
- \((a2 \mapsto t)\): the value of register \(a0\) is equal to \(t\),
- \((\text{if} \mapsto 1)\): the flag register with interrupt enabled.
\[
(p, g) \Rightarrow \left\{ \begin{array}{l}
[t] \ast (if \mapsto 0) \ast oJ(t) \\
[t] \ast (if \mapsto 1)
\end{array} \right\} \vdash \Psi(pc + 1)
\]
\[\Psi, J \vdash \{(p, g)\} \text{ pc: sti} \quad \text{(STI)}\]
\[
(p, g) \Rightarrow \left\{ \begin{array}{l}
[t] \ast (if \mapsto 1) \\
[t] \ast (if \mapsto 0) \ast oJ(t)
\end{array} \right\} \vdash \Psi(pc + 1)
\]
\[\Psi, J \vdash \{(p, g)\} \text{ pc: cli} \quad \text{(CLI)}\]
\[
(p, g) \Rightarrow \left\{ \begin{array}{l}
[[\cdot]] \ast (a2 \mapsto t) \ast \langle t \rangle \ast (if \mapsto 0) \ast oJ(t)
\end{array} \right\} \vdash \Psi(pc + 1)
\]
\[\Psi, J \vdash \{(p, g)\} \text{ pc: iret} \quad \text{(IRET)}\]
\[
(p \land (if \mapsto 1), g) \Rightarrow \left\{ \begin{array}{l}
[t] \ast (a2 \mapsto t) \ast (if \mapsto 1) \\
[t] \ast (a2 \mapsto t) \ast (if \mapsto 1)
\end{array} \right\} \vdash (p, g)
\]
\[\Psi, J \vdash \{(p, g)\} \text{ pc: c} \quad \text{(INTR)}\]
\[
\Psi(1^{\text{intr}}) = (p, g)_{\text{intr}} \quad \forall f \in \text{dom}(C) \quad \Psi, J \vdash \{(\Psi(f))\} f : C(f) \quad \Psi, J \vdash \{(\Psi(f))\} \text{ pc: i} \quad \Psi, J \vdash C \quad \text{(CDHP)}
\]
\[\Psi(1^{\text{intr}}) = (p, g)_{\text{intr}} \quad \forall f \in \text{dom}(C) \quad \Psi, J \vdash \{(\Psi(f))\} f : C(f) \quad \Psi, J \vdash \{(\Psi(f))\} \text{ pc: i} \quad \Psi, J \vdash C \quad \text{(CDHP)}
\]
\[
\begin{align*}
\Psi(1^{\text{intr}}) &= (p, g)_{\text{intr}} \\
&= \exists f \in \text{dom}(C) \quad \Psi, J \vdash \{(\Psi(f))\} f : C(f) \quad \Psi, J \vdash \{(\Psi(f))\} \text{ pc: i} \quad \Psi, J \vdash C
\end{align*}
\]
\[
\text{where } (p, g)_{\text{intr}} \triangleq \left\{ \begin{array}{l}
[[\cdot]] \ast K(\text{istack, ihsize}) \ast \langle t \rangle \ast (a2 \mapsto t) \ast oJ(t)
\end{array} \right\} \vdash (p, g)
\]

Fig. 17. Extended inference rules of Ext. I
After interrupt, the current instruction should be verified again under the current specification.

The rule of (IRET) states if the ISR want to load the context of a new thread, it should have the following resources:
- $[[\cdot]]$: the current ISR resource,
- $\langle t \rangle$: the resource of a new ready thread,
- $(a2 \mapsto t)$: the value of register $a2$ pointing to $t$,
- $(if \mapsto 0)$: the flag register with interrupt enabled,
- $\diamond J(t)$: shared resources.

The rules of (STI) and (CLI) cause ownership transfer of the shared resources specified by $\diamond J(t)$, where $t$ is exactly the ID of current running thread.

The rule of (CDHP) is changed as well. Each instruction in $C$ should be checked by the rule (INTR). Moreover, the specification of the ISR, $(p,g)_{isr}$ is hardwired at the label $1^{\text{intr}}$. The specification of ISR specifies the precondition, which should have the following resources:
- $[[\cdot]]$: the current ISR resource,
- $\langle t \rangle$: the resource of the interrupted thread,
- $(a2 \mapsto t)$: the value of register $a2$ pointing to $t$,
- $(if \mapsto 0)$: the flag register with interrupt disabled,
- $K($istack,ihksize$)$: the special stack for ISR,
- $\diamond J(t)$: shared resources.

### 6.2 Invariant of global resources

**Running thread.** The concrete resources of a running thread are specified by a continuation $\text{Cont}$ with an additional condition, the running thread owns all registers. The parameter $pc$ points to the next instruction the thread is going to run.

$$\text{CThrd}\left(\Psi, t, pc\right) \triangleq \exists n. \text{Cont}(n, \Psi, pc) \land (\langle r \rangle \ast \exists R. \lfloor R \rfloor \ast \text{true})$$

**Ready thread.** Because the interrupt must be enabled before the processor could be interrupted, any ready thread won’t have the shared resources when running. Accordingly, we modify the definition of $\text{RThrd}$ as below:

$$\text{RThrd}\left(\Psi, t, R_t, pc_t\right) \triangleq \lfloor R_t \rfloor \ast (if \mapsto 1) \ast [r] \ast \rightarrow \text{CThrd}\left(\Psi, t, pc_t\right)$$

The definition of $\text{CThrd}$ is same to the definition in Section 4.5.

**Interrupt service routine.** When a thread is interrupted, the ISR will use a reserved block of stack space (with size of $\text{ihksize}$). Thus we define the resources of ISR as below:

$$\text{IThrd}(\Psi, pc) \triangleq \exists n. \text{Cont}(n, \Psi, pc) \land ([[\cdot]] \ast \exists R. \lfloor R \rfloor \ast \text{kspace}(\text{istack}, \text{ihksize}) \ast \text{true})$$

The global invariant $\text{GINV}$ is defined according to the resource layout. Its form of the definition depends the condition whether the processor is running in the ISR, or a thread. Note that the shared resources will be get out of the running thread if the interrupt is enabled, otherwise it will bring data racing for a simple reason that the shared resources are accessed from both threads and the ISR. $\text{GINV}$:
\[
\text{GINV}(\Psi, P, pc) \triangleq\text{CThrd}(\Psi, t, pc) \ast \text{kspace}(\text{istack}, \text{ihksize}) \ast \text{RThrd}(\Psi, t_0, R_0, pc_0) \\
\ast \cdots \ast \text{RThrd}(\Psi, t_n, R_n, pc_n) \\
\ast ((\text{if } \mapsto 1) \ast (\text{if } \mapsto 1) \ast I(t)) \ast ((\text{if } \mapsto 0) \ast (\text{if } \mapsto 0)) \\
\text{if } P = \{t : \text{run}, t_0 : (\text{rdy}, R_0, pc_0), \ldots, t_n : (\text{rdy}, R_n, pc_n)\}
\]

\[
\text{GINV}(\Psi, P, pc) \triangleq \text{IThrd}(\Psi, pc) \ast \text{RThrd}(\Psi, t_0, R_0, pc_0) \ast \cdots \ast \text{RThrd}(\Psi, t_n, R_n, pc_n) \\
\text{if } P = \{\text{isr} : \text{intr}, t_0 : (\text{rdy}, R_0, pc_0), \ldots, t_n : (\text{rdy}, R_n, pc_n)\}
\]

**Soundness.** The soundness theorem of proof system can be proved by preservation of the global invariant:

**Theorem 5 (Soundness of Ext.I).** For any machine configuration \((C, S, pc)\), if its code is verified \(\Psi, J \vdash C\) and its state satisfies the global invariant \(S \vdash \text{GINV}(\Psi, P, pc)\), then it is safe to run, \(\text{Safe}((C, S, pc))\).

**Machine translation.** To make the verification result based on the physical machine, the new abstract commands should be able to be translated to concrete code. In many realistic machine, there often are special instruction to turn on/off interrupt, so it’s easy to translate \text{sti} and \text{cli} to those intructions. But for \text{intr} and \text{iret}, since the two abstract instructions are responsible for saving and loading context data, each of them should be translated to a sequence of instructions, from the labels \text{entry_ISR} and \text{loadctx}:

\[
\begin{array}{ll}
\text{entry_ISR:} & | \text{loadctx:} \\
\text{sw ra, 0(a0) } | & \text{lv sp, 20(a0)} \\
\text{sw v0, 4(a0) } | & \text{lv a2, 16(a0)} \\
\text{sw a0, 8(a0) } | & \text{lv a1, 12(a0)} \\
\text{sw a1, 12(a0) } | & \text{lv a0, 8(a0)} \\
\text{sw a2, 16(a0) } | & \text{lv v0, 4(a0)} \\
\text{sw sp, 20(a0) } | & \text{lv a0, 8(a0)} \\
\text{sw k0, 24(a0) } | & \text{lv k0, 24(a0)} \\
\text{movi sp, istack } | & \text{iret}
\end{array}
\]

The operations done by hardware when an interrupt occurs are: the program counter is saved to the stack of interrupted thread, the processor jumps to \text{entry_ISR}. And then a sequence of built-in code saves the registers into TCB, and change the stack pointer to a special stack. When returns from the ISR, the processor will load the registers of a new thread, and then run the \text{iret} instruction to jump to the code whose label was saved on the stack before.

