Modular Verification of Concurrent Thread Management (Extended)

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
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

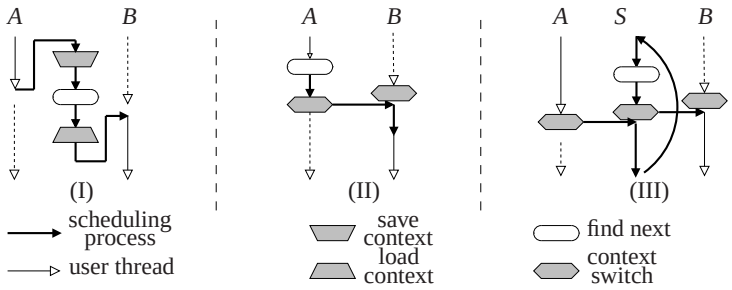

Fig. 1. Three patterns of scheduling

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

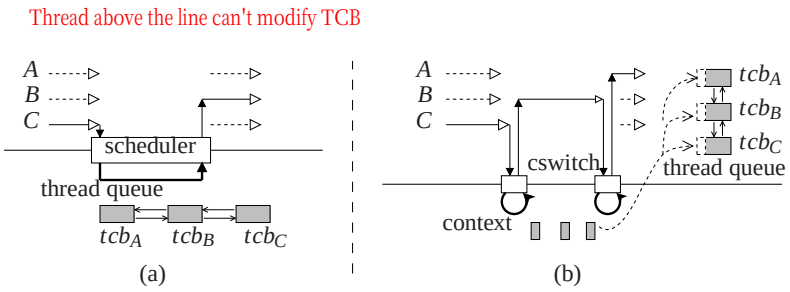

Fig. 2. Abstraction in verification framework

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 [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.

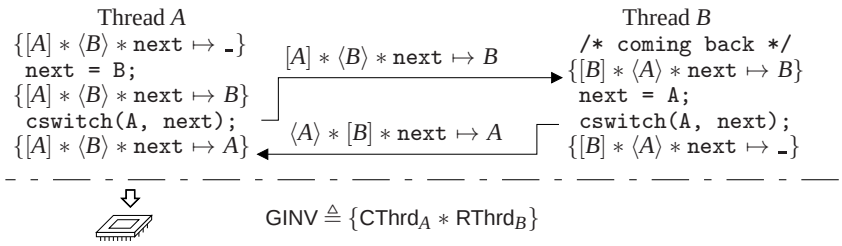

Fig. 3. Abstract thread res. vs. concrete thread res.

(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 [3,7] 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 `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 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 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.

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

abstract machine (logical abstract threads)

physical machine (no concept of thread)

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

no concept of thread

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

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.

$$
\begin{array}{rcl}
\text{Term } A, B ::= & \mathsf{Set} \mid \mathsf{Prop} \mid \mathsf{Type} & \\
\mid & X \mid \lambda X : A.B \mid A\,B \mid A \to B \mid A \wedge B \mid A \vee B \mid \mathsf{True} \mid \mathsf{False} & \\
\mid & \forall X : A.B \mid \exists X : A.B \mid \textit{inductive def.} \mid \cdots &
\end{array}
$$

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 Prop. In addition, Prop is the universe of all propositions, Set is the universe of all data sets, and 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 $\mathbb{W}$ consists of a code block $\mathbb{C}$, a mutable memory $\mathbb{M}$, a register file $\mathbb{R}$, 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 $\mathbb{C}$ 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 bits-arithmetic.

For all kinds of partial mappings, we use $\uplus$ to denote the disjoint union of two mappings, and use the notation like $R\{\mathtt{r} : \mathtt{w}\}$ to denote the updating of a mapping .

We use a relation $\mathbb{W} \longmapsto \mathbb{W}'$ to specify the operational semantics.

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

Fig. 5. Abstract machine model

$$
\frac{\mathbb{C}(\mathtt{pc}) = \mathtt{i} \qquad (\mathbb{M}, \mathbb{R}, \mathtt{pc}) \overset{\mathtt{i}}{\hookrightarrow} (\mathbb{M}', \mathbb{R}', \mathtt{pc}')}{(\mathbb{C}, \mathbb{M}, \mathbb{R}, \mathtt{pc}) \longmapsto (\mathbb{C}, \mathbb{M}', \mathbb{R}', \mathtt{pc}')}
$$

where the state transition relation $(\mathbb{M}, \mathbb{R}, \mathtt{pc}) \overset{\mathtt{i}}{\hookrightarrow} (\mathbb{M}', \mathbb{R}', \mathtt{pc}')$ 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 (`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. 8. In the physical machine, this abstract instruction can be implemented using normal machine instructions.

```
cswitch:
  sw ra,   0(a0)    |   lw ra, 0(a1)
  sw v0,   4(a0)    |   lw v0, 4(a1)
  sw a0,   8(a0)    |   lw a0, 8(a1)
  sw a1,  12(a0)    |   lw a2, 16(a1)
  sw a2,  16(a0)    |   lw sp, 20(a1)
  sw sp,  20(a0)    |   lw a1, 12(a1)
                    |   ret
```

The machine context, a register file, can be defined as the structure as below:

$(\mathbb{M},\mathbb{R},\mathtt{pc}) \overset{\mathtt{i}}{\hookrightarrow} (\mathbb{M}',\mathbb{R}',\mathtt{pc}')$	
if $\mathtt{i} =$	then
$\mathtt{add}\ \mathtt{r}_d,\ \mathtt{r}_s$	$\mathbb{M}'=\mathbb{M} \wedge \mathbb{R}'=\mathbb{R}\{\mathtt{r}_d : \mathbb{R}(\mathtt{r}_d)+\mathbb{R}(\mathtt{r}_s)\} \wedge \mathtt{pc}'=\mathtt{pc}+1$
$\mathtt{addi}\ \mathtt{r}_d,\ \mathtt{w}$	$\mathbb{M}'=\mathbb{M} \wedge \mathbb{R}'=\mathbb{R}\{\mathtt{r}_d : \mathbb{R}(\mathtt{r}_d)+\mathtt{w}\} \wedge \mathtt{pc}'=\mathtt{pc}+1$
$\mathtt{sub}\ \mathtt{r}_d,\ \mathtt{r}_s$	$\mathbb{M}'=\mathbb{M} \wedge \mathbb{R}'=\mathbb{R}\{\mathtt{r}_d : \mathbb{R}(\mathtt{r}_d)-\mathbb{R}(\mathtt{r}_s)\} \wedge \mathtt{pc}'=\mathtt{pc}+1$
$\mathtt{subi}\ \mathtt{r}_d,\ \mathtt{w}$	$\mathbb{M}'=\mathbb{M} \wedge \mathbb{R}'=\mathbb{R}\{\mathtt{r}_d : \mathbb{R}(\mathtt{r}_d)-\mathtt{w}\} \wedge \mathtt{pc}'=\mathtt{pc}+1$
$\mathtt{mov}\ \mathtt{r}_d,\ \mathtt{r}_s$	$\mathbb{M}'=\mathbb{M} \wedge \mathbb{R}'=\mathbb{R}\{\mathtt{r}_d : \mathbb{R}(\mathtt{r}_s)\} \wedge \mathtt{pc}'=\mathtt{pc}+1$
$\mathtt{movi}\ \mathtt{r}_d,\ \mathtt{w}$	$\mathbb{M}'=\mathbb{M} \wedge \mathbb{R}'=\mathbb{R}\{\mathtt{r}_d : \mathtt{w}\} \wedge \mathtt{pc}'=\mathtt{pc}+1$
$\mathtt{lw}\ \mathtt{r}_t,\ \mathtt{w}(\mathtt{r}_s)$	$\mathbb{M}'=\mathbb{M} \wedge \mathbb{R}'=\mathbb{R}\{\mathtt{r}_t : \mathbb{M}(\mathbb{R}(\mathtt{r}_s)+\mathtt{w})\} \wedge \mathtt{pc}'=\mathtt{pc}+1$
$\mathtt{sw}\ \mathtt{r}_t,\ \mathtt{w}(\mathtt{r}_s)$	$\mathbb{M}'=\mathbb{M}\{(\mathbb{R}(\mathtt{r}_s)+\mathtt{w}) : \mathbb{R}(\mathtt{r}_t)\} \wedge \mathbb{R}'=\mathbb{R} \wedge \mathtt{pc}'=\mathtt{pc}+1$
$\mathtt{call}\ \mathtt{f}$	$\mathbb{M}'=\mathbb{M} \wedge \mathbb{R}'=\mathbb{R}\{\mathtt{ra} : \mathtt{pc}+1\} \wedge \mathtt{pc}'=\mathtt{f}$
$\mathtt{jmp}\ \mathtt{f}$	$\mathbb{M}'=\mathbb{M} \wedge \mathbb{R}'=\mathbb{R} \wedge \mathtt{pc}'=\mathtt{f}$
$\mathtt{ret}$	$\mathbb{M}'=\mathbb{M} \wedge \mathbb{R}'=\mathbb{R} \wedge \mathtt{pc}'=\mathbb{R}(\mathtt{ra})$
$\mathtt{bz}\ \mathtt{r}_t,\ \mathtt{f}$	$\mathbb{M}'=\mathbb{M} \wedge \mathbb{R}'=\mathbb{R} \wedge \mathtt{pc}'=\mathtt{f}$ if $\mathbb{R}(\mathtt{r}_t)=0$ $\mathbb{M}'=\mathbb{M} \wedge \mathbb{R}'=\mathbb{R} \wedge \mathtt{pc}'=\mathtt{pc}+1$ if $\mathbb{R}(\mathtt{r}_t)\neq 0$

Fig. 6. Operational semantics of physical machine

```
struct context {
  int   ra;   int   v0;
  int   a0;   int   a1;
  int   a2;   int   sp;
};
```

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.

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 , $W \Downarrow \mathbb{W}$ 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, $\mathbb{C}_{cs}$. A thread pool P is translated into a block of memory with the context data of all threads, each of which is specified by $mc(t,\mathbb{R})$. 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:

(AbsMach)	W	::=	$(C, S, \mathtt{pc})$
(State)	S	::=	(M, R, P)
(AbsCode)	C	::=	$\{\mathtt{f} : \mathtt{c}\}^*$
(Mem)	M	::=	$\{\mathtt{l} : \mathtt{w}\}^*$
(RegFile)	R	::=	$\{\mathtt{r} : \mathtt{w}\}^*$
(TID)	t	::=	$\mathtt{w}$
(Pool)	P	::=	$\{t : T\}^*$
(Thrd)	T	::=	$run \mid (rdy, R)$
(AbsInstr)	$\mathtt{c}$	::=	$\mathtt{cswitch} \mid \mathtt{i}$

Fig. 7. Definition of abstract machine

$((M,R,P),\mathtt{pc}) \overset{\mathtt{c}}{\hookrightarrow} ((M',R',P'),\mathtt{pc}')$	
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
$\mathtt{add}\ \mathtt{r}_d,\ \mathtt{r}_s$	$M' = M \wedge R' = R\{\mathtt{r}_d : R(\mathtt{r}_d) + R(\mathtt{r}_s)\} \wedge \mathtt{pc}' = \mathtt{pc}+1$
$\mathtt{addi}\ \mathtt{r}_d,\ \mathtt{w}$	$M' = M \wedge R' = R\{\mathtt{r}_d : R(\mathtt{r}_d) + \mathtt{w}\} \wedge \mathtt{pc}' = \mathtt{pc}+1$
$\mathtt{mov}\ \mathtt{r}_d,\ \mathtt{r}_s$	$M' = M \wedge R' = R\{\mathtt{r}_d : R(\mathtt{r}_s)\} \wedge \mathtt{pc}' = \mathtt{pc}+1$
$\mathtt{movi}\ \mathtt{r}_d,\ \mathtt{w}$	$M' = M \wedge R' = R\{R(\mathtt{r}_d) : \mathtt{w}\} \wedge \mathtt{pc}' = \mathtt{pc}+1$
$\mathtt{lw}\ \mathtt{r}_t,\ \mathtt{w}(\mathtt{r}_s)$	$M' = M \wedge R' = R\{\mathtt{r}_t : M(R(\mathtt{r}_s)+\mathtt{w})\} \wedge \mathtt{pc}' = \mathtt{pc}+1$
$\mathtt{sw}\ \mathtt{r}_t,\ \mathtt{w}(\mathtt{r}_s)$	$M' = M\{(R(\mathtt{r}_s)+\mathtt{w}) : R(\mathtt{r}_t)\} \wedge R' = R \wedge \mathtt{pc}' = \mathtt{pc}+1$
$\mathtt{call}\ \mathtt{f}$	$M' = M \wedge R' = R\{\mathtt{ra} : \mathtt{pc}+1\} \wedge \mathtt{pc}' = \mathtt{f}$
$\mathtt{jmp}\ \mathtt{f}$	$M' = M \wedge R' = R \wedge \mathtt{pc}' = \mathtt{f}$
$\mathtt{ret}$	$M' = M \wedge R' = R \wedge \mathtt{pc}' = R(\mathtt{ra})$
$\mathtt{bz}\ \mathtt{r}_t,\ \mathtt{f}$	$M' = M \wedge R' = R \wedge \mathtt{pc}' = \mathtt{f}$ if $R(\mathtt{r}_t) = 0$ $M' = M \wedge R' = R \wedge \mathtt{pc}' = \mathtt{pc}+1$ if $R(\mathtt{r}_t) \neq 0$

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

Fig. 8. Operational semantics of abstract machine (part)

$$\frac{\forall \mathtt{f} \in \mathrm{dom}(C).\, C(\mathtt{f}) \Downarrow \mathbb{C}(\mathtt{f})}{C \Downarrow (\mathbb{C} \uplus \mathbb{C}_{cs})} \qquad \frac{}{\mathtt{i} \Downarrow \mathtt{i}} \qquad \frac{}{\mathtt{cswitch} \Downarrow \mathtt{call}\ \mathtt{f}_{\mathtt{cswitch}}}$$

$$\frac{\forall \mathtt{r}.\, R(\mathtt{r}) = \mathbb{R}(\mathtt{r})}{R \Downarrow \mathbb{R}} \qquad \frac{\forall \mathtt{l} \in \mathrm{dom}(M).\, M(\mathtt{l}) = \mathbb{M}(\mathtt{l})}{M \Downarrow \mathbb{M}}$$

$$\frac{P \Downarrow \mathbb{M} \quad P' \Downarrow \mathbb{M}'}{(P \uplus P') \Downarrow (\mathbb{M} \uplus \mathbb{M}')} \qquad \frac{\mathbb{M} = mc(t, \mathbb{R})}{\{t : (rdy, \mathbb{R})\} \Downarrow \mathbb{M}} \qquad \frac{\exists \mathbb{R}'.\, \mathbb{M} = mc(t, \mathbb{R}')}{\{t : (run)\} \Downarrow \mathbb{M}}$$

$$\frac{C \Downarrow \mathbb{C} \quad M \Downarrow \mathbb{M}_1 \quad P \Downarrow \mathbb{M}_2 \quad R \Downarrow \mathbb{R} \quad \mathbb{M} \Downarrow \mathbb{M}_1 \uplus \mathbb{M}_2}{(C, (M, R, P), \mathtt{pc}) \Downarrow (\mathbb{C}, \mathbb{M}, \mathbb{R}, \mathtt{pc})}$$

where $mc(\mathtt{l}, \mathbb{R}) \triangleq \{\mathtt{l} : \mathbb{R}(\mathtt{ra}), \mathtt{l}{+}4 : \mathbb{R}(\mathtt{v0}), \mathtt{l}{+}8 : \mathbb{R}(\mathtt{a0}),$
$\mathtt{l}{+}12 : \mathbb{R}(\mathtt{a1}), \mathtt{l}{+}16 : \mathbb{R}(\mathtt{a2}), \mathtt{l}{+}20 : \mathbb{R}(\mathtt{sp})\}$

Fig. 9. Relation between abstract machine and physical machine

$$\begin{array}{lll}
Safei(i{+}1, W) & \triangleq & (\exists W'.\, W \longmapsto W') \wedge \forall W'.\, W \longmapsto W' \rightarrow Safei(i, W') \\
Safei(0, W) & \triangleq & \mathsf{True} \\
Safe(W) & \triangleq & \forall i.\, Safei(i, W) \\
Safei(i{+}1, \mathbb{W}) & \triangleq & (\exists \mathbb{W}'.\, \mathbb{W} \longmapsto \mathbb{W}') \wedge \forall W'.\, W \longmapsto \mathbb{W}' \rightarrow Safei(i, \mathbb{W}') \\
Safei(0, \mathbb{W}) & \triangleq & \mathsf{True} \\
Safe(\mathbb{W}) & \triangleq & \forall i.\, Safei(i, \mathbb{W})
\end{array}$$

If a machine configuration is safe, it can run forever without going stuck.

