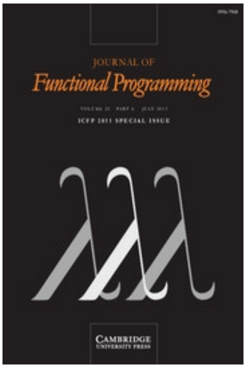

Modular verification of preemptive OS kernels

ALEXEY GOTSMAN and HONGSEOK YANG

Journal of Functional Programming / Volume 23 / Special Issue 04 / July 2013, pp 452 - 514
DOI: 10.1017/S0956796813000075, Published online: 29 October 2013

Link to this article: http://journals.cambridge.org/abstract_S0956796813000075

How to cite this article:
ALEXEY GOTSMAN and HONGSEOK YANG (2013). Modular verification of preemptive OS kernels. Journal of Functional Programming, 23, pp 452-514 doi:10.1017/S0956796813000075

JFP **23** (4): 452–514, 2013. © Cambridge University Press 2013
doi:10.1017/S0956796813000075

Modular verification of preemptive OS kernels

ALEXEY GOTSMAN
IMDEA Software Institute
(*e-mail:* alexey.gotsman@imdea.org)

HONGSEOK YANG
University of Oxford
(*e-mail:* Hongseok.Yang@cs.ox.ac.uk)

Abstract

Most major OS kernels today run on multiprocessor systems and are preemptive: it is possible for a process running in the kernel mode to get descheduled. Existing modular techniques for verifying concurrent code are not directly applicable in this setting: they rely on scheduling being implemented correctly, and in a preemptive kernel, the correctness of the scheduler is interdependent with the correctness of the code it schedules. This interdependency is even stronger in mainstream kernels, such as those of Linux, FreeBSD or Mac OS X, where the scheduler and processes interact in complex ways. We propose the first logic that is able to decompose the verification of preemptive multiprocessor kernel code into verifying the scheduler and the rest of the kernel separately, even in the presence of complex interdependencies between the two components. The logic hides the manipulation of control by the scheduler when reasoning about preemptable code and soundly inherits proof rules from concurrent separation logic to verify it thread-modularly. We illustrate the power of our logic by verifying an example scheduler, which includes some of the key features of the scheduler from Linux 2.6.11 challenging for verification.

1 Introduction

Developments in formal verification now allow us to consider the full verification of an operating system (OS) kernel, one of the most crucial components in any system today. Several recent projects have demonstrated that a formal verification of can tackle realistic OS kernels, such as a variant of the L4 microkernel (Klein *et al.*, 2009) and Microsoft's Hyper-V hypervisor (Cohen *et al.*, 2010). However, these projects only dealt with relatively small microkernels; tackling today's mainstream operating systems, such as Windows and Linux, remains a daunting task. A way to approach this problem is to verify OS kernels *modularly*, i.e., by considering each of their components in isolation. In this paper, we tackle a major challenge OS kernels present for modular reasoning—handling kernel preemption in a multiprocessor system. Most major OS kernels are designed to run with multiple CPUs and are *preemptive*: it is possible for a process running in the kernel mode to get descheduled. Reasoning about such kernels is difficult for the following reasons.

First of all, in a multiprocessor system several invocations of a system call may be running concurrently in a shared address space, so reasoning about the call needs to consider all possible interactions among them. This is a notoriously difficult problem; however, we now have a number of logics that can reason about concurrent code (O'Hearn, 2007; Gotsman *et al.*, 2007; Feng *et al.*, 2007a; Vafeiadis & Parkinson, 2007; Dinsdale-Young *et al.*, 2010; Cohen *et al.*, 2010; Dinsdale-Young *et al.*, 2013). The way the logics make verification tractable is by using *thread-modular* reasoning principles that consider every thread of computation in isolation under some assumptions about its environment and thus avoid direct reasoning about all possible interactions (Jones, 1983; Pnueli, 1985).

The problem is that all these logics can verify code only under so-called interleaving semantics, expressed by the well-known operational semantics rule:

$$\frac{C_k \longrightarrow C'_k}{C_1 \parallel \ldots \parallel C_k \parallel \ldots \parallel C_n \longrightarrow C_1 \parallel \ldots \parallel C'_k \parallel \ldots \parallel C_n}$$

This rule effectively assumes an abstract machine where every process C_k has its own CPU, whereas in an OS, the processes are multiplexed onto available CPUs by a scheduler, which is part of the OS kernel. Furthermore, in a preemptive kernel, the correctness of the scheduler is *interdependent* with the correctness of the rest of the kernel (which, in the following, we refer to as just the kernel). This is because, when reasoning about a system call implementation in a preemptive kernel, we have to consider the possibility of context-switch code getting executed at almost every program point. Upon a context switch, the state of the system call will be stored in kernel data structures and subsequently loaded for execution again, possibly on a different CPU. A bug in the context switch code can load an incorrect state of the system call implementation upon a context switch, and a bug in the system call can corrupt the scheduler's data structures. It is, of course, possible to reason about the kernel together with the scheduler as a whole, using one of the available logics. However, in a mainstream kernel, where kernel preemption is enabled most of the time, such reasoning would quickly become intractable.

Contributions. In this paper, we propose a logic that is able to decompose the verification of safety properties of preemptive OS code into verifying preemptable code and the scheduler (including the context-switch code) separately. This is the first logic that can achieve this in the presence of interdependencies between the scheduler and the kernel typical for mainstream OS kernels, such as those of Linux, FreeBSD and Mac OS X. The modularity of the logic is reflected in the structure of its proof system, which is partitioned into high-level and low-level parts. The high-level proof system verifies preemptable code assuming that the scheduler is implemented correctly (Section 5.2). It hides the complex manipulation of control by the context-switch code, which stores program counters of processes, describing their continuations, and jumps to one of them. In this way, the high-level proof system provides the illusion of an abstract machine where every process has its own virtual CPU—the control moves from one program point in the process code to the next without changing its state. This illusion is justified by verifying the scheduler code separately from the kernel in the low-level proof system (Section 5.3). Achieving this level of modularity requires us to cope with several technical challenges.

First, the setting of a preemptive OS kernel introduces an obligation to prove that the scheduler and the kernel do not corrupt each other's data structures. A common way to achieve this is by introducing the notion of *ownership* of memory areas: only the component owning an area of memory has the right to access it (Clarke *et al.*, 2001; Reynolds, 2002). A major difficulty of decomposing the verification of the mainstream OS kernels mentioned above lies in the fact that, in such kernels, there is no static address space separation between data structures owned by the scheduler and the rest of the kernel: the boundary between these changes according to a protocol for transferring the ownership of memory cells and permissions to access them in a certain way. For example, when an implementation of the `fork` system call asks the scheduler to make a new process runnable, the scheduler usually gains the ownership of the process descriptor provided by the system call implementation.

To deal with this, we base our proof systems on concurrent separation logic (O'Hearn, 2007), which we recap in Section 4. The logic allows us to track the dynamic memory partitioning between the scheduler and the rest of the kernel and prohibit memory accesses that cross the partitioning boundary. For example, assertions in the high-level proof system talk only about the memory belonging to the kernel and completely hide the memory belonging to the scheduler. A *frame property*, validated by concurrent separation logic, implies that in this case any memory not mentioned in the assertions, e.g., the memory belonging to the scheduler, is guaranteed not to be changed by the kernel. A realistic interface between the scheduler and the kernel is supported by proof rules for ownership transfer of logical assertions between the two components, describing permissions to access memory cells.

Using an off-the-shelf logic, however, is not enough to prove the correctness of a scheduler, as this requires domain-specific reasoning: e.g., we need to be able to check that a scheduler restores the state of a preempted process correctly when it resumes the process. To this end, the low-level proof system for reasoning about schedulers includes special $\mathsf{Process}$ assertions about the continuation of every OS process the scheduler manages, describing the states from which it can be safely resumed. A novelty of these assertions is the semantics of the separating conjunction connective on them, which treats the assertions affinely and allows us to interpret them as exclusive permissions to schedule the corresponding processes. This enables reasoning about scheduling on multiprocessors, as it allows checking that the scheduler invocations on different CPUs coordinate decisions about process scheduling correctly. Another interesting feature of $\mathsf{Process}$ assertions is that they describe only the part of the process state the scheduler is supposed to access and hide the rest; this ensures that the scheduler indeed cannot corrupt the latter. In this our $\mathsf{Process}$ assertions are similar to abstract predicates (Parkinson & Bierman, 2005).

Even though a scheduler is supposed to provide an illusion of running on a dedicated *virtual* CPU to every process, in practice, some features available to the kernel code can break through this abstraction: e.g., a process can disable preemption and become aware of the *physical* CPU on which it is currently executing. For example, some OS kernels use this to implement *per-CPU data structures* (Bovet & Cesati, 2005)—arrays indexed by CPU identifiers such that a process can only access an entry in an array when it runs on the corresponding CPU. We demonstrate that our approach can deal with such implementation exposures by extending the high-level proof system for the kernel code with axioms that allow reasoning about per-CPU data structures (Section 7).

In reasoning about mainstream operating systems, assertions describing the state transferred between the scheduler and the kernel can be complicated. The resulting ownership transfers make even formalising the notion of scheduler correctness non-trivial, as they are difficult to accommodate in an operational semantics of the abstract machine with one CPU per process the scheduler is supposed to implement (Gotsman *et al.*, 2011). We resolve this problem in the following way. In our logic, the desired properties of OS code are proved with respect to the abstract machine using the high-level proof system; the low-level system then relates the concrete and the abstract machines. However, proofs in neither of the two systems are interpreted with respect to any semantics alone. Instead, our soundness statement (Section 8) interprets a proof of the kernel in the high-level system and a proof of the scheduler in the low-level one together with respect to the semantics of the concrete machine. To this end, the statement has to construct a global property of the machine from local assertions about OS components in a non-trivial way.

Even though all of the OS verification projects carried out so far had to deal with a scheduler (see Section 9 for a discussion), to our knowledge they have not produced methods for handling practical multiprocessor schedulers with a complicated scheduler/ kernel interface. We illustrate the power of our logic by verifying an example scheduler (Sections 2.2 and 6), which includes some of the key features of the scheduler from Linux 2.6.11 exhibiting the issues mentioned above.

Limitations. Our goal here is not to verify an industrial-strength preemptive OS kernel—such an endeavour is beyond the scope of a single paper. Rather, we develop *principles* of how a given logic for verifying concurrent programs can be extended to verify preemptive kernel code with *real-world features*. These principles can then be used in verification projects that target different operating systems. To communicate the proposed principles cleanly and understandably, we present our results in a simplified setting:

- Instead of a realistic processor, such as x86, we consider an idealised machine (Sections 2.1 and 3).
- Since we are primarily interested in interactions of components within an OS kernel, our machine does not make a distinction between the user mode and the kernel mode.
- We base our logic for verifying OS kernels on one of the simplest logics for concurrent code—concurrent separation logic (O'Hearn, 2007). This logic would not be able to handle complicated concurrency mechanisms employed in modern OS kernels (Bovet & Cesati, 2005). However, as we argue in Section 8.2, our development can be adapted to its more advanced derivatives (Feng *et al.*, 2007a; Gotsman *et al.*, 2007; Vafeiadis & Parkinson, 2007; Dinsdale-Young *et al.*, 2010; Dinsdale-Young *et al.*, 2013).
- Since we concentrate on modular reasoning about concurrency and preemptive scheduling, our logic provides only rudimentary means of modular reasoning about sequential code and, in particular, procedures. We discuss ways of addressing this problem in Section 10.
- We consider scheduling interfaces providing only the basic services—context switch and process creation. We discuss how our logic can be extended to schedulers with more elaborate interfaces in Sections 5.5 and 10.
- Due to our focus on scheduling, we ignore many other aspects of an OS kernel, such as virtual memory and interrupts not related to scheduling.
- Our logic is designed for proving safety properties only. Proof methods for liveness properties usually rely on modular methods for safety properties. Thus, our logic is a prerequisite for attacking liveness in the future.

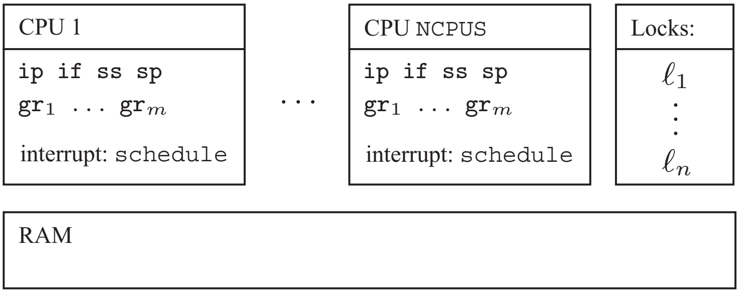

Fig. 1. The target machine.

Even though we develop our logic in this simplified setting, we hope that its modular nature makes it more likely that it will compose with logics for features and abstractions that we currently do not handle. A preliminary indication of this is our ability to deal with per-CPU data structures, despite the fact that these break through the abstraction implemented by a scheduler.

2 Informal development

We first explain our results informally, sketching the machine we use for formalising them (Section 2.1), illustrating the challenges of reasoning about schedulers by an example (Section 2.2) and describing the approach we take in our program logic (Section 2.3).

2.1 Example machine

We formalise our results for a simple machine, defined in Section 3. Here, we present it informally to the extent needed for understanding the rest of this section. We summarise its features in Figure 1.

The machine has multiple CPUs, identified by integers from 1 to NCPUS, communicating via the shared memory. We assume that the program the machine executes is stored separately from the heap and may not be modified; its commands are identified by labels. For simplicity, we also assume that programs can synchronise using a set of built-in locks (in a real system they would be implemented as spin-locks). Every CPU has a single interrupt, with its handler located at a distinguished label schedule (the same for all CPUs). A scheduler can use the interrupt to trigger a context switch. There are four special-purpose registers, ip, if, ss and sp, and m general-purpose ones, $\mathtt{gr}_1,\dots,\mathtt{gr}_m$. The ip register is the instruction pointer. The if register controls interrupts: they are disabled on the corresponding CPU when if stores zero, and enabled otherwise. As if affects only one CPU, we might have several instances of the scheduler code executing in parallel on different CPUs. Upon an interrupt, the CPU sets if to 0, which prevents nested interrupts. The ss register keeps the starting address of the stack, and sp points to the top of the stack, i.e., its first free slot. The stack grows upwards, so we always have $\mathtt{ss} \leqslant \mathtt{sp}$. As we noted before, our machine does not make a distinction between the user mode and the kernel mode—all processes can potentially access all available memory and execute all commands.

The machine executes programs in a minimalistic assembly-like programming language. It is described in full in Section 3; for now it suffices to say that the language includes standard commands for accessing registers and memory, and the following special ones:

- `lock`(ℓ) and `unlock`(ℓ) acquire and release the lock ℓ.
- `savecpuid`(e) stores the identifier of the CPU executing it at the address e.
- `call`(l) is a call to the function that starts at the label l. It pushes the label of the next instruction in the program and the values of the general-purpose registers onto the stack, and jumps to the label l.
- `icall`(l) behaves the same as `call`(l), except that it also disables interrupts by modifying the `if` register.
- `ret` is the return command. It pops the return label and the saved general-purpose registers off the stack, updates the registers with the new values, and jumps to the return label.
- `iret` is a variant of `ret` that additionally enables interrupts.

