Modular Verification of Concurrent Thread Management

Yu Guo[1], Xinyu Feng[1], Zhong Shao[2], and Peizhi Shi[1]

[1] University of Science and Technology of China
{guoyu,xyfeng}@ustc.edu.cn sea10197@mail.ustc.edu.cn
[2] Yale University
zhong.shao@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 create 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 about 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 [15]. On the other hand, in many real-world systems such as Linux-2.6.10 [12] and FreeBSD-5.2 [13], 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 [12,1] the thread scheduling itself is implemented as a separate thread that runs concurrently 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.* [15] verified both the thread switch and the threads in one logic [14], 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 [3,6] 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 [11,9] 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 [3,6]. The majority 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

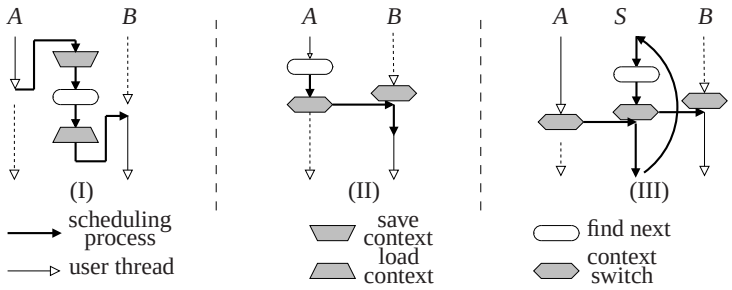

Fig. 1. Three patterns of scheduling

their use by concurrent programs can be reasoned about modularly following concurrent separation logic (CSL) [16]. 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 [3,6] can only verify one of them).
– In our extended TR [7], 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. 6, and then conclude.

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

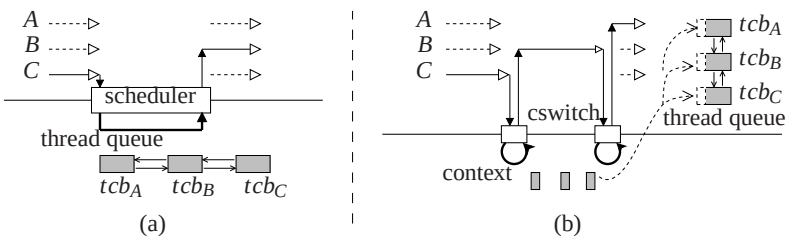

Fig. 2. Abstraction in verification framework

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 [8] 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 [12], 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.

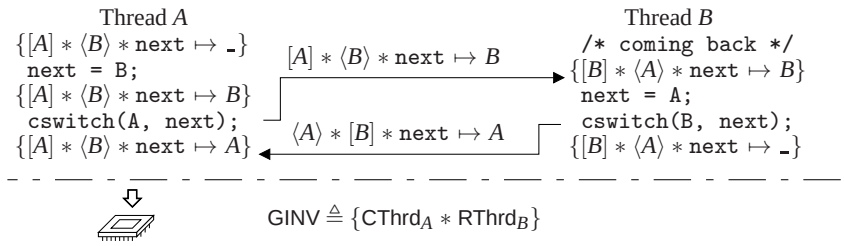

Fig. 3. Abstract thread res. vs. concrete thread res.

(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 [3,6] 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 access 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 does it fulfill 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 that 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 their correctness is still important to the safety of the kernel, *e.g.,* the pointers organizing 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 `next` (*i.e.*, $\langle B \rangle$), we know it is safe to switch to it via the operation `cswitch(A,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 `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 concrete threads. The concrete representation of abstract threads, including stack, TCBs *etc.*, can be defined globally. 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 CThrd_A and the other is a ready thread 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.

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.

Physical machine. The formal definition of the physical machine is shown in Fig. 4 (left side). A machine configuration $\mathbb{W}$ consists of a code block $\mathbb{C}$, a memory block $\mathbb{M}$, a register file $\mathbb{R}$ and a program counter `pc`. The machine has 6 general registers. Some common instructions are defined to write programs in this paper. Their meanings, as well as the operational semantics, follow the conventions. For simplicity, we omit many realistic hardware details, *e.g.,* address alignment and bits-arithmetic.

Abstract machine. The abstract machine is shown in Fig. 4 (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 thread pool P is a partial mapping from thread