### 6.3 Verification

We modify the definition of the two invariants of shared resources and give them below:

\[
J(t) \triangleq \text{ptcb}(t) \ast (\text{cur} \mapsto t) \ast \text{RQ}(rq)
\]

Then we can verify a small interrupt handler by the inference rules presented. The assembly code and selected assertions are shown in Figure 18.
intr_handler:

{[[1]]} (if \( \rightarrow 0 \)) \* ptcb(\( t \)) \* (cur \( \rightarrow t \)) \* RQ(rq) \* K(istack, ihksize)
+ (a0 \( \rightarrow t \)) \* (v0, a1, a2, ra \( \rightarrow _{-} \))}

movi a1, a0
movi a0, rq

{[[1]]} (if \( \rightarrow 0 \)) \* ptcb(\( t \)) \* (cur \( \rightarrow t \)) \* RQ(rq) \* K(istack, ihksize)
+ (a0 \( \rightarrow rq \)) \* (a1 \( \rightarrow t \)) \* (v0, a2, ra \( \rightarrow _{-} \))}
call deq
bz v0, ihret

{[[1]]} (if \( \rightarrow 0 \)) \* ptcb(\( t \)) \* (cur \( \rightarrow t \)) \* RQ(rq) \* K(istack, ihksize)
+ (v0 \( \rightarrow t' \)) \* (\( t' \)) \* ptcb(\( t' \)) \* (a0 \( \rightarrow rq \)) \* (a1 \( \rightarrow t \)) \* (a2, ra \( \rightarrow _{-} \))}
call enq

{[[1]]} (if \( \rightarrow 0 \)) \* (cur \( \rightarrow t' \)) \* ((\( t' \)) \( \rightarrow RQ(rq) \)) \* K(istack, ihksize)
+ (\( t' \)) \* ptcb(\( t' \)) \* (a0 \( \rightarrow rq \)) \* (a1 \( \rightarrow t \)) \* (v0, a2, ra \( \rightarrow _{-} \))}
movi a2, cur
sw v0, 0(a2)

{[[1]]} (if \( \rightarrow 0 \)) \* (cur \( \rightarrow t' \)) \* ((\( t' \)) \( \rightarrow RQ(rq) \)) \* K(istack, ihksize)
+ (\( t' \)) \* ptcb(\( t' \)) \* (a0 \( \rightarrow rq \)) \* (a1 \( \rightarrow t \)) \* (v0, a2, ra \( \rightarrow _{-} \))}
hret:

{[[1]]} (if \( \rightarrow 0 \)) \* K(istack, ihksize) \* (v0, a1, a2, ra \( \rightarrow _{-} \))
+ \( \exists t' \). ((\( a0 \rightarrow t' \)) \* ptcb(\( t' \)) \* (cur \( \rightarrow t' \)) \* RQ(rq))}

iret

Fig. 18. Verification of intr_handler()
Discussion. While in some realistic system, context data and the program counter of processor are usually pushed into stack. This pattern is different from ours. But it is easy to modify the machine to support it. In the kernel of Minix [9], its scheduling pattern has another slight difference from the pattern verified in this section. Minix uses a fixed kernel stack. When a user process is trapped into the kernel, the kernel will switch to the kernel stack thereafter and do the kernel jobs. Our framework is also easy to be adapted to this pattern by modifying the definition of \texttt{IThrd}: which will be the composition of \texttt{CThrd} and \texttt{RThrd} without stack borrowing.
7 Extension 2: dynamic thread (de)allocation

In this section, we extend our verification framework with thread dynamic creation and deallocation support to verify a mini thread manager with the following thread operations: thread create, thread exit. See our technical report for other operations: thread yield, thread kill and thread join.

The implementation of mini thread manager is given in Figure 19. To support dynamic thread allocation, the stack location should be stored in the TCB. To support thread joining, a field \texttt{joiner} is added so as to identify another thread which is waiting for this thread. Thus, this thread can be notified if thread is terminated. Please note that any thread can be waited by only one thread. In this mini thread manager, we adopt the scheduling pattern (III), and add a separate thread \texttt{sched} for doing scheduling. Besides the global pointers \texttt{cur} and \texttt{rq}, we add one more thread queue for collecting dead threads, pointed by \texttt{dq}.

We need three auxiliary functions for adding or removing nodes from thread queues and two functions about allocating/deallocating fixed-sized thread stacks. The function \texttt{create()} is for thread creation. It initializes a TCB and puts it into the ready thread queue. Sometimes, the scheduler may choose this new thread to run. The entry point of thread is put into the context data in TCB.

The function \texttt{schedth()} does scheduling job in a separate thread. There is a slight difference between this version with the version in Section ?? that the scheduler only gets a new thread out of the ready thread queue, and doesn’t put the old thread back into the ready thread.

When a thread has finished its task, it can call \texttt{exit()} to terminate itself. The \texttt{exit()} function will firstly check whether there is a thread waiting for it. If so, the function puts this thread into the ready thread queue and cleans the \texttt{joiner} field in the TCB of current thread. Then \texttt{exit()} function will put current thread into the dead thread queue, load the context of the scheduler thread and hand over the control of processor.

The function \texttt{join()} implements the synchronization of threads. When invoked, it firstly checks whether the waited thread has been dead. If so, the function will free the stack of the dead thread, get the TCB of waited thread and return. Otherwise, the function checks whether the waited thread is in the ready thread. If that, modify the \texttt{joiner} field (of the waited thread) to the ID of current running thread. Otherwise, it does context switch to the scheduler thread and loops when coming back.

The function \texttt{kill()} can terminate any thread in the system no matter what it is. When invoked, it checks whether the victim has been dead already. If not, the function gets the victim thread out of the ready thread queue, and puts it into the dead thread queue. And the function checks whether some thread is waiting for the victim thread, if so, the killer thread put the waiter thread into the ready thread queue.

7.1 Abstract machine

To support thread dynamic allocation and deallocation, we separate the stack space \( K \) from the shared heap \( H \) in the machine state \( S \). The space of stacks is for all threads, but cannot be shared between threads. The stack space \( K \) is a mapping from labels to words like heap \( H \). To specify a dead thread, we add a new type of abstract thread, \((\text{dead}, k)\). In
struct tcb {
    struct context ctxt;
    int *stack;
    struct tcb *prev, *next;
    struct tcb *joiner;
};

struct tcb sched;
struct tcb *cur, *rq, *dq;

void create(struct tcb *new, void (*f)())
{
    int *ss; ss = kalloc();
    new->stack = ss;
    new->ctxt.ra = (int)f;
    new->ctxt.sp = (int)(ss + KMAXSIZE);
    new->joiner = NULL;
    enq(rq,new);
    return;
}

void exit()
{
    struct tcb *old;
    if(cur->joiner!=NULL) {
        enq(rq, cur->joiner);
        cur->joiner = NULL;
    }
    enq(dq, cur);
    loadctx(&sched);
}

void yield()
{
    enq(rq,cur);
    cswitch(cur, &sched);
    return;
}

int *kalloc();
void kfree(int *p);
bool inq(struct tcb *q, *p);
void rmq(struct tcb *q, *p);
struct tcb *deq(struct tcb *q);
void enq(struct tcb *q, *p);

int kill(struct tcb *t)
{
    if (inq(dq, t) == 0)
        return 0;
    if (inq(rq, t) == -1)
        return -1;
    rmq(rq, t);
    if (t->joiner != NULL)
        enq(rq,t->joiner);
    t->joiner = NULL;
    enq(dq,t);
    return 0;
}

int join(struct tcb *t)
{
    while(rmq(dq, t) == -1)
    {
        if (inq(rq, t) == -1)
            return -1;
        if (t->joiner != NULL)
            return -1;
        rmq(rq, t);
        t->joiner = cur;
        cswitch(cur, &sched);
    }
    kfree(t->stack);
    return 0;
}

void schedth()
{
    while(1){
        cur = dequeue(rq);
        if (cur == NULL)
            continue;
        cswitch(&sched, cur);
    }
}

Fig. 19. Ext. 2: The C code of a mini thread manager

34
any abstract thread, there is an additional value named $k$ to record the start address of the stack. And we introduce three new abstract instructions: $\texttt{tnew}$ to make a new thread resource from heap; $\texttt{tdel}$ to recycle a dead thread resource back to heap; and $\texttt{quit}$ to terminate current running thread and load the context of another thread. The extended abstract machine is shown in Figure 20.