The following lemma states that the operational semantics of physical machine is deterministic. In other words, any execution trace is linear.

Lemma 1 (Deterministic physical machine). *For any physical machine configuration $\mathbb{W}$, if there exists $\mathbb{W}'$ and $\mathbb{W}''$ such that $\mathbb{W} \longmapsto \mathbb{W}'$ and $\mathbb{W} \longmapsto \mathbb{W}''$, then the two machine configurations are $\mathbb{W}' = \mathbb{W}''$.*

Proof. By induction on the transition relation of physical machine. □

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

Lemma 2 (Mapping of context switch). *If $W = (C, S, \mathtt{pc})$, $\mathbb{W}$, $W \Downarrow \mathbb{W}$, and $C(\mathtt{pc}) =$ `cswitch`, if W can run one step to W', then $\mathbb{W}$ can also run n steps ($n > 0$) to some $\mathbb{W}'$ such that $\mathbb{W} \longmapsto^n \mathbb{W}'$ and $W' \Downarrow \mathbb{W}'$.*

Proof. By the operational semantics of the instruction `cswitch` and the implementation of context switch routine. □

Lemma 3 (Mapping of One Step). *For any abstract machine configuration W and physical machine configuration $\mathbb{W}$, if they satisfies the translation relation $W \Downarrow \mathbb{W}$, and for any W' such that $W \longmapsto W'$, then there must exists a $\mathbb{W}'$ that $\mathbb{W} \longmapsto^* \mathbb{W}'$ and $W' \Downarrow \mathbb{W}'$.*

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* $\mathbb{W}$*, if* $W \Downarrow \mathbb{W}$ *and* $Safe(W)$ *then* $Safe(\mathbb{W})$.

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 [18], are *shallowly embedded* in the meta language , as shown in Fig. 10. The notation of emp 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 $p -\!\!* q$ 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 $[t]$ 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 $\mathtt{r} \mapsto \mathtt{w}$ 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] 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 `ret` instruction. For all instructions in C, all (p,g) pairs are put in a global mapping, Ψ, from labels to specifications.

$$\begin{array}{lrl} \text{(Assert)} & p,q & \in \quad \text{State} \to \text{Prop} \\ \text{(Action)} & g & \in \quad \text{State} \to \text{State} \to \text{Prop} \\ \text{(Spec)} & \Psi & ::= \quad \{\mathtt{f} : (p,g)\}^* \end{array}$$

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{emp} & \triangleq & \lambda(M,R,P).\, M=\{\cdot\} \wedge R=\{\cdot\} \wedge P=\{\cdot\} \\
\mathsf{true} & \triangleq & \lambda S.\ \mathsf{True} \\
\mathsf{false} & \triangleq & \lambda S.\ \mathsf{False} \\
p \mathbin{\mathbb{\wedge}} q & \triangleq & \lambda S.\, (p\ S) \wedge (q\ S) \\
p \mathbin{\mathbb{\vee}} q & \triangleq & \lambda S.\, (p\ S) \vee (q\ S) \\
\exists v.\, p & \triangleq & \lambda S.\, \exists v.\, p\ S \\
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 \\
 & & \wedge\, p\ (M_1,R_1,P_1) \wedge q\ (M_2,R_2,P_2) \\
p \mathrel{-\!\!*} 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) \\
 & & \rightarrow p\ (M_1,R_1,P_1) \rightarrow q\ (M',R',P') \\
\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{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\} \\
\mathtt{r} \mapsto \mathtt{w} & \triangleq & \lambda(M,R,P).\, R=\{\mathtt{r}:\mathtt{w}\} \wedge M=\{\cdot\} \wedge P=\{\cdot\} \\
\mathtt{r} \mapsto \_ & \triangleq & \lambda(M,R,P).\, \exists \mathtt{w}.\, 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 \_ & \triangleq & \lambda(M,R,P).\, \exists \mathtt{w}.\, M=\{\mathtt{l}:\mathtt{w}\} \wedge \mathtt{l} \neq \mathsf{NULL} \wedge R=\{\cdot\} \wedge P=\{\cdot\} \\
\mathtt{l} \hookrightarrow \mathtt{w} & \triangleq & \lambda(M,R,P).\, \exists M'.\, M=\{\mathtt{l}:\mathtt{w}\} \uplus M' \wedge \mathtt{l} \neq \mathsf{NULL} \\
[t] & \triangleq & \lambda(M,R,P).\, P=\{t:\mathit{run}\} \wedge M=\{\cdot\} \wedge R=\{\cdot\} \\
\langle t \rangle & \triangleq & \lambda(M,R,P).\, P=\{t:(\mathit{rdy},\_)\} \wedge M=\{\cdot\} \wedge R=\{\cdot\} \\
\mathtt{r}_1,\ldots,\mathtt{r}_n \mapsto \mathtt{w}_1,\ldots,\mathtt{w}_n & \triangleq & (\mathtt{r}_1 \mapsto \mathtt{w}_1) * \cdots * (\mathtt{r}_n \mapsto \mathtt{w}_n) \\
\mathtt{l} \mapsto {}^{(n)}\_ & \triangleq & (\mathtt{l} \mapsto \_) * (\mathtt{l}+4 \mapsto \_) * \cdots * (\mathtt{l}+4(n-1) \mapsto \_)
\end{array}
$$

Fig. 10. Definition of assertion constructs

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

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. For example, the action of an addition instruction `addi a0, 2` can be specified as follows:

$$\left\{\begin{array}{l} \texttt{a0} \mapsto v \\ \texttt{a0} \mapsto v{+}2 \end{array}\right\}^{(v)}$$

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 $\texttt{a0} \mapsto v$, the state after addition will satisfy $\texttt{a0} \mapsto v{+}2$. We define a binary operator for composing two pairs into one.

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

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:

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

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:

$$\left\{\begin{array}{l} \texttt{a0} \mapsto v \\ \texttt{a0} \mapsto v{+}2 \end{array}\right\}^{(v)} \triangleright \left\{\begin{array}{l} \texttt{a0} \mapsto v' \\ \texttt{a0} \mapsto v'{+}3 \end{array}\right\}^{(v')} \Rightarrow \left\{\begin{array}{l} \texttt{a0} \mapsto v \\ \texttt{a0} \mapsto v{+}5 \end{array}\right\}^{(v)}$$

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: $\Psi, I \vdash \{(p,g)\}\ \mathtt{pc} : \mathtt{c}$, where Ψ, I are implementation-specific. 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. The inference rules are shown in Fig. 11.

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

The rules of (ADD), (ADDI), (MOV), (MOVI), (LW), and (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 (ADD), the premise says that the specification (p,g) implies the action of the `add` instruction composed with the specification of the next instruction, $\Psi(\mathtt{pc}+1)$. The action of `add` instruction is that if the destination register $\mathtt{r}_d$ contains the value of w_1, and the source register $\mathtt{r}_s$ contains the value of w_2, then after the instruction, $\mathtt{r}_d$ will contain the sum of w_1 and w_2, while $\mathtt{r}_s$ will keep 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 $\Psi(\mathtt{f})$, (3) the action of instruction `ret`, and (4) the specification of the next instruction $\Psi(\mathtt{pc}+1)$. The (RET) rule says that the specification (p,g) implies an empty action, which means the actions of the current function should be fulfilled. 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, (CSW). It states that the precondition of `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:

– $[t]$: 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;
– $\diamond I(t'', t)$: 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 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 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*.

Running thread. 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.

$$\begin{aligned}\lfloor R \rfloor \triangleq\ & (\mathtt{ra} \mapsto \_) * (\mathtt{v0} \mapsto R(\mathtt{v0})) * (\mathtt{sp} \mapsto R(\mathtt{sp})) \\ & * (\mathtt{a0} \mapsto R(\mathtt{a0})) * (\mathtt{a1} \mapsto R(\mathtt{a1})) * (\mathtt{a2} \mapsto R(\mathtt{a2}))\end{aligned}$$

Ready thread. For a *ready thread* (or a runnable thread), its concrete resources are defined by separation 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. It 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`.

$$\frac{\forall \mathtt{f} \in \mathrm{dom}(C).\quad \Psi, I \vdash \{\Psi(\mathtt{f})\}\mathtt{f} : C(\mathtt{f})}{\Psi, I \vdash C}\ (\textsc{cdhp})$$

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

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

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

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

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

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

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

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

$$\frac{(p \between (r_t \hookrightarrow 0), g) \Rightarrow \Psi(\mathtt{f}) \qquad (p \between (r_t \hookrightarrow \mathtt{w}) * \sharp(\mathtt{w} \neq 0), g) \Rightarrow \Psi(\mathtt{pc}{+}1)}{\Psi, I \vdash \{(p,g)\}\ \mathtt{pc} : \mathtt{bz}\ r_t,\ \mathtt{f}}\ (\textsc{bz})$$

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

Fig. 11. Inference rules selected)

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

Fig. 12. Concrete threads and the global invariant

a logical partition

Invariant 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, $(\mathsf{GINV}(\Psi,P,\mathtt{pc})\ (M,R,P))$.

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 [8].

Lemma 4 (Context switch over shared invariant). *For any machine configuration* $(C,(M,R,P),\mathtt{pc})$*, and kernel specification* Ψ*,* $\Psi,I,I \vdash C$*, state satisfies the global invariant*

$$(M,R,P) \Vdash \lfloor R \rfloor * [t] * \langle t' \rangle * I(t,t') * p'$$

then after machine run a context switch $(M,R,P) \overset{\mathtt{cswitch}}{\hookrightarrow} (M',R',P')$*, the machine state will satisfy*

$$(M',R',P') \Vdash \lfloor R' \rfloor * [t'] * I(t') * p',$$

where $R(\mathtt{a0}) = t$ *and* $R(\mathtt{a1}) = t'$.

Proof. Immediate. □

Lemma 5 (Context switch over global invariant). *For any machine configuration* $(C,(M,R,P),\mathtt{pc})$*, and kernel specification* Ψ*,* $\Psi,I,I \vdash C$*, state satisfies the global invariant*

$$(M,R,P) \Vdash \mathsf{GINV}(\Psi,P,\mathtt{pc}),$$

if machine can run a context switch command, $(M,R,P) \overset{\mathtt{cswitch}}{\hookrightarrow} (M',R',P')$ *then the state after the command still satisfies the global invariant:*

$$(M',R',P') \Vdash \mathsf{GINV}(\Psi,P',\mathtt{pc}).$$

Proof. Immediate. □

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

Proof. □

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

Proof. □

Theorem 4 (Soundness). *For any machine configuration ,*$(C,(M,R,P),\mathtt{pc})$*, if its code is verified,* $\Psi,I \vdash C$*, and its state satisfies the global invariant,* $(M,R,P) \Vdash \mathsf{GINV}(\Psi,P,\mathtt{pc})$*, 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.

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

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

$$t \overset{prev}{\longmapsto} p \triangleq t+\texttt{OFF\_PREV} \mapsto p$$
$$t \overset{next}{\longmapsto} q \triangleq t+\texttt{OFF\_NEXT} \mapsto q$$
$$\mathsf{ptcb}(t) \triangleq (t \overset{prev}{\longmapsto} \_) * (t \overset{next}{\longmapsto} \_)$$

The notation $t \overset{next}{\longmapsto} p$ and $t \overset{prev}{\longmapsto} q$ specify the memory cell of two pointers of the TCB. The notation $\mathsf{ptcb}(t)$ specifies a part of TCB including the fields of `next` and `prev`. We use the predicate $\mathsf{RQ}(q)$ to describe a double linked list as a thread queue pointed by q. Please note that the resources specified by $\mathsf{RQ}q$ include the ready thread resources $\langle t \rangle$ inside. Thus we can know that every node in the thread queue is exactly a TCB of a *ready* thread.

$$\mathsf{RQseg}(t',t) \triangleq \langle t \rangle * (t \overset{prev}{\longmapsto} t') * \exists t''.(t \overset{next}{\longmapsto} t'') * \mathsf{RQseg}(t,t'')$$
$$\mathsf{RQseg}(t',t) \triangleq \natural(t = \mathsf{NULL})$$
$$\mathsf{RQ}(rq) \triangleq \exists t.(rq \mapsto t) * \mathsf{RQseg}(\mathsf{NULL},t)$$

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

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

The precondition states the following resources:

- $[t]$: the current thread resource with thread ID t;
- $\mathsf{ptcb}(t)$: the two pointers in TCB of current thread t;
- $(\mathtt{cur} \mapsto t)$: the global variable `cur` pointing to current thread;
- $\mathsf{RQ}(\mathtt{rq})$: a ready thread which includes ready thread resources;
- $(\mathtt{ra} \mapsto ret)$: the register `ra` containing the return address;
- $\mathsf{K}(bp, 20)$: the stack register `sp` and a block of memory as an unused stack, with base address of bp and the size of 20;
- $(\mathtt{v0}, \mathtt{a0}, \mathtt{a1}, \mathtt{a2} \mapsto \_)$: all of other registers used for the function.