$$
\begin{array}{rrcl}
\text{(PhyMach)} & \mathbb{W} & ::= & (\mathbb{C},\mathbb{M},\mathbb{R},\mathtt{pc}) \\
\text{(PhyCode)} & \mathbb{C} & ::= & \{\mathtt{f}:\mathtt{i}\}^* \\
\text{(PhyMem)} & \mathbb{M} & ::= & \{\mathtt{l}:\mathtt{w}\}^* \quad (\mathtt{l}=4n) \\
\text{(PhyRegFile)} & \mathbb{R} & ::= & \{\mathtt{r}:\mathtt{w}\}^* \\
\text{(Register)} & \mathtt{r} & ::= & \mathtt{v0} \mid \mathtt{a0} \mid \mathtt{a1} \mid \mathtt{a2} \mid \mathtt{sp} \mid \mathtt{ra} \\
\text{(Instruction)} & \mathtt{i} & ::= & \mathtt{add}\ \mathtt{r}_d,\ \mathtt{r}_s \mid \mathtt{addi}\ \mathtt{r}_d,\ \mathtt{w} \\
 & & & \mid \mathtt{mov}\ \mathtt{r}_d,\ \mathtt{r}_s \mid \mathtt{movi}\ \mathtt{r}_d,\ \mathtt{w} \\
 & & & \mid \mathtt{lw}\ \mathtt{r}_t,\ \mathtt{w}(\mathtt{r}_s) \mid \mathtt{sw}\ \mathtt{r}_t,\ \mathtt{w}(\mathtt{r}_s) \\
 & & & \mid \mathtt{jmp}\ \mathtt{f} \mid \mathtt{call}\ \mathtt{f} \mid \mathtt{ret} \\
 & & & \mid \mathtt{subi}\ \mathtt{r}_d,\ \mathtt{w} \mid \mathtt{bz}\ \mathtt{r}_t,\ \mathtt{f}
\end{array}
$$

$$
\begin{array}{rrcl}
\text{(AbsMach)} & W & ::= & (C,S,\mathtt{pc}) \\
\text{(State)} & S & ::= & (M,R,P) \\
\text{(AbsCode)} & C & ::= & \{\mathtt{f}:\mathtt{c}\}^* \\
\text{(Mem)} & M & ::= & \{\mathtt{l}:\mathtt{w}\}^* \\
\text{(RegFile)} & R & ::= & \{\mathtt{r}:\mathtt{w}\}^* \\
\text{(TID)} & t & ::= & \mathtt{w} \\
\text{(Pool)} & P & ::= & \{t:T\}^* \\
\text{(Thrd)} & T & ::= & \mathit{run} \mid (\mathit{rdy},R) \\
\text{(AbsInstr)} & \mathtt{c} & ::= & \mathtt{cswitch} \mid \mathtt{i} \\
\text{(TIDList)} & L & ::= & t :: L \mid \mathsf{nil}
\end{array}
$$

Fig. 4. Physical and abstract machine models

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 (`cswitch`) and other physical machine instructions defined on the left. We model the operational semantics using the step transition relation $W \longmapsto W'$ defined in Fig. 5. The abstract instruction `cswitch` requires two thread IDs passed as arguments in `a0` and `a1`, one of which is tagged by *run* and the other is taged by *rdy* 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.

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 (in the TR). The code block at the abstract machine level is extended with the code of implementation of context switch, and the abstract instruction `cswitch` is translated to a call instruction that invokes the implementation code of context switch. The memory block at the abstract machine level is translated to physical memory block by being merged with the memory where context data is stored. By the translation, it can be proved that any safe program over the abstract machine is safe over the physical machine.

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 verification of concurrent code with modification of TCBs at the assembly level.

$((M,R,P),\mathtt{pc}) \overset{\mathtt{c}}{\hookrightarrow} ((M',R',P'),\mathtt{pc}')$	
if $\mathtt{c}=$	then
$\mathtt{i}$	$((M,R),\mathtt{pc}) \overset{\mathtt{i}}{\hookrightarrow} ((M',R'),\mathtt{pc}') \wedge P=P'$
$\mathtt{cswitch}$	$\exists R'',P''.M=M' \wedge R''=R\{\mathtt{ra}:\mathtt{pc}+1\} \wedge t=R(\mathtt{a0})$ $\wedge t'=R(\mathtt{a1}) \wedge \mathtt{pc}'=R'(\mathtt{ra})$ $\wedge P=\{t:run,\ t':(rdy,R')\} \uplus P''$ $\wedge P'=\{t:(rdy,R''),\ t':run\} \uplus P''$ R and R' is complete.

$((M,R),\mathtt{pc}) \overset{\mathtt{i}}{\hookrightarrow} ((M',R'),\mathtt{pc}')$	
if $\mathtt{i}=$	then
$\mathtt{add}\ \mathtt{r}_d,\ \mathtt{r}_s$	$M'=M \wedge R'=R\{\mathtt{r}_d:R(\mathtt{r}_d)+R(\mathtt{r}_s)\} \wedge \mathtt{pc}'=\mathtt{pc}+1$
$\mathtt{call}\ \mathtt{f}$	$M'=M \wedge R'=R\{\mathtt{ra}:\mathtt{pc}+1\} \wedge \mathtt{pc}'=\mathtt{f}$
$\mathtt{jmp}\ \mathtt{f}$	$M'=M \wedge R'=R \wedge \mathtt{pc}'=\mathtt{f}$
$\mathtt{ret}$	$M'=M \wedge R'=R \wedge \mathtt{pc}'=R(\mathtt{ra})$

$$\frac{C(\mathtt{pc})=\mathtt{c} \qquad (S,\mathtt{pc}) \overset{\mathtt{c}}{\hookrightarrow} (S',\mathtt{pc}')}{(C,S,\mathtt{pc}) \longmapsto (C,S',\mathtt{pc}')}$$