**Operational semantics.** The extended operational semantics are presented in Figure 21. The state transition of the command $\texttt{cswitch}$ is exchanging the tags of two thread entities: changing the running thread to ready and changing a ready thread to running. The two thread entities are referenced by the registers, $a_0$ and $a_1$. The command $\texttt{quit}$ also exchanges the tags of two thread entities: changing the running thread to dead and changing a ready thread to running. The ready thread should be referenced by the register $a_0$. The instruction $\texttt{tnew}$ turns a heap block $\texttt{mck}(1, R)$ into a ready thread entity, with ID stored in the register $r$. And the instruction $\texttt{tdel}$ turns a dead abstract thread back into the heap.

### 7.2 Assertion language

We add an assertion construct to represent the resource of a dead thread, $\langle \! \langle \cdot \rangle \! \rangle$. Since we separate heap from stack space, we use $1 \leadsto w$ to specify a cell in stack space and use $1 \mapsto w$ to specify a cell in heap. A shared assertion doesn’t contain any resource of register and stack. We change the meaning of $\diamond p$ to that: a shared assertion $p$ doesn’t specify any resource of registers or resource on stack.

$$\langle \! \langle \cdot \rangle \! \rangle \triangleq \lambda(H,K,R,P).P=\{t:\{\text{dead,}_.\}\} \land H=\{\cdot\} \land K=\{\cdot\} \land R=\{\cdot\}$$

$$1 \leadsto w \triangleq \lambda(H,K,R,P).K=\{1:w\} \land l \neq \text{NULL} \land H=\{\cdot\} \land R=\{\cdot\} \land P=\{\cdot\}$$

$$1 \mapsto w \triangleq \lambda(H,K,R,P).H=\{1:w\} \land l \neq \text{NULL} \land K=\{\cdot\} \land R=\{\cdot\} \land P=\{\cdot\}$$

$$\diamond p \triangleq \lambda(H,K,R,P).p(H,K,R,P) \land R=\{\cdot\} \land K=\{\cdot\}$$

The definition of the notation $\texttt{kspace}(bp,n)$ need to be changed for specifying unused stack space, and $K(bp,n,\cdots)$ specifying the shape of a stack frame:

$$\texttt{kspace}(bp,n) \triangleq (bp \leadsto_.) \ast (bp+4 \leadsto_.) \ast \cdots \ast (bp+4(n-1) \leadsto_.)$$

$$\texttt{hspace}(l,n) \triangleq (l \mapsto_.) \ast (l+4 \mapsto_.) \ast \cdots \ast (l+4(n-1) \mapsto_.)$$

$$K(bp,n,w_0::w_1::\cdots::w_m) \triangleq \exists sp.(bp \mapsto sp) \ast \not\exists (sp=bp+4n)$$

$$\ast \texttt{ksp} \texttt{ace}(bp,n) \ast (sp \leadsto w_0) \ast (sp+4 \leadsto w_1) \ast \cdots \ast (sp+4m \leadsto w_m)$$
\[
C(pc) = c \quad (S, pc) \xrightarrow{c} (S', pc') \\
(C, S, pc) \xrightarrow{c} (C', S', pc')
\]

\[
((H, K, R, P), pc) \xrightarrow{c} ((H', K', R', P'), pc')
\]

if \( c = 0 \) then

\[
P = P' \land ((H, R), pc) \xrightarrow{c} ((H', R'), pc')
\]

\[
P = P' \land ((K, R), pc) \xrightarrow{c} ((K', R'), pc')
\]

cswitch

\[
H = H' \land K = K' \land R = R'[ra: pc + 1] \land pc' = R'(ra)
\]

\[
\land t = R(a0) \land t' = R(a1)
\land P = \{ t : run, t' : (rdy, R') \} \cup P''
\land P' = \{ t : (rdy, R'), t' : run \} \cup P''
\land R and R' is complete.
\]

quit

\[
H = H' \land K = K' \land t = R(a0) \land pc' = R'(ra)
\]

\[
\land P = \{ t : run, t' : (rdy, R') \} \cup P''
\land P' = \{ t : dead, t' : (run) \} \cup P''
\]

tnew r

\[
H = H' \uplus mck(t_x, R_x, k_x) \uplus blk(k_x, k\text{maxsize})
\]

\[
\land P = P' \uplus \{ t_x : (rdy, R_x) \} \land R = R' \land t_x = R(r)
\land K' = K \uplus blk(k_x, k\text{maxsize}) \land pc' = pc + 1
\]

tdel r

\[
H' = H \uplus mck(t_x, R, k_x) \uplus blk(k_x, k\text{maxsize})
\]

\[
\land P = P' \uplus \{ t_x : \text{dead} \} \land R = R' \land t_x = R(r)
\land K = K' \uplus blk(k_x, k\text{maxsize}) \land pc' = pc + 1
\]

\[
mck(1, R, k) \triangleq mc(1, R) \uplus \{ 1 + 24 : k \}
\]

\[
blk(k, s) \triangleq \{ k : \_, k + 4 : \_, \ldots, k + 4(s - 1) : \_ \}
\]

**Fig. 21.** Operational semantics
We use the following notations to specify data in TCBs:

\[ t^{\text{stack}} \mapsto k \triangleq t + \text{OFF}_\text{STACK} \mapsto k \]
\[ t^{\text{next}} \mapsto p \triangleq t + \text{OFF}_\text{NEXT} \mapsto p \]
\[ t^{\text{prev}} \mapsto q \triangleq t + \text{OFF}_\text{PREV} \mapsto q \]
\[ t^{\text{joiner}} \mapsto j \triangleq t + \text{OFF}_\text{JOINER} \mapsto j \]

Then we define notations for specifying the exposed part of TCB. The notation \( rtcb(t) \) specifies three pointers, among which the pointer of joiner points to a waiting chain, specified by WQt. The notation \( dtcb(t) \) specifies three pointers, but the pointer of joiner should be NULL. A dead thread should be joined immediately, and it is unreasonable that the dead thread is waited for by other thread. Since a thread A can wait for a thread B, which can still wait for another thread C, the waiting chain will be a linked list organized by the field joiner in TCBs. Its definition includes ready thread resources and we don’t discriminate abstract waiting threads with ready threads in our framework. The notation \( tcb(t) \) is used to specify an empty TCB with all fields.

\[ rtcb(t) \triangleq (t^{\text{prev}} \mapsto \_ \mapsto t^{\text{next}} \mapsto \_ \mapsto \exists j. (t^{\text{joiner}} \mapsto j) \mapsto WQ(j) \]
\[ dtcb(t) \triangleq (t^{\text{prev}} \mapsto \_ \mapsto t^{\text{next}} \mapsto \_ \mapsto (t^{\text{joiner}} \mapsto j) \mapsto \text{NULL} \]
\[ WQ(t) \triangleq (t \mapsto rtcb(t)) \not\in \exists (t = \text{NULL}) \]
\[ tcb(t) \triangleq (t^{\text{ctxt}} \mapsto \_ \mapsto t^{\text{stack}} \mapsto \_ \mapsto dtcb(t) \]

Since a ready thread may be connected to a waiting chain, thus we re-define the ready thread queue: each thread in the ready thread queue is connected to a waiting chain, which could be empty to indicate no waiting threads for this thread.

\[ RQseg(t, t') \triangleq \langle t \rangle \ast (t^{\text{prev}} \mapsto \_ \mapsto t^{\text{next}} \mapsto \_ \mapsto \exists j. (t^{\text{joiner}} \mapsto j) \mapsto WQ(j) \]
\[ RQseg(t, t') \triangleq \#(t = \text{NULL}) \]
\[ RQ(q) \triangleq \exists t. (q \mapsto t) \ast RQseg(t, \text{NULL}) \]

We add two new invariants, \( X \) and \( J \). The first one \( X \) is used to specify the shared resources before quit. Like \( I \), \( X \) also takes two parameters, i.e., the ID of the thread quitting and the new thread. The second one \( J \) is used to specify the shared resources after context switch. The reason why we don’t \( I \) here is that the last thread could be a dead thread, or a ready thread. The two invariants also need to be precise and should satisfy the following requirement:

\[ \forall t, t'. X(t, t') \ast \langle t \rangle \Rightarrow J(t') \]
\[ \forall t, t'. I(t, t') \ast \langle t \rangle \Rightarrow J(t') \]

Since \( X \) specifies the invariant before one thread quit, if it is added a dead resource of the thread \( \langle t \rangle \), the separation conjunction will imply the invariant of \( J(t) \). The invariant \( X \) shouldn’t specify any stack resource since stack resources are always private. If not, the private stack space of one thread will be taken by other threads and thus this thread will be unable to be released and hard to tackle.

37
7.3 Inference rules

The extended inference rules of new instructions are shown in Figure 22. We add a notation $\Delta$ to specify the set of specification of new threads, each of which, we assume, is a static specification $(p, g)_{init}$, for a new-born thread.