When the `if` register is set to a non-zero value, an interrupt can fire at any time. This has the same effect as executing `icall`(`schedule`).

2.2 Motivating example

The challenge. Figures 2–3 present an implementation of the scheduler that we use as a running example. Our goal is to be able to verify safety properties of OS processes managed by this scheduler using off-the-shelf concurrency logics, i.e., as though every process has its own virtual CPU. To show what this entails, consider the piece of code in Figure 4, which could be a part of a system call implementation in the kernel (we explain its proof later). The code removes some of the nodes from a cyclic doubly-linked list with a sentinel head node pointed to by `request_queue`. Here and in the following, we use some library functions: e.g., `remove_node` deletes a node from the doubly-linked list it belongs to. We assume that the code can be called concurrently by multiple processes, and thus protect the list with a lock `request_lock`: the list can only be accessed by the process that holds this lock.

There are a number of properties we might want to prove about this code: e.g., that pointer manipulations done by the invocation of `remove_node` in line 25 in Figure 4 do not overwrite a memory cell storing critical information elsewhere in the kernel, or that the doubly-linked list shape of `request_queue` is preserved. Modern concurrency logics can prove such properties by considering every process in isolation. Namely, every assertion in Figure 4 describes information about the state of the program relevant to the process executing the code. These assertions are justified using essentially sequential reasoning, and in particular, using the classical proof rule for sequential composition:

$$\frac{\{P_1\}\, C_1\, \{P_2\} \qquad \{P_2\}\, C_2\, \{P_3\}}{\{P_1\}\, C_1;C_2\, \{P_3\}} \tag{1}$$

We would like such reasoning (and the proof in Figure 4) to be sound even when the code is managed by the scheduler in Figures 2–3. Hence, we need to be able to ignore the fact that the control flow of the code in Figure 4 can jump to the schedule function in Figure 2 at any time, with the state of the former stored in kernel data structures and loaded from them again later. Furthermore, we need to achieve this even though the scheduler and the system call implementation execute in a shared address space and can thus potentially access each other's data structures.

Example scheduler. The example scheduler in Figures 2–3 includes some of the key features of the scheduler from Linux 2.6.11 (Bovet & Cesati, 2005) that are challenging for verification, as detailed below.[1]

The scheduler's interface consists of two functions: schedule and create. The former is called as the interrupt handler or directly by a process and is responsible for switching the process running on the CPU and migrating processes between CPUs. The latter can be called by the kernel implementation of the fork system call and is responsible for inserting a newly created process into the scheduler's data structures, thereby making it runnable. Both functions are called by processes using the icall command that disables interrupts, so that the scheduler routines always execute with interrupts disabled.

Programming language. Even though we formalise our results for a machine executing a minimalistic programming language, we present the example in C. We now explain how a C program, such as the one in Figures 2–3, is mapped to our machine.

We assume that every global variable x is allocated at a fixed address &x in memory. Local variable declarations allocate local variables on the stack in the activation records of the corresponding procedures; these variables are then addressed via the sp register. When the variables go out of scope, they are removed from the stack by decrementing the sp register. The general-purpose registers are used to store intermediate values while computing complex expressions. In our C programs, we allow the ss and sp registers to be accessed directly as _ss and _sp. Function calls and returns are implemented using the call and ret commands of the machine. By default, parameters and return values are passed via the stack; in particular, a zero-filled slot for a return value is allocated on the stack before calling a function. Parameters of functions annotated with _regparam (such as create, line 64 in Figure 3) are passed via registers. We assume macros lock, unlock, savecpuid and iret for the corresponding machine commands.

Data structures. Every process is associated with a process descriptor of type Process. Its prev and next fields are used by the scheduler to connect descriptors into doubly-linked lists of processes it manages (*runqueues*). The scheduler uses per-CPU runqueues with dummy head nodes pointed to by the entries in the runqueue array. These are protected by the locks in the runqueue_lock array. The entries in the current array point to the descriptors of the processes running on the corresponding CPUs; these descriptors are not members of any runqueue. Thus, every process descriptor is either in the current array or in some runqueue. Note that every CPU always has at least one process to run—the

[1] We took an older version of the Linux kernel (from 2005) as a reference because its scheduler uses simpler data structures. Newer versions use more efficient data structures (Love, 2010) that would only complicate our running example without adding anything interesting.

```
#define StackSize ... // size of the stack
#define FORK_FRAME sizeof(Process*)
#define SCHED_FRAME sizeof(Process*)+sizeof(int)

struct Process {
  Process *prev, *next;
  word kernel_stack[StackSize];
  word *saved_sp;
  int timeslice;
};

Lock *runqueue_lock[NCPUS];
Process *runqueue[NCPUS];
Process *current[NCPUS];

void init() {
  for (int cpu = 0; cpu < NCPUS; cpu++) {
    Process* dummy = alloc(sizeof(Process));
    Process* process0 = alloc(sizeof(Process));
    dummy->prev = dummy->next = dummy;
    process0->timeslice = SCHED_QUANTUM;
    ... // initialise the stack of process0
    runqueue[cpu] = dummy;
    current[cpu] = process0;
  }
}

void schedule() {
  int cpu;
  Process *old_process;
  savecpuid(&cpu);
  load_balance(cpu);
  old_process = current[cpu];
  ... // update the timeslice of old_process
  if (old_process->timeslice) iret();
  old_process->timeslice = SCHED_QUANTUM;
  lock(runqueue_lock[cpu]);
  insert_node_after(runqueue[cpu]->prev, old_process);
  current[cpu] = runqueue[cpu]->next;
  remove_node(current[cpu]);
  old_process->saved_sp = _sp;
  _sp = current[cpu]->saved_sp;
  savecpuid(&cpu);
  _ss = current[cpu]->kernel_stack;
  unlock(runqueue_lock[cpu]);
  iret();
}
```

Fig. 2. The example scheduler (continued in Figure 3).

```
void load_balance(int cpu) {
  int cpu2;
  Process *proc;
  if (random(0, 1)) return;
  do { cpu2 = random(0, NCPUS-1); } while (cpu == cpu2);
  if (cpu < cpu2) {
    lock(runqueue_lock[cpu]); lock(runqueue_lock[cpu2]); }
  else { lock(runqueue_lock[cpu2]); lock(runqueue_lock[cpu]); }
  if (runqueue[cpu2]->next != runqueue[cpu2]) {
    proc = runqueue[cpu2]->next;
    remove_node(proc);
    insert_node_after(runqueue[cpu], proc);
  }
  unlock(runqueue_lock[cpu]);
  unlock(runqueue_lock[cpu2]);
}

_regparam void create(Process *new_process) {
  int cpu;
  savecpuid(&cpu);
  new_process->timeslice = SCHED_QUANTUM;
  lock(runqueue_lock[cpu]);
  insert_node_after(runqueue[cpu], new_process);
  unlock(runqueue_lock[cpu]);
  iret();
}

int fork() {
  Process *new_process;
  new_process = alloc(sizeof(Process));
  memcpy(new_process->kernel_stack, _ss, StackSize);
  new_process->saved_sp = new_process->kernel_stack+
      _sp-_ss-FORK_FRAME+SCHED_FRAME;
  _icall create(new_process);
  return 1;
}
```

Fig. 3. The example scheduler (continued).

one in the corresponding slot of the `current` array. Every process has its own kernel stack of a fixed size `StackSize`, represented by the `kernel_stack` field of its descriptor. When a process is preempted, the `saved_sp` field is used to save the value of the stack pointer register `sp`; the other registers are saved on the stack. Finally, while a process is running, the `timeslice` field gives the remaining time from its scheduling time quantum and is periodically updated by the scheduler. The `init` function in Figure 2 sketches code that could be used to initialise the scheduler data structures.

Apart from the data structures described above, a realistic kernel would contain many others not related to scheduling, including additional fields in process descriptors. The kernel data structures reside in the same address space as the ones belonging to the scheduler;

```
#define StackSize ... // size of the stack
struct Request {
  Request *prev, *next;
  int data;
};
Request *request_queue; // a cyclic doubly−linked list with a sentinel node
Lock *request_lock; // protects the list

...
Request *req, *tmp;
$\{req \Vdash \mathsf{sp}..(\mathsf{ss}+\mathtt{StackSize}-1) \mapsto \_ \wedge \mathsf{sp}=\mathsf{ss}+2\cdot\mathtt{sizeof}(\mathtt{Request}*)\}$
lock(request_lock);
$\{req \Vdash \exists x,y,z.\ \&\mathtt{request\_queue} \mapsto z * z.\mathtt{prev} \mapsto y * z.\mathtt{next} \mapsto x * z.\mathtt{data} \mapsto \_ *$
 $\mathsf{dll}_\Lambda(x,z,z,y) * \mathsf{locked}(\mathtt{request\_lock}) *$
 $\mathsf{sp}..(\mathsf{ss}+\mathtt{StackSize}-1) \mapsto \_ \wedge \mathsf{sp}=\mathsf{ss}+2\cdot\mathtt{sizeof}(\mathtt{Request}*)\}$
req = request_queue->next;
$\{req \Vdash \exists y,z.\ \&\mathtt{request\_queue} \mapsto z * z.\mathtt{prev} \mapsto y * z.\mathtt{next} \mapsto req * z.\mathtt{data} \mapsto \_ *$
 $\mathsf{dll}_\Lambda(req,z,z,y) * \mathsf{locked}(\mathtt{request\_lock}) *$
 $\mathsf{sp}..(\mathsf{ss}+\mathtt{StackSize}-1) \mapsto \_ \wedge \mathsf{sp}=\mathsf{ss}+2\cdot\mathtt{sizeof}(\mathtt{Request}*)\}$
while (req != request_queue) {
  $\{req \Vdash \exists x,y,z,u,v.\ \&\mathtt{request\_queue} \mapsto z * z.\mathtt{prev} \mapsto v * z.\mathtt{next} \mapsto x * z.\mathtt{data} \mapsto \_ *$
   $\mathsf{dll}_\Lambda(x,z,req,y) * req.\mathtt{prev} \mapsto y * req.\mathtt{next} \mapsto u * req.\mathtt{data} \mapsto \_ * \mathsf{dll}_\Lambda(u,req,z,v) *$
   $\mathsf{locked}(\mathtt{request\_lock}) * \mathsf{sp}..(\mathsf{ss}+\mathtt{StackSize}-1) \mapsto \_ \wedge$
   $\mathsf{sp}=\mathsf{ss}+2\cdot\mathtt{sizeof}(\mathtt{Request}*)\}$
  if (stale_data(req->data)) remove_node(req);
  $\{req \Vdash \exists x,y,z,u,v.\ \&\mathtt{request\_queue} \mapsto z * z.\mathtt{prev} \mapsto v * z.\mathtt{next} \mapsto x * z.\mathtt{data} \mapsto \_ *$
   $\mathsf{dll}_\Lambda(x,z,u,y) * req.\mathtt{prev} \mapsto y * req.\mathtt{next} \mapsto u * req.\mathtt{data} \mapsto \_ * \mathsf{dll}_\Lambda(u,y,z,v) *$
   $\mathsf{locked}(\mathtt{request\_lock}) * \mathsf{sp}..(\mathsf{ss}+\mathtt{StackSize}-1) \mapsto \_ \wedge$
   $\mathsf{sp}=\mathsf{ss}+2\cdot\mathtt{sizeof}(\mathtt{Request}*)\}$
  tmp = req;
  req = req->next;
  free(tmp);
  $\{req \Vdash \exists x,y,z,v.\ \&\mathtt{request\_queue} \mapsto z * z.\mathtt{prev} \mapsto v * z.\mathtt{next} \mapsto x * z.\mathtt{data} \mapsto \_ *$
   $\mathsf{dll}_\Lambda(x,z,req,y) * \mathsf{dll}_\Lambda(req,y,z,v) * \mathsf{locked}(\mathtt{request\_lock}) *$
   $\mathsf{sp}..(\mathsf{ss}+\mathtt{StackSize}-1) \mapsto \_ \wedge \mathsf{sp}=\mathsf{ss}+2\cdot\mathtt{sizeof}(\mathtt{Request}*)\}$
}
$\{req \Vdash \exists x,y,z.\ \&\mathtt{request\_queue} \mapsto z * z.\mathtt{prev} \mapsto y * z.\mathtt{next} \mapsto x * z.\mathtt{data} \mapsto \_ *$
 $\mathsf{dll}_\Lambda(x,z,z,y) * \mathsf{locked}(\mathtt{request\_lock}) *$
 $\mathsf{sp}..(\mathsf{ss}+\mathtt{StackSize}-1) \mapsto \_ \wedge \mathsf{sp}=\mathsf{ss}+2\cdot\mathtt{sizeof}(\mathtt{Request}*)\}$
unlock(request_lock);
$\{req \Vdash \mathsf{sp}..(\mathsf{ss}+\mathtt{StackSize}-1) \mapsto \_ \wedge \mathsf{sp}=\mathsf{ss}+2\cdot\mathtt{sizeof}(\mathtt{Request}*)\}$
```

Fig. 4. An example system call part. The assertions are explained in Section 4.

thus, while verifying the OS, we have to prove that the two components do not corrupt each other's data structures.

The `schedule` function. According to the semantics of our machine, when `schedule` starts executing, interrupts are disabled and the previous values of `ip` and the general-purpose registers are saved on the top of the stack. The scheduler uses the empty slots on the stack of the process it has preempted to store activation records of its procedures

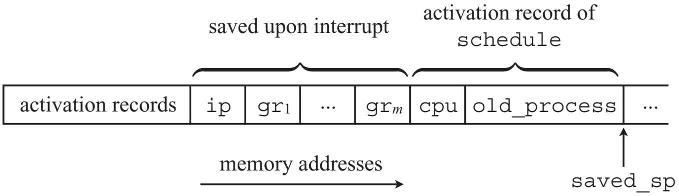

Fig. 5. The invariant of the stack of a preempted process.

and thus expects the kernel to leave enough of these. Intuitively, while a process is running, only this process has the right to access its stack, i.e., *owns* it. When the scheduler preempts the process, the right to access the empty slots on the stack (their *ownership*) is transferred to the scheduler. When the scheduler returns the control to this process, it transfers the ownership of the stack slots back. This is one example of ownership transfer we have to reason about.

The schedule function first calls load_balance (line 32 in Figure 2), which migrates processes between CPUs to balance the load; we describe it below. It then updates the timeslice of the currently running process, and if it becomes zero, proceeds to schedule another one (line 35); here we abstract from the particular way the timeslice is updated. The processes are scheduled in a round-robin fashion; thus, the function inserts the current process at the end of the local runqueue (line 38) and dequeues the process at the front of the runqueue, making it current (line 40).[2] The former is done using a library function insert_node_after, which inserts the node given as its second argument after the list node given as its first argument. The schedule function also refills the scheduling quantum of the process being descheduled (line 36). The runqueue manipulations are done with the corresponding lock held (lines 37 and 45). Note that in a realistic OS choosing a process to run would be more complicated, but still based on scheduler-private data structures protected by runqueue locks.