Fig. 5. Operational semantics of abstract machine

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 [17], are *shallowly embedded* in the meta language , as shown in Fig. 6. In our assertion language, there are two special assertion constructs for abstract threads. One of them is $\langle t \rangle$ specifying a ready thread and the other is $[t]$ specifying a 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 $\mathtt{r} \mapsto \mathtt{w}$ specifies a register with the value of $\mathtt{w}$. The notation $\mathtt{r}_1,\ldots,\mathtt{r}_n \mapsto \mathtt{w}_1,\ldots,\mathtt{w}_n$ is a compact form for multiple registers.

We borrow the idea from SCAP [4] and use a (p,g) pair 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 the end of the current function. For all instructions in C, their (p,g) pairs are put in Ψ, a global mapping from labels to specifications. 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,

$$
\begin{array}{rcl}
\mathsf{true} & \triangleq & \lambda(M,R,P).\ \mathsf{True} \\
\mathsf{false} & \triangleq & \lambda(M,R,P).\ \mathsf{False} \\
\mathsf{emp} & \triangleq & \lambda(M,R,P).M=\{\cdot\} \wedge R=\{\cdot\} \wedge P=\{\cdot\} \\
p * q & \triangleq & \lambda(M,R,P).\exists M_1,M_2,R_1,R_2,P_1,P_2.M=M_1 \uplus M_2 \wedge R=R_1 \uplus R_2 \wedge P=P_1 \uplus P_2 \\
 & & \quad \wedge p\ (M_1,R_1,P_1) \wedge q\ (M_2,R_2,P_2) \\
p -\!\!* q & \triangleq & \lambda(M,R,P).\forall M_1,R_1,P_1,M',R',P'.(M'=M_1 \uplus M \wedge R'=R_1 \uplus R \wedge P'=P_1 \uplus P) \\
 & & \quad \rightarrow p\ (M_1,R_1,P_1) \rightarrow q\ (M',R',P') \\
p \mathbin{\mathbb{M}} q & \triangleq & \lambda S.(p\ S) \wedge (q\ S) \\
p \mathbin{\mathbb{W}} q & \triangleq & \lambda S.(p\ S) \vee (q\ S) \\
\exists v.p & \triangleq & \lambda S.\exists v.p\ S \\
\sharp p & \triangleq & \lambda(M,R,P).p \wedge M=\{\cdot\} \wedge R=\{\cdot\} \wedge P=\{\cdot\} \\
\diamond p & \triangleq & \lambda(M,R,P).p\ (M,R,P) \wedge R=\{\cdot\} \\
\mathtt{r} \mapsto \mathtt{w} & \triangleq & \lambda(M,R,P).R=\{\mathtt{r}:\mathtt{w}\} \wedge M=\{\cdot\} \wedge P=\{\cdot\} \\
\mathtt{r} \hookrightarrow \mathtt{w} & \triangleq & \lambda(M,R,P).\exists R'.R=\{\mathtt{r}:\mathtt{w}\} \uplus R' \\
\mathtt{l} \mapsto \mathtt{w} & \triangleq & \lambda(M,R,P).M=\{\mathtt{l}:\mathtt{w}\} \wedge \mathtt{l} \neq \mathsf{NULL} \wedge R=\{\cdot\} \wedge P=\{\cdot\} \\
[t] & \triangleq & \lambda(M,R,P).P=\{t:run\} \wedge t \neq \mathsf{NULL} \wedge M=\{\cdot\} \wedge R=\{\cdot\} \\
\langle t \rangle & \triangleq & \lambda(M,R,P).P=\{t:(rdy,\_)\} \wedge t \neq \mathsf{NULL} \wedge M=\{\cdot\} \wedge R=\{\cdot\}
\end{array}
$$

Fig. 6. Definition of selected assertion constructs

$$
\left\{ \begin{array}{l} p \\ q \end{array} \right\}^{(v_1,\ldots,v_n)} \triangleq (\lambda S.\exists v_1,\ldots,v_n.(p(v_1,\ldots,v_n) * \mathsf{true})\ S,\ \lambda S,S'.\forall p'.\forall v_1,\ldots,v_n.(p(v_1,\ldots,v_n) * p')\ S \rightarrow (q(v_1,\ldots,v_n) * p')\ S')
$$

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. We define a binary operator for composing two pairs into one.

$$
(p,g) \triangleright (p',g') \triangleq (\lambda S.p\ S \wedge (\forall S'.g\ S\ S' \rightarrow p'\ S'),\ \lambda S,S''.p\ S \rightarrow (\exists S'.g\ S\ S' \wedge g'\ S'\ S''))
$$