$\Delta \triangleq \{ \xi : (p, g)_{init} \}^*$

The specification $(p, g)_{init}$ states the initial resources of new threads: the current thread resource, all registers and a block of stack space with a fixed size, $k_{maxsize}$.

$(p, g)_{init} \triangleq \{ [t] \ast [R] \ast \text{kspace}(R(\text{sp}), k_{maxsize}) \}^{(t, R)}$

The rule of (QUIT) states that if a thread wants to exit, it should have the following resources:
- $[t]$: the current thread resource;
- $\langle t' \rangle$: resource of a ready thread;
- $(a0 \mapsto t')$: the register $a0$ with value of $t'$;
- $\odot X(t, t')$: shared resources except the resources of two threads, $t$ and $t'$.

The rule of (TNEW) states that the union of the following resources can be turned into an abstract resource of a ready thread:
- $(t \xrightarrow{\text{stack}} k, R)$: resources of machine context and a cell of memory with value $k$;
- $\text{RThrd}(\Delta, t, R, k)$: resources satisfying the predicate of a concrete ready thread.

$\text{NThrd}(\Delta, t, R) \triangleq \exists k. t \xrightarrow{\text{stack}} k \ast \text{hspace}(k, k_{maxsize}) \ast (\text{kspace}(k, k_{maxsize}) \rightarrow \text{RThrd}(\Delta, t, R, k))$

The former is a part of TCB of a new thread, and the latter is actually what the new thread requires to run. Similarly, the rule of (TDEL) is for reasoning about recycling the concrete resources of a dead thread, specified by $\text{DThrd}(t, k)$.

7.4 Invariant of global resources

To ensure that the private stack of a thread is not shared to others, we add a stack space constraint in the definition of $\text{CThrd}$:

$\text{CThrd}(\Psi, t, pc) \triangleq \exists k. (t \xrightarrow{\text{stack}} k) \ast \exists n. \text{Cont}(n, \Psi, pc)$

$\& ([t] \ast \exists R. [R] \ast \text{kspace}(k, k_{maxsize}) \ast T)$

The definition of a concrete ready thread is same with one in Sec. 4.5 except it takes one more parameter, the stack location $k$.

$\text{RThrd}(\Psi, t, R) \triangleq [R] \ast [t] \ast \odot I(t) \rightarrow \text{CThrd}(\Psi, t, R(\text{ra}))$

We add a new definition of a concrete dead thread $\text{DThrd}$, which is just equal to the stack space of thread starting at $k$. 38
The invariant of global resources is defined as below:

\[ \text{GINV} \]

7.5 Soundness

For any machine configuration \((C,S,pc)\), if its code is verified \( \Psi,\Delta,I,1,X \vdash C \) and its state satisfies the global invariant \( S \vdash \text{GINV}(\Psi,P,pc) \), and if the state can be mapped to a physical state, \( S \not\vdash S \), then it is safe to run \( \text{Safe}(C,S,pc) \).

Machine translation. The translation from abstract machine defined in this section to physical machine defined in Section ?? is simple to define. The new commands \( \text{tnew} \) and \( \text{tdel} \) can be translated to a nop like instruction, addi \( r,0 \), which doesn’t change the state. The command \( \text{quit} \) can be implemented by instructions easily.
tquit:
  lw sp, 20(a0)
  lw a2, 16(a0)
  lw a1, 12(a0)
  lw a0, 8(a0)
  lw v0, 4(a0)
  lw ra, 0(a0)
  ret

7.6 Verification example

We define three invariants of shared resources as below:

\[ I(t,t') \triangleq \sharp(t = \text{sched}) \ast (\text{cur} \mapsto t') \ast \text{rtcb}(t') \ast \text{RQ}(\text{rq}) \ast \text{DQ}(\text{dq}) \]
\[ \forall \not\exists(t' = \text{sched}) \ast (\text{cur} \mapsto t) \ast (t \mapsto \text{RQ}(\text{rq})) \ast \text{DQ}(\text{dq}) \]

\[ X(t,t') \triangleq \sharp(t' = \text{sched}) \ast (\text{cur} \mapsto t) \ast \text{RQ}(\text{rq}) \ast (t \mapsto \text{DQ}(\text{dq})) \]

\[ I(t) \triangleq \sharp(t = \text{sched}) \ast (\text{cur} \mapsto \_\_\_\_) \ast \text{RQ}(\text{rq}) \ast \text{DQ}(\text{dq}) \]
\[ \forall \not\exists(t \neq \text{sched}) \ast \text{rtcb}(t) \ast (\text{cur} \mapsto t) \ast (t \mapsto \text{RQ}(\text{rq})) \ast \text{DQ}(\text{dq}) \]

The code and assertions are presented in Figure ??–??.
create:

\[
\begin{align*}
&\{ t \} \ast \text{tcb}(t') \ast \text{RQ}(rq) \ast \not{\exists}(t' \in \text{dom}(\Delta)) \\
&\ast(a_0, a_1, a_2, v_0, ra \mapsto t', f_{\ldots}, \ldots, \text{ret}) \ast K(bp, 20) \}
\end{align*}
\]

\begin{align*}
\text{subi \enspace sp, \ 4} \\
\text{sw \ ra, \ 0(sp)} \\
\text{call \ kalloc}
\end{align*}

\[
\begin{align*}
&\{ t \} \ast \text{tcb}(t') \ast \text{RQ}(rq) \ast \text{hspace}(k, k\text{maxsize}) \\
&\ast(a_0, a_1, a_2, v_0, ra \mapsto t', f_{\ldots}, k_{\ldots}) \ast \not{\exists}(t' \in \text{dom}(\Delta)) \ast K(bp, 19, \text{ret}) \}
\end{align*}
\]

\begin{align*}
\text{sw \ v_0, \ OFF\_STACK(a_0)} \\
\text{sw \ a_1, \ 0(a_0)} \\
\text{addi \ v_0, \ KMAXSIZE-1} \\
\text{sw \ v_0, \ OFF\_SP(a_0)} \\
\text{movi \ v_0, \ 0} \\
\text{sw \ v_0, \ OFF\_JOINER(a_0)} \\
\text{tnew \ a_0}
\end{align*}

\[
\begin{align*}
&\{ t \} \ast (t') \ast \text{rtcb}(t') \ast \text{RQ}(rq) \\
&\ast(a_0, a_1, a_2, v_0, ra \mapsto t', f_{\ldots}, k_{\ldots}) \ast K(bp, 19, \text{ret}) \}
\end{align*}
\]

\begin{align*}
\text{mov \ a_1, \ a_0} \\
\text{movi \ a_0, \ rq} \\
\text{call \ enq}
\end{align*}

\[
\begin{align*}
&\{ t \} \ast \text{RQ}(rq) \ast (a_0, a_1, a_2, v_0, ra \mapsto \ldots, k_{\ldots}) \ast K(bp, 19, \text{ret}) \}
\end{align*}
\]

\begin{align*}
\text{lw \ ra, \ 0(sp)} \\
\text{addi \ sp, \ 4}
\end{align*}

\[
\begin{align*}
&\{ t \} \ast \text{RQ}(rq) \ast (a_0, a_1, a_2, v_0, ra \mapsto \ldots, \ldots) \ast K(bp, 19, \text{ret}) \}
\end{align*}
\]

\text{ret}

\textbf{Fig. 23.} Verification of create()
exit:

\[
[t] \rightarrow rtcb(t) \rightarrow (cur \rightarrow t) \rightarrow RQ(rq) \rightarrow DQ(dq) \rightarrow \langle sched \rangle \\
\rightarrow (a0, a1, a2, v0, ra \rightarrow 0, 0, 0, 0) \rightarrow K(bp, 20)
\]

movi a2, cur
lw a0, 0(a2)
lw a1, OFF_STACK(a0)
bz a1, exit
quit

exit_quit:

\[
[t] \rightarrow \langle \text{prev} \rightarrow \text{prev} \rangle \rightarrow \langle \text{next} \rightarrow \text{next} \rangle \rightarrow \langle \text{joiner} \rightarrow \text{joiner} \rangle \rightarrow \langle t' \rightarrow rtcb(t') \rangle \\
\rightarrow (cur \rightarrow t) \rightarrow RQ(rq) \rightarrow DQ(dq) \rightarrow \langle sched \rangle \\
\rightarrow (a0, a1, a2, v0, ra \rightarrow t, t', cur, joker) \rightarrow K(bp, 20)
\]

movi a0, rq
call enq
movi a1, 0
sw a1, 36(a0)

exit_quit:

\[
[t] \rightarrow rtcb(t) \rightarrow (cur \rightarrow t) \rightarrow RQ(rq) \rightarrow DQ(dq) \rightarrow \langle sched \rangle \\
\rightarrow (a0, a1, a2, v0, ra \rightarrow t, 0, 0, 0) \rightarrow K(bp, 20)
\]