To save the state of the process being preempted, schedule copies sp into the saved_sp field of the process descriptor (line 41 in Figure 2). This field, together with the kernel_stack of the process, forms its saved state. The stack of a preempted process contains the activation records of functions called before the process was preempted, the label of the instruction to resume the process from and the values of general-purpose registers, saved upon the interrupt, as well as the activation record of schedule, as shown in Figure 5. This invariant holds for descriptors of all preempted processes.

The actual context switch is performed by the assignment to sp (line 42), which switches the current stack to another one satisfying the invariant in Figure 5. Since this changes the activation record of schedule, the function has to update the cpu variable (line 43), which lets it then retrieve and load the new value of ss (line 44). The iret command at the end of schedule (line 46) loads the values of the registers stored on the stack and enables interrupts, thus completing the context switch.

[2] Actually, in Linux 2.6.11 the descriptor of the current process stays in the runqueue. We dequeue it because this simplifies the following formal treatment of the example.

The `load_balance` function checks if the CPU given as its parameter is underloaded and, if it is the case, tries to migrate a process from another CPU to this one. The particular way the function performs the check and chooses the process is irrelevant for our purposes, and is thus abstracted by a random choice (line 50 in Figure 3). To migrate a process, the function chooses a runqueue to steal a process from (line 51) and locks it together with the current runqueue in the order determined by the corresponding CPU identifiers, to avoid deadlocks (lines 52–54). The function then removes one process from the victim runqueue (line 57), if it is non-empty (line 55), and inserts the process into the runqueue of the CPU it runs on (line 58). Note that two concurrent scheduler invocations executing `load_balance` on different CPUs may try to access the same runqueue. While verifying the scheduler, we have to ensure that they synchronise their accesses correctly. We also need to deal with the fact that, due to `load_balance`, processes cannot be tied to a CPU statically.

The `create` function inserts the descriptor of a newly created process with the address given as its parameter into the runqueue of the current CPU. We pass the parameter via a register, as this simplifies the following treatment of the example. The descriptor must be initialised like that of a preempted process, and hence its stack must satisfy the invariant in Figure 5. Upon a call to `create`, the ownership of the descriptor is transferred from the kernel to the scheduler. The `create` function must be called using `icall`, which disables interrupts; if interrupts were enabled, `schedule` could be called while `create` holds the lock for the current runqueue, resulting in a deadlock.

The `fork` function is formally not part of the scheduler. It illustrates how the rest of the kernel can use `create` to implement a common system call that creates a clone of the current process. This function allocates a new descriptor (line 76), copies the stack of the current process to it (line 77) and initialises the stack as expected by `create` (Figure 5). This amounts to discarding the topmost activation record of `fork` and pushing a fake activation record of `schedule` (line 78). We do not initialise the latter record, since `schedule` refreshes the values of the variables (line 43) when it receives control. Note that the values of registers in the initial state of the new process have been saved on the stack upon the call to `fork`. Since stack slots for return values are initialised with zeros, this is what `fork` in the child process will return; we return 1 in the parent process.

The need for modularity. We could try to verify the scheduler and the rest of the kernel (including, say, the system call in Figure 4) as a whole, modelling every CPU as a process in one of the existing program logics for concurrency (O'Hearn, 2007; Gotsman *et al.*, 2007; Feng *et al.*, 2007a; Vafeiadis & Parkinson, 2007; Dinsdale-Young *et al.*, 2010; Cohen *et al.*, 2010; Dinsdale-Young *et al.*, 2013). However, in this case our proofs would have to consider the possibility of the control flow going from any statement in a process to the `schedule` function, and from there to any other process. Thus, in reasoning about the system call implementation in Figure 4 we would end up having to reason explicitly about invariants and actions of both `schedule` and all other processes, making the reasoning non-modular and, most likely, intractable. In the rest of the paper, we propose a logic that avoids this pitfall.

2.3 Approach

Before presenting our logic for preemptive kernels in detail, we give an informal overview of the reasoning principles behind it. The goal of the logic is to reason about the kernel and the scheduler separately. Following previous work on OS verification (Feng *et al.*, 2008b; Yang & Hawblitzel, 2010), our logic uses different proof systems for this purpose: the high-level one for the kernel and the low-level one for the scheduler.

Modular reasoning via memory partitioning. The first challenge we have to deal with in separating the reasoning about the kernel and the scheduler is the fact that they share the same address space. To this end, our logic partitions the memory into two disjoint parts. The memory cells in each of the parts are *owned* by the corresponding component, meaning that only this component can access them. In our running example, the runqueues from Figure 2 will belong to the scheduler, and the request queue from Figure 4 to the kernel. It is important to note that this partitioning does not exist in the semantics, but is enforced by proofs in the logic to enable modular reasoning about the system. Modular reasoning becomes possible because, while reasoning about one component, one does not have to consider the memory partition owned by the other, since it cannot influence the behaviour of the component. An important feature of our logic, required for handling schedulers from mainstream kernels, is that the memory partitioning is not required to be static: the logic permits ownership transfer of memory cells between the areas owned by the scheduler and the kernel according to an axiomatically defined interface. For example, in reasoning about the scheduler of Section 2.2, the logic permits the transfer of the descriptor for a new process from the kernel to the scheduler at a call to `create`; this descriptor then becomes part of a runqueue owned by the scheduler. Of course, assigning the ownership of parts of memory to OS components requires checking that a component does not access the memory it does not own. To this end, our logic implements a form of rely-guarantee reasoning between the scheduler and the kernel, where one component assumes that the other does not touch its memory partition and provides well-formed pieces of memory at ownership transfer points.

Concurrent separation logic. Our logic is based on concurrent separation logic (O'Hearn, 2007), which we recap in Section 4. In particular, this logic provides us with means for modular reasoning within a given component, i.e., either among concurrent OS processes or concurrent scheduler invocations on different CPUs. The choice of concurrent separation logic was guided by the convenience of presentation; see Section 8 for a discussion of how more advanced logics can be integrated. However, the use of a version of separation logic is crucial, because we inherently rely on the *frame property* validated by the logic: the memory that is not mentioned by assertions in a proof of a command is guaranteed not to be changed by it. As we have noted, while reasoning about a component, we consider only the memory partition belonging to it. Hence, by the frame property we automatically know that the component cannot modify the others. This makes it easy to carry out the above-mentioned rely-guarantee reasoning between the scheduler and the kernel: one does not need to state assumptions about one component not

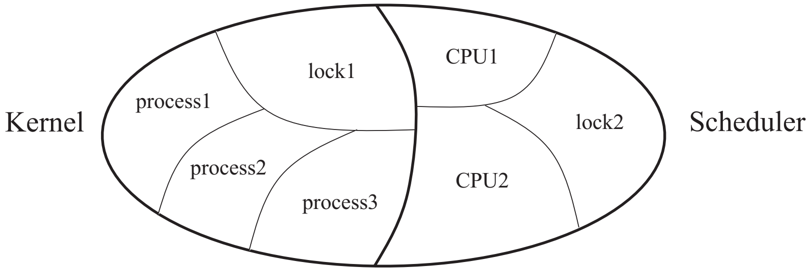

Fig. 6. The partitioning of the system state enforced by the logic. The memory is partitioned into two parts, owned by the scheduler and the kernel, respectively. The memory of each component is further partitioned into parts local to processes or scheduler invocations on a given CPU, and parts protected by locks.

modifying the memory of the other explicitly, as they will be automatically validated by the logic.

Concurrent separation logic lets us achieve modular reasoning within a given component by further partitioning the memory owned by it into disjoint process-local parts (one for each process or scheduler invocation on a given CPU) and protected parts (one for each free lock). A process-local part can only be accessed by the corresponding process or scheduler invocation, and a lock-protected part only when the process holds the lock. The resulting partitioning of the system state is illustrated in Figure 6. The frame property guarantees that a process cannot access the partition of the heap belonging to another one. To reason modularly about parts of the state protected by locks, the logic associates with every lock an assertion—its *lock invariant*—that describes the part of the state it protects. Lock invariants restrict how processes can change the protected state and, hence, allow reasoning about them in isolation. For example, in the program from Figure 4, the invariant of `request_lock` can state that it protects the `request_queue` variable and the doubly-linked list it identifies.

Scheduler-agnostic verification of kernel code. The high-level proof system (Section 5.2) reasons about preemptable code assuming an abstract machine where every process has its own virtual CPU with a dedicated set of registers. It relies on the partitioned view of memory described above to hide the state of the scheduler, with all the remaining state split among processes and locks accessible to them, as illustrated in Figure 7. We have primed process identifiers in the figure to emphasise that the state of the process can be represented differently in the abstract and physical machines: for example, if a process is not running, the values of its local registers can be stored in scheduler-private data structures, rather than in CPU registers.

Apart from hiding the state of the scheduler, the high-level system also hides the complex manipulation of the control flow performed by its context-switch code: the proof system assumes that the control moves from one point in the process code to the next without changing its state, ignoring the possibility of the context-switch code getting executed upon an interrupt. This is expressed by handling sequential composition in the proof system essentially using the standard rule (1) from Hoare logic. Explicit calls to the scheduler are treated as if they were executed atomically.

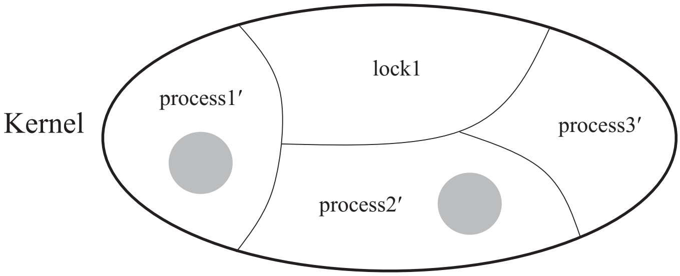

Fig. 7. The state of the abstract system with one virtual CPU per process. Process identifiers are primed to emphasise that the state of the process can be represented differently in the abstract and physical machines (cf. Figure 6). Dark regions illustrate the parts of process state that are tracked by a scheduler invocation running on a particular physical CPU.

Technically, the proof system is a straightforward adaptation of concurrent separation logic, which is augmented with proof rules axiomatising the effect of scheduler routines explicitly called by processes. The novelty here is that we can use such a scheduler-agnostic logic in this context at all. This is made possible by verifying the scheduler implementation using the low-level proof system (Section 5.3).

Proving schedulers correct. Intuitively, a correct scheduler provides an illusion of the above-mentioned abstract machine with one CPU per process, but what formal obligations does this entail? First, a process should not notice any effects of being preempted and then scheduled again: whenever the scheduler sets up a process to run on a CPU, it has to restore the state of the CPU registers to the one the process had last time it was preempted. The low-level proof system ensures this by recording these values in a special predicate Process, which can be viewed as an assertion about the continuation of a process describing the states from which it can be safely resumed.

In more detail, when a process is preempted and the control is given to the context-switch routine of the scheduler (`schedule` in the example from Section 2.2), a Process predicate recording the current values of the CPU registers appears in its precondition. When the context-switch routine terminates and the control is given to the process it resumes, the proof system requires the postcondition of the routine to exhibit a Process predicate with register values equal to the ones loaded onto the CPU. Roughly speaking, we thus require the following judgement to hold of the context-switch routine `schedule`:

$$\{\exists \vec{r}.\, \mathsf{Process}(\vec{r}) \wedge \vec{\mathtt{r}} = \vec{r} \ldots\}\ \mathtt{schedule}\ \{\exists \vec{r'}.\, \mathsf{Process}(\vec{r'}) \wedge \vec{\mathtt{r}} = \vec{r'} \ldots\}, \qquad (2)$$

where $\vec{\mathtt{r}}$ is the vector of CPU register names. The Process predicate in the postcondition may correspond to a different process than the predicate in the precondition. For example, when the scheduler from Section 2.2 preempts a process and links its descriptor into a runqueue, assertions about the runqueue in the proof system can record the corresponding Process predicate with register values equal to the ones stored in the descriptor. This predicate can then be used to establish the postcondition of the context-switch routine when it decides to schedule the process again.

In the case of multiprocessors, ensuring that the scheduler preserves the state of a preempted process is not enough for it to be correct. The scheduler should also be prevented from duplicating processes at will: it would be incorrect to preempt one process and then schedule two copies of it on two CPUs at the same time. To check that this does not happen, the proof system interprets a Process predicate as not merely recording a process state, but serving as an exclusive permission for the scheduler invocation owning it to schedule the corresponding process. Technically, it treats Process predicates affinely, prohibiting their duplication. Judgement (2) is then interpreted as stating that the scheduler gets the ownership of a Process predicate when it preempts a process and gives it up when scheduling that process again. This ensures that, at any time, only a scheduler invocation on a single CPU can own a Process predicate for a given process and, hence, can schedule it. In terms of Figure 6, a Process predicate can only belong to one partition in the scheduler-owned memory at a time.

We note that the problem of interdependence between the correctness of the scheduler and the rest of the kernel that we address in this paper also arises in preemptive *uniprocessor* kernels. The Process predicate also allows us to verify schedulers on uniprocessors; however, its affine treatment described above is not relevant in this case.

Assertions of the low-level proof system can be thought of as relating the states of the concrete machine and the abstract one the scheduler is supposed to implement. An important feature of our logic is that this relation is *local*, in the sense that it does not describe the whole state of the two machines. Namely, since we use concurrent separation logic to reason about concurrent execution of scheduler invocations on different CPUs, an assertion in the low-level proof system describes only the state owned by a scheduler invocation on a particular CPU (e.g., the region marked CPU1 in Figure 6). Similarly, the Process predicates describe only the registers of the processes a scheduler invocation has permission to schedule (shown by the dark regions in Figure 7), but not the memory they own. Since the assertions about the scheduler cannot talk about the memory owned by kernel processes, the frame property automatically ensures that the scheduler cannot corrupt it.

Soundness. We establish the soundness of our logic using an approach atypical for the kind of setting we consider. Since a scheduler is supposed to provide an illusion of an abstract machine with one CPU per process, to formalise its correctness, we could define an operational semantics of such an abstract machine and prove that it reproduces any behaviour of the concrete machine with the scheduler, thus establishing a *refinement* between the two machines. However, for realistic OS schedulers defining a semantics for the abstract machine that a scheduler implements is difficult. This is because, in reasoning about mainstream operating systems, the state transferred between the scheduler and the kernel can be described by complicated assertions; in such cases, defining ownership transfer operationally is difficult (we discuss this further in Section 8).

To resolve this problem with stating soundness, we do not define the semantics of the abstract machine operationally; instead, we describe its behaviour only by the high-level proof system, thus giving it an axiomatic semantics. As expected, the low-level proof system is used to reason about the correspondence between the concrete and the abstract machines, with its assertions relating their states. However, proofs in neither of the two systems are interpreted with respect to any semantics alone: our soundness statement

(Section 8) interprets a proof of the kernel in the high-level system and a proof of the scheduler in the low-level one together with respect to the semantics of the concrete machine. Thus, instead of relating sets of executions of the two machines, like in the classical refinement, the soundness statement relates logical statements about the abstract machine (given by high-level proofs) to logical statements about the concrete one (given by a constraint on concrete states). Note that in this case the soundness statement for the logic does not yield a semantic statement of correctness for the scheduler being considered in isolation. Rather, its correctness is established indirectly by the fact that reasoning in the high-level proof system, which assumes the abstract one-CPU-per-process machine, is sound with respect to the concrete machine.