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:

$$
(p,g) \Rightarrow (p',g') \triangleq \forall S.p\ S \rightarrow p'\ S \wedge (\forall S'.g'\ S\ S' \rightarrow g\ S\ S')
$$

i.e., the precondition p be stronger than p' and the action g be weaker than g'.

$$
\begin{array}{llcl}
\text{(Assert)} & p,q & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{emp} \mid p * q \mid p -\!\!* q \mid p \mathbin{\mathbb{M}} q \mid p \mathbin{\mathbb{W}} q \mid \exists v.p \mid \mathtt{l} \mapsto \mathtt{w} \\
 & & \mid & [t] \mid \langle t \rangle \mid \sharp p \mid \diamond p \mid \mathtt{r} \mapsto \mathtt{w} \mid \mathtt{r} \hookrightarrow \mathtt{w} \\
\text{(Action)} & g & \in & \mathsf{State} \rightarrow \mathsf{State} \rightarrow \mathsf{Prop} \\
\text{(Spec)} & \Psi & ::= & \{\mathtt{f}:(p,g)\}^*
\end{array}
$$

4.2 Invariant for shared resources and inference rules

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.

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 to verify various programs, as long as the instantiation satisfies the requirement of being *precise* [17].

Precisely, the invariant $I(t_s, t_d)$ describes the shared resources when the context switch is 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 depend on the source and the destination threads of a context switch. This is quite different from concurrent code at user-level, where a context switch is non-deterministic and the scheduling algorithm is abstracted away.

The judgment for instructions in our proof system is of the following form: $\Psi, I \vdash \{(p,g)\}\ \texttt{pc} : \texttt{c}$, where Ψ and I are given as specifications. The judgement states that an instruction sequence, started with `c` at the label of `pc` and ended with a `ret`, satisfies specification (p,g) under Ψ and I. Some selected inference rules for instructions are shown in Fig. 7.

In the rule of (ADD), the premise says that the specification (p,g) implies the action of the `add` instruction composed with the specification of the next instruction, $\Psi(\texttt{pc}+1)$. The action of `add` instruction is that if the destination register $\texttt{r}_d$ contains the value of w_1, and the source register $\texttt{r}_s$ contains the value of w_2, then after the instruction, $\texttt{r}_d$ will contain the sum of w_1 and w_2, while $\texttt{r}_s$ will remain unchanged.

Functions are reasoned with the rules of (CALL) and (RET). The (CALL) rule says that the specification (p,g) implies the action that is composed by (1) the action of instruction `call`, (2) the specification of the *function* invoked $\Psi(\texttt{f})$, (3) the action of instruction `ret`, and (4) the specification of the next instruction $\Psi(\texttt{pc}+1)$. The (RET) rule says that the specification (p,g) implies an empty action, which means the actions of the current function should be fulfilled.

The most important rule is (CSW). The precondition of `cswitch` requires the following resources: the current thread resource, the registers `a0` containing the current thread ID t and `a1` containing the destination thread ID t', and the shared resource satisfying the invariant $\diamond I(t,t')$. After return from context switch, the current thread will own the shared resources (satisfying $\diamond I(t'',t)$ for some t'') again.

4.3 Invariant of global resources and soundness

Each abstract thread corresponds to the part of global resources representing the concrete resources allocated for this thread. For example, for an abstract thread $\langle t \rangle$, there

$$\frac{(p,g) \Rightarrow \left\{\begin{array}{l}(\mathtt{r}_d \mapsto w1) * (\mathtt{r}_s \mapsto w2)\\ (\mathtt{r}_d \mapsto w1{+}w2) * (\mathtt{r}_s \mapsto w_2)\end{array}\right\}^{(w1,w2)} \triangleright \Psi(\mathtt{pc}{+}1)}{\Psi, I \vdash \{(p,g)\}\ \mathtt{pc} : \mathtt{add}\ \mathtt{r}_d,\ \mathtt{r}_s}\ (\textsc{add})$$

$$\frac{(p,g) \Rightarrow \left\{\begin{array}{l}\mathtt{ra} \mapsto \_\\ \mathtt{ra} \mapsto \mathtt{pc}{+}1\end{array}\right\} \triangleright \Psi(\mathtt{f}) \triangleright \left\{\begin{array}{l}\mathtt{ra} \mapsto \mathtt{pc}{+}1\\ \mathtt{ra} \mapsto \_\end{array}\right\} \triangleright \Psi(\mathtt{pc}{+}1)}{\Psi, I \vdash \{(p,g)\}\ \mathtt{pc} : \mathtt{call}\ \mathtt{f}}\ (\textsc{call})$$

$$\frac{(p,g) \Rightarrow \left\{\begin{array}{l}\mathsf{emp}\\ \mathsf{emp}\end{array}\right\}}{\Psi, I \vdash \{(p,g)\}\ \mathtt{pc} : \mathtt{ret}}\ (\textsc{ret}) \qquad \frac{(p,g) \Rightarrow \Psi(\mathtt{f})}{\Psi, I \vdash \{(p,g)\}\ \mathtt{pc} : \mathtt{jmp}\ \mathtt{f}}\ (\textsc{jmp})$$