mov a1, a0
movi a0, rq
call enq
movi a0, sched

\[
[t] \rightarrow (cur \rightarrow t) \rightarrow RQ(rq) \rightarrow \langle \{\rightarrow \text{DQ(dq)}\} \rightarrow \langle sched \rangle \\
\rightarrow (a0, a1, a2, v0, ra \rightarrow sched, \_\_\_\_) \rightarrow K(bp, 20)
\]

quit

Fig. 24. Verification of `exit()`
join:
\[
\{ [t] \circ \text{rtcb}(t) \circ (\text{cur} \mapsto t) \circ \text{RQ}(\text{rq}) \circ \text{DQ}(\text{dq}) \circ \langle \text{sched} \rangle \\
\circ (a_0 \mapsto t') \circ (v_0,a_1,a_2 \mapsto \_ ) \circ (ra \mapsto w_r) \circ K(bp, 20, .) \}
\]
addi sp -12
sw ra 8(sp)
sw a0 4(sp)
movi a2 cur
lw a0, 0(a2)
sw a0, 0(sp)

join_loop:
\[
\{ [t] \circ \text{rtcb}(t) \circ (\text{cur} \mapsto t) \circ \text{RQ}(\text{rq}) \circ \text{DQ}(\text{dq}) \circ \langle \text{sched} \rangle \\
\circ (a_0 \mapsto \_ ) \circ (v_0,a_1,a_2,ra \mapsto \_ ) \\
K(bp, 17, t :: t' :: w_r) \}
\]
movi a0, dq
lw a1, 4(sp)
call rmq
bz v0, join_free

\[
\{ [t'] \circ \text{rtcb}(t') \circ (\text{cur} \mapsto t') \circ \text{RQ}(\text{rq}) \circ \text{DQ}(\text{dq}) \circ \langle \text{sched} \rangle \\
\circ (a_0 \mapsto \_ ) \circ (a_1 \mapsto t') \circ (v_0,a_2,ra \mapsto \_ ) \circ K(bp, 17, t :: t' :: w_r) \}
\]
movi a0, rq
call inq
bz v0, join_in
jmp join_ret

join_in:
\[
\{ [t] \circ \text{rtcb}(t) \circ (\text{cur} \mapsto t) \circ (t') \circ \text{rtcb}(t') \circ \text{DQ}(\text{dq}) \circ \langle \text{sched} \rangle \\
\circ (t') \circ \text{rtcb}(t') \circ \text{RQ}(\text{rq}) \circ (a_0 \mapsto \_ ) \circ (a_1 \mapsto t') \circ (v_0,a_2,ra \mapsto \_ ) \circ K(bp, 17, t :: t' :: w_r) \}
\]
lw a0, 36(a1)
bz a0, join_wait
movi v0, -1
jmp join_ret

Fig. 25. Verification of join() (part 1.)
join_wait:

\[
[t] \ast rtcb(t) \ast (cur \mapsto t) \ast \{t'\} \ast dtcb(t') \ast DQ(dq) \ast \{\text{sched}\}
\ast \{(t') \ast rtcb(t') \mapsto RQ(rq)\}
\ast (a0 \mapsto 0) \ast (a1 \mapsto t') \ast (v0,a2,ra \mapsto _) \ast K(bp,17,t' :: w_r)\}
\]

<table>
<thead>
<tr>
<th>lw</th>
<th>a0,</th>
<th>0(sp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw</td>
<td>a0, 36(a1)</td>
<td></td>
</tr>
<tr>
<td>movi</td>
<td>a1, sched</td>
<td></td>
</tr>
</tbody>
</table>

\[
[t] \ast (cur \mapsto t) \ast (t \mapsto RQ(rq)) \ast DQ(dq) \ast \{\text{sched}\}
\ast (a0 \mapsto t) \ast (a1 \mapsto \text{sched}) \ast (v0,a2,ra \mapsto _)
\ast K(bp,17,t :: t' :: w_r)\}
\]

cswitch  
jmp  join_loop

join_free:

\[
[t] \ast rtcb(t) \ast (cur \mapsto t) \ast RQ(rq) \ast \{t'\} \ast dtcb(t') \ast DQ(dq)
\ast (a0 \mapsto _) \ast (a1 \mapsto t') \ast (v0,a2,ra \mapsto _) \ast \{\text{sched}\}
\ast K(bp,17,t :: t' :: w_r)\}
\]

<table>
<thead>
<tr>
<th>tdel</th>
<th>a1</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>a0, 24(a1)</td>
</tr>
<tr>
<td>call</td>
<td>kfree</td>
</tr>
</tbody>
</table>

\[
[t] \ast rtcb(t) \ast tcb(t') \ast (cur \mapsto t) \ast RQ(rq) \ast DQ(dq) \ast \{\text{sched}\}
\ast (a0,a1 \mapsto _) \ast (v0,a1,a2,ra \mapsto _) \ast K(bp,17,t :: t' :: w_r)\}
\]

| movi  | v0, 0 |

join_ret:

\[
[t] \ast rtcb(t) \ast (cur \mapsto t) \ast RQ(rq) \ast DQ(dq) \ast \{\text{sched}\}
\ast (a0,a1,a2,ra \mapsto _) \ast K(bp,17,t :: t' :: w_r)
\ast (v0 \mapsto 0) \lor (v0 \mapsto -1)\}
\]

<table>
<thead>
<tr>
<th>lw</th>
<th>ra, 8(sp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>sp, 12</td>
</tr>
</tbody>
</table>

\[
[t] \ast rtcb(t) \ast (cur \mapsto t) \ast RQ(rq) \ast DQ(dq) \ast \{\text{sched}\}
\ast (a0,a1,a2 \mapsto _) \ast (ra \mapsto w_r) \ast K(bp,20,\cdot)
\ast ((v0 \mapsto 0) \ast tcb(t')) \lor (v0 \mapsto -1)\}
\]

ret

Fig. 26. Verification of join() (part 2.)
schedth:

\[
\{ [\text{sched}] \ast (\text{cur} \mapsto t) \ast (t) \ast (t) \mapsto \text{RQ(rq)}) \ast K(bp, 10, \cdot) \\
\ast (v0 \mapsto \_ \cdot a0 \mapsto \_ \cdot a1 \mapsto \_ \cdot a2 \mapsto \_ \cdot (ra \mapsto \_)) \}
\]

movi a0, rq

\[
\{ [\text{sched}] \ast (\text{cur} \mapsto t) \ast \text{RQ(rq)} \ast K(bp, \cdot, \cdot) \\
\ast (v0 \mapsto \_ \cdot a0 \mapsto \_ \cdot a1 \mapsto \_ \cdot a2 \mapsto \_ \cdot (ra \mapsto \_)) \}
\]

call deq

bz v0, sched

\[
\{ [\text{sched}] \ast (\text{cur} \mapsto t) \ast (t') \ast \text{ptcb}(t') \ast \text{RQ(rq)} \ast K(bp, 10, \cdot) \\
\ast (v0 \mapsto t' \cdot a0 \mapsto \_ \cdot a1 \mapsto \_ \cdot a2 \mapsto \_ \cdot (ra \mapsto \_)) \}
\]

movi a2, cur

sw v0, 0(a2)

mov a1, v0

lw a0, sched

\[
\{ [\text{sched}] \ast (t') \ast (\text{cur} \mapsto t) \ast \text{ptcb}(t') \ast \text{RQ(rq)} \ast K(bp, 10, \cdot) \\
\ast (v0 \mapsto t' \cdot a0 \mapsto \_ \cdot a1 \mapsto \_ \cdot a2 \mapsto \_ \cdot (ra \mapsto \_)) \}
\]

cswitch

\[
\{ [\text{sched}] \ast \exists t'' . (\text{cur} \mapsto t'') \ast ((t'') \mapsto \text{RQ(rq)}) \ast K(bp, 10, \cdot) \\
\ast (v0 \mapsto t' \cdot a0 \mapsto \_ \cdot a1 \mapsto t' \cdot a2 \mapsto \_ \cdot (ra \mapsto \_)) \}
\]

jmp sched th

yield:

\[
\{ [t] \ast \text{rtcb}(t) \ast (\text{cur} \mapsto t) \ast \text{RQ(rq)} \ast K(bp, 10, \cdot) \\
\ast (v0 \mapsto \_ \cdot a0 \mapsto \_ \cdot a1 \mapsto \_ \cdot a2 \mapsto \_ \cdot (ra \mapsto \_)) \}
\]

movi a0, rq

movi a2, cur

lw a1, 0(a2)

call enq

movi a2, cur

lw a0, 0(a2)

movi a1, sched
csw

ret

Fig. 27. Verification of schedth() and yield()
Fig. 28. Verification of `kill()`
8 Extension 3: Scheduling over SMP

In this section, we extend our proof system to verify a scheduler with symmetrical multiprocessing support and simple load balancing mechanism.