To formulate the soundness statement in the above way, we need to construct a global property about the whole system state shown in Figure 6 from local assertions about its components, corresponding to partitions in Figure 6. This does not just boil down to combining the assertions about the partitions using the separating conjunction, since the high-level and low-level proof systems work on different levels of abstraction. In particular, when conjoining assertions about the scheduler and the kernel, we need to make sure that the view of the parts of kernel state in the assertions about the scheduler (dark regions in Figure 7) is consistent with those about the kernel. This requires a delicate construction, combining relational composition and separating conjunction. Similar constructions can potentially be used for justifying other program logics working on several levels of abstraction at the same time.

3 Machine semantics

In this section, we give a formal semantics to the example machine informally presented in Section 2.1.

3.1 Storage model

Figure 8 gives a model for the set of configurations Config that can arise during an execution of the machine. A machine configuration is a triple with the components describing the values of registers of the CPUs in the machine (its *global context*), the state of the heap and the set of locks taken by some CPU (the *lockset* of the machine). Note that we allow the global context or the heap to be a partial function. However, the corresponding configurations are not encountered in the semantics we define in this section. They come in handy in Sections 5 and 8 to give a semantics to the assertion language and express the soundness of our logic.

In this paper, we use the following notation for partial functions: $f[x:y]$ is the function that has the same value as f everywhere, except for x, where it has the value y; $[]$ is a nowhere-defined function; $f \uplus g$ is the union of the disjoint partial functions f and g.

3.2 Programming language

We consider a low-level language where programs are represented by structures similar to control-flow graphs. The programs are constructed from primitive commands c, whose syntax we define in Figure 9. In addition to the primitive commands listed in Section 2,

$$
\begin{array}{llll}
k \in \mathsf{CPUid} & = \{1,\ldots,\mathtt{NCPUS}\} & \mathtt{r} \in \mathsf{Reg} & = \{\mathtt{ip},\mathtt{if},\mathtt{ss},\mathtt{sp},\mathtt{gr}_1,\ldots,\mathtt{gr}_m\} \\
r \in \mathsf{Context} & = \mathsf{Reg} \rightarrow \mathsf{Val} & R \in \mathsf{GContext} & = \mathsf{CPUid} \rightharpoonup \mathsf{Context} \\
\ell \in \mathsf{Lock} & = \{\ell_1,\ell_2,\ldots,\ell_n\} & L \in \mathsf{Lockset} & = \mathscr{P}(\mathsf{Lock}) \\
h \in \mathsf{Heap} & = \mathsf{Loc} \rightharpoonup \mathsf{Val} & (R,h,L) \in \mathsf{Config} & = \mathsf{GContext} \times \mathsf{Heap} \times \mathsf{Lockset}
\end{array}
$$

Fig. 8. The set of machine configurations Config. We assume sets Loc of valid memory addresses and Val of values such that Loc ⊆ Val.

$$
\begin{array}{lcl}
\mathtt{r} & \in & \mathsf{Reg} - \{\mathtt{ip},\mathtt{if}\} \\
\ell & \in & \mathsf{Lock} \\
l & \in & \mathsf{Label} = \mathbb{N} \\
e & ::= & \mathtt{r} \mid 0 \mid 1 \mid 2 \mid \ldots \mid e+e \mid e-e \\
b & ::= & e=e \mid e \leqslant e \mid b \wedge b \mid b \vee b \mid \neg b \\
c & ::= & \mathtt{skip} \mid \mathtt{r} := e \mid \mathtt{r} := [e] \mid [e] := e \mid \mathtt{assume}(b) \mid \mathtt{lock}(\ell) \mid \mathtt{unlock}(\ell) \\
 & \mid & \mathtt{savecpuid}(e) \mid \mathtt{call}(l) \mid \mathtt{icall}(l) \mid \mathtt{ret} \mid \mathtt{iret}
\end{array}
$$

Fig. 9. Primitive commands.

we have the following ones: `skip` and $\mathtt{r} := e$ have the standard meaning; $\mathtt{r} := [e]$ reads the contents of a heap cell e and assigns the value read to `r`; $[e] := e'$ updates the contents of cell e by e'; $\mathtt{assume}(b)$ acts as a filter on the state space of programs, choosing states satisfying b. The `assume` command is used to treat branches in conditionals and loops uniformly with the other primitive commands, as we explain below. We write PComm for the set of primitive commands. Note that primitive commands cannot access the `ip` register directly. Also, only `icall` and `iret` can affect the `if` register, a restriction that we lift in Section 7.

Commands C are partial maps from Label to PComm $\times \mathscr{P}(\mathsf{Label})$. Intuitively, if $C(l) = (c,X)$, then c is labelled by l in C and can be followed by any command with a label in X. In this case, we let $\mathsf{comm}(C,l) = c$ and $\mathsf{next}(C,l) = X$. We denote the domain of C by $\mathsf{labels}(C)$ and the set of all commands by Comm.

The language constructs used in the example scheduler of Section 2, such as sequential composition, loops and conditionals, can be expressed as commands in a standard way, with conditions translated using `assume`. We illustrate this in Figure 10, where we represent the mapping C by a graph, with nodes annotated by labels and primitive commands and edges defining the next function.

3.3 Operational semantics

We now give a standard operational semantics to our programming language. We interpret primitive commands c using a transition relation $\rightsquigarrow_c$ of the following type:

$$
\begin{aligned}
\mathsf{State} &= \mathsf{Context} \times \mathsf{Heap} \times \mathsf{Lockset}; \qquad (3) \\
\rightsquigarrow_c &\subseteq \big(\mathsf{CPUid} \times \mathsf{State} \times \mathsf{Label} \times \mathsf{Label}\big) \times \big((\mathsf{State} \times \mathsf{Label}) \cup \{\top\}\big).
\end{aligned}
$$

The input $(k,(r,h,L),l,l')$ to $\rightsquigarrow_c$ consists of the following components:

- $k \in \mathsf{CPUid}$ is the identifier of the CPU executing the command.

```
if (x > 0) {
  y = 1;
} else {
  z = 1;
}
w = 2;
```

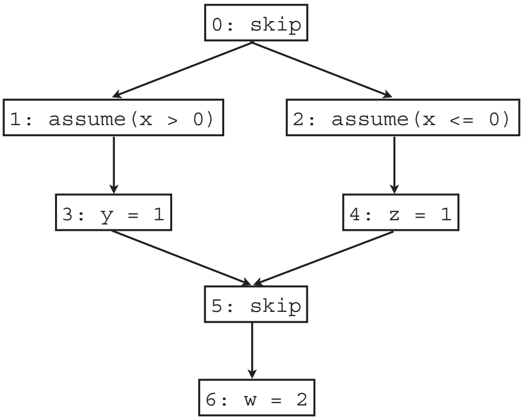

Fig. 10. Representing sequential composition and conditionals in our low-level language.

- (r,h,L) is the configuration of the system projected to this CPU, which we call a *state*. It includes the context of the CPU and the information about the shared resources—the heap and locks.
- $l \in \mathsf{Label}$ is the label of the command c.
- $l' \in \mathsf{Label}$ is the label of a primitive command following c in the program.

Given this input, the transition relation $\rightsquigarrow_c$ for c computes the next state of the CPU after running c, together with the label of the primitive command to run next. The former may be a special $\top$ state signalling a machine crash. The latter may be different from l' when c is a call or a return.

The relation $\rightsquigarrow_c$ is defined in Figure 11. In the figure and in the rest of the paper, we write _ for an expression whose value is irrelevant and implicitly existentially quantified and $\vec{\texttt{gr}}$ for the vector of general-purpose registers. The relation follows the informal meaning of primitive commands given in Sections 2.1 and 3.2. We have omitted standard definitions for `skip` and most of assignments (Reynolds, 2002). We have also omitted them for `icall` and `iret`: the definitions are the same as for `call` and `ret`, but additionally modify `if`. Note that $\rightsquigarrow_c$ may yield no poststate for a given prestate. Unlike a transition to the $\top$ state, this represents the command getting stuck. For example, according to Figure 11, acquiring the same lock twice leads to a deadlock, and releasing a lock that is not held crashes the system.

The program our machine executes is given by a command C that includes a primitive command labelled `schedule`, serving as the entry point of the interrupt handler. For such a command C, we give its meaning using a small-step operational semantics, formalised by the transition relation $\rightarrow_C \subseteq \mathsf{Config} \times (\mathsf{Config} \cup \{\top\})$ in Figure 12. The first rule in the figure describes a normal execution, where the value l of the `ip` register of CPU k is used to choose the primitive command c to run. After choosing c, the machine nondeterministically picks a label $l' \in \mathsf{next}(C,l)$ identifying the command to follow c, runs c according to the semantics $\rightsquigarrow_c$, and uses the result of this run to update the registers of CPU k and the heap and the lockset of the machine. For example, when a CPU executes the program in Figure 10 from label `0`, both labels `1` and `2` following it will be explored; however, only the branch where the `assume` condition evaluates to true will proceed further.

$$
\begin{array}{lll}
(k,(r,h[[\![e]\!]r:u],L),l,l') & \rightsquigarrow_{\mathtt{r}:=[e]} & ((r[\mathtt{r}:u],h[[\![e]\!]r:u],L),l') \\
(k,(r,h,L),l,l') & \rightsquigarrow_{\mathtt{assume}(b)} & ((r,h,L),l'),\ \text{if } [\![b]\!]r=\mathsf{true} \\
(k,(r,h,L),l,l') & \not\rightsquigarrow_{\mathtt{assume}(b)} & \qquad\qquad \text{if } [\![b]\!]r=\mathsf{false} \\
(k,(r,h,L),l,l') & \rightsquigarrow_{\mathtt{lock}(\ell)} & ((r,h,L\cup\{\ell\}),l'),\ \text{if } \ell\notin L \\
(k,(r,h,L),l,l') & \not\rightsquigarrow_{\mathtt{lock}(\ell)} & \qquad\qquad\qquad \text{if } \ell\in L \\
(k,(r,h,L),l,l') & \rightsquigarrow_{\mathtt{unlock}(\ell)} & ((r,h,L-\{\ell\}),l'),\text{if } \ell\in L \\
(k,(r,h[[\![e]\!]r:\_],L),l,l') & \rightsquigarrow_{\mathtt{savecpuid}(e)} & ((r,h[[\![e]\!]r:k],L),l') \\
(k,(r,h[r(\mathtt{sp})..(r(\mathtt{sp})+m):\_],L),l,l') & \rightsquigarrow_{\mathtt{call}(l'')} & \\
\multicolumn{3}{r}{((r[\mathtt{sp}:r(\mathtt{sp})+m+1],h[r(\mathtt{sp}):l',(r(\mathtt{sp})+1)..(r(\mathtt{sp})+m):r(\vec{\mathtt{gr}})],L),l'')} \\
\multicolumn{3}{l}{(k,(r,h[r(\mathtt{sp})-m-1:l'',(r(\mathtt{sp})-m)..(r(\mathtt{sp})-1):\vec{g}],L),l,l')\rightsquigarrow_{\mathtt{ret}}} \\
\multicolumn{3}{r}{((r[\mathtt{sp}:r(\mathtt{sp})-m-1,\vec{\mathtt{gr}}:\vec{g}],h[r(\mathtt{sp})-m-1:l'',(r(\mathtt{sp})-m)..(r(\mathtt{sp})-1):\vec{g}],L),l'')} \\
(k,(r,h,L),l,l') & \rightsquigarrow_{c} & \top,\ \text{otherwise}
\end{array}
$$

Fig. 11. Semantics of primitive commands. The notation $\rightsquigarrow_c \top$ indicates that the command c crashes, and $\not\rightsquigarrow_c$ that it does not crash, but gets stuck. The function $[\![\cdot]\!]r$ evaluates expressions with respect to the context r.

$$
\frac{r(\mathtt{ip})=l\in\mathsf{labels}(C) \qquad l'\in\mathsf{next}(C,l) \qquad (k,(r,h,L),l,l')\rightsquigarrow_{\mathsf{comm}(C,l)}((r',h',L'),l'')}{(R[k:r],h,L)\rightarrow_C(R[k:r'[\mathtt{ip}:l'']],h',L')}
$$

$$
\frac{r(\mathtt{ip})=l\in\mathsf{labels}(C) \qquad r(\mathtt{if})=1 \qquad (k,(r,h,L),l,l)\rightsquigarrow_{\mathtt{icall(schedule)}}((r',h',L'),l'')}{(R[k:r],h,L)\rightarrow_C(R[k:r'[\mathtt{ip}:l'']],h',L')}
$$

$$
\frac{r(\mathtt{ip})=l\in\mathsf{labels}(C) \qquad l'\in\mathsf{next}(C,l) \qquad (k,(r,h,L),l,l')\rightsquigarrow_{\mathsf{comm}(C,l)}\top}{(R[k:r],h,L)\rightarrow_C\top}
$$

$$
\frac{r(\mathtt{ip})\notin\mathsf{labels}(C)}{(R[k:r],h,L)\rightarrow_C\top}
$$

$$
\frac{r(\mathtt{if})=1 \qquad \{r(\mathtt{sp}),\dots,r(\mathtt{sp})+m\}\not\subseteq\mathsf{dom}(h)}{(R[k:r],h,L)\rightarrow_C\top}
$$

Fig. 12. Operational semantics of the machine.

The second rule in Figure 12 concerns interrupts. Upon an interrupt, the interrupt handler label `schedule` is loaded into `ip`, and the label of the command to execute after the handler returns is pushed onto the stack together with the values of the general-purpose registers. The remaining rules deal with crashes arising from erroneous execution of primitive commands, undefined command labels and a stack overflow upon an interrupt.