The postcondition of the schedule function is same with the precondition. Here we use a notation $\mathsf{K}(bp, n, w :: w' :: \cdots)$ 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). 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:

$$\begin{array}{l}\mathsf{K}(bp, n, \mathtt{w}_0 :: \mathtt{w}_1 :: \ldots :: \mathtt{w}_m) \triangleq \exists sp\,.\,(\mathtt{sp} \mapsto sp) * \sharp(sp = bp + 4n) \\ \qquad * (bp \mapsto {}^{(n)}\_) * (sp \mapsto \mathtt{w}_0) * (sp{+}4 \mapsto \mathtt{w}_1) * \cdots * (sp{+}4m \mapsto \mathtt{w}_m) \\ \mathsf{K}(bp, n) \triangleq \mathsf{K}(bp, n, \cdot)\end{array}$$

Note that the stack register `sp` is included in the $\mathsf{K}(\cdots)$.

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

$$\left\{\begin{array}{l}\mathsf{RQ}(q) * \mathsf{ptcb}(t) * (\mathtt{a0}, \mathtt{a1}, \mathtt{a2}, \mathtt{v0}, \mathtt{ra} \mapsto q, t, w, \_, ret) * \mathsf{K}(bp, 10) \\ (\langle t \rangle \mathrel{-\!\!*} \mathsf{RQ}(q)) * (\mathtt{a0}, \mathtt{a1}, \mathtt{a2}, \mathtt{v0} \mapsto q, t, w, ret) * (\mathtt{v0} \mapsto 0) * \mathsf{K}(bp, 10)\end{array}\right\}^{(q, t, w, ret)}$$

$$\left\{\begin{array}{l}\mathsf{RQ}(q) * (\mathtt{a0}, \mathtt{a1}, \mathtt{a2}, \mathtt{ra} \mapsto q, w1, w2, ret) * \mathsf{K}(bp, 10) * (\mathtt{v0} \mapsto \_) \\ \mathsf{RQ}(q) * (\mathtt{a0}, \mathtt{a1}, \mathtt{a2}, \mathtt{ra} \mapsto q, w1, w2, ret) * \mathsf{K}(bp, 10) \\ \qquad\qquad * ((\exists t\,.\,(\mathtt{v0} \mapsto t) * \langle t \rangle * \mathsf{ptcb}(t)) \mathbb{W} (\mathtt{v0} \mapsto 0))\end{array}\right\}^{(q, w1, w2, ret)}$$

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

$$I(t, t') \triangleq \mathsf{ptcb}(t') * (\mathtt{cur} \mapsto t') * (\langle t \rangle \mathrel{-\!\!*} \mathsf{RQ}(\mathtt{rq}))$$

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

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

`schedule_p2:`

$\{[t] * \mathsf{ptcb}(t) * (\mathtt{cur} \mapsto t) * \mathsf{RQ}(\mathtt{rq}) * (\mathtt{a0},\mathtt{a1},\mathtt{v0},\mathtt{ra} \mapsto \_,\_,\_,ret) * \mathsf{K}(bp,20)\}$

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

$\{[t] * \mathsf{ptcb}(t) * (\mathtt{cur} \mapsto t) * \mathsf{RQ}(\mathtt{rq}) * \mathsf{K}(bp,17,t :: \_ :: ret)$
$* (\mathtt{a0},\mathtt{a1},\mathtt{v0},\mathtt{ra} \mapsto \mathtt{cur},\_,t,\_)\}$

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

$\{[t] * \mathsf{ptcb}(t) * \langle t' \rangle * \mathsf{ptcb}(t') * \mathsf{RQ}(\mathtt{rq}) * \mathsf{K}(bp,17,t :: \_ :: ret) * (\mathtt{cur} \mapsto t)$
$* (\mathtt{a0},\mathtt{a1},\mathtt{v0},\mathtt{ra} \mapsto \mathtt{rq},\_,t',\_)\}$

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

$\{[t] * \langle t' \rangle * \mathsf{ptcb}(t') * (\langle t \rangle \mathrel{-\!\!*} \mathsf{RQ}(\mathtt{rq})) * \mathsf{K}(bp,17,t :: t' :: ret) * (\mathtt{cur} \mapsto t)$
$* (\mathtt{a0},\mathtt{a1},\mathtt{v0},\mathtt{ra} \mapsto \mathtt{rq},t,0,\_)\}$

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

$\{[t] * \langle t' \rangle * (\langle t \rangle \mathrel{-\!\!*} \mathsf{RQ}(\mathtt{rq})) * \mathsf{ptcb}(t') * \mathsf{K}(bp,17,t :: t' :: ret) * (\mathtt{cur} \mapsto t')$
$* (\mathtt{a0},\mathtt{a1},\mathtt{v0},\mathtt{ra} \mapsto t,t',0,\_)\}$

```
        cswitch
```

$\{[t] * \mathsf{ptcb}(t) * \exists t''.\, \langle t'' \rangle * (\langle t'' \rangle \mathrel{-\!\!*} \mathsf{RQ}(\mathtt{rq})) * \mathsf{K}(bp,17,t :: \_ :: ret) * (\mathtt{cur} \mapsto t)$
$* (\mathtt{a0},\mathtt{a1},\mathtt{v0},\mathtt{ra} \mapsto t,t',\_,\_)\}$

`Ls_ret:`

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

$\{[t] * \mathsf{ptcb}(t) * (\mathtt{cur} \mapsto t) * \mathsf{RQ}(\mathtt{rq}) * (\mathtt{a0},\mathtt{a1},\mathtt{v0},\mathtt{ra} \mapsto \_,\_,\_,ret) * \mathsf{K}(bp,20)\}$

```
        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 a infinite loop. A stub function is needed `schedule_p3()` for being invoked by other threads.

```
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:

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

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

- $[t]$: the current thread with thread ID t;
- $\mathsf{ptcb}(t)$: the two pointers of TCB of current thread t;
- $(\mathtt{cur} \mapsto t)$: the global variable `cur` is pointing to current thread;
- $(\mathtt{ra} \mapsto \mathit{ret})$: the register containing the return address;
- $\mathsf{K}(\mathit{bp}, 10)$: stack register and a block of memory as a empty stack, with base address of bp and size of 10;
- $(\mathtt{a0}, \mathtt{a1} \mapsto \_)$: two registers;
- $\langle \mathtt{sched} \rangle$: 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 $\langle \mathtt{sched} \rangle$ only. The specification of `schedth()` function is shown below:

$$\left\{\begin{array}{l}[\mathtt{sched}] * (\mathtt{cur} \mapsto \_) * \mathsf{RQ}(\mathtt{rq}) * (\mathtt{a0}, \mathtt{a1}, \mathtt{a2}, \mathtt{v0}, \mathtt{ra} \mapsto \_, \_, \_, \_, \_, \_) * \mathsf{K}(\mathit{bp}, 10) \\ \mathsf{false}\end{array}\right\}^{(\mathit{bp})}$$

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

schedth:

$\{[\texttt{sched}] * (\texttt{cur} \mapsto \_) * \mathsf{RQ}(\texttt{rq}) * \mathsf{K}(bp,10) * (\texttt{a0},\texttt{a1},\texttt{v0},\texttt{ra} \mapsto \_,\_,\_,\_)\}$

```
movi    a0,   rq
call    deq
bz      v0,   schedth
```

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

```
movi    a2,   cur
sw      v0,   0(a2)
mov     a1,   v0
lw      a0,   sched
```

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

```
cswitch
```

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

```
movi    a0,   rq
lw      a1,   0(a2)
```

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

```
call    enq
```

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

```
jmp     schedth
```

schedule_p3:

$\{[t] * \mathsf{ptcb}(t) * \langle \texttt{sched} \rangle * (\texttt{cur} \mapsto t) * \mathsf{K}(bp,10) * (\texttt{a0},\texttt{a1},\texttt{ra} \mapsto \_,\_,ret)\}$

```
subi    sp,   4
sw      ra,   0(sp)
movi    a1,   cur
lw      a0,   0(a1)
movi    a1,   sched
```

$\{[t] * \mathsf{ptcb}(t) * \langle \texttt{sched} \rangle * (\texttt{cur} \mapsto t) * \mathsf{K}(bp,9,ret) * (\texttt{a0},\texttt{a1},\texttt{ra} \mapsto t,\texttt{sched},ret)\}$

```
cswitch
```

$\{[t] * \mathsf{ptcb}(t) * \langle \texttt{sched} \rangle * (\texttt{cur} \mapsto t) * \mathsf{K}(bp,9,ret) * (\texttt{a0},\texttt{a1},\texttt{ra} \mapsto \_,\_,ret)\}$

```
lw      ra,   0(sp)
addi    sp,   4
```

$\{[t] * \mathsf{ptcb}(t) * \langle \texttt{sched} \rangle * (\texttt{cur} \mapsto t) * \mathsf{K}(bp,10) * (\texttt{a0},\texttt{a1},\texttt{ra} \mapsto \_,\_,ret)\}$

```
ret
```

Fig. 14. Verification of `schedule_p3()`

– [sched]: the current thread with thread ID t;
– $(\texttt{cur} \mapsto \_)$: the global variable cur pointing to current thread;
– $(\texttt{ra} \mapsto ret)$: the register containing the return address;
– $\texttt{K}(bp, 10)$: the stack register sp and a block of memory as empty stack frame, with base address of bp and the size of 20;
– $(\texttt{v0}, \texttt{a0}, \texttt{a1}, \texttt{a2}, \texttt{ra} \mapsto \_)$: 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:

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

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.

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

struct tcb *cur;       |   void intr_handler()
struct tcb *rq;        |   {
int istack[IKSIZE];    |     struct tcb *new;
void f()               |     savectxt(cur);
{                      |     new = deq(rq);
  struct tcb *t;       |     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:

$$
\begin{array}{lllll}
(\text{AbsMach}) & W & ::= & (C, S, \texttt{pc}) \\
(\text{State}) & S & ::= & (H, R, F, P) \\
(\text{Flag}) & F & ::= & \{\texttt{if} : \texttt{w}\} \\
(\text{Pool}) & P & ::= & \{t : T\}^* \\
(\text{TID}) & t & ::= & \texttt{w} \mid \mathit{isr} \\
(\text{Thrd}) & T & ::= & \mathit{run} \mid (\mathit{rdy}, R, \texttt{pc}) \mid (\mathit{intr}, t) \\
(\text{AbsInstr}) & \texttt{c} & ::= & \texttt{i} \mid \texttt{iret} \mid \texttt{sti} \mid \texttt{cli}
\end{array}
$$

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 insterrupt 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 occurrs, 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 $W \longmapsto W'$ specifies the one-step operational semantics of the machine. The relation $(S, \texttt{pc}) \overset{\texttt{intr}}{\hookrightarrow} (S', \texttt{pc}')$ specifies the state transition when an interrupt occurrs; while the relation $(S, \texttt{pc}) \overset{\texttt{iret}}{\hookrightarrow} (S', \texttt{pc}')$ 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.

$$\frac{C(\mathtt{pc}) = \mathtt{c} \quad (S,\mathtt{pc}) \overset{\mathtt{c}}{\hookrightarrow} (S',\mathtt{pc}')}{(C,S,\mathtt{pc}) \longmapsto (C,S',\mathtt{pc}')} \qquad \frac{S.F(\mathtt{if}) = 1 \quad (S,\mathtt{pc}) \overset{\mathtt{intr}}{\hookrightarrow} (S',\mathtt{pc}')}{(C,S,\mathtt{pc}) \longmapsto (C,S',\mathtt{pc}')}$$

$((M,R,F,P),\mathtt{pc}) \overset{\mathtt{intr}}{\hookrightarrow} ((M',R',F',P'),\mathtt{pc}')$
$M=M' \wedge R'=R \wedge t=R(\mathtt{a2}) \wedge \mathtt{pc}'=\mathtt{l}^{\mathsf{intr}} \wedge F(\mathtt{if})=1 \wedge F'(\mathtt{if})=0$ $\wedge P = \{t : run\} \uplus P''$ $\wedge P' = \{t : (rdy,R,\mathtt{pc})\} \uplus \{isr : intr\} \uplus P''$

$((M,R,F,P),\mathtt{pc}) \overset{\mathtt{c}}{\hookrightarrow} ((M',R',F',P'),\mathtt{pc}')$	
if c =	then
i	$((M,R),\mathtt{pc}) \overset{\mathtt{i}}{\hookrightarrow} ((M',R'),\mathtt{pc}') \wedge P=P' \wedge F=F'$
iret	$M=M' \wedge t=R(\mathtt{a2}) \wedge \mathtt{pc}'=\mathtt{pc}_t \wedge F(\mathtt{if})=0 \wedge F'(\mathtt{if})=1$ $\wedge P = \{t : (rdy,R_t,\mathtt{pc}_t)\} \uplus \{isr : intr\} \uplus P''$ $\wedge P' = \{t : run\} \uplus P''$
sti	$M=M' \wedge R=R' \wedge F(\mathtt{if})=0 \wedge F'(\mathtt{if})=1 \wedge P=P'$
cli	$M=M' \wedge R=R' \wedge F(\mathtt{if})=1 \wedge F'(\mathtt{if})=0 \wedge P=P'$

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 $\mathtt{if} \mapsto b$ specifies the flag register, and $\mathtt{if} \hookrightarrow b$ specifies the flag register and other uninteresting resources. We change the definition of $\diamond p$ here, which states that a shared assertion p doesn't specify any resource of general register or flag register.

$$\begin{array}{rcl} [[\cdot]] & \triangleq & \lambda(M,R,F,P).\, P=\{isr : intr\} \wedge M=F=R=\{\cdot\} \\ \mathtt{if} \mapsto b & \triangleq & \lambda(M,R,F,P).\, F=\{\mathtt{if} : b\} \wedge M=R=P=\{\cdot\} \\ \mathtt{if} \hookrightarrow b & \triangleq & \lambda(M,R,F,P).\, F=\{\mathtt{if} : b\} \\ \diamond p & \triangleq & \lambda(M,R,F,P).\, p\,(M,R,F,P) \wedge R=F=\{\cdot\} \end{array}$$

6.1 Inference rules

We extend our proof system with the following four rules for new instruction in Figure 17. The invariant of context switch $J(t)$ 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 (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,
- $(\mathtt{a2} \mapsto t)$: the value of register a0 is equal to t,
- $(\mathtt{if} \mapsto 1)$: the flag register with interrupt enabled.

$$\frac{(p,g) \Rightarrow \left\{ \begin{array}{l} [t] * (\texttt{if} \mapsto 0) * \diamond J(t) \\ [t] * (\texttt{if} \mapsto 1) \end{array} \right\}^{(t)} \triangleright \Psi(\texttt{pc}+1)}{\Psi, J \vdash \{(p,g)\}\ \texttt{pc} : \texttt{sti}}\ (\textsc{sti})$$

$$\frac{(p,g) \Rightarrow \left\{ \begin{array}{l} [t] * (\texttt{if} \mapsto 1) \\ [t] * (\texttt{if} \mapsto 0) * \diamond J(t) \end{array} \right\}^{(t)} \triangleright \Psi(\texttt{pc}+1)}{\Psi, J \vdash \{(p,g)\}\ \texttt{pc} : \texttt{cli}}\ (\textsc{cli})$$

$$\frac{(p,g) \Rightarrow \left\{ \begin{array}{l} [[\cdot]] * (\texttt{a2} \mapsto t) * \langle t \rangle * (\texttt{if} \mapsto 0) * \diamond J(t) \\ \bot \end{array} \right\}^{(t)}}{\Psi, J \vdash \{(p,g)\}\ \texttt{pc} : \texttt{iret}}\ (\textsc{iret})$$

$$\frac{(p \Lsh (\texttt{if} \hookrightarrow 1), g) \Rightarrow \left\{ \begin{array}{l} [t] * (\texttt{a2} \mapsto t) * (\texttt{if} \mapsto 1) \\ [t] * (\texttt{a2} \mapsto t) * (\texttt{if} \mapsto 1) \end{array} \right\}^{(t)} \triangleright (p,g)}{\Psi, J \vdash_{ih} \{(p,g)\}\ \texttt{pc} : \texttt{c}}\ (\textsc{intr})$$

$$\frac{\Psi(\texttt{1}^{\mathsf{intr}}) = (p,g)_{isr} \quad \forall \texttt{f} \in \mathrm{dom}(C).\ \ \Psi, J \vdash_{ih} \{\Psi(\texttt{f})\}\texttt{f} : C(\texttt{f}) \qquad \Psi, J \vdash \{\Psi(\texttt{f})\}\ \texttt{pc} : \texttt{i}}{\Psi, J \vdash C}\ (\textsc{cdhp})$$

$$\text{where } (p,g)_{isr} \triangleq \left\{ \begin{array}{l} [[\cdot]] * \mathsf{K}(\texttt{istack}, \mathit{ihksize}) * \langle t \rangle * (\texttt{a2} \mapsto t) * \diamond J(t) \\ \qquad * (\texttt{v0}, \texttt{a0}, \texttt{a1}, \texttt{a2}, \texttt{ra} \mapsto \_, \_, \_, \_, \_) \\ \bot \end{array} \right\}^{(t)}$$

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,
- $(\mathtt{a2} \mapsto t)$: the value of register `a2` pointing to t,
- $(\mathtt{if} \mapsto \mathtt{0})$: the flag register with interrupt enabled,
- $\diamond J(t)$: shared resources.