We show a simple example that doing thread scheduling over multiple processors in Figure ???. We introduce a new struct for each processor to record the meta-data of the processor. Since the processors cause concurrency, we also need some synchronization operations, spin locks, to protect shared memory. The scheduler follows the pattern (II) explained in the Section ???. Different from the version on uniprocessor, each processor has its own thread run queue. The scheduler finds the next runnable thread in the local thread run queue, and a load balancer can move threads between run queues of different processors.

In this example, we omit interrupts for simplicity and we assume there is a special operation `cpuid()` from that we can have the identity of the processor. We assume that the identity numbers of processor are from zero to $NCPU - 1$, where $NCPU$ is the total number of processors. We extend the thread control block with one more field, `cpuid`, to record which processor the thread runs on.

If a thread run queue owned by some processor is empty, The function `loadbalance()` can be invoked to pull some runnable threads from the run queue of the busiest processor. Therefore, each run queue structure has one special lock, which is implemented as a spin lock, to protect the run queue. If a thread attempts to acquire a spin lock while it is contended, the thread will busy spins waiting for the lock to become available. The spinning prevents more than one thread of execution from entering the critical region at any one time. We implement a naive algorithm of spin lock with an atomic compare-and-swap instruction `xchg`.

In the struct `cpu`, we add containing processor related information. The pointer `curr` points to the current running thread on the processor and `rq` points to the thread queue of the processor.

To support SMP, the scheduler function `schedule()` should be upgraded as follows:

- The pointer of thread queue of the current processor should be obtained via calling `cpuid()`.
- Before accessing thread run queue, the code must acquire its lock.
- If there is no more thread in the current thread run queue, the scheduler will calling `loadbalance()` to move some threads from some busy processor.
- The scheduler get the next thread from the current thread run queue. If fails, the function will return immediately. If succeeds, it will set the `cpuid` of the next thread, change the pre-CPU variable `curr` and perform context switch.
- Most importantly, the scheduler need to get the CPU-ID again after returning back, for the current thread might be migrated to another processor and wake up there.
- The scheduler need to release the lock of the current run queue before return. Readers may notice the subtle problem, a thread may release the lock that it didn’t acquire because of thread migration. Interestingly, as long as every thread releases the lock of the the thread run queue of the current thread, the world is going to be in order.
struct tcb {
    struct context ctxt;
    struct tcb *prev;
    struct tcb *next;
};

struct spinlock {
    int locked;
};

struct cpu {
    struct spinlock lock;
    struct tcb *curr;
    struct queue rq;
} cpus[NCPU];

void lock(struct spinlock *lk){
    while(xchg(&lk->locked, 1) != 0);
}

void unlock(struct spinlock *lk){
    xchg(&lk->locked, 0);
}

void schedule(){
    struct tcb *curr, *next;
    struct cpu *cpu;
    cpu = &cpus[cpuid()];
    curr = cpu->curr;
    lock(&cpu->lock);
    next = deq(&cpu->rq);
    if (next == NULL) {
        unlock(&cpu->lock);
        loadbalance();
        lock(&cpu->lock);
        next = deq(&cpu->rq);
        if (next == NULL)
            goto sched_ret;
    }
    enq(&cpu->rq, curr);
    cswitch(curr, next);
    cpu = &cpus[cpuid()];
    sched_ret:
    unlock(&cpu->lock);
    return;
}

void loadbalance(){
    struct cpu cpu0, cpu1;
    struct tcb *mt;
    struct cpu *cpua, *cpub;
    cpu0 = cpus[cpuid()];
    cpu1 = busiest_cpu(cpu0);
    if (cpu1 == NULL) goto Llb_end;
    cpua = cpu0;
    cpub = cpu1;
    if(cpu0 > cpu1)
        swap(cpua, cpub);
    lock(&cpua->lock);
    lock(&cpub->lock);
    mt = deq(cpu1->rq);
    if (mt == NULL) goto Llb_end;
    enq(&cpu0->rq, mt);
    Llb_end:
    unlock(&cpub->lock);
    unlock(&cpua->lock);
    return;
}

Fig. 29. Implementation of scheduler over SMP
8.1 Abstract machine and operational semantics

To support multicore machine, we add a notation $X$ to specify a set of processors, each of them has a register file and a program counter. The running abstract thread is extended with a processor-ID, $u$, which is just a machine word. The instruction set of the machine is extended with an atomic instruction $\text{xchg } r_d, [r_s + w]$ to exchange the value (in the register $r_s$) with the value in memory (addressed by $r_s$).

The operational semantics of instructions is shown in Fig. 32. The new instruction $\text{cpuid } r$ puts the CPU-ID into the register $r$. And the other new instruction $\text{xchg}$ exchanges the value in a register with the value in a memory cell. All of other instructions are same to the definitions in Fig. ??. For the abstract instruction, $\text{cswitch}$, it set the CPU-ID of the current processor to the next ready thread. The predicate $\text{atomic}$ is used...
\[(u,R,pc,M) \xrightarrow{i} (u',R',pc',M')\]

If \(i = 0\) then
\[
xchg \, r_d, w(r_s) \quad M' = M(R(r_s) + w : R(r_d)) \land pc' = pc + 1 \\
\land R' = R\{r_d : M(R(r_s) + w)\}
\]

otherwise \[(R,pc,M) \xrightarrow{} (R',pc',M')\]

\[(u,R,pc,M,P) \xrightarrow{c} (u',R',pc',M',P')\]

if \(c = 0\) then
\[
cswitch \exists R''', P'''. \quad M = M' \land t = R(a_0) \land t' = R(a_1) \\
\land P = \{t : \text{(run, } u\text{)} \land t' : \text{(rdy, } R'')\} \uplus P'''
\land P' = \{t : \text{(rdy, } R'')\} \land t' : \text{(run, } u\text{)} \uplus P'''
\land pc''' = pc + 1 \land R''' = R\{r_a : pc + 1\}
\]
\(R \text{ and } R' \text{ is complete.}\)

atomic(\(i\)) \(\triangleq i \in \{\text{xchg } r_d, [r_s + w]\}\)

Fig. 32. Operational semantics of instructions (smp)

to specify whether an instruction is an atomic instruction, which locks the data bus to prevent memory from being accessed by other processors. Currently, only the xchg is atomic.

The operational semantics of our multicore machine is shown in Fig. 32. One execution step of a processor is either an atomic step or a concurrent step. If a processor runs one atomic step, it goes forward without stop. While if it runs one concurrent step, it may go forward or be stopped by other atomic instruction on other processors. The step relation for the whole multicore machine is in the form of:

\[(C,Y,M,P) \rightarrow (C',Y',M',P')\]

If one processor runs an atomic instruction, all of other processors are stopped from execution. And if no atomic instruction at a time, then all processors can run one step or not.

The notation of an abstract running thread is changed to \([t \circ u]\), adding the current processor-ID. The notation of an abstract ready thread

\[\langle t \rangle \triangleq \lambda(M,R,P). M = \{\} \land R = \{\} \land P = \{t : \text{(run, } u\text{)}\}\]

8.2 Inference Rules

We only show inference rules new in Figure 32. Other rules are similar with the rules defined previously. The judgments of this version of verification framework is of the form:

\[\vdash \]

50
We define an operator to append the shared resources to a thread's local state. In concurrent separation logic, the invariants in our proof system are abstract and can be instantiated to concrete definitions according to the concrete implementation of kernels. In multicore machines, there are two kinds of shared resources.

The first kind is the global resources shared by all processors. We use a notation \( \Sigma \) to specify the invariant of global resources. If one thread accesses the resources specified by \( I_2 \), it should use kernel synchronization operations, e.g., spin locks, to protect accessing. The invariant \( \Sigma \) is a partial map from lock locations to assertions.

\[ \Sigma \in \text{Word} \rightarrow \text{Assert} \]

The notation \( I_2 \) is used to specify all the global resources, and it is defined as a big separation conjunction assertion of all of the assertions in the \( \Sigma \):

\[ I_2 \triangleq \bigcirc_{l \in \text{dom}(\Sigma)} (l \mapsto 0 \ast \text{emp}) \lor (l \mapsto 1 \ast \Sigma(l)) \]

We define an operator to append the shared resources to a \((p, g)\) pair so as to specify actions performed by atomic instructions, like `xchg` etc.

\[ (p, g) \ast I_2 \triangleq (p \ast I_2, (\lambda S, S'. \forall S_1, S_2. S = S_1 \uplus S_2 \rightarrow pS_1 \rightarrow I_2S_2 \rightarrow \exists S'_1, S'_2. S' = S'_1 \uplus S'_2 \land gS_1S'_1 \land I_2S'_2)) \]
∀ \xi ∈ \text{dom}(\circ) \Rightarrow \Psi, \Sigma, I \vdash (p, g) \xi : C(\xi)

Fig. 34. Inference rules of Ext. 3

Despite the two forms of concurrency, one thread can also voluntarily perform context switch to relinquish processor. The invariants $I$ and $I$ defined in the Sec. ?? specify the resources, coming either from global shared resources, per-CPU shared resources, or thread local resources, transferred through context switch. They need one more parameter, $u$ to relate resources with processors.

$$I \in \text{CID} \rightarrow \text{TID} \rightarrow \text{TID} \rightarrow \text{Assert}$$

The rule (XCHG) is for atomic instruction xchg. Atomic instruction can access memory with bus locked, and then have the resources of global shared memory. So the specification of this instruction is the separation conjunction of local resources and shared resources $I_\Sigma$. The requirement is that the resources of global shared memory should satisfy its invariant before and after the accessing by atomic instructions.