$$\frac{(p,g) \Rightarrow \left\{\begin{array}{l}[t] * (\mathtt{a0}, \mathtt{a1}, \mathtt{ra} \mapsto t, t', \_) * \langle t' \rangle * \diamond I(t,t')\\ [t] * (\mathtt{a0}, \mathtt{a1}, \mathtt{ra} \mapsto t, t', \_) * \exists t''.\, \langle t'' \rangle * \diamond I(t'',t)\end{array}\right\}^{(t,t')} \triangleright \Psi(\mathtt{pc}{+}1)}{\Psi, I \vdash \{(p,g)\}\ \mathtt{pc} : \mathtt{cswitch}}\ (\textsc{csw})$$

Fig. 7. Inference rules (selected)

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 GINV, defined in Fig. 8, describes the partition of all resources globally. The invariant is the key for proving the soundness theorem of our proof system.

First, for each thread, we define a predicate 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 is required to 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 Ψ at pc, such that the resources of the thread satisfies p; and g guarantees that the thread will continue to satisfy Cont recursively after it returns to the address *retaddr*.

The concrete resources of a *running thread* are specified by a continuation 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.

For a *ready thread* (or a runnable thread), its concrete resources are defined by separating implication $-\!\!*$: if given (1) the resources of saved machine context $\lfloor R \rfloor$, (2) the abstract resource of itself $[t]$, (3) another ready thread t' and (4) shared resources specified by $\diamond I(t',t)$, the resources of the ready thread can be transformed into the resources of a running thread. Its thread ID is specified by the second parameter of 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.

The whole machine state can be partitioned, and each part is owned by one thread, which is either running or ready. Thus, the global invariant GINV is defined in the form of separating conjunction by CThrd and RThrd. The structure of GINV is isomorphic to the thread pool P: the abstract running thread is mapped to the resource specified by

$$
\begin{array}{rcl}
\lfloor R \rfloor & \triangleq & (\mathtt{ra} \mapsto \_) * (\mathtt{v0} \mapsto R(\mathtt{v0})) * (\mathtt{sp} \mapsto R(\mathtt{sp})) \\
& & \quad * (\mathtt{a0} \mapsto R(\mathtt{a0})) * (\mathtt{a1} \mapsto R(\mathtt{a1})) * (\mathtt{a2} \mapsto R(\mathtt{a2})) \\
\mathsf{Cont}(n{+}1, \Psi, \mathtt{pc}) & \triangleq & \lambda S.\, \Psi(\mathtt{pc}) = (p, g) \wedge (p\ S) \\
& & \wedge (\forall S'.\, g\ S\ S' \rightarrow (\exists\, \mathit{retaddr}.\, (\mathtt{ra} \hookrightarrow \mathit{retaddr}) \mathbin{-\!\!\!*\!\!\!\wedge} \mathsf{Cont}(n, \Psi, \mathit{retaddr}))\ S') \\
\mathsf{Cont}(0, \Psi, \mathtt{pc}) & \triangleq & \lambda S.\, \Psi(\mathtt{pc}) = (p, g) \wedge (p\ S) \wedge (\forall S'.\, g\ S\ S' \rightarrow \mathsf{False}) \\
\mathsf{CThrd}(\Psi, t, \mathtt{pc}) & \triangleq & \exists n.\, \mathsf{Cont}(n, \Psi, \mathtt{pc}) \mathbin{-\!\!\!*\!\!\!\wedge} ([t] * \exists R.\, \lfloor R \rfloor * \mathsf{true}) \\
\mathsf{RThrd}(\Psi, t, R) & \triangleq & \lfloor R \rfloor * [t] * \exists t'.\, \langle t' \rangle * \diamond I(t', t) \mathbin{-\!\!*} \mathsf{CThrd}(\Psi, t, R(\mathtt{ra})) \\
\mathsf{GINV}(\Psi, P, \mathtt{pc}) & \triangleq & \mathsf{CThrd}(\Psi, t, \mathtt{pc}) * \mathsf{RThrd}(\Psi, t_0, R_0) * \cdots * \mathsf{RThrd}(\Psi, t_n, R_n) \\
& & \text{where } P = \{t : \mathit{run},\ t_0 : (\mathit{rdy}, R_0),\ \ldots,\ t_n : (\mathit{rdy}, R_n)\}
\end{array}
$$

Fig. 8. Concrete threads and the global invariant

```
struct tcb {           | void schedule_p2()
  struct context ctxt; |  {
  struct tcb *prev;    |    struct tcb *old, *new;
  struct tcb *next;    |    old = cur;
};                     |    new = deq(&rq);
struct queue {         |    if (new == NULL) return;
  struct tcb *head;    |    enq(&rq, old);
  struct tcb *tail;    |    cur = new;
};                     |    cswitch(old,new);
struct tcb *cur;       |    return;
struct queue rq;       | }
```

Fig. 9. Pseudo C code for `schedule_p2()`

CThrd; an abstract ready thread is mapped to a resource specified by 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, ($\mathsf{GINV}(\Psi, P, \mathtt{pc})\ (M, R, P)$).