4 Baseline concurrency logic

We start by presenting the variation of concurrent separation logic on which our logic for verifying preemptive kernels is based (Section 5). The logic we present in this section assumes that interrupts are disabled on all CPUs. Thus, we assume that all `if` registers are initially set to zero and only consider programs that do not use `icall`, `iret` and `savecpuid` commands. One can thus think of a single process having been pinned to every

$$
\begin{array}{rcl}
x, y & \in & \mathsf{NVar} \\
\gamma & \in & \mathsf{CVar} \\
\mathtt{r} & \in & \mathsf{Reg} - \{\mathtt{ip}, \mathtt{if}\} \\
\mathbf{r} & \in & \{\mathbf{ip}, \mathbf{if}, \mathbf{ss}, \mathbf{sp}, \mathbf{gr}_1, \dots, \mathbf{gr}_m\} \\
E & ::= & x \mid \mathtt{r} \mid 0 \mid 1 \mid \dots \mid E+E \mid E-E \mid G(\mathbf{r}) \\
G & ::= & \gamma \mid [\mathbf{ip} : E, \mathbf{if} : E, \mathbf{ss} : E, \mathbf{sp} : E, \vec{\mathbf{gr}} : \vec{E}] \\
\Sigma & ::= & \varepsilon \mid E \mid \Sigma\Sigma \\
B & ::= & E = E \mid \Sigma = \Sigma \mid G = G \mid E \leqslant E \mid B \wedge B \mid B \vee B \mid \neg B \\
P & ::= & B \mid \mathsf{true} \mid P \wedge P \mid \neg P \mid \exists x. P \mid \exists \gamma. P \\
 & \mid & \mathsf{emp} \mid E \mapsto E \mid E..E \mapsto \Sigma \mid P * P \mid \mathsf{dll}_\Lambda(E, E, E, E) \mid \mathsf{locked}(\ell)
\end{array}
$$

$$
\begin{array}{lll}
(r, h, L) \models_\eta B & \text{iff} & [\![B]\!]_\eta r = \mathit{true} \\
(r, h, L) \models_\eta P_1 \wedge P_2 & \text{iff} & (r, h, L) \models_\eta P_1 \text{ and } (r, h, L) \models_\eta P_2 \\
(r, h, L) \models_\eta \mathsf{emp} & \text{iff} & h = [\,] \text{ and } L = \emptyset \\
(r, h, L) \models_\eta E_0 \mapsto E_1 & \text{iff} & h = [[\![E_0]\!]_\eta r : [\![E_1]\!]_\eta r] \text{ and } L = \emptyset \\
(r, h, L) \models_\eta E_0..E_1 \mapsto \Sigma & \text{iff} & \exists j \geqslant 0. \exists u_1, \dots, u_j \in \mathsf{Val}.\ L = \emptyset,\ j = [\![E_1]\!]_\eta r - [\![E_0]\!]_\eta r + 1, \\
 & & u_1 u_2 \dots u_j = [\![\Sigma]\!]_\eta r \text{ and } h = [[\![E_0]\!]_\eta r : u_1, \dots, [\![E_1]\!]_\eta r : u_j] \\
(r, h, L) \models_\eta \mathsf{locked}(\ell) & \text{iff} & h = [\,] \text{ and } L = \{\ell\} \\
(r, h, L) \models_\eta P_1 * P_2 & \text{iff} & \exists h_1, h_2, L_1, L_2.\ h = h_1 \uplus h_2,\ L = L_1 \uplus L_2, \\
 & & (r, h_1, L_1) \models_\eta P_1 \text{ and } (r, h_2, L_2) \models_\eta P_2
\end{array}
$$

Predicate dll_Λ is the least one satisfying the equivalence below:

$$
\begin{aligned}
\mathsf{dll}_\Lambda(E_h, E_p, E_n, E_t) \iff \exists x. (E_h = E_n \wedge E_p = E_t \wedge \mathsf{emp}) \vee \\
E_h.\mathtt{prev} \mapsto E_p * E_h.\mathtt{next} \mapsto x * \Lambda(E_h) * \mathsf{dll}_\Lambda(x, E_h, E_n, E_t)
\end{aligned}
$$

Fig. 13. Syntax and semantics of assertions of the baseline logic. We have omitted the standard clauses for most of the first-order connectives. The function $[\![\cdot]\!]_\eta r$ evaluates expressions with respect to the context r and the logical variable environment η.

CPU, so that we do not have to consider scheduling. Our logic for preemptive kernels in Section 5 lifts this restriction.

4.1 Assertion language

Mathematically, assertions denote sets of states as defined by (3). However, they describe properties of a single process, rather than the whole machine. Hence, unlike in Section 3.3, here a heap can be a partial function, with its domain defining the part of the heap owned by the process. Similarly, a lockset is now meant to contain only the set of locks that the process has permission to release.

We use a minor extension of the assertion language of separation logic (Reynolds, 2002), whose syntax and semantics are defined in Figure 13. We denote the set of assertions by $\mathsf{Assert}_\mathsf{H}$. We assume disjoint sets NVar and CVar containing logical variables for values and contexts, respectively. The latter is needed for the extension of the current logic to reasoning about preemptive kernels (Section 5). A context G is either a logical variable or

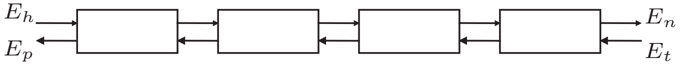

Fig. 14. An illustration of the $\mathsf{dll}_\Lambda(E_h, E_p, E_n, E_t)$ predicate.

a finite map from register labels $\mathbf{r}$ to expressions. Note that we use $\mathbf{r}$ to range over register names in context expressions, but $\mathtt{r}$ elsewhere in assertions and in programs. Expressions E and Booleans B are similar to those in programs, except that they allow logical variables to appear and include the lookup $G(\mathbf{r})$ of the value of the register $\mathbf{r}$ in the context G. Let a logical variable environment η be a mapping from $\mathsf{NVar} \cup \mathsf{CVar}$ to $\mathsf{Val} \cup \mathsf{Context}$ that respects the types of variables. Assertions denote sets of states from State as defined by the satisfaction relation $\models_\eta$ in Figure 13. For an environment η and an assertion P, we denote the set of states satisfying P by $[\![P]\!]_\eta$.

The assertions in the first line of the definition of P are connectives from the first-order logic with the standard semantics. We can define the missing connectives from the given ones. The assertions in the second line up to dll_Λ are standard assertions of separation logic (Reynolds, 2002). Informally, emp describes the empty heap, and $E \mapsto E'$ the heap with only one cell at the address E containing E'. The assertion $E..E' \mapsto \Sigma$ is the generalisation of the latter to several consecutive cells at the addresses from E to E' inclusive containing the sequence of values Σ. For a value u of a C type $\mathtt{t}$ taking several cells, we shorten $E..(E + \mathtt{sizeof(t)} - 1) \mapsto u$ to just $E \mapsto u$. For a field $\mathtt{f}$ of a C structure, we use $E.\mathtt{f} \mapsto E'$ as a shorthand for $(E + \mathit{off}) \mapsto E'$, where off is the offset of $\mathtt{f}$ in the structure. The separating conjunction $P_1 * P_2$ talks about the splitting of the local state, which consists of the heap and the lockset of the process. It says that a pair (h, L) can be split into two disjoint parts, such that one part (h_1, L_1) satisfies P_1 and the other (h_2, L_2) satisfies P_2.

The assertion $\mathsf{dll}_\Lambda(E_h, E_p, E_n, E_t)$ is an inductive predicate describing a segment of a doubly-linked list (Figure 14). We included it to describe the runqueues of the scheduler in our example; predicates for other data structures can be added straightforwardly (Reynolds, 2002). The predicate assumes a C structure definition with fields $\mathtt{prev}$ and $\mathtt{next}$. Here, E_h is the address of the head of the list, E_t is the address of its tail, E_p is the pointer in the $\mathtt{prev}$ field of the head node, and E_n is the pointer in the $\mathtt{next}$ field of the tail node. The Λ parameter is a formula with one free logical variable describing the shape of each node in the list, excluding the $\mathtt{prev}$ and $\mathtt{next}$ fields; the logical variable defines the address of the node. For instance, the request queue from Figure 4 can be described by the following assertion:

$$\exists x, y, z.\, \&\mathtt{request\_queue} \mapsto z * z.\mathtt{prev} \mapsto y * z.\mathtt{next} \mapsto x * z.\mathtt{data} \mapsto \_ * \mathsf{dll}_\Lambda(x, z, z, y), \quad (4)$$

where $\Lambda(x) = x.\mathtt{data} \mapsto \_$.

Finally, the assertion $\mathsf{locked}(\ell)$ is specific to reasoning about concurrent programs and denotes states with an empty local heap and the lockset consisting of ℓ, i.e., it denotes a permission to release the lock ℓ. Note that $\mathsf{locked}(\ell) * \mathsf{locked}(\ell)$ is inconsistent: acquiring the same lock twice leads to a deadlock.

In the following, we write $\mathit{var} \Vdash P$ for a local C variable or procedure parameter $\mathtt{var}$ instead of $\exists \mathit{var}.\, (\mathtt{sp} - \mathit{var\_off}) \mapsto \mathit{var} * P$, where $\mathit{var\_off}$ is the offset of $\mathtt{var}$ with respect to the top of the stack in the activation record of the function where it is declared (note that here `var` is a program variable, whereas *var* is a logical one). Thus, the assertion at line 13 of Figure 4 states that the local state of a process executing the system call consists of the local variables `req` and `tmp`, stored on its stack, the free part of the stack, the doubly-linked list `request_queue` and a permission to release `request_lock`.

To summarise, our assertion language extends that of concurrent separation logic with expressions to denote contexts and locked assertions to keep track of permissions to release locks.

4.2 Proof system

The proof system for the baseline logic is obtained by adapting concurrent separation logic to our low-level language. The judgements of the proof system are of the form $I, \Delta \vdash C$. Here, C specifies the code executed by processes on all CPUs; note that even though C is the same for all of them, the processes can still execute different programs if they start from different program points in C. We explain I below. The parameter $\Delta : \mathsf{Label} \to \mathsf{Assert_H}$ in our judgement specifies local states of a process given the program point it is at; these states correspond to process partitions in Figure 7. It thus induces pre- and postconditions for all primitive commands in C. The top-level rule PROG of the proof system requires us to prove $I, \Delta \rhd_{l'} \{\Delta(l)\}\ c\ \{\Delta(l')\}$ for every primitive command c in C and the label l' of a command following c. This informally means that if c is run from an initial state satisfying $\Delta(l)$, then it accesses only the memory specified by $\Delta(l)$ and either terminates normally and ends up in a state satisfying $\Delta(l')$, or jumps to a label l'' whose assertion $\Delta(l'')$ holds in the current state. The proof rules for the above kind of judgements are also given in Figure 15. They include the standard separation logic axioms for primitive commands, such as ASSUME and STORE; see Reynolds (2002) for the others. Note that PROG treats sequential composition, represented by Δ as illustrated in Figure 10, in the same way as the classical proof rule (1).

The fact that $I, \Delta \rhd_{l'} \{\Delta(l)\}\ c\ \{\Delta(l')\}$ guarantees that c accesses only the memory specified by $\Delta(l)$ validates the frame property (Section 2.3): a process will not step out of the boundaries of its partition in Figure 7. This also allows us to include the FRAME rule of separation logic, which states that executing a command in a bigger local state does not change its behaviour. The rule is useful to restrict the reasoning about primitive commands to the memory they actually access. The rules CONSEQ, DISJ and EXISTS are standard rules of Hoare logic. To keep the logic sound we have to forbid applying EXISTS and FRAME to calls or returns.

The LOCK and UNLOCK axioms are inherited from concurrent separation logic and provide tools for modular reasoning about concurrent processes. They use the mapping $I : \mathsf{Lock} \rightharpoonup \mathsf{Assert_H}$, which specifies the invariants of locks that can be used in C (see Section 2.3). An example of a lock invariant is the assertion (4), which states that the lock `request_lock` from Figure 4 protects a non-empty cyclic doubly-linked list of `Request` nodes with the head node at address `request_queue`. We do not allow lock invariants to contain registers or free occurrences of logical variables and require them to have an empty lockset: $\forall \ell, \eta, (r,h,L) \in [\![I(\ell)]\!]_\eta . L = \emptyset$. The latter does not allow us to prove programs where a lock is released by a CPU other than the one that acquired it, which our machine semantics allows. We put this restriction to simplify the explanation of soundness

$$\frac{\forall l \in \mathsf{labels}(C).\, \forall l' \in \mathsf{next}(C,l).\, (I,\Delta \rhd_{l'} \{\Delta(l)\}\ \mathsf{comm}(C,l)\ \{\Delta(l')\})}{I,\Delta \vdash C}\ \textsc{Prog}$$

$$\frac{}{I,\Delta \rhd_l \{P\}\ \mathtt{assume}(b)\ \{P \wedge b\}}\ \textsc{Assume}$$

$$\frac{}{I,\Delta \rhd_l \{e \mapsto \_\}[e]{:=}e'\{e \mapsto e'\}}\ \textsc{Store}$$

$$\frac{I,\Delta \rhd_l \{P\}c\{Q\} \quad \mathsf{mod}(c) \cap \mathsf{free}(F) = \emptyset \quad c \text{ is not one of } \mathtt{call},\ \mathtt{icall},\ \mathtt{ret} \text{ and } \mathtt{iret}}{I,\Delta \rhd_l \{P * F\}c\{Q * F\}}\ \textsc{Frame}$$

$$\frac{P \implies P' \quad I,\Delta \rhd_l \{P'\}\ c\ \{Q'\} \quad Q' \implies Q}{I,\Delta \rhd_l \{P\}\ c\ \{Q\}}\ \textsc{Conseq}$$

$$\frac{I,\Delta \rhd_l \{P\}\ c\ \{Q\} \quad c \text{ is not one of } \mathtt{call},\ \mathtt{icall},\ \mathtt{ret} \text{ and } \mathtt{iret}}{I,\Delta \rhd_l \{\exists x.P\}\ c\ \{\exists x.Q\}}\ \textsc{Exists}$$

$$\frac{I,\Delta \rhd_l \{P_1\}\ c\ \{Q_1\} \quad I,\Delta \rhd_l \{P_2\}\ c\ \{Q_2\}}{I,\Delta \rhd_l \{P_1 \vee P_2\}\ c\ \{Q_1 \vee Q_2\}}\ \textsc{Disj}$$

$$\frac{}{I,\Delta \rhd_l \{\mathsf{emp}\}\ \mathtt{lock}(\ell)\ \{I(\ell) * \mathsf{locked}(\ell)\}}\ \textsc{Lock}$$

$$\frac{}{I,\Delta \rhd_l \{I(\ell) * \mathsf{locked}(\ell)\}\ \mathtt{unlock}(\ell)\ \{\mathsf{emp}\}}\ \textsc{Unlock}$$

$$\frac{(P * (\mathtt{sp}..(\mathtt{sp}+m) \mapsto l\, \mathtt{gr}_1 \ldots \mathtt{gr}_m)) \implies (\Delta(l')[(\mathtt{sp}+m+1)/\mathtt{sp}])}{I,\Delta \rhd_l \{P * (\mathtt{sp}..(\mathtt{sp}+m) \mapsto \_)\}\ \mathtt{call}(l')\ \{Q\}}\ \textsc{Call}$$

$$\frac{\forall l' \in \mathsf{Label}.(P * ((\mathtt{sp}-m-1)..(\mathtt{sp}-1) \mapsto E'\vec{E}) \wedge E'{=}l') \implies (\Delta(l')[(\mathtt{sp}-m-1)/\mathtt{sp}][\vec{E}/\vec{\mathtt{gr}}])}{I,\Delta \rhd_l \{P * ((\mathtt{sp}-m-1)..(\mathtt{sp}-1) \mapsto E'\vec{E})\}\mathtt{ret}\{Q\}}\ \textsc{Ret}$$

Fig. 15. Proof system of the baseline logic. Here, $\mathsf{mod}(c)$ is the set of registers modified by c, and $\mathsf{free}(F)$ is the set of registers appearing in F.

in Section 8. We consider a version of concurrent separation logic where lock invariants are allowed to be imprecise (O'Hearn, 2007) at the expense of excluding the conjunction rule from the proof system (Gotsman *et al.*, 2011).

The Lock axiom says that upon acquiring a lock, the process gets the ownership of its invariant and a permission to release it. In terms of Figure 7, we can think of the corresponding lock partition becoming part of the process-local one, allowing the process to modify it at will. According to Unlock, before releasing the lock, the process must have the corresponding permission and must re-establish the lock invariant. When the lock is released, the process gives up the ownership of the permission and the invariant. In terms of Figure 7, the lock partition gets split off the process-local one.