The rule (CSWTICH) requires that the transferred shared resources should satisfy $I(u, t, t')$ with respect to the current processor ID. After context switch to the same thread, it may wake up on a different processor with different a processor ID.

8.3 Concrete thread resources and soundness

After changing of the concept of abstract running threads, we need change the definitions of concrete threads likewise. The continuation of a thread is same to the definition in Section. We list the definitions that only need to be changed here.

Running thread. The definition of a concrete running thread is the resources asserted by the assertion $p$ and the continuation. The resources should include the abstract resource of the current running thread and all registers.

$$\text{CThrd}(\Psi, u, t, pc) \triangleq \exists n. \text{Cont}(n, \Psi, pc) \land ([t \circ u] + [-] + \text{true})$$

52
**Ready thread.** For a ready thread, its concrete resources can be defined by separation implication →: if given the resources of machine context data, \( |R| \), (2) its own running thread abstract resource, \( [t \odot u] \), on an unknown processor, and (3) shared resources specified by \( \circ I(t) \), a ready thread can become a running thread.

\[
\text{RThrd}(\Psi, t, R) \triangleq [R] \ast \exists u \cdot [t \odot u] \ast \exists t' \cdot \langle t' \rangle \ast \circ I(u, t', t) \rightarrow \text{CThrd}(\Psi, u, t, R(ra))
\]

The second parameter of RThrd is the ID of the thread and the third parameter is the machine context data saved in its TCB. Please note that the program counter of a runnable thread is saved into the register ra.

**Invariant of global resources.** The invariant of global resources is defined with respect to the structure of the thread pool.

\[
\text{GINV}(\Psi, Y, P, pc) \triangleq ([R_0] \rightarrow \text{CThrd}(\Psi, u_0, t_0, pc_0)) \ast \cdots \ast ([R_n] \rightarrow \text{CThrd}(\Psi, u_n, t_n, pc_n)) \ast \text{RThrd}(\Psi, t(n+1), R(n+1)) \ast \cdots \ast \text{RThrd}(\Psi, t(n+1), R(n+1))
\]

where \( Y = \{u_0 : (R_0, pc_0), \ldots, u_n : (R_n, pc_n)\} \)

where \( P = \{t_0 : (run, u_0), \ldots, t_n : (run, u_n), t_{n+1} : (rdy, R_{n+1}), \ldots, t_{n+m} : (rdy, R_{n+m})\} \)

### 8.4 Soundness
The global invariant of the whole machine is defined as below: The memory resources and abstract thread resources can be partitioned into \( n \) parts, where \( n \) is the number of processors. Each processor along with its resources satisfies the interpretation of abstract threads that it owns.

### 8.5 Verification of scheduler and load balancer
The assembly code of scheduler and load balancer are shown in Figure 35–39. The global shared resources are the thread run queues owned by all processors. The invariant \( \Sigma \) is defined as a map from \( lk_i \) to assertion specified by \( RQ(u) \).

\[
I(u, t, t') \triangleq \text{curr}(u, t') \ast \text{ptcb}(t') \ast \langle t \rangle \rightarrow \text{RQ}(u)
\]

\[
\Sigma \triangleq \{lk_0 : \text{RQ}(0), \ldots, lk_n : \text{RQ}(n)\}
\]

where \( lk_u = \text{CPU} \times u \times \text{SIZE_CPU} \times \text{OFF_LOCK} \)

\[
\text{ptcb}(t) \triangleq \langle t \xrightarrow{\text{prev}} \_ \rangle \ast \langle t \xrightarrow{\text{next}} \_ \rangle
\]

\[
\text{RQ}(u) \triangleq \exists \text{rq} : \{ \text{rq} = \text{CPU} \times u \times \text{SIZE_CPU} \times \text{OFF_RQ} \} \ast \text{RQ}(rq)
\]

\[
\text{curr}(u, t) \triangleq \langle \text{CPU} \times u \times \text{SIZE_CPU} \times \text{OFF_CURR} \rangle \rightarrow t
\]

where CPU is the starting location of the array of cpu structs; SIZE_CPU is the size of the cpu struct; OFF_LOCK, OFF_RQ and OFF_CURR are the offsets of fields in the cpu struct defined in Figure 29.
\( \{ t \circ u \} + ptcb(t) + curr(u,t) + (a0,a1,a2,v0,ra \mapsto w1,w2,w3,_,ret) + K(bp.20) \}

**schedule:**

```assembly
subi sp, 28
sw ra, 24(sp)
sw a2, 20(sp)
sw a1, 16(sp)
sw a0, 12(sp)
```

\( \{ t \circ u \} + ptcb(t) + curr(u,t) + (a0,a1,a2,v0,ra \mapsto \_\_\_\_\_\_\_\_ \}
* K(bp.13,\_ \_ \_ :: \_ :: w1 :: w2 :: w3 :: ret) \}

cpuid a0
call getcpu

\( \{ t \circ u \} + ptcb(t) + curr(u,t) + (a0,a1,a2,v0,ra \mapsto u,\_\_\_\_\_\_\_\_CPUS[u] \_\_\_ \}
* K(bp.13,\_ :: :: CPUS[u] :: w1 :: w2 :: w3 :: ret) \}

sw v0, 8(sp)
call lock

\( \{ t \circ u \} + ptcb(t) + curr(u,t) + RQ(u) + (a0,a1,a2,v0,ra \mapsto \_\_\_\_\_\_\_\_CPUS[u] \_\_\_ \)
* K(bp.13,\_ :: :: CPUS[u] :: w1 :: w2 :: w3 :: ret) \}

1w a0, OFF_LOCK(v0)
call deq
bz v0, sched_lb
jmp sched_cont

\( \{ t \circ u \} + ptcb(t) + curr(u,t) + RQ(u) + (a0,a1,a2,v0,ra \mapsto \_\_\_\_\_\_\_\_CPUS[u] \_\_\_NULL,\_ \)
* K(bp.13,\_ :: :: CPUS[u] :: w1 :: w2 :: w3 :: ret) \}

**sched_lb:**

1w a2, 8(sp)
1w a0, OFF_LOCK(a2)
call unlock

\( \{ t \circ u \} + ptcb(t) + curr(u,t) + (a0,a1,a2,v0,ra \mapsto \_\_\_\_\_\_\_\_CPUS[u] \_\_\_NULL,\_ \)
* K(bp.13,\_ :: :: CPUS[u] :: w1 :: w2 :: w3 :: ret) \}

call loadbalance

\( \{ t \circ u \} + ptcb(t) + curr(u,t) + (a0,a1,a2,v0,ra \mapsto \_\_\_\_\_\_\_\_CPUS[u] \_\_\_ \)
* K(bp.13,\_ :: :: CPUS[u] :: w1 :: w2 :: w3 :: ret) \}

1w a2, 8(sp)
1w a0, OFF_LOCK(a2)
call lock

\( \{ t \circ u \} + ptcb(t) + curr(u,t) + RQ(u) + (a0,a1,a2,v0,ra \mapsto \_\_\_\_\_\_\_\_CPUS[u] \_\_\_ \)
* K(bp.13,\_ :: :: CPUS[u] :: w1 :: w2 :: w3 :: ret) \}

1w a2, 8(sp)
1w a0, OFF_LOCK(a2)
call deq
bz v0, sched_ret

\( \{ t \circ u \} + ptcb(t) + curr(u,t) + RQ(u) + \exists t' \cdot (t' \cdot ptcb(t')) + (a0,a1,a2,v0,ra \mapsto \_\_\_\_t',NULL,\_ \)
* K(bp.13,\_ :: :: CPUS[u] :: w1 :: w2 :: w3 :: ret) \}

---

*Fig. 35. Ext. 3: Verification of scheduler (part one)*
\[ \{ t \in u \} \cup \text{ptcb}(t) \cup \text{cur}(u,t) \cup \text{RQ}(u) \cup \exists \text{t}' \cup \exists \text{t'} \cup \text{ptcb}(\text{t'}) \]