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 [7].

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. We give the code written in pseudo C to explain

the programs and their specifications. The corresponding assembly code and selected assertions of the two schedulers are shown in Fig. 10.

Scheduler as function. The scheduler function `schedule_p2()` (see Fig. 9) 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 to choose a candidate from the thread queue and then perform context switch from the current thread to the candidate. There are two global variables, `cur` and `rq`. The variable `cur` points to the TCB of the running thread; `rq` points to the thread queue containing TCBs of all other runnable (ready) threads.

The notation $t \overset{field}{\longmapsto} w$ specifies a named field in the structure. The notation $\mathsf{ptcb}(t)$ specifies a part of TCB including the fields of `next` and `prev`. The predicate $\mathsf{RQ}(q,L)$ specifies a doubly linked list as a thread queue pointed to by q, where L is a list of thread IDs of the thread queue. We also use $\langle L\rangle$ as an abbreviation for $\langle t_0\rangle * \langle t_1\rangle * \cdots * \langle t_n\rangle$, if L is $t_0 :: t_1 :: \cdots :: t_n :: \mathsf{nil}$, and use $\mathtt{l} \mapsto {}^{(n)}\_$ to specify n continuous memory cells.

$$
\begin{array}{rcl}
\mathtt{l} \overset{field}{\longmapsto} w & \triangleq & (l + \textit{offset of the field in the struct}) \mapsto w \\
\mathsf{ptcb}(t) & \triangleq & (t \overset{prev}{\longmapsto} \_) * (t \overset{next}{\longmapsto} \_) \\
\mathsf{RQseg}(pv, tl, t, \mathsf{nil}) & \triangleq & (t \overset{prev}{\longmapsto} pv) * \exists t' . (t \overset{next}{\longmapsto} \mathsf{NULL}) * \sharp(t = tl) \\
\mathsf{RQseg}(pv, tl, t, t' :: L') & \triangleq & (t \overset{prev}{\longmapsto} pv) * (t \overset{next}{\longmapsto} t') * \mathsf{RQseg}(t, tl, t', L') \\
\mathsf{RQ}(q, \mathsf{nil}) & \triangleq & (q \overset{head}{\longmapsto} \mathsf{NULL}) * (q \overset{tail}{\longmapsto} \mathsf{NULL}) \\
\mathsf{RQ}(q, t :: L) & \triangleq & \exists pv . \exists tl . (q \overset{head}{\longmapsto} t) * (q \overset{tail}{\longmapsto} tl) * \mathsf{RQseg}(pv, tl, t, L) \\
\mathsf{K}(bp, n, \mathtt{w}_0 :: \mathtt{w}_1 :: \ldots :: \mathtt{w}_m :: \mathsf{nil}) & \triangleq & \exists sp . (\mathtt{sp} \mapsto sp) * \sharp(sp = bp + 4n) * (bp \mapsto {}^{(n)}\_) \\
 & & * (sp \mapsto \mathtt{w}_0) * (sp+4 \mapsto \mathtt{w}_1) * \cdots * (sp+4m \mapsto \mathtt{w}_m) \\
\mathsf{K}(bp, n) & \triangleq & \mathsf{K}(bp, n, \mathsf{nil})
\end{array}
$$

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

$$
\left\{
\begin{array}{l}
[t] * \mathsf{ptcb}(t) * (\mathtt{cur} \mapsto t) * \exists L . \mathsf{RQ}(\mathtt{rq}, L) * \langle L\rangle * (\mathtt{ra} \mapsto ret) \\
\qquad\qquad * \mathsf{K}(bp, 20) * (\mathtt{v0}, \mathtt{a0}, \mathtt{a1} \mapsto \_, \_, \_) \\
[t] * \mathsf{ptcb}(t) * (\mathtt{cur} \mapsto t) * \exists L . \mathsf{RQ}(\mathtt{rq}, L) * \langle L\rangle * (\mathtt{ra} \mapsto ret) \\
\qquad\qquad * \mathsf{K}(bp, 20) * (\mathtt{v0}, \mathtt{a0}, \mathtt{a1} \mapsto \_, \_, \_)
\end{array}
\right\}^{(t, ret, bp)}
$$

Here we use a notation $\mathsf{K}(bp, n, w :: w' :: \cdots)$ 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). 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 abstract invariant I is instantiated to a concrete definition specifying the shared resources *before* and *after* context switch for this implementation of scheduler.

$$
I(t, t') \triangleq \mathsf{ptcb}(t') * (\mathtt{cur} \mapsto t') * \exists L . \mathsf{RQ}(\mathtt{rq}, t :: L) * \langle L\rangle
$$