The Call and Ret axioms mirror the operational semantics of `call` and `ret` (see Section 2.1 and Figure 11). Call requires us to provide enough space on the stack to store the values of registers before a call. The precondition together with the modified stack then has to establish the assertion given by Δ at the target label. The premiss of Ret requires us to make a case-split on all possible labels l' we could return to; the precondition has to establish the assertion at every such label after the values of general-purpose registers and `ip` (denoted by $\vec{E}$ and E') have been loaded from the stack.

The axioms CALL and RET provide only a very rudimentary treatment of procedures. In particular, our logic does not have analogues of the usual modular Hoare proof rules for procedures and does not allow applying the EXISTS and FRAME rules over a procedure call. This is because soundly formulating such proof rules in the setting where the stack is visible to procedure code and can potentially be modified by it is non-trivial. Since we are concerned with scheduler verification, in this paper we opted for a simplistic solution and did not include more powerful proof rules for procedures (Feng *et al.*, 2006). As we discuss in Section 10, this issue also represents a promising direction of future work.

The soundness statement of the logic (presented in Section 8.1) constrains the states obtained by running the machine with interrupts disabled: when CPUs are at given program points $l_1, \ldots, l_{\mathtt{NCPUS}}$, the state of the machine can be obtained by combining the assertions $\Delta(l_1), \ldots, \Delta(l_{\mathtt{NCPUS}})$, describing their local states, and the lock invariants of free locks, as per Figure 7. The set of free locks can be determined based on occurrences of locked predicates in the assertions $\Delta(l_1), \ldots, \Delta(l_{\mathtt{NCPUS}})$.

Figure 4 gives an example proof of a concurrent program in the baseline logic, where every CPU executes the code shown in the figure. The assertions shown define the local state of a process and thus the required mapping Δ; the lock invariant of `request_lock` is (4). Note that the assertions describe the stack of a process explicitly. We introduced the $\Vdash$ notation in Section 4.1. When a process acquires `request_lock`, it gets the ownership of the doubly-linked list it protects, together with the corresponding locked predicate. After performing manipulations on the list, the process gives up its ownership upon releasing the lock. Our proof ensures that the code in Figure 4 preserves the doubly-linked list shape of `request_queue` and does not access memory cells other than those specified by the assertions.

5 Logic for preemptive kernels

In this paper we consider schedulers whose interface consists of two routines: `create` and `schedule`. Like in our example scheduler (Section 2.2), `create` makes a new process runnable, and `schedule` performs a context switch. We discuss how our results can be extended when new scheduler routines are introduced in Section 5.5 below. Our logic thus reasons about programs of the form:

$$\mathsf{C} \uplus [l_c : (\mathtt{iret}, \{l_c + 1\})] \uplus \mathsf{S} \uplus [l_s : (\mathtt{iret}, \{l_s + 1\})] \uplus \mathsf{K}. \qquad (\mathsf{OS})$$

where C and S are pieces of code implementing the `create` and `schedule` routines of the scheduler, l_c and l_s are their exit points, and K is the rest of the kernel code. Our high-level proof system is designed for proving K, and the low-level system for proving C and S.

We make several assumptions about programs:

- We require that C and S define primitive commands labelled `create` and `schedule`, which are meant to be the entry points for the corresponding scheduler routines. The `create` routine expects the address of the descriptor of the new process to be stored in the register $\mathtt{gr}_1$. By our convention `schedule` also marks the entry point of the interrupt handler. Thus, `schedule` may be called both directly by a process or by an interrupt.

- For now, we ensure that the kernel may not affect the status of interrupts, become aware of the particular CPU it is executing on, or change the stack address. Thus, K may not contain `savecpuid`, `icall` and `iret` (except calls to the scheduler routines `schedule` and `create`), and assignments writing to `ss`. In reality, a kernel might need to disable interrupts, and we generalise our results to handle this in Section 7.
- To ensure that the scheduler routines execute with interrupts disabled, we require that C and S may not contain `icall` and `iret`.
- We require that the kernel K and the scheduler C and S access disjoint sets of locks. This condition simplifies the soundness statement in Section 8 and can be lifted.
- For simplicity, we assume that the scheduler data structures are properly initialised when the program starts executing.

5.1 Interface parameters

As we noted in Section 2.3, our logic can be viewed as implementing a form of rely-guarantee reasoning between the scheduler and the kernel. In particular, interactions between them involve ownership transfer of memory cells at points where the control crosses the boundary between the two components. Hence, the high- and low-level proof systems have to agree on the description of the memory areas being transferred and the properties they have to satisfy. These descriptions form the specification of the interface between the scheduler and the kernel and, correspondingly, between the two proof systems. Here we describe parameters used to formulate it.

When the kernel calls the `create` routine of the scheduler, the latter might need to get the ownership of the process descriptor supplied as the parameter. In the two proof systems, we specify this descriptor using an assertion $\mathsf{desc}(d,\gamma) \in \mathsf{Assert}_{\mathsf{H}}$ with two free logical variables and no register occurrences. Our intention is that it describes the descriptor of a process with the context γ, allocated at the address d. However, the user of our logic is free to choose any assertion, depending on a particular scheduler implementation being verified. This flexibility has an impact on the soundness statement of the logic, as we discuss in Section 8. Since the scheduler and the kernel access disjoint sets of locks, we require that $[\![\mathsf{desc}(d,\gamma)]\!]$ have an empty lockset (Section 4.2). We assume that `create` does not transfer anything back to the kernel at its return, and thus do not introduce an interface parameter for this case.

The `schedule` routine can be called by the kernel explicitly, or as a consequence of an interrupt at any time. Due to the latter case, `schedule` cannot make that many assumptions about the state in which it is called. Therefore, rather than specifying the state to be transferred from the kernel to `schedule` upon an interrupt abstractly, like in the case of `create`, we fix it to be the free part of the stack of the process being preempted—the minimum `schedule` needs to execute. The scheduler returns the ownership of this memory to the process when it schedules the process again. The corresponding interface parameters determine the size of the part of the stack being transferred: the size of the stack `StackSize` $\in \mathbb{N}$ and the upper bound `StackBound` $\in \mathbb{N}$ on the stack usage by the kernel (excluding the scheduler). To ensure that the stack does not overflow while calling

$$\frac{\begin{array}{l}\forall l \in \mathsf{labels}(C).\, \forall l' \in \mathsf{next}(C,l).\, (I,\Delta \rhd_{l'} \{\Delta(l)\}\ \mathsf{comm}(C,l)\ \{\Delta(l')\}) \\ \forall l \in \mathsf{Label}(C).\, \exists P \in \mathsf{Assert}_{\mathsf{H}}.\, \Delta(l) \iff \\ \qquad ((0 \leqslant \mathsf{sp} - \mathsf{ss} \leqslant \mathtt{StackBound}) \wedge (\mathsf{sp}..(\mathsf{ss} + \mathtt{StackSize} - 1) \mapsto \_) * P)\end{array}}{I,\Delta \vdash C}\ \textsc{Prog-H}$$

$$\frac{}{I,\Delta \rhd_l \{P\}\ \mathtt{icall}\ (\mathtt{schedule})\ \{P\}}\ \textsc{Sched}$$

$$\frac{\begin{array}{l}\mathsf{free}(P) \cap \mathsf{Reg} = \emptyset \\ P \text{ has an empty lockset} \\ \forall l' \in \mathsf{Label}.\, (\exists \gamma.\, \mathsf{id} = \gamma \wedge \gamma(\mathbf{ip}) = l' \wedge \mathsf{sp}..(\mathsf{ss} + \mathtt{StackSize} - 1) \mapsto \_ * P) \implies \Delta(l')\end{array}}{I,\Delta \rhd_l \{\exists \gamma.\, \gamma(\mathbf{if}) = 1 \wedge \mathsf{desc}(\mathsf{gr}_1, \gamma) * P * Q\}\ \mathtt{icall}(\mathtt{create})\ \{\exists \gamma.\, Q\}}\ \textsc{Create}$$

Fig. 16. The rules specific to the high-level proof system. Here, $\mathsf{id} = [\mathbf{ip} : \_, \mathbf{if} : 1, \mathbf{ss} : \mathsf{ss}, \mathbf{sp} : \mathsf{sp}, \vec{\mathbf{gr}} : \vec{\mathsf{gr}}]$.

an interrupt hander, we require that $\mathtt{StackSize} - \mathtt{StackBound} \geqslant m + 1$, where m is the number of general-purpose registers.

In the following, we first present the high-level proof system used for verifying kernel code, which adapts the baseline concurrency logic from Section 4. We then present the low-level proof system for verifying scheduler code, which extends the high-level system.

5.2 High-level proof system

The high-level proof system is obtained from that of Section 4 with minimal changes. The proof system reasons under an illusion that every process runs on a separate virtual CPU with its own registers (but not memory), and its assertions now describe properties of processes under this assumption, as illustrated in Figure 7. Whereas in Section 4 we justified such reasoning by requiring processes to be pinned to physical CPUs by disabling interrupts, here we do not make this assumption.

The judgements of the proof system are of the same form as before: $I, \Delta \vdash C$, where C specifies all the code executed by kernel processes. As before, different processes can execute different programs by starting at different program points in C. The high-level proof system is meant for verifying the K part of the OS program and, hence, Δ in $I, \Delta \vdash C$ is now meant to give assertions only for the kernel code. When combining proofs in the high-level and low-level proof systems in Section 5.4, we enforce this by restricting Δ so that it is false everywhere except at labels in the kernel code. Similarly, I describes invariants for locks accessible in the kernel code only.

The changes to the logic from Section 4 are as follows. The Prog rule from Figure 15 gets replaced by a similar rule Prog-h, shown in Figure 16, and the rest of the rules in Figure 15 are left without changes. Prog-h inherits the premiss of Prog, and thus subsumes the usual sequential composition rule of Hoare logic: it assumes that the control follows the structure of the process code, even though the scheduler code can get executed due to an interrupt at any time. This possibility is accounted for by the second premiss of Prog-h. Recall from Section 5.1 that the kernel is supposed to transfer the ownership of the free part of the stack to the scheduler at an interrupt, and get it back when it is scheduled again. The second premiss of Prog-h ensures this by requiring all assertions in Δ to satisfy some restrictions regarding stack usage, formulated using parameters `StackSize` and `StackBound` from Section 5.1:

- the free part of the stack of the process must always be in its local state so that it can be transferred to the interrupt handler at any time;
- this part must always be large enough for the handler to run without a stack overflow; and
- the assertions should be independent of any changes to the empty slots of the stack, which may be modified by the handler.

The latter condition is similar to that of stability in logics based on rely-guarantee (Feng *et al.*, 2007a; Vafeiadis & Parkinson, 2007).

To complete the illusion of uninterrupted control flow in a process, the high-level proof system treats explicit calls to the `create` and `schedule` routines of the scheduler as primitive commands, axiomatising their effect using SCHED and CREATE. These axioms are formulated as if after the corresponding `icall` commands the control just proceeded to the next statement in the program, instead of jumping to the implementation of the routines. This is despite the fact that, after a call to `schedule`, the process may be preempted and the control given to any other process in the system. In this way, the axioms abstract from the scheduler implementation.

The SCHED axiom states that invoking `schedule` has no effect from the point of view of the process—if it is preempted, the scheduler resumes it in the same context, and no other process can touch its local state. The axiom does not place any requirements on the process, as the preconditions necessary for the execution of `schedule`, which can anyway be invoked at any time as the interrupt handler, are established by the second premiss of PROG-H.

The CREATE axiom is more complicated. First, it requires the caller of `create` to provide a new descriptor $\mathsf{desc}(\mathtt{gr}_1, \gamma)$ for the process being created with the context γ. We pass the parameter via the register $\mathtt{gr}_1$ and not via the stack, as this simplifies the following technical presentation. The context is required to have `if` set, since after the context switch is finished, the process starts executing with interrupts enabled. Note that the descriptor is not present in the postcondition: it gets transferred to the scheduler and reappears in the precondition of the implementation of `create` (Section 5.4). The axiom also allows us to transfer the ownership of the part of the heap given by P to the newly created process, thus providing it with an initial local state. This is a typical idiom for high-level reasoning about processes in separation logics (Gotsman *et al.*, 2007). The premiss of the rule correspondingly requires that, after the registers and the stack are properly initialised, the state P we are transferring should establish the assertion at the label the process starts executing from. The effect of loading registers from γ is formulated using the context id.

For the example scheduler in Section 2.2, $\mathsf{desc}(d, \gamma)$ should describe a process descriptor with the stack initialised according to the invariant of a preempted process pictured in Figure 5:

$$\mathsf{desc}(d,\gamma) = d.\mathtt{prev} \mapsto \_ * d.\mathtt{next} \mapsto \_ * \mathsf{desc}_0(d,\gamma),$$

where

$$\begin{aligned}\mathsf{desc}_0(d,\gamma) \;=\;& (\gamma(\mathbf{if}) = 1) \wedge (\gamma(\mathbf{ss}) = d.\mathtt{kernel\_stack}) \wedge \\ & (0 \leqslant \gamma(\mathbf{sp}) - \gamma(\mathbf{ss}) \leqslant \mathtt{StackBound}) \wedge d.\mathtt{timeslice} \mapsto \_ * \\ & d.\mathtt{saved\_sp} \mapsto (\gamma(\mathbf{sp}) + m + 1 + \mathtt{SCHED\_FRAME}) * \\ & \gamma(\mathbf{sp})..(\gamma(\mathbf{sp}) + m) \mapsto \gamma(\mathbf{ip})\gamma(\vec{\mathbf{gr}}) * \\ & (\gamma(\mathbf{sp}) + m + 1)..(\gamma(\mathbf{ss}) + \mathtt{StackSize} - 1) \mapsto \_\end{aligned}$$

and SCHED_FRAME is the size of the activation record of schedule (Figure 2). The descriptor does not include filled stack slots; they can be passed to the process directly in the precondition P.

Now assume that we want to create a process that will start executing from a label l_0 with an empty stack and the ownership of a cell at the address stored in the register $\mathtt{gr}_1$, so that

$$\Delta(l_0) = (\mathtt{ss} = \mathtt{sp} \wedge \mathtt{ss}..(\mathtt{ss} + \mathtt{StackSize} - 1) \mapsto \_ * \mathtt{gr}_1 \mapsto \_).$$

To apply CREATE, we let

$$P = (\gamma(\mathbf{ip}) = l_0 \wedge \gamma(\mathbf{ss}) = \gamma(\mathbf{sp}) \wedge \gamma(\mathbf{gr}_1) \mapsto \_).$$

Then the left-hand side of the implication in the last premiss of CREATE is false for all $l' \neq l_0$, and in this case the implication holds trivially; it is easy to check that the implication also holds for $l' = l_0$.

The proof in Figure 4, previously done using the baseline concurrency logic, is also a valid proof in the high-level logic for $\mathtt{StackBound} = 2 \cdot \mathtt{sizeof(Request*)}$.

To summarise, the high-level proof system provides modern tools for modular reasoning about concurrent processes using proof rules of concurrent separation logic and hides the control flow of the scheduler by treating its routines as primitive commands. The soundness of such an illusion is established by verifying the scheduler code using a low-level proof system, which we describe next.