The rules of (STI) and (CLI) cause ownership tranfer 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 $\mathtt{l}^{\mathsf{intr}}$. 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,
- $(\mathtt{a2} \mapsto t)$: the value of register `a2` pointing to t,
- $(\mathtt{if} \mapsto \mathtt{0})$: the flag register with interrupt disabled,
- $\mathsf{K}(\mathtt{istack}, \mathit{ihksize})$: 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 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.

$$\mathsf{CThrd}(\Psi, t, \mathtt{pc}) \triangleq \exists n.\, \mathsf{Cont}(n, \Psi, \mathtt{pc}) \mathbin{\mathbb{\&}} ([t] * \exists R.\, \lfloor R \rfloor * \mathsf{true})$$

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 RThrd as below:

$$\mathsf{RThrd}(\Psi, t, R_t, \mathtt{pc}_t) \triangleq \lfloor R_t \rfloor * (\mathtt{if} \mapsto 1) * [t] \mathrel{-\!\!*} \mathsf{CThrd}(\Psi, t, \mathtt{pc}_t)$$

The definition of 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 *ihksize*). Thus we define the resources of ISR as below:

$$\mathsf{IThrd}(\Psi, \mathtt{pc}) \triangleq \exists n.\, \mathsf{Cont}(n, \Psi, \mathtt{pc}) \mathbin{\mathbb{\&}} ([[\cdot]] * \exists R.\, \lfloor R \rfloor * \mathsf{kspace}(\mathtt{istack}, \mathit{ihksize}) * \mathsf{true})$$

The global invariant 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. GINV:

$$\begin{aligned}
\mathsf{GINV}(\Psi,P,\mathtt{pc}) \triangleq\ & \mathsf{CThrd}(\Psi,t,\mathtt{pc}) * \mathsf{kspace}(\mathtt{istack},\mathit{ihksize}) * \mathsf{RThrd}(\Psi,t_0,R_0,\mathtt{pc}_0) \\
& * \cdots * \mathsf{RThrd}(\Psi,t_n,R_n,\mathtt{pc}_n) \\
& *((\mathtt{if} \mapsto 1) -\!\!* (\mathtt{if} \mapsto 1) * I(t)) \between ((\mathtt{if} \mapsto 0) -\!\!* (\mathtt{if} \mapsto 0)) \\
& \text{if } P = \{t : \mathit{run}, t_0 : (\mathit{rdy}, R_0, \mathtt{pc}_0), \ldots, t_n : (\mathit{rdy}, R_n, \mathtt{pc}_n)\} \\
\mathsf{GINV}(\Psi,P,\mathtt{pc}) \triangleq\ & \mathsf{IThrd}(\Psi,\mathtt{pc}) * \mathsf{RThrd}(\Psi,t_0,R_0,\mathtt{pc}_0) * \cdots * \mathsf{RThrd}(\Psi,t_n,R_n,\mathtt{pc}_n) \\
& \text{if } P = \{\mathit{isr} : \mathit{intr}, t_0 : (\mathit{rdy}, R_0, \mathtt{pc}_0), \ldots, t_n : (\mathit{rdy}, R_n, \mathtt{pc}_n)\}
\end{aligned}$$

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

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

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 `sti` and `cli` to those intructions. But for `intr` and `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 `entry_ISR` and `loadctx`:

```
entry_ISR:              |   loadctx:
     sw ra, 0(a0)       |       lw sp, 20(a0)
     sw v0, 4(a0)       |       lw a2, 16(a0)
     sw a0, 8(a0)       |       lw a1, 12(a0)
     sw a1, 12(a0)      |       lw a0, 8(a0)
     sw a2, 16(a0)      |       lw v0, 4(a0)
     sw sp, 20(a0)      |       lw ra, 0(a0)
     sw k0, 24(a0)      |       lw k0, 24(a0)
     movi sp, istack    |       iret
```

The operations done by hardware when an interrupt occurrs are: the program counter is saved to the stack of interrupted thread, the processor jumps to `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 `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:

$$J(t) \triangleq \mathsf{ptcb}(t) * (\mathtt{cur} \mapsto t) * \mathsf{RQ}(\mathtt{rq})$$

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:`

$\{[[\cdot]] * (\texttt{if} \mapsto 0) * \mathsf{ptcb}(t) * \langle t \rangle * (\texttt{cur} \mapsto t) * \mathsf{RQ}(\texttt{rq}) * \mathsf{K}(\texttt{istack}, \mathit{ihksize}) * (\texttt{a0} \mapsto t) * (\texttt{v0}, \texttt{a1}, \texttt{a2}, \texttt{ra} \mapsto \_)\}$

```
movi    a1,   a0
movi    a0,   rq
```

$\{[[\cdot]] * (\texttt{if} \mapsto 0) * \mathsf{ptcb}(t) * \langle t \rangle * (\texttt{cur} \mapsto t) * \mathsf{RQ}(\texttt{rq}) * \mathsf{K}(\texttt{istack}, \mathit{ihksize}) * (\texttt{a0} \mapsto \texttt{rq}) * (\texttt{a1} \mapsto t) * (\texttt{v0}, \texttt{a2}, \texttt{ra} \mapsto \_)\}$

```
call    deq
bz      v0,   ihret
```

$\{[[\cdot]] * (\texttt{if} \mapsto 0) * \mathsf{ptcb}(t) * \langle t \rangle * (\texttt{cur} \mapsto t) * \mathsf{RQ}(\texttt{rq}) * \mathsf{K}(\texttt{istack}, \mathit{ihksize}) * (\texttt{v0} \mapsto t') * \langle t' \rangle * \mathsf{ptcb}(t') * (\texttt{a0} \mapsto \texttt{rq}) * (\texttt{a1} \mapsto t) * (\texttt{a2}, \texttt{ra} \mapsto \_)\}$

```
movi    a2,   cur
sw      v0,   0(a2)
```

$\{[[\cdot]] * (\texttt{if} \mapsto 0) * \mathsf{ptcb}(t) * \langle t \rangle * (\texttt{cur} \mapsto t') * \mathsf{RQ}(\texttt{rq}) * \mathsf{K}(\texttt{istack}, \mathit{ihksize}) * (\texttt{v0} \mapsto t') * \langle t' \rangle * \mathsf{ptcb}(t') * (\texttt{a0} \mapsto \texttt{rq}) * (\texttt{a1} \mapsto t) * (\texttt{a2}, \texttt{ra} \mapsto \_)\}$

```
call    enq
```

$\{[[\cdot]] * (\texttt{if} \mapsto 0) * (\texttt{cur} \mapsto t') * (\langle t \rangle \mathrel{-\!\!*} \mathsf{RQ}(\texttt{rq})) * \mathsf{K}(\texttt{istack}, \mathit{ihksize}) * \langle t' \rangle * \mathsf{ptcb}(t') * (\texttt{a0} \mapsto \texttt{rq}) * (\texttt{a1} \mapsto t) * (\texttt{v0}, \texttt{a2}, \texttt{ra} \mapsto \_)\}$

```
movi    a2,   cur
lw      a0,   0(a2)
```

$\{[[\cdot]] * (\texttt{if} \mapsto 0) * (\texttt{cur} \mapsto t') * (\langle t \rangle \mathrel{-\!\!*} \mathsf{RQ}(\texttt{rq})) * \mathsf{K}(\texttt{istack}, \mathit{ihksize}) * \langle t' \rangle * \mathsf{ptcb}(t') * (\texttt{a0} \mapsto t') * (\texttt{v0}, \texttt{a1}, \texttt{a2}, \texttt{ra} \mapsto \_)\}$

`ihret:`

$\{[[\cdot]] * (\texttt{if} \mapsto 0) * \mathsf{K}(\mathit{istack}, \mathit{ihksize}) * (\texttt{v0}, \texttt{a1}, \texttt{a2}, \texttt{ra} \mapsto \_) * \exists t' . ((\texttt{a0} \mapsto t') * \mathsf{ptcb}(t') * (\texttt{cur} \mapsto t') * \mathsf{RQ}(\texttt{rq}))\}$

```
iret
```

Fig. 18. Verification of `intr_handler()`

Disscussion. 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 adpated to this pattern by modifying the definition of IThrd: which will be the composition of CThrd and 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 `joiner` is added so as to identify another thread which is waiting for this thead. 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 `sched` for do scheduling. Besides the global pointers `cur` and `rq`, we add one more thread queue for collecting dead threads, pointed by `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 `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 `schedth()` does scheduling job in a separate thread. There a slight difference between this function 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 `exit()` to terminate itself. The `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 `joiner` field in the TCB of current thread. Then `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 `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 `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 comming back.

The function `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, $(dead, k)$. In

```
struct tcb {
  struct context ctxt;
  int *stack;
  struct tcb *prev, *next;
  struct tcb *joiner;
};

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

void 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;
}
```

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

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

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

$$
\begin{array}{llll}
(\mathsf{Mach}) & W & ::= & (C,S,\mathtt{pc}) \\
(\mathsf{State}) & S & ::= & (H,K,R,P) \\
(\mathsf{Heap}) & H & ::= & \{\mathtt{l}:\mathtt{w}\}^* \\
(\mathsf{Stack}) & K & ::= & \{\mathtt{l}:\mathtt{w}\}^* \\
(\mathsf{Thrd}) & T & ::= & \mathit{run} \mid (\mathit{rdy},R) \mid \mathit{dead} \\
(\mathsf{AbsIntr}) & \mathtt{c} & ::= & \mathtt{cswitch} \mid \mathtt{tnew\,r} \mid \mathtt{tdel\,r} \mid \mathtt{quit} \mid \mathtt{i}
\end{array}
$$

Fig. 20. Abstract machine

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: `tnew` to make a new thread resource from heap; `tdel` to recycle a dead thread resource back to heap; and `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 `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, `a0` and `a1`. The command `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 `a0`. The instruction `tnew r` turns a heap block $\mathit{mck}(\mathtt{l},\mathbb{R})$ into a ready thread entity, with ID stored in the register `r`. And the instruction `tdel r` 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, $\langle t \rangle$. Since we separate heap from stack space, we use $\mathtt{l} \rightsquigarrow \mathtt{w}$ to specify a cell in stack space and use $\mathtt{l} \mapsto \mathtt{w}$ 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.

$$
\begin{array}{rcl}
\langle t \rangle & \triangleq & \lambda(H,K,R,P).\, P = \{t : (\mathit{dead}, \_)\} \wedge H = \{\cdot\} \wedge K = \{\cdot\} \wedge R = \{\cdot\} \\
\mathtt{l} \rightsquigarrow \mathtt{w} & \triangleq & \lambda(H,K,R,P).\, K = \{\mathtt{l}:\mathtt{w}\} \wedge l \neq \mathsf{NULL} \wedge H = \{\cdot\} \wedge R = \{\cdot\} \wedge P = \{\cdot\} \\
\mathtt{l} \mapsto \mathtt{w} & \triangleq & \lambda(H,K,R,P).\, H = \{\mathtt{l}:\mathtt{w}\} \wedge l \neq \mathsf{NULL} \wedge K = \{\cdot\} \wedge R = \{\cdot\} \wedge P = \{\cdot\} \\
\diamond p & \triangleq & \lambda(H,K,R,P).\, p\ (H,K,R,P) \wedge R = \{\cdot\} \wedge K = \{\cdot\}
\end{array}
$$

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

$$
\begin{array}{l}
\mathsf{kspace}(\mathit{bp},n) \triangleq (\mathit{bp} \rightsquigarrow \_) * (\mathit{bp}+4 \rightsquigarrow \_) * \cdots * (\mathit{bp}+4(n-1) \rightsquigarrow \_) \\
\mathsf{hspace}(l,n) \triangleq (l \mapsto \_) * (l+4 \mapsto \_) * \cdots * (l+4(n-1) \mapsto \_) \\
\mathsf{K}(\mathit{bp},n,\mathtt{w}_0 :: \mathtt{w}_1 :: \ldots :: \mathtt{w}_m) \triangleq \exists \mathit{sp}.\, (\mathtt{sp} \mapsto \mathit{sp}) * \sharp(\mathit{sp} = \mathit{bp}+4n) \\
\qquad * \mathsf{kspace}(\mathit{bp},n) * (\mathit{sp} \rightsquigarrow \mathtt{w}_0) * (\mathit{sp}+4 \rightsquigarrow \mathtt{w}_1) * \cdots * (\mathit{sp}+4m \rightsquigarrow \mathtt{w}_m)
\end{array}
$$