\* \text{K}(\text{bp},13: \text{u} \leftarrow \text{CPUS}[u] \leftarrow \text{w}1 \leftarrow \text{w}2 \leftarrow \text{w}3 \leftarrow \text{ret})}

\begin{align*}
sch\text{cont} \colon & \text{sw} \quad v0, \quad 0(\text{sp}) \\
& \text{lw} \quad a2, \quad 8(\text{sp}) \\
& \text{lw} \quad a1, \quad \text{OFF\_CURR}(a2) \\
& \text{sw} \quad a1, \quad 4(\text{sp}) \\
& \text{lw} \quad a0, \quad \text{OFF\_RQ}(a2)
\end{align*}

\[ \{ t \in u \} \cup \text{ptcb}(t) \cup \text{cur}(u,t) \cup \text{RQ}(u) \cup \exists \text{t}' \cup \exists \text{t'} \cup \text{ptcb}(\text{t'}) \]

\* \text{K}(\text{bp},13: \text{u} \leftarrow \text{CPUS}[u] \leftarrow \text{w}1 \leftarrow \text{w}2 \leftarrow \text{w}3 \leftarrow \text{ret})}

\text{call} \quad \text{enq}

\[ \{ t \in u \} \cup \text{ptcb}(t) \cup \text{cur}(u,t) \cup \text{RQ}(u) \cup \exists \text{t}' \cup \exists \text{t'} \cup \text{ptcb}(\text{t'}) \]

\* \text{K}(\text{bp},13: \text{u} \leftarrow \text{CPUS}[u] \leftarrow \text{w}1 \leftarrow \text{w}2 \leftarrow \text{w}3 \leftarrow \text{ret})}

\text{cs\text{witch}}

\[ \{ u' \colon t \in u' \} \cup \text{ptcb}(t) \cup \text{cur}(u',t) \cup \exists \text{t}'' \cup \exists \text{t'} \cup \text{ptcb}(\text{t'}) \]

\* \text{K}(\text{bp},13: \text{u} \leftarrow \text{CPUS}[u] \leftarrow \text{w}1 \leftarrow \text{w}2 \leftarrow \text{w}3 \leftarrow \text{ret})}

\text{cp\text{uid}} \quad a0

\text{call} \quad \text{get\text{cpu}}

\[ \{ u' \colon t \in u' \} \cup \text{ptcb}(t) \cup \text{cur}(u',t) \cup \exists \text{t}'' \cup \exists \text{t'} \cup \text{ptcb}(\text{t'}) \]

\* \text{K}(\text{bp},13: \text{u} \leftarrow \text{CPUS}[u] \leftarrow \text{w}1 \leftarrow \text{w}2 \leftarrow \text{w}3 \leftarrow \text{ret})}

\text{sw} \quad v0, \quad 8(\text{sp})

\text{lw} \quad a0, \quad \text{OFF\_LOCK}(v0)

\text{call} \quad \text{unlock}

\[ \{ u' \colon t \in u' \} \cup \text{ptcb}(t) \cup \text{cur}(u',t) \cup \exists \text{t}'' \cup \exists \text{t'} \cup \text{ptcb}(\text{t'}) \]

\* \text{K}(\text{bp},13: \text{u} \leftarrow \text{CPUS}[u'] \leftarrow \text{w}1 \leftarrow \text{w}2 \leftarrow \text{w}3 \leftarrow \text{ret})}

\text{lw} \quad a0, \quad 12(\text{sp})

\text{lw} \quad a1, \quad 16(\text{sp})

\text{lw} \quad a2, \quad 20(\text{sp})

\text{lw} \quad ra, \quad 24(\text{sp})

\text{addi} \quad \text{sp}, \quad 28

\[ \{ u' \colon t \in u' \} \cup \text{ptcb}(t) \cup \text{cur}(u',t) \cup \exists \text{t}'' \cup \exists \text{t'} \cup \text{ptcb}(\text{t'}) \]

\* \text{K}(\text{bp},13: \text{u} \leftarrow \text{CPUS}[u'] \leftarrow \text{w}1 \leftarrow \text{w}2 \leftarrow \text{w}3 \leftarrow \text{ret}) \star \text{K}(\text{bp},20))}

\text{ret}

\text{Fig. 36. Ext. 3: Verification of scheduler (part two)}
\[
\{(a_0, a_1, r_a \mapsto l_k, w_1, r_t) \ast K(bp, 1) \ast \# (l_k \in \text{dom}(\Sigma))\}
\]

lock : /* a simplest spin lock */
  
  addi sp, -4
  sw a1 0(sp)

\[
\{(a_0, a_1, r_a \mapsto l_k, w_1, r_t) \ast K(bp, 0, w_1) \ast \# (l_k \in \text{dom}(\Sigma))\}
\]

lock_loop:
  movi a1, 1

\[
\{(a_0, a_1, r_a \mapsto l_k, 1, r_t) \ast K(bp, 0, w_1) \ast \# (l_k \in \text{dom}(\Sigma))\}
\]

xchg a1, 0(a0)

\[
\{(a_0, a_1, r_a \mapsto l_k, x, r_t) \ast K(bp, 0, w_1) \ast \left((\# x = 1) \lor (\# x = 0) \ast \Delta(l_k)\right) \ast \# (l_k \in \text{dom}(\Sigma))\}
\]

bz a1, lock_ret
jmp lock_loop

\[
\{(a_0, a_1, r_a \mapsto l_k, 0, r_t) \ast K(bp, 0, w_1) \ast \Delta(l_k) \ast \# (l_k \in \text{dom}(\Sigma))\}
\]

lock_ret:
  lw a1, 0(sp)
  addi sp, 4
  ret

\[
\{(a_0, a_1, r_a \mapsto l_k, w_1, r_t) \ast K(bp, 1) \ast \# (l_k)\}
\]

unlock:
  addi sp, -4
  sw a1, 0(sp)
  movi a1, 0
  xchg a1, 0(a0)

\[
\{(a_0, a_1, r_a \mapsto l_k, 0, r_t) \ast K(bp, 0, w_1)\}
\]

lw a1, 0(sp)
addi sp, 4

\[
\{(a_0, a_1, r_a \mapsto l_k, w_1, r_t) \ast K(bp, 1)\}
\]

ret

Fig. 37. Ext. 3: Verification of lock/unlock
Fig. 38. Ext. 3: Verification of loadbalance (part one)
Llb: lw a1, 4(sp)  
call lock  
lw a1, 8(sp)  
lw a0, OFF_LOCK(a1)  
call lock

Llb_end: sw a1, 16(sp)  
sw ra, 20(sp)  
addi sp, 24

Fig. 39. Ext. 3: Verification of loadbalance (part two)
9 Related work

Gotsman and Yang [7] proposed a two-layer framework to verify schedulers. The proof system in the lower-layer is for verifying code manipulating TCBs, while the upper-layer is for verifying the rest concurrent code of the kernel. Since thread queues and TCBs are hidden from the upper-layer, one thread could not have any knowledge of the others, thus their proof system is unable to verify the scheduling pattern of II and III. Similar to our assertion $RThrd(\cdots)$, they introduced a primitive predicate $Process(G)$ to relate TCBs in the lower-layer with threads in the upper-layer, but there is no counterpart of $\langle i \rangle$ in their framework.

Feng et al. also verified a kernel prototype [3] in a two-layer framework. Code manipulating TCBs needs to be verified in the lower-layer of their framework. The TCBs are connected with actual threads in the upper layer by an interpretation function of their framework. Our use of global invariant is similar to their use of the interpretation function. In the upper-layer, information of threads is completely hidden. Thus, their framework also fails to support the verification of the scheduler pattern of II and III.

Ni et al. verified a small thread manager with a logic system [16,15] that supports modular reasoning about code including embedded code pointers. In their logic, however, there is no abstraction of threads. Multithreaded programs are seen as sequential interleaving of pieces of code in low-level continuation passing style. Therefore, TCBs with embedded code pointers can be treated as normal data. But since the reasoning level of their method is too low without any abstraction, TCBs have to be specified by over-complicated logic expressions and then it is very difficult to apply their method to realistic code.

Klein et al. verified a micro-kernel, seL4 [12], where the kernel code runs sequentially. Thus they used a sequential proof system to verify most of the kernel code. The scheduling pattern of seL4 is similar to our pattern I, but they trusted the code doing context saving and loading, and left it unverified. Since they do not verify user processes upon the kernel, they need not relate TCBs in the kernel with actual user processes.

Gargano et al. used a framework CVM [6] to build verified kernels in the Verisoft project. CVM is a computational model for concurrent user processes, which interleave through a micro-kernel. Starostin and Tsyban presented a formal approach [20] to reason about context switch between user processes. The context switch code and proofs are integrated in a framework for building verified kernels (CVM) [11]. Their framework keeps a global invariant, weak consistency, to relate TCBs in the kernel with user processes outside the kernel. Since the kernel itself is sequential, their process scheduling follows pattern I. The other two patterns cannot be verified.

10 Conclusion

In this paper, we proposed a novel approach to verify concurrent thread management code, which allows multiple threads to modify their own thread control blocks. The assertions of the code and inference rules of the proof system are straightforward and easy to follow. Moreover, it can be easily extended to support other kernel features (e.g., preemptive scheduling, multi-core systems, synchronizations) and be practically applied to realistic OS code.
References