5.3 Low-level proof system

The low-level proof system is used for proving that the commands C and S of the OS program implement scheduling correctly. This boils down to checking the two obligations explained in Section 2.3:

1. A scheduler resumes a process with the state of the CPU registers it had the last time it was preempted.
2. A scheduler does not duplicate processes arbitrarily.

To reason about these, we extend the assertion language of Section 4.1 with an additional predicate:

$$P \;::=\; \ldots \mid \mathsf{Process}(G),$$

where G ranges over context expressions. The predicate records the reference values G of registers in between the time a process is preempted and scheduled again. In Section 5.4 below, we use it to formulate the proof obligation on the context-switch routine of the

$$\begin{array}{lll}
(r,h,L),M \models_\eta \mathsf{Process}(G) & \text{iff} & h=[\,],\ L=\emptyset,\ M=\{[\![G]\!]_\eta r\} \\
(r,h,L),M \models_\eta P*Q & \text{iff} & \exists h_1,h_2,L_1,L_2,M_1,M_2.h{=}h_1 \uplus h_2, L=L_1 \uplus L_2, M=M_1 \uplus M_2, \\
 & & (r,h_1,L_1),M_1 \models_\eta P \text{ and } (r,h_2,L_2),M_2 \models_\eta Q \\
(r,h,L),M \models_\eta \mathsf{emp} & \text{iff} & h=[\,],\ L=\emptyset \text{ and } M=\emptyset \\
(r,h,L),M \models_\eta P \wedge Q & \text{iff} & (r,h,L),M \models_\eta P \text{ and } (r,h,L),M \models_\eta Q
\end{array}$$

Fig. 17. Semantics of low-level assertions. The $\uplus$ operation on multisets adds up the number of occurrences of each element in its operands.

scheduler formalising (2) from Section 2.3 and thus ensuring property 1 above. We denote the extended set of assertions by $\mathsf{Assert}_\mathsf{L}$.

The addition of the Process predicate changes objects described by assertions: they now denote relations defined by subsets of

$$\mathsf{SchedState} = \mathsf{State} \times \mathscr{M}(\mathsf{Context}),$$

where $\mathscr{M}(\mathsf{Context})$ is the set of all finite multisets of contexts. Here, an element of State describes a state local to a scheduler invocation on a CPU (Figure 6) and that of $\mathscr{M}(\mathsf{Context})$ interprets Process predicates. We give the formal semantics of assertions using the satisfaction relation $\models_\eta$ in Figure 17, parameterised by environments η. The first two cases in the figure are the most interesting ones. The assertion $\mathsf{Process}(G)$ describes a scheduler invocation having the empty heap and lockset and a permission to schedule a single process with the register values G. The separating conjunction $P*Q$ splits all parts of the state-multiset pair except the current scheduler context such that the first part satisfies P and the second Q. This definition of $*$ prohibits duplicating Process and thus ensures property 2 above:

$$\neg(\mathsf{Process}(G) \implies \mathsf{Process}(G) * \mathsf{Process}(G)).$$

The semantic definitions for the remaining assertions are obtained from the corresponding cases in our high-level proof system (Figure 13) either by requiring the multiset component M to be empty, like in the case of emp, or by propagating M to their sub-assertions, like in the case of $P \wedge Q$. For example, the assertion $\exists \gamma.\, \mathsf{desc}(d,\gamma) * \mathsf{Process}(\gamma)$ denotes a descriptor of a preempted process with a Process predicate matching the state stored in it and thus certifying its validity. We denote the set of states satisfying P by $[\![P]\!]_\eta$.

Relations in SchedState can be thought of as connecting the states of the concrete machine and the abstract machine with one CPU per process. As we have noted in Section 2.3, these relations do not describe the full state of the machines. The first component in a relation describes the local state of a scheduler invocation running on a CPU, including its context and the heap and the lockset local to it (e.g., the region marked CPU1 in Figure 6). The multiset in the second part records information about the states of processes described by Process predicates in the assertion (cf. the dark regions in Figure 7), which includes their contexts, but excludes local heaps and locksets. The low-level logic we present in this section is based on separation logic and, hence, the invisibility of these parts of process state to the scheduler automatically guarantees that it cannot access them.

The judgements of the low-level proof system have the form $I, \Delta \vdash_k C$, where $k \in \mathsf{CPUid}$, $I : \mathsf{Lock} \rightharpoonup \mathsf{Assert}_\mathsf{L}$ is a vector of invariants for locks accessible to the scheduler, and

$\Delta : \mathsf{Label} \to \mathsf{Assert_L}$ is a mapping from program positions to low-level assertions. When considering a complete system in Section 5.4, we restrict Δ so that it is false everywhere except at labels in the scheduler code. Since we allow the scheduler to use the `savecpuid` command, the judgement includes the identifier k of the CPU executing the code C.

The intuitive meaning of the judgements is the same as in the high-level system (Section 5.2), with the component describing process states unchanged during the execution of scheduler commands. The judgements thus express how the scheduler code changes the relationship between the state of the scheduler on the CPU k and those of processes running on the machine. The proof rule for deriving our judgements is identical to PROG from Figure 15, modulo the addition of k:

$$\frac{\forall l \in \mathsf{labels}(C).\, \forall l' \in \mathsf{next}(C,l).\, I, \Delta \rhd^k_{l'} \{\Delta(l)\}\ \mathsf{comm}(C,l)\ \{\Delta(l')\}}{I, \Delta \vdash_k C}\ \text{PROG-L}$$

Note that the syntactic structure of the OS program (see the beginning of Section 5) ensures that the scheduler always executes with interrupts disabled. Thus, in the rule we are able to follow the control flow of C. The low-level system inherits the proof rules for deriving judgements for primitive commands $I, \Delta \rhd_l \{P\}\ c\ \{Q\}$ in Figure 15, adding the superscript k to $\rhd_l$ and ignoring the rules for `icall(schedule)` and `icall(create)`. It also has a rule for `savecpuid`, which makes use of the index k:

$$\frac{}{I, \Delta \rhd^k_l \{e \mapsto \_\}\ \mathtt{savecpuid}(e)\ \{e \mapsto k\}}\ \text{CPUID}$$

5.4 Putting the two proof systems together

The proof systems presented in Sections 5.2 and 5.3 allow us to reason about the kernel and the scheduler code. We now describe a rule for combining judgements from the two systems, which defines proof obligations for the OS components. This allows us to prove the OS program defined at the beginning of Section 5.

As can be seen from the example of Section 2.2, a scheduler might need to maintain some data structures related to every CPU, which can be accessed by a scheduler invocation running on it. A data structure of this kind in our example scheduler is the element of the `current` array corresponding to the current CPU. Let J_k be an invariant of such data structures for CPU k, which is meant to hold when the scheduler is not running on it. Similar to lock invariants, we do not allow J_k to contain free logical variables or registers, except `ss`. In this case, we can allow `ss` because we have previously required that the kernel cannot modify it. We denote the vector of invariants J_k by J.

Consider assertions $I_\mathsf{K}, \Delta_\mathsf{K}$ and $I_\mathsf{S}, \Delta^k_\mathsf{S}$ for all $k \in \mathsf{CPUid}$, corresponding to the kernel and the scheduler code, respectively:

- $\mathsf{dom}(I_\mathsf{K}) \cap \mathsf{dom}(I_\mathsf{S}) = \emptyset$;
- $\forall l.\, l \notin \mathsf{dom}(\mathsf{K}) \implies \Delta_\mathsf{K}(l) = \mathsf{false}$;
- $\forall l.\, l \notin \mathsf{dom}(\mathsf{S}) \uplus \mathsf{dom}(\mathsf{C}) \uplus \{l_s, l_c\} \implies \Delta^k_\mathsf{S}(l) = \mathsf{false}$.

The proof rule for the program OS is as follows:

$$\frac{\begin{array}{l} I_{\mathsf{K}}, \Delta_{\mathsf{K}} \vdash \mathsf{K} \\ \forall k \in \mathsf{CPUid}.\ I_{\mathsf{S}}, \Delta_{\mathsf{S}}^{k} \vdash_{k} \mathsf{S},\ \ I_{\mathsf{S}}, \Delta_{\mathsf{S}}^{k} \vdash_{k} \mathsf{C} \\ \forall k \in \mathsf{CPUid}.\, \Delta_{\mathsf{S}}^{k}(\texttt{schedule}) = \Delta_{\mathsf{S}}^{k}(l_s) = \Delta_{\mathsf{S}}^{k}(l_c) = \mathsf{SchedState}_{k} \\ \forall k \in \mathsf{CPUid}.\, \Delta_{\mathsf{S}}^{k}(\texttt{create}) {=} (\exists \gamma.\, \gamma(\mathbf{if}) {=} 1 \wedge \mathsf{SchedState}_{k} {*} \mathsf{desc}(\texttt{gr}_1, \gamma) {*} \mathsf{Process}(\gamma)) \end{array}}{I_{\mathsf{K}}, \Delta_{\mathsf{K}} \mid I_{\mathsf{S}}, \{\Delta_{\mathsf{S}}^{k}\}_{k \in \mathsf{CPUid}} \mid J \vdash (\mathsf{S}, \mathsf{C}, \mathsf{K})}\ \mathrm{OS}$$

where

$$\begin{aligned} \mathsf{SchedState}_{k} \ = \ & \exists l, \vec{g}.\, 0 \leqslant \texttt{sp} - \texttt{ss} - m - 1 \leqslant \texttt{StackBound} \wedge \\ & (\texttt{sp} - m - 1)..(\texttt{sp} - 1) \mapsto l\vec{g} * \texttt{sp}..(\texttt{ss} + \texttt{StackSize} - 1) \mapsto \_ * \\ & \qquad J_{k} * \mathsf{Process}([\mathbf{ip} : l, \mathbf{if} : 1, \mathbf{ss} : \texttt{ss}, \mathbf{sp} : \texttt{sp} - m - 1, \vec{\mathbf{gr}} : \vec{g}]). \end{aligned}$$

The first two premisses require us to prove the kernel and the scheduler code in their respective proof systems. The rest define pre- and postconditions for `schedule` and `create` by fixing the assertions at the corresponding labels. This is done using the predicate $\mathsf{SchedState}_{k}$, which describes the state of a scheduler invocation at CPU k just after it is called using `icall` or before it returns by executing `iret`.

According to the penultimate premiss of the proof rule, when `schedule` is called the stack satisfies the bound on stack usage. The scheduler gets the ownership of the per-CPU data structure J_k, a part of the stack of the process being preempted (which contains the values of registers saved upon the call together with the empty slots), and a Process predicate consistent with the registers saved on the stack. The predicate certifies that, when the scheduler starts executing, the state of the preempted process in the machine corresponds to its state in the abstract machine. The `schedule` routine has to re-establish the same assertion before returning. In the case when it schedules a different process, this will be done using a different Process predicate. However, since the scheduler can only get a Process predicate in the precondition of `schedule` (and when a new process is created; see below), its postcondition guarantees that the process being scheduled has the same register values it had last time it was preempted. Note that the pre- and postconditions of `schedule` mirror the second premiss of the PROG-H rule. Thus, the assumptions it makes about the kernel are justified by the proof of the latter in the high-level system. Also, the per-CPU state of the scheduler J_k is treated similarly to a piece of state protected by a lock: a scheduler invocation gets its ownership when the scheduler starts executing, and gives it up after giving the control back to a process.

The precondition of `create` is similar to that of `schedule`, but additionally assumes a process descriptor for a new process with the address in $\texttt{gr}_1$, and a corresponding Process assertion initialised according to the information in the descriptor. This descriptor is guaranteed to be provided by the kernel by the precondition of the CREATE rule. Adding the new Process assertion can be understood intuitively as creating a fresh virtual CPU for the new process in the abstract machine.

5.5 Extending the logic

Our logic considers scheduling interfaces providing only the fundamental routines for context switch and process creation. However, our simple treatment of scheduler routines allows extending the logic when routines are added to its interface by:

- adding axioms, similar to SCHED and CREATE, for the new routines to the high-level proof system, specifying the pieces of state transferred between the scheduler and the kernel at calls to and returns from the routines; and
- adding new obligations for the routines to the proof rule from Section 5.4, to be discharged using the low-level proof system.

In this case, the pre- and postconditions of the routines in the low-level proof system should mirror those of the axioms in the high-level proof system, similarly to how this is the case for `schedule` and `create`. In Section 7 we demonstrate how a similar approach can be used to deal with features that break through the virtual CPU abstraction implemented by a scheduler, such as access to the interrupt status flag by the kernel.

6 Verifying the example scheduler

We have used the logic to manually construct a proof of the example scheduler of Section 2.2, establishing the judgements about `schedule` and `create` required by the OS proof rule from Section 5.4.[3] By the soundness theorem for our logic (presented in Section 8), this implies that any property of a piece of high-level code proved in concurrent separation logic, including memory safety and functional correctness, holds of the code when it is managed by the example scheduler. In particular, this is true of the properties of the code in Figure 4 described in Section 4. The full proof is given in a Supplementary Appendix available at http://dx.doi.org/10.1017/S0956796813000075 (Gotsman & Yang, 2013). Here, we present only lock and per-CPU scheduler invariants, together with a sketch of the proof of `schedule`.

The invariants of runqueue locks are as follows:

$$I(\texttt{runqueue\_lock}[k]) = \exists x,y,z.\, \texttt{runqueue}[k] \mapsto z * \\ \mathsf{desc}_0(z,\_) * z.\texttt{prev} \mapsto y * z.\texttt{next} \mapsto x * \mathsf{dll}_\Lambda(x,z,z,y),$$

where $\Lambda(d) = \exists \gamma.\, \mathsf{desc}_0(d,\gamma) * \mathsf{Process}(\gamma)$ and desc_0 is defined in Section 5.2. Thus, a runqueue for a CPU k contains a list of descriptors of preempted processes together with Process predicates matching the state stored in them. The per-CPU scheduler invariants are:

$$J_k = \exists d.\, (d.\texttt{kernel\_stack} = \texttt{ss}) \wedge \texttt{current}[k] \mapsto d * \\ d.\texttt{prev} \mapsto \_ * d.\texttt{next} \mapsto \_ * d.\texttt{timeslice} \mapsto \_ * d.\texttt{saved\_sp} \mapsto \_.$$

Thus, the invariant for CPU k includes the descriptor of the process currently running on the CPU. We also know that the current stack is the one identified by its descriptor (recall that the kernel cannot modify the `ss` register).

[3] Since we do not support modular reasoning about procedures, we constructed a proof *schema* for the body of `fork` in Figure 3, which is meant to be instantiated and inlined at every use.

Below we give a sketch of the proof of the context-switch routine `schedule` with the following main idea. When an invocation of `schedule` acquires the runqueue lock and removes a descriptor from the runqueue, it gets the ownership of the corresponding Process predicate, which lets it schedule the process by establishing the postcondition $\mathsf{SchedState}_k$ of `schedule`. When the process is preempted again, `schedule` receives the Process predicate in its precondition $\mathsf{SchedState}_k$. This predicate and the state in J_k let the scheduler insert the descriptor back into the runqueue while maintaining its invariant.