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

$((H,K,R,P),\mathtt{pc}) \overset{\mathtt{c}}{\hookrightarrow} ((H',K',R',P'),\mathtt{pc}')$	
if $\mathtt{c} =$	then
`i`	$P = P' \wedge ((H,R),\mathtt{pc}) \overset{\mathtt{i}}{\hookrightarrow} ((H',R'),\mathtt{pc}')$
	$P = P' \wedge ((K,R),\mathtt{pc}) \overset{\mathtt{i}}{\hookrightarrow} ((K',R'),\mathtt{pc}')$
`cswitch`	$H = H' \wedge K = K' \wedge R'' = R\{\mathtt{ra} : \mathtt{pc}+1\} \wedge \mathtt{pc}' = R'(\mathtt{ra})$ $\wedge t = R(\mathtt{a0}) \wedge t' = R(\mathtt{a1})$ $\wedge P = \{t : run, t' : (rdy, R')\} \uplus P''$ $\wedge P' = \{t : (rdy, R''), t' : run\} \uplus P''$ $\wedge R$ and R' is complete.
`quit`	$H = H' \wedge K = K' \wedge t' = R(\mathtt{a0}) \wedge \mathtt{pc}' = R'(\mathtt{ra})$ $\wedge P = \{t : run, t' : (rdy, R')\} \uplus P''$ $\wedge P' = \{t : dead, t' : run\} \uplus P''$
`tnew r`	$H = H' \uplus mck(t_x, R_x, k_x) \uplus blk(k_x, kmaxsize)$ $\wedge P' = P \uplus \{t_x : (rdy, \mathbb{R}_x)\} \wedge R = R' \wedge t_x = R(\mathtt{r})$ $\wedge K' = K \uplus blk(k_x, kmaxsize) \wedge \mathtt{pc}' = \mathtt{pc}+1$
`tdel r`	$H' = H \uplus mck(t_x, R_x, k_x) \uplus blk(k_x, kmaxsize)$ $\wedge P = P' \uplus \{t_x : dead\} \wedge R = R' \wedge t_x = R(\mathtt{r})$ $\wedge K = K' \uplus blk(k_x, kmaxsize) \wedge \mathtt{pc}' = \mathtt{pc}+1$

$$\begin{array}{lcl} mck(\mathtt{l}, R, k) & \triangleq & mc(\mathtt{l}, R) \uplus \{\mathtt{l}+24 : k\} \\ blk(k, s) & \triangleq & \{k : \_, k+4 : \_, \ldots, k+4(s-1) : \_\} \end{array}$$

Fig. 21. Operational semantics

We use the following notations to specify data in TCBs:

$$
\begin{array}{lcl}
t \overset{stack}{\longmapsto} k & \triangleq & t+\texttt{OFF\_STACK} \mapsto k \\
t \overset{next}{\longmapsto} p & \triangleq & t+\texttt{OFF\_NEXT} \mapsto p \\
t \overset{prev}{\longmapsto} q & \triangleq & t+\texttt{OFF\_PREV} \mapsto q \\
t \overset{joiner}{\longmapsto} j & \triangleq & t+\texttt{OFF\_JOINER} \mapsto j
\end{array}
$$

Then we define notations for specifying the exposed part of TCB. The notation $\mathsf{rtcb}(t)$ specifies three pointers, among which the pointer of `joiner` points to a waiting chain, specified by $\mathsf{WQ}\texttt{t}$. The notation $\mathsf{dtcb}(t)$ 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 $\mathsf{tcb}(t)$ is used to specify an empty TCB with all fields.

$$
\begin{array}{lcl}
\mathsf{rtcb}(t) & \triangleq & (t \overset{prev}{\longmapsto} \_) * (t \overset{next}{\longmapsto} \_) * \exists j\,.\,(t \overset{joiner}{\longmapsto} j) * \mathsf{WQ}(j) \\
\mathsf{dtcb}(t) & \triangleq & (t \overset{prev}{\longmapsto} \_) * (t \overset{next}{\longmapsto} \_) * (t \overset{joiner}{\longmapsto} \mathsf{NULL}) \\
\mathsf{WQ}(t) & \triangleq & (\langle t \rangle * \mathsf{rtcb}(t)) \mathbin{\mathbb{W}} \natural(t = \mathsf{NULL}) \\
\mathsf{tcb}(t) & \triangleq & (t \overset{ctxt}{\longmapsto} \_) * (t \overset{stack}{\longmapsto} \_) * \mathsf{dtcb}(t)
\end{array}
$$

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.

$$
\begin{array}{lcl}
\mathsf{RQseg}(t,t') & \triangleq & \langle t \rangle * (t \overset{prev}{\longmapsto} t') * \exists t''\,.\,(t \overset{next}{\longmapsto} t'') * \mathsf{RQseg}(t'',t) \\
 & & * \exists j\,.\,(t \overset{joiner}{\longmapsto} j) * \mathsf{WQ}(j) \\
\mathsf{RQseg}(t,t') & \triangleq & \natural(t = \mathsf{NULL}) \\
\mathsf{RQ}(q) & \triangleq & \exists t\,.\,(q \mapsto t) * \mathsf{RQseg}(t, \mathsf{NULL})
\end{array}
$$

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 quiting 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:

$$\forall t,t'\,.\,X(t,t') * \langle\!|t|\!\rangle \Rightarrow J(t') \qquad\qquad \forall t,t'\,.\,I(t,t') * \langle\!|t|\!\rangle \Rightarrow J(t')$$

Since X specifies the invariant before one thread `quit`, if it is added a dead resource of the thread $\langle\!|t|\!\rangle$, the separation conjuction will imply the invariant of $J(t)$. The invariant X shouldn't specify any stack resouce 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.

7.3 Inference rules

The extended inference rules of new instructions are shown in Figure 22. We add a notation Δ 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.

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

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, *kmaxsize*.

$$(p,g)_{init} \triangleq \left\{ \begin{array}{l} [t] * \lfloor R \rfloor * \mathsf{kspace}(R(\mathtt{sp}), kmaxsize) \\ \bot \end{array} \right\}^{(t,R)}$$

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;
- $(\mathtt{a0} \mapsto t')$: the register a0 with value of t';
- $\diamond X(t,t')$: 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:

- $(t \overset{ctxt,k}{\longmapsto} R,k)$: resources of machine context and a cell of memory with value k;
- $\mathsf{RThrd}(\Delta,t,R,k)$: resources satisfying the predicate of a concrete ready thread.

$$\mathsf{NThrd}(\Delta,t,R) \triangleq \exists k\,.\, t \overset{stack}{\longmapsto} k * \mathsf{hspace}(k, kmaxsize) * (\mathsf{kspace}(k, kmaxsize) \mathrel{-\!\!*} \mathsf{RThrd}(\Delta,t,R,k))$$

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 $\mathsf{DThrd}(t,k)$.

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 CThrd:

$$\begin{array}{l} \mathsf{CThrd}(\Psi,t,\mathtt{pc}) \triangleq \exists k\,.\,(t \overset{stack}{\longmapsto} k) * \exists n\,.\,\mathsf{Cont}(n,\Psi,\mathtt{pc}) \\ \qquad\qquad \mathbin{\&\!\!\&} ([t] * \exists R\,.\,\lfloor R \rfloor * \mathsf{kspace}(k, kmaxsize) * \top) \end{array}$$

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.

$$\mathsf{RThrd}(\Psi,t,R) \triangleq \lfloor R \rfloor * [t] * \diamond I(t) \mathrel{-\!\!*} \mathsf{CThrd}(\Psi,t,R(\mathtt{ra}))$$

We add a new definition of a concrete dead thread DThrd, which is just equal to the stack space of thread starting at k.

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

$$\frac{(p,g) \Rightarrow \left\{ \begin{array}{l} [t] * \langle t' \rangle * (\mathtt{a0} \mapsto t') * \diamond X(t,t') \\ \bot \end{array} \right\}^{(t,t')} \triangleright \Psi(\mathtt{pc}+1)}{\Psi, \Delta, I, J, X \vdash \{(p,g)\}\ \mathtt{pc} : \mathtt{quit}} \ (\textsc{tquit})$$

$$\frac{(p,g) \Rightarrow \left\{ \begin{array}{l} (\mathtt{r} \mapsto t) * \exists R.\, (t \overset{ctxt}{\longmapsto} R) * \mathsf{NThrd}(\Delta, t, R) \\ (\mathtt{r} \mapsto t) * \langle t \rangle \end{array} \right\}^{(t)} \triangleright \Psi(\mathtt{pc}+1)}{\Psi, \Delta, I, J, X \vdash \{(p,g)\} \mathtt{pc} : \mathtt{tnew\ r}} \ (\textsc{tnew})$$

$$\frac{(p,g) \Rightarrow \left\{ \begin{array}{l} (\mathtt{r} \mapsto t) * \langle\!| t |\!\rangle \\ (\mathtt{r} \mapsto t) * (t \overset{ctxt}{\longmapsto} \_) * \exists k.\, (t \overset{stack}{\longmapsto} k) * \mathsf{hspace}(k, kmaxsize) \end{array} \right\}^{(t)} \triangleright \Psi(\mathtt{pc}+1)}{\Psi, \Delta, I, J, X \vdash \{(p,g)\}\ \mathtt{pc} : \mathtt{tdel\ r}} \ (\textsc{tdel})$$

$$\frac{\begin{array}{c} \forall \mathtt{f} \in \mathrm{dom}(\Psi).\quad \Psi, \Delta, I, J, X \vdash \{\Psi(\mathtt{f})\}\mathtt{f} : C(\mathtt{f}) \\ \Delta \subseteq \Psi \qquad \forall \mathtt{f} \in \mathrm{dom}(\Delta).\, \Delta(\mathtt{f}) \Rightarrow (p,g)_{init} \end{array}}{\Psi, \Delta, I, J, X \vdash C} \ (\textsc{cdhp})$$

Fig. 22. Extended inference rules

$$\mathsf{DThrd}(t) \ \triangleq \ \exists k.\, (t \overset{stack}{\longmapsto} k) * \mathsf{kspace}(k, kmaxsize)$$

The invariant of global resources is defined as below:

$$\begin{array}{l} \mathsf{GINV}(\Psi, P, \mathtt{pc}) \ \triangleq \ \mathsf{CThrd}(\Psi, t, \mathtt{pc}) * \mathsf{RThrd}(\Psi, t_0, R_0) * \cdots * \mathsf{RThrd}(\Psi, t_m, R_m) \\ \qquad * \mathsf{DThrd}(t_{m+1}) * \cdots * \mathsf{DThrd}(t_n) * \top \\ \text{where } P = \{t : run, t_0 : (rdy, R_0), \ldots, t_m : (rdy, R_m), t_{m+1} : dead, \ldots, t_n : dead\} \end{array}$$

7.5 Soundness

By the invariant GINV, we can prove the soundness theorem of this extension of our proof system.

Theorem 6. *For any machine configuration* $(C, S, \mathtt{pc})$*, if its code is verified* $\Psi, \Delta, I, I, X \vdash C$ *and its state satisfies the global invariant* $S \vdash \mathsf{GINV}(\Psi, P, \mathtt{pc})$*, and if the state can be mapped to a physical state,* $S \Downarrow \mathbb{S}$*, then it is safe to run* $Safe((C, S, \mathtt{pc}))$.

Machine translation. The translation from abstract machine defined in this section to physical machine defined in Section **??** is simple to define. The new commands `tnew` and `tdel` can be translated to a nop like instruction, `addi r, 0`, which doesn't change the state. The command `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:

$$
\begin{aligned}
I(t,t') &\triangleq \sharp(t=\texttt{sched}) * (\texttt{cur} \mapsto t') * \mathsf{rtcb}(t') * \mathsf{RQ}(\texttt{rq}) * \mathsf{DQ}(\texttt{dq}) \\
&\quad \mathbb{W}\, \sharp(t'=\texttt{sched}) * (\texttt{cur} \mapsto t) * (\langle t \rangle \mathrel{-\!\!*} \mathsf{RQ}(\texttt{rq})) * \mathsf{DQ}(\texttt{dq}) \\
X(t,t') &\triangleq \sharp(t'=\texttt{sched}) * (\texttt{cur} \mapsto t) * \mathsf{RQ}(\texttt{rq}) * (\langle\!|t|\!\rangle \mathrel{-\!\!*} \mathsf{DQ}(\texttt{dq})) \\
I(t) &\triangleq \sharp(t=\texttt{sched}) * (\texttt{cur} \mapsto \_) * \mathsf{RQ}(\texttt{rq}) * \mathsf{DQ}(\texttt{dq}) \\
&\quad \mathbb{W}\, \sharp(t \neq \texttt{sched}) * \mathsf{rtcb}(t) * (\texttt{cur} \mapsto t) * \langle \texttt{sched} \rangle * \mathsf{RQ}(\texttt{rq}) * \mathsf{DQ}(\texttt{dq})
\end{aligned}
$$

The code and assertions are presented in Figure **??**–**??**.

create:

$\{[t] * \mathsf{tcb}(t') * \mathsf{RQ}(\mathtt{rq}) * \sharp(t' \in \mathrm{dom}(\Delta))$
$\quad * (\mathtt{a0}, \mathtt{a1}, \mathtt{a2}, \mathtt{v0}, \mathtt{ra} \mapsto t', \mathtt{f}, \_, \_, \mathit{ret}) * \mathsf{K}(\mathit{bp}, 20)\}$

```
subi  sp,   4
sw    ra,   0(sp)
call  kalloc
```

$\{[t] * \mathsf{tcb}(t') * \mathsf{RQ}(\mathtt{rq}) * \mathsf{hspace}(k, \mathit{kmaxsize})$
$\quad * (\mathtt{a0}, \mathtt{a1}, \mathtt{a2}, \mathtt{v0}, \mathtt{ra} \mapsto t', \mathtt{f}, \_, k, \_) * \sharp(t' \in \mathrm{dom}(\Delta)) * \mathsf{K}(\mathit{bp}, 19, \mathit{ret})\}$

```
sw    v0,   OFF_STACK(a0)
sw    a1,   0(a0)
addi  v0,   KMAXSIZE-1
sw    v0,   OFF_SP(a0)
movi  v0,   0
sw    v0,   OFF_JOINER(a0)
tnew  a0
```

$\{[t] * \langle t' \rangle * \mathsf{rtcb}(t') * \mathsf{RQ}(\mathtt{rq})$
$\quad * (\mathtt{a0}, \mathtt{a1}, \mathtt{a2}, \mathtt{v0}, \mathtt{ra} \mapsto t', \_, \_, k, \_) * \mathsf{K}(\mathit{bp}, 19, \mathit{ret})\}$

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

$\{[t] * \mathsf{RQ}(\mathtt{rq}) \quad * (\mathtt{a0}, \mathtt{a1}, \mathtt{a2}, \mathtt{v0}, \mathtt{ra} \mapsto \_, \_, \_, k, \_) * \mathsf{K}(\mathit{bp}, 19, \mathit{ret})\}$

```
lw    ra,   0(sp)
addi  sp,   4
```

$\{[t] * \mathsf{RQ}(\mathtt{rq}) * (\mathtt{a0}, \mathtt{a1}, \mathtt{a2}, \mathtt{v0}, \mathtt{ra} \mapsto \_, \_, \_, \_, \_) * \mathsf{K}(\mathit{bp}, 19, \mathit{ret})\}$

```
ret
```

Fig. 23. Verification of `create()`

exit:

$\{[t] * \mathsf{rtcb}(t) * (\mathtt{cur} \mapsto t) * \mathsf{RQ}(\mathtt{rq}) * \mathsf{DQ}(\mathtt{dq}) * \langle \mathtt{sched} \rangle$
$* (\mathtt{a0}, \mathtt{a1}, \mathtt{a2}, \mathtt{v0}, \mathtt{ra} \mapsto \_, \_, \_, \_, \_) * \mathsf{K}(bp, 20)\}$

```
movi    a2,   cur
lw      a0,   0(a2)
lw      a1,   OFF_STACK(a0)
bz      a1,   exit_quit
```