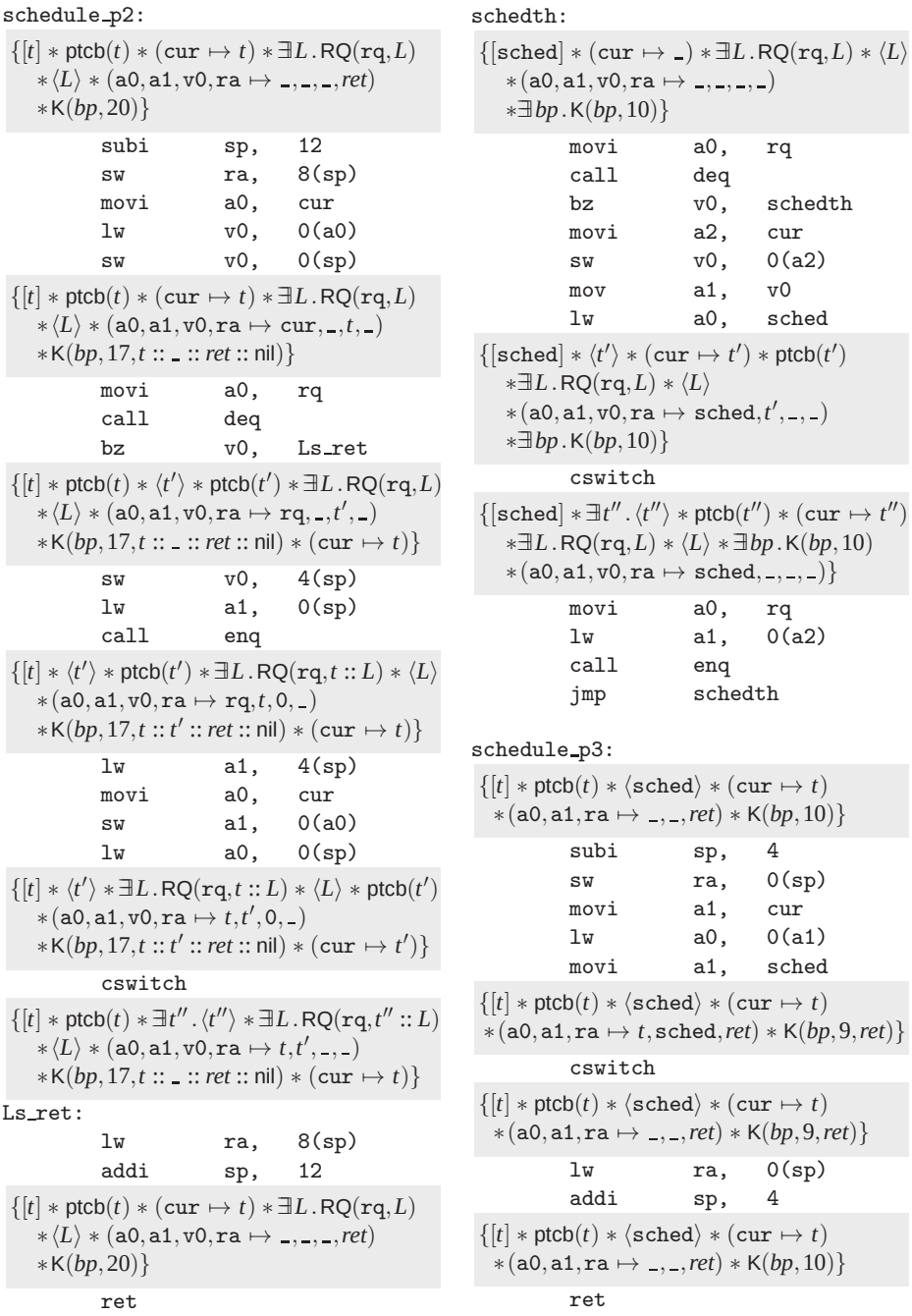

Fig. 10. Verification of `schedule_p2()`, `schedth()` and `schedule_p3()`

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

Fig. 11. Pseudo C code for `schedule_p3()`

Scheduler as a separated thread. A scheduler in the pattern (III) is implemented as a separated thread (see Fig. 11), which does scheduling jobs in an infinite loop. A global variable `sched` is added to represent the TCB of the scheduler thread. A stub function `schedule_p3()` can be invoked by other threads to do scheduling. As shown below, the specification of `schedule_p3()` function is different from the one of `schedule_p2()`. The schedule function in this implementation doesn't own the thread queue, which is owned by the scheduler thread $\langle \texttt{sched} \rangle$ instead since all of the operations over the thread queue are put into the separated thread.