$\{[t] * (t \overset{prev}{\longmapsto} \_) * (t \overset{next}{\longmapsto} \_) * (t \overset{joiner}{\longmapsto} t') * \langle t' \rangle * \mathsf{rtcb}(t')$
$* (\mathtt{cur} \mapsto t) * \mathsf{RQ}(\mathtt{rq}) * \mathsf{DQ}(\mathtt{dq}) * \langle \mathtt{sched} \rangle$
$* (\mathtt{a0}, \mathtt{a1}, \mathtt{a2}, \mathtt{v0}, \mathtt{ra} \mapsto t, t', \mathtt{cur}, \_, \mathit{joker}) * * \mathsf{K}(bp, 20)\}$

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

$\{[t] * \mathsf{dtcb}(t) * (\mathtt{cur} \mapsto t) * \mathsf{RQ}(\mathtt{rq}) * \mathsf{DQ}(\mathtt{dq}) * \langle \mathtt{sched} \rangle$
$* (\mathtt{a0}, \mathtt{a1}, \mathtt{a2}, \mathtt{v0}, \mathtt{ra} \mapsto t, 0, \_, \_, \_) * \mathsf{K}(bp, 20)\}$

exit_quit:

$\{[t] * \mathsf{dtcb}(t) * (\mathtt{cur} \mapsto t) * \mathsf{RQ}(\mathtt{rq}) * \mathsf{DQ}(\mathtt{dq}) * \langle \mathtt{sched} \rangle$
$* (\mathtt{a0}, \mathtt{a1}, \mathtt{a2}, \mathtt{v0}, \mathtt{ra} \mapsto t, 0, \_, \_, \_) * \mathsf{K}(bp, 20)\}$

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

$\{[t] * (\mathtt{cur} \mapsto t) * \mathsf{RQ}(\mathtt{rq}) * (\langle t \rangle \mathrel{-\!\!*} \mathsf{DQ}(\mathtt{dq})) * \langle \mathtt{sched} \rangle$
$* (\mathtt{a0}, \mathtt{a1}, \mathtt{a2}, \mathtt{v0}, \mathtt{ra} \mapsto \mathtt{sched}, \_, \_, \_, \_) * \mathsf{K}(bp, 20)\}$

```
quit
```

Fig. 24. Verification of `exit()`

```
join:
```
$\{[t] * \mathsf{rtcb}(t) * (\mathtt{cur} \mapsto t) * \mathsf{RQ}(\mathtt{rq}) * \mathsf{DQ}(\mathtt{dq}) * \langle \mathtt{sched} \rangle$
$\quad * (\mathtt{a0} \mapsto t') * (\mathtt{v0}, \mathtt{a1}, \mathtt{a2} \mapsto \_) * (\mathtt{ra} \mapsto w_r) * \mathsf{K}(bp, 20, \cdot)\}$
```
        addi  sp      -12
        sw    ra      8(sp)
        sw    a0      4(sp)
        movi  a2      cur
        lw    a0,     0(a2)
        sw    a0,     0(sp)
join_loop:
```
$\{[t] * \mathsf{rtcb}(t) * (\mathtt{cur} \mapsto t) * \mathsf{RQ}(\mathtt{rq}) * \mathsf{DQ}(\mathtt{dq}) * \langle \mathtt{sched} \rangle$
$\quad * (\mathtt{a0} \mapsto \_) * (\mathtt{v0}, \mathtt{a1}, \mathtt{a2}, \mathtt{ra} \mapsto \_)$
$\quad \mathsf{K}(bp, 17, t :: t' :: w_r)\}$
```
        movi  a0,     dq
        lw    a1,     4(sp)
        call  rmq
        bz    v0,     join_free
```
$\{[t] * \mathsf{rtcb}(t) * (\mathtt{cur} \mapsto t) * \mathsf{RQ}(\mathtt{rq}) * \mathsf{DQ}(\mathtt{dq}) * \langle \mathtt{sched} \rangle$
$\quad * (\mathtt{a0} \mapsto \_) * (\mathtt{a1} \mapsto t') * (\mathtt{v0}, \mathtt{a2}, \mathtt{ra} \mapsto \_) * \mathsf{K}(bp, 17, t :: t' :: w_r)\}$
```
        movi  a0,     rq
        call  inq
        bz    v0,     join_in
        jmp   join_ret
join_in:
```
$\{[t] * \mathsf{rtcb}(t) * (\mathtt{cur} \mapsto t) * \langle t' \rangle * \mathsf{rtcb}(t') * \mathsf{DQ}(\mathtt{dq}) * \langle \mathtt{sched} \rangle$
$\quad * (\langle t' \rangle * \mathsf{rtcb}(t') \mathrel{-\!\!*} \mathsf{RQ}(\mathtt{rq}))$
$\quad * (\mathtt{a0} \mapsto \_) * (\mathtt{a1} \mapsto t') * (\mathtt{v0}, \mathtt{a2}, \mathtt{ra} \mapsto \_) * \mathsf{K}(bp, 17, t :: t' :: w_r)\}$
```
        lw    a0,     36(a1)
        bz    a0,     join_wait
        movi  v0,     -1
        jmp   join_ret
```

Fig. 25. Verification of `join()` (part 1.)

```
join_wait:
```
$\{[t] * \mathsf{rtcb}(t) * (\mathtt{cur} \mapsto t) * \langle t' \rangle * \mathsf{dtcb}(t') * \mathsf{DQ}(\mathtt{dq}) * \langle \mathtt{sched} \rangle$
$\quad * (\langle t' \rangle * \mathsf{rtcb}(t') \mathrel{-\!\!*} \mathsf{RQ}(\mathtt{rq}))$
$\quad * (\mathtt{a0} \mapsto 0) * (\mathtt{a1} \mapsto t') * (\mathtt{v0}, \mathtt{a2}, \mathtt{ra} \mapsto \_) * \mathsf{K}(bp, 17, t :: t' :: w_r)\}$

```
        lw      a0,     0(sp)
        sw      a0,     36(a1)
        movi    a1,     sched
```
$\{[t] * (\mathtt{cur} \mapsto t) * (\langle t \rangle \mathrel{-\!\!*} \mathsf{RQ}(\mathtt{rq})) * \mathsf{DQ}(\mathtt{dq}) * \langle \mathtt{sched} \rangle$
$\quad * (\mathtt{a0} \mapsto t) * (\mathtt{a1} \mapsto \mathtt{sched}) * (\mathtt{v0}, \mathtt{a2}, \mathtt{ra} \mapsto \_)$
$\quad * \mathsf{K}(bp, 17, t :: t' :: w_r)\}$

```
        cswitch
        jmp     join_loop
join_free:
```
$\{[t] * \mathsf{rtcb}(t) * (\mathtt{cur} \mapsto t) * \mathsf{RQ}(\mathtt{rq}) * \langle\!| t' |\!\rangle * \mathsf{dtcb}(t') * \mathsf{DQ}(\mathtt{dq})$
$\quad * (\mathtt{a0} \mapsto \_) * (\mathtt{a1} \mapsto t') * (\mathtt{v0}, \mathtt{a2}, \mathtt{ra} \mapsto \_) * \langle \mathtt{sched} \rangle$
$\quad * \mathsf{K}(bp, 17, t :: t' :: w_r)\}$

```
        tdel    a1
        lw      a0,     24(a1)
        call    kfree
```
$\{[t] * \mathsf{rtcb}(t) * \mathsf{tcb}(t') * (\mathtt{cur} \mapsto t) * \mathsf{RQ}(\mathtt{rq}) * \mathsf{DQ}(\mathtt{dq}) * \langle \mathtt{sched} \rangle$
$\quad * (\mathtt{a0}, \mathtt{a1} \mapsto \_) * (\mathtt{v0}, \mathtt{a1}, \mathtt{a2}, \mathtt{ra} \mapsto \_) * \mathsf{K}(bp, 17, t :: t' :: w_r)\}$

```
        movi    v0,     0
join_ret:
```
$\{[t] * \mathsf{rtcb}(t) * (\mathtt{cur} \mapsto t) * \mathsf{RQ}(\mathtt{rq}) * \mathsf{DQ}(\mathtt{dq}) * \langle \mathtt{sched} \rangle$
$\quad * (\mathtt{a0}, \mathtt{a1}, \mathtt{a2}, \mathtt{ra} \mapsto \_) * \mathsf{K}(bp, 17, t :: t' :: w_r)$
$\quad * (\mathtt{v0} \mapsto 0) \mathbb{V} (\mathtt{v0} \mapsto \mathtt{-1})\}$

```
        lw      ra,     8(sp)
        addi    sp,     12
```
$\{[t] * \mathsf{rtcb}(t) * (\mathtt{cur} \mapsto t) * \mathsf{RQ}(\mathtt{rq}) * \mathsf{DQ}(\mathtt{dq}) * \langle \mathtt{sched} \rangle$
$\quad * (\mathtt{a0}, \mathtt{a1}, \mathtt{a2} \mapsto \_) * (\mathtt{ra} \mapsto w_r) * \mathsf{K}(bp, 20, \cdot)$
$\quad * ((\mathtt{v0} \mapsto 0) * \mathsf{tcb}(t')) \mathbb{V} (\mathtt{v0} \mapsto \mathtt{-1})\}$

```
        ret
```

Fig. 26. Verification of `join()` (part 2.)

`schedth:`

$\{[\texttt{sched}] * (\texttt{cur} \mapsto t) * \langle t \rangle * (\langle t \rangle \mathrel{-\!\!*} \mathsf{RQ}(\texttt{rq})) * \mathsf{K}(bp, 10, \cdot)$
$* (\texttt{v0} \mapsto \_) * (\texttt{a0} \mapsto \_) * (\texttt{a1} \mapsto \_) * (\texttt{a2} \mapsto \_) * (\texttt{ra} \mapsto \_)\}$

```
movi    a0,   rq
```

$\{[\texttt{sched}] * (\texttt{cur} \mapsto t) * \mathsf{RQ}(\texttt{rq}) * \mathsf{K}(bp, \cdot, \cdot)$
$* (\texttt{v0} \mapsto \_) * (\texttt{a0} \mapsto \texttt{rq}) * (\texttt{a1} \mapsto \_) * (\texttt{a2} \mapsto \_) * (\texttt{ra} \mapsto \_)\}$

```
call    deq
bz      v0,   schedth
```

$\{[\texttt{sched}] * (\texttt{cur} \mapsto t) * \langle t' \rangle * \mathsf{ptcb}(t') * \mathsf{RQ}(\texttt{rq}) * \mathsf{K}(bp, 10, \cdot)$
$* (\texttt{v0} \mapsto t') * (\texttt{a0} \mapsto \_) * (\texttt{a1} \mapsto \_) * (\texttt{a2} \mapsto \_) * (\texttt{ra} \mapsto \_)\}$

```
movi    a2,   cur
sw      v0,   0(a2)
mov     a1,   v0
lw      a0,   sched
```

$\{[\texttt{sched}] * \langle t' \rangle * (\texttt{cur} \mapsto t) * \mathsf{ptcb}(t') * \mathsf{RQ}(\texttt{rq}) * \mathsf{K}(bp, 10, \cdot)$
$* (\texttt{v0} \mapsto t') * (\texttt{a0} \mapsto \texttt{sched}) * (\texttt{a1} \mapsto t') * (\texttt{a2} \mapsto \_) * (\texttt{ra} \mapsto \_)\}$

```
cswitch
```

$\{[\texttt{sched}] * \exists t''.\, (\texttt{cur} \mapsto t'') * (\langle t'' \rangle \mathrel{-\!\!*} \mathsf{RQ}(\texttt{rq}) * \mathsf{K}(bp, 10, \cdot)$
$* (\texttt{v0} \mapsto t') * (\texttt{a0} \mapsto \texttt{sched}) * (\texttt{a1} \mapsto t') * (\texttt{a2} \mapsto \_) * (\texttt{ra} \mapsto \_)\}$

```
jmp     sched  th
```

`yield:`

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

```
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()`

```
kill:
```
$\{[t] * \mathsf{rtcb}(t) * \mathsf{RQ}(\mathtt{rq}) * \mathsf{DQ}(\mathtt{dq})$
$\quad * (\mathtt{a0} \mapsto t') * (\mathtt{v0}, \mathtt{a1}, \mathtt{a2} \mapsto \_) * (\mathtt{ra} \mapsto w_r) * \mathsf{K}(bp, 20, \cdot)\}$
```
        addi    sp      -8
        sw      ra      4(sp)
        sw      a0      0(sp)
```
$\{[t] * \mathsf{rtcb}(t) * \mathsf{RQ}(\mathtt{rq}) * \mathsf{DQ}(\mathtt{dq})$
$\quad * (\mathtt{a0} \mapsto t') * (\mathtt{v0}, \mathtt{a1}, \mathtt{a2}, \mathtt{ra} \mapsto \_) * \mathsf{K}(bp, 18, t' :: w_r)\}$
```
        mov     a1,     a0
        movi    a0,     dq
        call    inq
        bz      v0,     kill_ret
```
$\{[t] * \mathsf{rtcb}(t) * \mathsf{RQ}(\mathtt{rq}) * \mathsf{DQ}(\mathtt{dq})$
$\quad * (\mathtt{a0} \mapsto \mathtt{dq}) * (\mathtt{a1} \mapsto t') * (\mathtt{v0}, \mathtt{a2}, \mathtt{ra} \mapsto \_) * \mathsf{K}(bp, 18, t' :: w_r)\}$
```
        movi    a0,     rq
        call    inq
        bz      v0,     kill_inrq
```
$\{[t] * \mathsf{rtcb}(t) * \mathsf{RQ}(\mathtt{rq}) * \mathsf{DQ}(\mathtt{dq})$
$\quad * (\mathtt{a0} \mapsto \mathtt{rq}) * (\mathtt{a1} \mapsto t') * (\mathtt{v0}, \mathtt{a2}, \mathtt{ra} \mapsto \_) * \mathsf{K}(bp, 18, t' :: w_r)\}$
```
        jmp     kill_ret
kill_inrq:
        call    rmq
        bz      v0,     kill_ck
        jmp     kill_ret
kill_rmrq:
```
$\{[t] * \mathsf{rtcb}(t) * \langle t' \rangle * \mathsf{rtcb}(t') * \mathsf{RQ}(\mathtt{rq}) * \mathsf{DQ}(\mathtt{dq})$
$\quad * (\mathtt{a0} \mapsto \mathtt{rq}) * (\mathtt{a1} \mapsto t') * (\mathtt{v0}, \mathtt{a2}, \mathtt{ra} \mapsto \_) * \mathsf{K}(bp, 18, t' :: w_r)\}$
```
        lw      a2,     36(a1)
        bz      a2,     kill_endq
        mov     a1,     a2
        call    enq
kill_endq:
        lw      a1,     0(sp)
        sw      v0,     36(a1)
        movi    a0,     dq
        call    enq
kill_ret:
        lw      ra,     4(sp)
        addi    sp,     8
```
$\{[t] * \mathsf{rtcb}(t) * \mathsf{RQ}(\mathtt{rq}) * \mathsf{DQ}(\mathtt{dq})$
$\quad * (\mathtt{v0}, \mathtt{a0}, \mathtt{a1}, \mathtt{a2} \mapsto \_) * (\mathtt{ra} \mapsto w_r) * \mathsf{K}(bp, 20, \cdot)\}$
```
        ret
```

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 the 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 runqueue 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 actomic 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 runqueue. 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 peform 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 acuqire 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

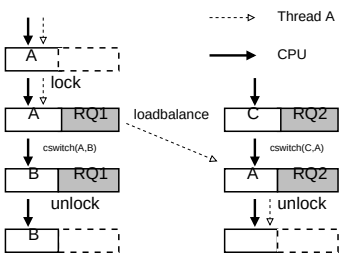

Fig. 30. Ownership transfer of load balancing

$$
\begin{array}{llll}
(\mathsf{Mach}) & W & ::= & (C, Y, M, P) \\
(\mathsf{CPU}) & X & ::= & (R, \mathtt{pc}) \\
(\mathsf{Thrd}) & T & ::= & (rdy, R) \mid (run, u) \\
(\mathsf{Pool}) & P & ::= & \{t : T\}^* \\
(\mathsf{CPUSet}) & Y & ::= & \{u : X\}^* \\
(\mathsf{CPUId}) & u, v & ::= & \mathtt{w} \\
(\mathsf{Mem}) & M & ::= & \{\mathtt{l} : \mathtt{w}\}^* \qquad (\mathtt{l} = 4n, n = 0, 1, 2, \ldots) \\
(\mathsf{Code}) & C & ::= & \{\mathtt{f} : \mathtt{c}\}^* \\
(\mathsf{Instr}) & \mathtt{c} & ::= & \mathtt{i} \mid \mathtt{cswitch} \mid \mathtt{cpuid\ r} \mid \mathtt{xchg\ r}_d, \mathtt{w}(\mathtt{r}_s)
\end{array}
$$

Fig. 31. Definition of abstract machine

The idea of *threads as resources* is still helpful to reason about load balancing. Thread resources bound with TCBs are migrated back and forth among processors. Thread resources don't only provide the knowledge of concrete threads in the system, but also provide the knowledge that which processor one thread is belong to. We extend the running thread with processor information, $[t \circlearrowright u]$, that means the thread t is running on the processor whose ID number is u.

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 $\mathtt{xchg\ r}_d, [\mathtt{r}_s + \mathtt{w}]$ to exchange the value (in the register $\mathtt{r}_s$) with the value in memory (addressed by $\mathtt{r}_s$).

The operational semantics of instructions is shown in Fig. 32. The new instruction `cpuid r` puts the CPU-ID into the register `r`. And the other new instruction `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, `cswitch`, it set the CPU-ID of the current processor to the next ready thread. The predicate atomic is used

$(u,R,\mathtt{pc},M) \overset{\mathtt{i}}{\hookrightarrow} (u,R',\mathtt{pc}',M')$	
if $\mathtt{i} =$	then
$\mathtt{xchg}\ \mathtt{r}_d, \mathtt{w}(\mathtt{r}_s)$	$M' = M\{R(\mathtt{r}_s)+\mathtt{w} : R(\mathtt{r}_d)\} \wedge \mathtt{pc}' = \mathtt{pc}+1$ $\wedge R' = R\{\mathtt{r}_d : M(R(\mathtt{r}_s)+\mathtt{w})\}$
otherwise	$(R,\mathtt{pc},M) \overset{\mathtt{i}}{\hookrightarrow} (R',\mathtt{pc}',M')$

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

$$\text{atomic}(\mathtt{i}) \triangleq \mathtt{i} \in \{\mathtt{xchg}\ \mathtt{r}_d, [\mathtt{r}_s+\mathtt{w}]\}$$

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. **??**. 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:

$$(C,Y,M,P) \longmapsto (C,Y',M',P')$$

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 $[t \circlearrowright u]$, adding the current processor-ID. The notation of an abstract ready thead

$$\begin{array}{rcl} [t \circlearrowright u] & \triangleq & \lambda(M,R,P).\, M = \{\cdot\} \wedge R = \{\cdot\} \wedge P = \{t : (run, u)\} \\ \langle t \rangle & \triangleq & \lambda(M,R,P).\, M = \{\cdot\} \wedge R = \{\cdot\} \wedge P = \{t : (rdy, X)\} \end{array}$$

8.2 Inference Rules

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

$$\frac{\mathtt{c}=C(\mathtt{pc}) \quad \text{atomic}(\mathtt{c}) \quad (u,R,\mathtt{pc},M,P) \overset{\mathtt{c}}{\hookrightarrow} (u,R',\mathtt{pc}',M',P')}{(C,u,(R,\mathtt{pc}),M,P) \longmapsto^{atomic} (C,u,(R',\mathtt{pc}'),M',P')}$$

$$\frac{\mathtt{c}=C(\mathtt{pc}) \quad \text{atomic}(\mathtt{c}) \quad (u,R,\mathtt{pc},M,P) \overset{\mathtt{c}}{\hookrightarrow} (u,R',\mathtt{pc}',M',P')}{(C,u,(R,\mathtt{pc}),M,P) \longmapsto^{conc} (C,u,(R',\mathtt{pc}'),M',P')}$$

$$\frac{}{(C,u,(R,\mathtt{pc}),M,P) \longmapsto^{conc} (C,u,(R,\mathtt{pc}),M,P)}$$

$$\frac{(C,u,X_k,M,P') \longmapsto^{atomic} (C,u,X_k'M',P') \quad Y(u)=X_k \quad Y'=Y\{u_k:X_k'\}}{(C,Y,M,P) \longmapsto (C,Y',M',P')}$$

$$\frac{\forall k \in \{1,\ldots,n\}.\,(C,u_k,X_k,M) \longmapsto^{conc} (C,u_k,X_k',M') \quad X=\{u_1:X_1,u_2:X_2,\ldots,u_n:X_n\} \quad X'=\{u_1:X_1',u_2:X_2',\ldots,u_n:X_n'\}}{(C,Y,M,P) \longmapsto (\mathbb{C},Y',M',P')}$$

Fig. 33. Operational semantics of abstract machine (Ext. 3)

$$\Psi,\Sigma,I \vdash \{(p,g)\}\ \mathtt{pc} : \mathtt{c}$$

To verify the code running on multicore machines, we draw ideas of ownership transfer from concurrent separation logic [17]. By defining invariants for shared resources, our proof system ensures safe operations. Like 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 Σ to specify the invariant of global resources. If one thread accesses the resources specified by I_Σ, it should use kernel synchronization operations, *e.g.,* spin locks, to protect accessing. The invariant Σ is a partial map from lock locations to assertions.

$$\Sigma \in \mathsf{Word} \rightharpoonup \mathsf{Assert}$$

The notation I_Σ 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 Σ:

$$I_\Sigma \triangleq \bigodot_{l \in \mathrm{dom}(\Sigma)} (l \mapsto 0 * \mathsf{emp}) \mathbb{W} (l \mapsto 1 * \Sigma(l))$$

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

$$\begin{aligned}(p,g) * I_\Sigma \triangleq\ & (p * I_\Sigma,\ (\lambda \mathbb{S},\mathbb{S}'.\forall \mathbb{S}_1,\mathbb{S}_2.\,\mathbb{S}=\mathbb{S}_1 \uplus \mathbb{S}_2 \to p\,\mathbb{S}_1 \\ & \to I_\Sigma\,\mathbb{S}_2 \to \exists \mathbb{S}_1',\mathbb{S}_2'.\,\mathbb{S}'=\mathbb{S}_1' \uplus \mathbb{S}_2' \wedge g\,\mathbb{S}_1\,\mathbb{S}_1' \wedge I_\Sigma\,\mathbb{S}_2'))\end{aligned}$$

$$\frac{\forall \mathtt{f} \in \mathrm{dom}(\mathbb{C}) \,.\, \Psi, \Sigma, I \vdash \{(p,g)\}\ \mathtt{f} : C(\mathtt{f})}{\Psi, \Sigma, I \vdash \mathbb{C}}\ (\textsc{cdhp})$$

$$\frac{(p,g) * I_\Sigma \Rightarrow \left\{ \begin{array}{l} (\mathtt{r}_t \mapsto w) * (\mathtt{r}_s{+}\mathtt{w} \mapsto w') \\ (\mathtt{r}_t \mapsto w') * (\mathtt{r}_s{+}\mathtt{w} \mapsto w) \end{array} \right\}^{(w,w')} \triangleright \Psi(\mathtt{pc}{+}1)}{\Psi, \Sigma, I \vdash \{(p,g)\}\ \mathtt{pc} : \mathtt{xchg}\ \mathtt{r}_t, [\mathtt{r}_s{+}\mathtt{w}]}\ (\textsc{xchg})$$

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

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.

$$I \ \in\ \mathsf{CID} \to \mathsf{TID} \to \mathsf{TID} \to \mathsf{Assert}$$

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_Σ. 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 4. 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.

$$\mathsf{CThrd}(\Psi, u, t, \mathtt{pc}) \ \triangleq\ \exists n \,.\, \mathsf{Cont}(n, \Psi, \mathtt{pc}) \Rrightarrow ([t \circlearrowright u] * \lfloor \_ \rfloor * \mathsf{true})$$

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

$$\begin{aligned}\mathsf{RThrd}(\Psi,t,R) \triangleq\ & \lfloor R \rfloor * \exists u\,.\,[t \circlearrowright u] * \exists t'\,.\,\langle t' \rangle * \diamond I(u,t',t) \\ & -\!\!* \mathsf{CThrd}(\Psi,u,t,R(\mathtt{ra}))\end{aligned}$$

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.

$$\begin{aligned}\mathsf{GINV}(\Psi,Y,P,\mathtt{pc}) \triangleq\ & (\lfloor R_0 \rfloor -\!\!* \mathsf{CThrd}(\Psi,u_0,t_0,\mathtt{pc}_0)) * \cdots * (\lfloor R_n \rfloor -\!\!* \mathsf{CThrd}(\Psi,u_n,t_n,\mathtt{pc}_n)) \\ & * \mathsf{RThrd}(\Psi,t_{(}n{+}1),R_{(}n{+}1)) * \cdots * \mathsf{RThrd}(\Psi,t_{n+1},R_{n+1}) \\ \text{where } Y =\ & \{u_0 : (R_0,\mathtt{pc}_0),\ \ldots,\ u_n : (R_n,\mathtt{pc}_n)\} \\ \text{where } P =\ & \{t_0 : (run,u_0)\ \ldots,\ t_n : (run,u_n), t_{n+1} : (rdy,R_{n+1}),\ \ldots,\ t_{n+m} : (rdy,R_{n+m})\}\end{aligned}$$

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 Σ is defined as a map from lk_i to assertion specified by $RQ(u)$.

$$\begin{aligned}I(u,t,t') &\triangleq \mathsf{curr}(u,t') * \mathsf{ptcb}(t') * (\langle t \rangle -\!\!* \mathsf{RQ}(u)) \\ \Sigma &\triangleq \{lk_0 : \mathsf{RQ}(0),\ \ldots,\ lk_n : \mathsf{RQ}(n)\} \\ &\qquad \text{where } lk_u = \mathtt{CPUS} + u \times \mathtt{SIZE\_CPU} + \mathtt{OFF\_LOCK} \\ \mathsf{ptcb}(t) &\triangleq (t \overset{prev}{\longmapsto} \_) * (t \overset{next}{\longmapsto} \_) \\ \mathsf{RQ}(u) &\triangleq \exists rq\,.\,\sharp(rq = \mathtt{CPUS} + u \times \mathtt{SIZE\_CPU} + \mathtt{OFF\_RQ}) * \mathsf{RQ}(rq) \\ \mathsf{curr}(u,t) &\triangleq ((\mathtt{CPUS} + u \times \mathtt{SIZE\_CPU} + \mathtt{OFF\_CURR}) \mapsto t)\end{aligned}$$

where CPUS 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.

$\{[t \circlearrowleft u] * \mathsf{ptcb}(t) * \mathsf{curr}(u,t) * (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto w1,w2,w3,\_,ret) * \mathsf{K}(bp,20)\}$

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

$\{[t \circlearrowleft u] * \mathsf{ptcb}(t) * \mathsf{curr}(u,t) * (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto \_,\_,\_,\_,\_)$
$* \mathsf{K}(bp,13,\_ :: \_ :: \_ :: w1 :: w2 :: w3 :: ret)\}$

```
    cpuid   a0
    call    getcpu
```

$\{[t \circlearrowleft u] * \mathsf{ptcb}(t) * \mathsf{curr}(u,t) * (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto u,\_,\_,\mathtt{CPUS[u]},\_)$
$* \mathsf{K}(bp,13,\_ :: \_ :: \_ :: w1 :: w2 :: w3 :: ret)\}$

```
    sw      v0,         8(sp)
    lw      a0,         OFF_LOCK(v0)
    call    lock
```

$\{[t \circlearrowleft u] * \mathsf{ptcb}(t) * \mathsf{curr}(u,t) * \mathsf{RQ}(u) * (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto \mathtt{CPUS}[u],\_,\_,\mathtt{CPUS}[u],\_)$
$* \mathsf{K}(bp,13,\_ :: \_ :: \mathtt{CPUS}[u] :: w1 :: w2 :: w3 :: ret)\}$

```
    lw      a0,         OFF_RQ(v0)
    call    deq
    bz      v0,         sched_lb
    jmp     sched_cont
```

$\{[t \circlearrowleft u] * \mathsf{ptcb}(t) * \mathsf{curr}(u,t) * \mathsf{RQ}(u) * (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto \mathtt{CPUS}[u],\_,\_,\mathsf{NULL},\_)$
$* \mathsf{K}(bp,13,\_ :: \_ :: \mathtt{CPUS[u]} :: w1 :: w2 :: w3 :: ret)\}$

```
sched_lb:
    lw      a2,         8(sp)
    lw      a0,         OFF_LOCK(a2)
    call    unlock
```

$\{[t \circlearrowleft u] * \mathsf{ptcb}(t) * \mathsf{curr}(u,t) * (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto \mathtt{CPUS}[u],\_,\_,\mathsf{NULL},\_)$
$* \mathsf{K}(bp,13,\_ :: \_ :: \mathtt{CPUS[u]} :: w1 :: w2 :: w3 :: ret)\}$

```
    call    loadbalance
```

$\{[t \circlearrowleft u] * \mathsf{ptcb}(t) * \mathsf{curr}(u,t) * (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto \mathtt{CPUS}[u],\_,\_,\_,\_)$
$* \mathsf{K}(bp,13,\_ :: \_ :: \mathtt{CPUS[u]} :: w1 :: w2 :: w3 :: ret)\}$