$$\left\{ \begin{array}{l} [t] * \mathsf{ptcb}(t) * (\texttt{cur} \mapsto t) * \langle \texttt{sched} \rangle * (\texttt{a0}, \texttt{a1}, \texttt{ra} \mapsto \_, \_, \mathit{ret}) * \mathsf{K}(\mathit{bp}, 10) \\ [t] * \mathsf{ptcb}(t) * (\texttt{cur} \mapsto t) * \langle \texttt{sched} \rangle * (\texttt{a0}, \texttt{a1}, \texttt{ra} \mapsto \_, \_, \mathit{ret}) * \mathsf{K}(\mathit{bp}, 10) \end{array} \right\}^{(t, \mathit{bp}, \mathit{ret})}$$

The specification of `schedth()` function is shown below:

$$\left\{ \begin{array}{l} [\texttt{sched}] * (\texttt{cur} \mapsto \_) * \exists L \,.\, \mathsf{RQ}(\texttt{rq}, L) * \langle L \rangle \\ \qquad\qquad * (\texttt{a0}, \texttt{a1}, \texttt{a2}, \texttt{v0}, \texttt{ra} \mapsto \_, \_, \_, \_, \_) * \exists \mathit{bp} \,.\, \mathsf{K}(\mathit{bp}, 10) \\ \mathsf{false} \end{array} \right\}$$

Since the ready thread queue is only owned by the scheduler thread, it does not need to be shared by other threads and occur in the invariant for the shared resources, I:

$$I(t, t') \triangleq (\sharp(t' = \texttt{sched}) * (\texttt{cur} \mapsto t) * \mathsf{ptcb}(t)) \curlyvee (\sharp(t = \texttt{sched}) * (\texttt{cur} \mapsto t') * \mathsf{ptcb}(t'))$$

The invariant $I(t, t')$ is defined by two cases 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 Related work and conclusions

Gotsman and Yang [6] 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 $\mathsf{RThrd}(\cdots)$, 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 $\langle t \rangle$ 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 [15,14] supporting 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 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 [11], 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 [5] 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 [18] to reason about context switch between user processes. The context switch code and proofs are integrated in a framework for building verified kernels (CVM) [10]. 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.

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 to be practically applied to realistic OS code.

Acknowledgements. We thank anonymous referees for suggestions and comments on an earlier version of this paper. Yu Guo, Xinyu Feng and Peizhi Shi are supported in part by grants from National Natural Science Foundation of China (Nos. 61073040, 61202052 and 61229201), the Fundamental Research Funds for the Central Universities (Nos. WK0110000018 and WK0110000025), and Program for New Century Excellent Talents in Universities (NCET). Zhong Shao is supported in part by DARPA under agreement numbers FA8750-10-2-0254 and FA8750-12-2-0293, and by NSF grants CNS-0910670, CNS-0915888, and CNS-1065451. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. R. S. Engelschall. Portable multithreading: the signal stack trick for user-space thread creation. In *Proc. of ATEC'00*, pages 20–20, Berkeley, CA, USA, 2000. USENIX Association.
2. D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.
3. X. Feng, Z. Shao, Y. Guo, and Y. Dong. Combining domain-specific and foundational logics to verify complete software systems. In *Proc. VSTTE'08*, pages 54–69, Toronto, Canada, October 2008.
4. X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *Proc. PLDI'06*, pages 401–414, June 2006.
5. M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In J. Hurd and T. F. Melham, editors, *Proc. TPHOLs'05*, volume 3603 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.
6. A. Gotsman and H. Yang. Modular verification of preemptive os kernels. In *Proc. ICFP'11*, pages 404–417, Tokyo, Japan, 2011. ACM.
7. Y. Guo, X. Feng, Z. Shao, and P. Shi. Modular verification of concurrent thread management (technical report and coq proof). `http://kyhcs.ustcsz.edu.cn/~guoyu/sched/`, June 2012.
8. J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Minix 3: a highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 40:80–89, July 2006.
9. M. Hohmuth and H. Tews. The vfiasco approach for a verified operating system. In *Proceedings of the 2nd ECOOP Workshop on Programming Languages and Operating Systems*, 2005.
10. T. In der Rieden and A. Tsyban. CVM – A verified framework for microkernel programmers. In *Proc. SSV'08*, volume 217C of *Electronic Notes in Theoretical Computer Science*, pages 151–168. Elsevier Science B.V., 2008.
11. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. SOSP'09*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.
12. R. Love. *Linux Kernel Development (2nd Edition) (Novell Press)*. Novell Press, 2005.
13. M. K. McKusick and G. V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004.
14. Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. POPL'06*, pages 320–333, Jan. 2006.
15. Z. Ni, D. Yu, and Z. Shao. Using XCAP to certify realistic systems code: Machine context management. In *Proc. TPHOLs'07*, volume 4732 of *Lecture Notes in Computer Science*, pages 189–206. Springer-Verlag, September 2007.
16. P. W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
17. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.
18. A. Starostin and A. Tsyban. Verified process-context switch for C-programmed kernels. In J. Woodcock and N. Shankar, editors, *Proc. VSTTE'08*, volume 5295 of *Lecture Notes in Computer Science*, pages 240–254, Toronto, Canada, Oct. 2008. Springer.