```
    lw      a2,         8(sp)
    lw      a0,         OFF_LOCK(a2)
    call    lock
```

$\{[t \circlearrowleft u] * \mathsf{ptcb}(t) * \mathsf{curr}(u,t) * \mathsf{RQ}(u)$
$* (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto \mathtt{CPUS}[u],\_,\_,\_,\_)$
$* \mathsf{K}(bp,13,\_ :: \_ :: \mathtt{CPUS}[u] :: w1 :: w2 :: w3 :: ret)\}$

```
    lw      a2,         8(sp)
    lw      a0,         OFF_RQ(a2)
    call    deq
    bz      v0,         sched_ret
```

$\{[t \circlearrowleft u] * \mathsf{ptcb}(t) * \mathsf{curr}(u,t) * \mathsf{RQ}(u) * \exists t' \,.\, * \langle t' \rangle * \mathsf{ptcb}(t')$
$* (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto \_,\_,t',\mathsf{NULL},\_)$
$* \mathsf{K}(bp,13,\_ :: \_ :: \mathtt{CPUS}[u] :: w1 :: w2 :: w3 :: ret)\}$

Fig. 35. Ext. 3: Verification of scheduler (part one)

$\{[t \circlearrowright u] * \mathsf{ptcb}(t) * \mathsf{curr}(u,t) * \mathsf{RQ}(u) * \exists t'.\ * \langle t' \rangle * \mathsf{ptcb}(t') * (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto \_,\_,t',\mathsf{NULL},\_)$
$* \mathsf{K}(bp, 13, \_ :: \_ :: \mathtt{CPUS[u]} :: w1 :: w2 :: w3 :: ret)\}$

```
sched_cont:
    sw      v0,   0(sp)
    lw      a2,   8(sp)
    lw      a1,   OFF_CURR(a2)
    sw      a1,   4(sp)
    lw      a0,   OFF_RQ(a2)
```

$\{[t \circlearrowright u] * \mathsf{ptcb}(t) * \mathsf{curr}(u,t) * \mathsf{RQ}(u) * \exists t'.\ * \langle t' \rangle * \mathsf{ptcb}(t')$
$* (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto \mathtt{CPUS}[u]+\mathtt{OFF\_RQ}, t, \mathtt{CPUS}[u], \mathsf{NULL}, \_)$
$* \mathsf{K}(bp, 13, t' :: t :: \mathtt{CPUS}[u] :: w1 :: w2 :: w3 :: ret)\}$

```
    call    enq
```

$\{[t \circlearrowright u] * (\langle t \rangle -\!\!* \mathsf{RQ}(u)) * \mathsf{curr}(u,t) * \exists t'.\ * \langle t' \rangle * \mathsf{ptcb}(t')$
$* (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto \mathtt{CPUS}[u]+\mathtt{OFF\_RQ}, t, \mathtt{CPUS}[u], \_, \_)$
$* \mathsf{K}(bp, 13, t' :: t :: \mathtt{CPUS[u]} :: w1 :: w2 :: w3 :: ret)\}$

```
    lw      a2,   8(sp)
    lw      a1,   0(sp)
    sw      a1,   OFF_CURR(sp)
    lw      a0,   4(sp)
```

$\{[t \circlearrowright u] * (\langle t \rangle -\!\!* \mathsf{RQ}(u)) * \mathsf{curr}(u,t') * \exists t'.\ * \langle t' \rangle * \mathsf{ptcb}(t')$
$* (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto t, t', \mathtt{CPUS}[u], \_, \_)$
$* \mathsf{K}(bp, 13, t' :: t :: \mathtt{CPUS[u]} :: w1 :: w2 :: w3 :: ret)\}$

```
    cswitch
```

$\{\exists u'.[t \circlearrowright u'] * \mathsf{ptcb}(t) * \mathsf{curr}(u',t) * \exists t''.\ * \langle t'' \rangle * (ANGt'' -\!\!* \mathsf{RQ}(u'))$
$* (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto \_,\_,\_,\_,\_) * \mathsf{K}(bp, 13, \_ :: \_ :: \_ :: w1 :: w2 :: w3 :: ret)\}$

```
    cpuid   a0
    call    getcpu
```

$\{\exists u'.[t \circlearrowright u'] * \mathsf{ptcb}(t) * \mathsf{curr}(u',t) * \exists t''.\ * \langle t'' \rangle * (ANGt'' -\!\!* \mathsf{RQ}(u'))$
$* (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto u', \_, \_, \mathtt{CPUS}[u'], \_) * \mathsf{K}(bp, 13, \_ :: \_ :: \_ :: w1 :: w2 :: w3 :: ret)\}$

```
    sw      v0,   8(sp)
    lw      a0,   OFF_LOCK(v0)
    call    unlock
```

$\{\exists u'.[t \circlearrowright u'] * \mathsf{ptcb}(t) * \mathsf{curr}(u',t) * (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto \_,\_,\_,\mathtt{CPUS}[u'],\_)$
$* \mathsf{K}(bp, 13, \_ :: \_ :: \mathtt{CPUS}[u'] :: w1 :: w2 :: w3 :: ret)\}$

```
    lw      a0,   12(sp)
    lw      a1,   16(sp)
    lw      a2,   20(sp)
    lw      ra,   24(sp)
    addi    sp,   28
```

$\{\exists u'.[t \circlearrowright u'] * \mathsf{ptcb}(t) * \mathsf{curr}(u',t) * (\mathtt{a0},\mathtt{a1},\mathtt{a2},\mathtt{v0},\mathtt{ra} \mapsto w1, w2, w3, \_, ret) * \mathsf{K}(bp, 20)\}$

```
    ret
```

Fig. 36. Ext. 3: Verification of scheduler (part two)

$\{(\texttt{a0},\texttt{a1},\texttt{ra} \mapsto lk, w1, ret) * \mathsf{K}(bp,1) * \sharp(lk \in \mathrm{dom}(\Sigma))\}$

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

$\{(\texttt{a0},\texttt{a1},\texttt{ra} \mapsto lk, \_, ret) * \mathsf{K}(bp,0,w1) * \sharp(lk \in \mathrm{dom}(\Sigma))\}$

```
lock   _loop:
       movi    a1, 1
```

$\{(\texttt{a0},\texttt{a1},\texttt{ra} \mapsto lk, 1, ret) * \mathsf{K}(bp,0,w1) * \sharp(lk \in \mathrm{dom}(\Sigma))\}$

```
       xchg    a1,     0(a0)
```

$\{\exists x.\,(\texttt{a0},\texttt{a1},\texttt{ra} \mapsto lk, x, ret) * \mathsf{K}(bp,0,w1) * ((\sharp x = 1) \mathbb{V} (\sharp x = 0) * \Delta(lk)) * \sharp(lk \in \mathrm{dom}(\Sigma))\}$

```
       bz      a1,     lock_ret
       jmp     lock_loop
```

$\{(\texttt{a0},\texttt{a1},\texttt{ra} \mapsto lk, 0, ret) * \mathsf{K}(bp,0,w1) * \Delta(lk) * \sharp(lk \in \mathrm{dom}(\Sigma))\}$

```
lock   _ret:
       lw      a1,     0(sp)
       addi    sp,     4
       ret
```

$\{(\texttt{a0},\texttt{a1},\texttt{ra} \mapsto lk, w1, ret) * \mathsf{K}(bp,1) * \Delta(lk)\}$

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

$\{(\texttt{a0},\texttt{a1},\texttt{ra} \mapsto lk, 0, ret) * \mathsf{K}(bp,0,w1)\}$

```
       lw      a1,     0(sp)
       addi    sp,     4
```

$\{(\texttt{a0},\texttt{a1},\texttt{ra} \mapsto lk, w1, ret) * \mathsf{K}(bp,1)\}$

```
       ret
```

Fig. 37. Ext. 3: Verification of lock/unlock

$\{[t \circlearrowright u] * \mathsf{ptcb}(t) * (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \_, w1, \_, ret) * \mathsf{K}(bp, 10)\}$

```
loadbalance:
    subi        sp,         24
    sw          ra,         20(sp)
    sw          a1,         16(sp)
```

$\{[t \circlearrowright u] * \mathsf{ptcb}(t) * (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \_, w1, \_, ret) * \mathsf{K}(bp, 4, \_ :: \_ :: \_ :: \_ :: w1 :: ret)\}$

```
    cpuid       a0
    call        getcpu
```

$\{[t \circlearrowright u] * \mathsf{ptcb}(t) * (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto u, w1, \mathtt{CPUS}[u], \_) * \mathsf{K}(bp, 4, \_ :: \_ :: \_ :: \_ :: w1 :: ret)\}$

```
    sw          v0,         12(sp)
    mov         a0, v0
    call        busiest_cpu
    bz          Llb_end
```

$\{[t \circlearrowright u] * \mathsf{ptcb}(t) * \exists u' . (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \mathtt{CPUS}[u], w1, \mathtt{CPUS}[u'], \_) * (\sharp u' \neq u)$
$* \mathsf{K}(bp, 4, \_ :: \_ :: \_ :: \mathtt{CPUS}[u] :: w1 :: ret)\}$

```
    sw          v0,         8(sp)
    lw          a0,         12(sp)
```

$\{[t \circlearrowright u] * \mathsf{ptcb}(t) * \exists u' . (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \mathtt{CPUS}[u], w1, \mathtt{CPUS}[u'], \_) * (\sharp u' \neq u)$
$* \mathsf{K}(bp, 4, \_ :: \_ :: \mathtt{CPUS}[u'] :: \mathtt{CPUS}[u] :: w1 :: ret)\}$

```
    sub         v0,         a0
    bgt         v0,         Llb_swap
```

$\{[t \circlearrowright u] * \mathsf{ptcb}(t) * \exists u' . (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \mathtt{CPUS}[u], w1, \_, \_) * (\sharp u > u')$
$* \mathsf{K}(bp, 4, \_ :: \_ :: \mathtt{CPUS}[u'] :: \mathtt{CPUS}[u] :: w1 :: ret)\}$

```
    sw          a0,         4(sp)
    lw          a1,         8(sp)
    sw          a1,         0(sp)
```

$\{[t \circlearrowright u] * \mathsf{ptcb}(t) * \exists u' . (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \mathtt{CPUS}[u], \mathtt{CPUS}[u'], \mathtt{CPUS}[u'], \_) * (\sharp u > u')$
$* \mathsf{K}(bp, 4, \mathtt{CPUS}[u'] :: \mathtt{CPUS}[u] :: \mathtt{CPUS}[u'] :: \mathtt{CPUS}[u] :: w1 :: ret)\}$

```
    jmp         Llb_lb
Llb_swap:
    sw          a0,         0(sp)
    lw          a1,         8(sp)
    sw          a1,         4(sp)
```

$\{[t \circlearrowright u] * \mathsf{ptcb}(t) * \exists u' . (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \mathtt{CPUS}[u], \mathtt{CPUS}[u'], \mathtt{CPUS}[u'], \_) * (\sharp u > u')$
$* \mathsf{K}(bp, 4, \mathtt{CPUS}[u] :: \mathtt{CPUS}[u'] :: \mathtt{CPUS}[u'] :: \mathtt{CPUS}[u] :: w1 :: ret)\}$

Fig. 38. Ext. 3: Verification of loadbalance (part one)

$\{[t \circlearrowright u] * \mathsf{ptcb}(t) * \exists u' . (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \mathtt{CPUS}[u], \mathtt{CPUS}[u'], \mathtt{CPUS}[u'], \_) * (\sharp u > u')$
$* \mathsf{K}(bp, 4, cpua :: cpub :: \mathtt{CPUS}[u'] :: \mathtt{CPUS}[u] :: w1 :: ret)$
$* (\sharp u > u' \wedge cpua = \mathtt{CPUS}[u] \wedge cpub = \mathtt{CPUS}[u']) \mathbb{V} (\sharp u < u' \wedge cpua = \mathtt{CPUS}[u'] \wedge cpub = \mathtt{CPUS}[u])\}$

```
Llb_lb:
    lw    a1,   4(sp)
    lw    a0,   OFF_LOCK(a1)
    call  lock
    lw    a1,   8(sp)
    lw    a0,   OFF_LOCK(a1)
    call  lock
```

$\{[t \circlearrowright u] * \mathsf{ptcb}(t) * \exists u' . (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \mathtt{CPUS}[u], \mathtt{CPUS}[u'], \mathtt{CPUS}[u'], \_) * (\sharp u > u')$
$* \mathsf{K}(bp, 4, cpua :: cpub :: \mathtt{CPUS}[u'] :: \mathtt{CPUS}[u] :: w1 :: ret)$
$* (\sharp u > u' \wedge cpua = \mathtt{CPUS}[u] \wedge cpub = \mathtt{CPUS}[u']) \mathbb{V} (\sharp u < u' \wedge cpua = \mathtt{CPUS}[u'] \wedge cpub = \mathtt{CPUS}[u]) * \Sigma(u) * \Sigma(u')\}$

```
    lw    a1,   8(sp)
    lw    a0,   OFF_LOCK(a1)
    call  deq
    bz    v0,   Llb_end
    lw    a1,   12(sp)
    lw    a0,   OFF_LOCK(a1)
    mov   a1,   v0
    call  enq
    lw    a1,   8(sp)
    lw    a0,   OFF_LOCK(a1)
    call  unlock
    lw    a1,   4(sp)
    lw    a0,   OFF_LOCK(a1)
    call  unlock
```

$\{[t \circlearrowright u] * \mathsf{ptcb}(t) * (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \mathtt{CPUS}[u], \mathtt{CPUS}[u'], \mathtt{CPUS}[u'], \_) * (\sharp u > u')$
$* \mathsf{K}(bp, 4, \_ :: \_ :: \mathtt{CPUS}[u'] :: \mathtt{CPUS}[u] :: w1 :: ret)\}$

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

$\{[t \circlearrowright u] * \mathsf{ptcb}(t) * (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \mathtt{CPUS}[u], \mathtt{CPUS}[u'], \mathtt{CPUS}[u'], \_) * \mathsf{K}(bp, 10)\}$

```
    ret
```

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 $\mathsf{RThrd}(\cdots)$, 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 [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 to be practically applied to realistic OS code.

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. B. Ford and S. Susarla. Cpu inheritance scheduling. In *Proc. OSDI'96*, OSDI '96, pages 91–105, New York, NY, USA, 1996. ACM.
6. 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.
7. A. Gotsman and H. Yang. Modular verification of preemptive os kernels. In *Proc. ICFP'11*, pages 404–417, Tokyo, Japan, 2011. ACM.
8. Y. Guo, X. Feng, Z. Shao, and P. Shi. Modular verification of concurrent thread management (technical report). `http://kyhcs.ustcsz.edu.cn/~guoyu/sched/`, June 2012.
9. 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.
10. 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.
11. 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.
12. 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.
13. R. Love. *Linux Kernel Development (2nd Edition) (Novell Press)*. Novell Press, 2005.
14. M. K. McKusick and G. V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004.
15. Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. POPL'06*, pages 320–333, Jan. 2006.
16. 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.
17. P. W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
18. 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.
19. J. H. Saltzer and M. F. Kaashoek. *Principles of Computer System Design: An Introduction*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009.

20. 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.