To make sure that the kernel leaves enough space on the stack for the activation records of `schedule` and `load_balance` or `create`, we assume that

$$\mathtt{StackSize} - \mathtt{StackBound} \geqslant 2 \cdot m + 2 + 4 \cdot \mathtt{sizeof}(\mathtt{int}) + 2 \cdot \mathtt{sizeof}(\mathtt{Process}*).$$

We abbreviate `SCHED_FRAME` to s. Below k is the identifier of the CPU for which the proof is done.

$$\{\mathsf{SchedState}_k\}$$

```
int cpu;
Process *old_process;
```

$$\begin{array}{l}
\{cpu, old\_process \Vdash \exists l, \vec{g}, d.\, \mathtt{if} = 0 \land \\
\;d.\mathtt{kernel\_stack} = \mathtt{ss} \land \\
\;0 \leqslant \mathtt{sp} - \mathtt{ss} - m - s - 1 \leqslant \mathtt{StackBound} \land \\
\;\mathtt{current}[k] \mapsto d * d.\mathtt{prev} \mapsto \_ * d.\mathtt{next} \mapsto \_ * \\
\;d.\mathtt{timeslice} \mapsto \_ * d.\mathtt{saved\_sp} \mapsto \_ * \\
\;(\mathtt{sp} - s - m - 1)..(\mathtt{sp} - s - 1) \mapsto l\vec{g} * \\
\;\mathtt{sp}..(\mathtt{ss} + \mathtt{StackSize} - 1) \mapsto \_ * \\
\;\mathsf{Process}([\mathbf{ip} : l, \mathbf{if} : 1, \mathbf{ss} : \mathtt{ss}, \mathbf{sp} : \mathtt{sp} - s - m - 1, \vec{\mathbf{gr}} : \vec{g}])\}
\end{array}$$

Unpack $\mathsf{SchedState}_k$.
Local variables `cpu` and `old_process` are allocated on the stack.
The assertion $d.\mathtt{timeslice} \mapsto \_$ allows us to prove the following commands manipulating the `timeslice` field of the current descriptor.

```
savecpuid(&cpu);
load_balance(cpu);
old_process = current[cpu];
```

... // *update the timeslice of old_process*

```
if (old_process->timeslice) iret();
old_process->timeslice = SCHED_QUANTUM;
```

$$\begin{array}{l}
\{cpu, old\_process \Vdash \exists l, \vec{g}.\, \mathtt{if} = 0 \land \\
\;old\_process.\mathtt{kernel\_stack} = \mathtt{ss} \land \\
\;cpu = k \land 0 \leqslant \mathtt{sp} - \mathtt{ss} - m - s - 1 \leqslant \mathtt{StackBound} \land \\
\;\mathtt{current}[k] \mapsto old\_process * \\
\;old\_process.\mathtt{prev} \mapsto \_ * old\_process.\mathtt{next} \mapsto \_ * \\
\;old\_process.\mathtt{timeslice} \mapsto \_ * \\
\;old\_process.\mathtt{saved\_sp} \mapsto \_ * \\
\;(\mathtt{sp} - s - m - 1)..(\mathtt{sp} - s - 1) \mapsto l\vec{g} * \\
\;\mathtt{sp}..(\mathtt{ss} + \mathtt{StackSize} - 1) \mapsto \_ * \\
\;\mathsf{Process}([\mathbf{ip} : l, \mathbf{if} : 1, \mathbf{ss} : \mathtt{ss}, \mathbf{sp} : \mathtt{sp} - s - m - 1, \vec{\mathbf{gr}} : \vec{g}])\}
\end{array}$$

`cpu` is now equal to k, and `old_process` points to the descriptor of the current process.

```
lock(runqueue_lock[cpu]);
```

$$\begin{array}{l}
\{cpu, old\_process \Vdash \mathsf{locked}(\mathtt{runqueue\_lock}[k]) * \\
\;\exists l, \vec{g}.\, \mathtt{if} = 0 \land old\_process.\mathtt{kernel\_stack} = \mathtt{ss} \land \\
\;cpu = k \land 0 \leqslant \mathtt{sp} - \mathtt{ss} - m - s - 1 \leqslant \mathtt{StackBound} \land \\
\;\mathtt{current}[k] \mapsto old\_process * \\
\;old\_process.\mathtt{prev} \mapsto \_ * old\_process.\mathtt{next} \mapsto \_ * \\
\;old\_process.\mathtt{timeslice} \mapsto \_ * \\
\;old\_process.\mathtt{saved\_sp} \mapsto \_ * \\
\;(\mathtt{sp} - s - m - 1)..(\mathtt{sp} - s - 1) \mapsto l\vec{g} * \\
\;\mathtt{sp}..(\mathtt{ss} + \mathtt{StackSize} - 1) \mapsto \_ * \\
\;\mathsf{Process}([\mathbf{ip} : l, \mathbf{if} : 1, \mathbf{ss} : \mathtt{ss}, \mathbf{sp} : \mathtt{sp} - s - m - 1, \vec{\mathbf{gr}} : \vec{g}]) *
\end{array}$$

Acquiring the lock gets us ownership of the locked predicate and the lock invariant.
This allows us to prove the commands below that manipulate the runqueue.

$$\begin{array}{l}
\exists x,y,z.\, \texttt{runqueue}[k] \mapsto z * \\
\mathsf{desc}_0(z,\_) * z.\texttt{prev} \mapsto y * z.\texttt{next} \mapsto x * \mathsf{dll}_\Lambda(x,z,z,y)\}
\end{array}$$

```
insert_node_after(runqueue[cpu]->prev,
    old_process);
current[cpu] = runqueue[cpu]->next;
remove_node(current[cpu]);
old_process->saved_sp = _sp;
```

$$\begin{array}{l}
\{(cpu, old\_process \Vdash \mathsf{locked}(\texttt{runqueue\_lock}[k]) * \\
\exists l,\vec{g}.\, \texttt{if} = 0 \land old\_process.\texttt{kernel\_stack} = \texttt{ss} \land \\
cpu = k \land 0 \leqslant \texttt{sp} - \texttt{ss} - m - s - 1 \leqslant \texttt{StackBound} \land \\
\texttt{current}[k] \mapsto x * old\_process.\texttt{prev} \mapsto y * \\
old\_process.\texttt{next} \mapsto z * old\_process.\texttt{timeslice} \mapsto \_ * \\
old\_process.\texttt{saved\_sp} \mapsto \texttt{sp} * \\
(\texttt{sp} - s - m - 1)..(\texttt{sp} - s - 1) \mapsto l\vec{g} * \\
\texttt{sp}..(\texttt{ss} + \texttt{StackSize} - 1) \mapsto \_ * \\
\mathsf{Process}([\mathbf{ip} : l, \mathbf{if} : 1, \mathbf{ss} : \texttt{ss}, \mathbf{sp} : \texttt{sp} - s - m - 1, \vec{\mathbf{gr}} : \vec{g}]) * \\
\exists x,y,z,w,\gamma.\, \texttt{runqueue}[k] \mapsto z * \mathsf{desc}_0(z,\_) * \\
z.\texttt{prev} \mapsto old\_process * z.\texttt{next} \mapsto w * \\
\mathsf{desc}_0(x,\gamma) * \mathsf{Process}(\gamma) * x.\texttt{prev} \mapsto \_ * x.\texttt{next} \mapsto \_ * \\
\mathsf{dll}_\Lambda(w,z,old\_process,y)) \lor \ldots\}
\end{array}$$

`old_process` is now at the end of the runqueue, and process that was at the front is now the current one. We are still using the stack of the old process.

We have omitted the case corresponding to the runqueue being originally empty.

```
_sp = current[cpu]->saved_sp;
savecpuid(&cpu);
_ss = current[cpu]->kernel_stack;
```

$$\begin{array}{l}
\{(cpu, old\_process \Vdash \mathsf{locked}(\texttt{runqueue\_lock}[k]) * \\
\exists l,\vec{g},d.\, \texttt{if} = 0 \land d.\texttt{kernel\_stack} = \texttt{ss} \land \\
cpu = k \land 0 \leqslant \texttt{sp} - \texttt{ss} - m - s - 1 \leqslant \texttt{StackBound} \land \\
\texttt{current}[k] \mapsto d * d.\texttt{prev} \mapsto \_ * d.\texttt{next} \mapsto \_ * \\
d.\texttt{timeslice} \mapsto \_ * d.\texttt{saved\_sp} \mapsto \texttt{sp} * \\
(\texttt{sp} - s - m - 1)..(\texttt{sp} - s - 1) \mapsto l\vec{g} * \\
\texttt{sp}..(\texttt{ss} + \texttt{StackSize} - 1) \mapsto \_ * \\
\mathsf{Process}([\mathbf{ip} : l, \mathbf{if} : 1, \mathbf{ss} : \texttt{ss}, \mathbf{sp} : \texttt{sp} - s - m - 1, \vec{\mathbf{gr}} : \vec{g}]) * \\
\exists x,y,z.\, \texttt{runqueue}[k] \mapsto z * \\
\mathsf{desc}_0(z,\_) * z.\texttt{prev} \mapsto y * z.\texttt{next} \mapsto x * \mathsf{dll}_\Lambda(x,z,z,y)\}
\end{array}$$

We are now using the stack of the new process.
The descriptor of the old process has been merged into the dll predicate representing the runqueue.

```
unlock(runqueue_lock[cpu]);
```

$$\begin{array}{l}
\{cpu, old\_process \Vdash \exists l,\vec{g},d.\, \texttt{if} = 0 \land \\
d.\texttt{kernel\_stack} = \texttt{ss} \land \\
0 \leqslant \texttt{sp} - \texttt{ss} - m - s - 1 \leqslant \texttt{StackBound} \land \\
\texttt{current}[k] \mapsto d * d.\texttt{prev} \mapsto \_ * d.\texttt{next} \mapsto \_ * \\
d.\texttt{timeslice} \mapsto \_ * d.\texttt{saved\_sp} \mapsto \_ * \\
(\texttt{sp} - s - m - 1)..(\texttt{sp} - s - 1) \mapsto l\vec{g} * \\
\texttt{sp}..(\texttt{ss} + \texttt{StackSize} - 1) \mapsto \_ * \\
\mathsf{Process}([\mathbf{ip} : l, \mathbf{if} : 1, \mathbf{ss} : \texttt{ss}, \mathbf{sp} : \texttt{sp} - s - m - 1, \vec{\mathbf{gr}} : \vec{g}])\}
\end{array}$$

After releasing the lock, we give up the ownership of the runqueue and the locked predicate.

```
// We deallocate local variables here
```

$\{\mathsf{SchedState}_k\}$

```
iret();
```

Note that the above proof would not go through if we forgot to acquire the runqueue lock before accessing it in `schedule`. This is because, according to the STORE axiom from Figure 15, we need to have ownership of a memory cell in order to access it. Without acquiring the lock, we would not get the ownership of the doubly-linked list representing the runqueue and, hence, would not be able to justify the correctness of runqueue manipulations.

Our logic is not tied to the particular scheduler implementation we consider here. For example, if we represented the runqueue using a red-black tree sorted by process priorities, like in the newer versions of the Linux kernel (Love, 2010), then we would be able to verify the resulting scheduler as above, but with a new runqueue lock invariant.

7 Breaking through the scheduler abstraction: per-CPU data structures

Even though a scheduler is supposed to provide an illusion of running on a dedicated *virtual* CPU to every process, in practice, some features available to the kernel code can break through this abstraction: e.g., a process can disable preemption (which for our machine corresponds to disabling interrupts) and become aware of the *physical* CPU on which it is currently executing. So far we have ignored this possibility by not allowing the kernel to access the `if` register or execute the `savecpuid` command (Section 5). One way in which OS kernels, such as Linux, use preemption disabling is for implementing so-called *per-CPU data structures* (Bovet & Cesati, 2005)—arrays indexed by CPU identifiers such that a process can only access an entry in an array when it runs on the corresponding CPU. This is widely used to implement CPU-local caches of data, which can be accessed without synchronisation with processes running on other CPUs.

The code in Figure 18, whose proof we explain below, illustrates this by the example of a memory allocator, whose routines can be called concurrently by multiple processes. The allocator manages nodes of type `Node`, which it stores in a doubly-linked list `free_list`. Since multiple invocations of the allocator routines can try to access the free list concurrently, such accesses have to be protected by `list_lock`. To avoid this synchronisation in most cases, the deallocation routine shown in Figure 18 first stores nodes in a CPU-local cache; only when this cache overflows does the routine acquire `list_lock` and move the nodes from the cache to the free list. An allocation routine, which we have omitted, could benefit from a similar optimisation, by trying to allocate a node from the CPU-local cache first and accessing the shared free list only when this fails. For the manipulations of a CPU-local cache to be safe, we need to make sure that at most one allocator invocation can access it at a time. We achieve this by disabling interrupts, and hence, preemption, for the duration of the access, using a pair of new commands `cli` (for disabling interrupts) and `sti` (for enabling them). We also use the `savecpuid` command to index into the array of per-CPU caches.

The use of per-CPU data structures makes it more challenging to separate the verification of the kernel from that of the scheduler, as this exposes the notion of a physical CPU that a scheduler is meant to hide. We now show that we can deal with such implementation exposures while preserving the level of abstraction our logic has enabled so far. Instead of exposing the low-level meaning of concepts such as interrupts and physical CPUs in the logic, our approach is to hide them behind an axiomatic interface that allows only reasoning about their intended uses in the kernel, such as per-CPU data structures.

We extend the set of primitive commands from Figure 9 with the above-mentioned commands for disabling and enabling interrupts, meant for the use by the kernel code only:

$$c \ ::= \ \ldots \mid \texttt{cli} \mid \texttt{sti}$$

```
struct Node {
  Node *prev, *next;
  int data;
};

Node *free_list; // a cyclic doubly−linked list with a sentinel node
Lock *list_lock; // protects the list
Node *free_cache[NCPUS]; // CPU−local caches of free nodes
                         // (cyclic doubly−linked lists with sentinel nodes)
int count[NCPUS]; // number of nodes in each CPU−local cache

void free(Node *n) {
  int cpu;
  $\{n, cpu \Vdash n.\mathtt{prev} \mapsto \_ * n.\mathtt{next} \mapsto \_ * n.\mathtt{data} \mapsto \_ *$
   $\mathtt{sp}..(\mathtt{ss}+\mathtt{StackSize}-1) \mapsto \_ \wedge \mathtt{sp} = \mathtt{ss}+2\cdot\mathtt{sizeof}(\mathtt{Request}*)\}$
  cli();
  $\{n, cpu \Vdash \exists x,y,z,i,k.\, \&\mathtt{count}[k] \mapsto i * \&\mathtt{free\_cache}[k] \mapsto z * z.\mathtt{prev} \mapsto y * z.\mathtt{next} \mapsto x *$
   $z.\mathtt{data} \mapsto \_ * \mathsf{dll}^i_\Lambda(x,z,z,y) * \mathsf{CPU}(k) * n.\mathtt{prev} \mapsto \_ * n.\mathtt{next} \mapsto \_ * n.\mathtt{data} \mapsto \_ *$
   $\mathtt{sp}..(\mathtt{ss}+\mathtt{StackSize}-1) \mapsto \_ \wedge \mathtt{sp} = \mathtt{ss}+2\cdot\mathtt{sizeof}(\mathtt{Request}*)\}$
  savecpuid(&cpu);
  $\{n, cpu \Vdash \exists x,y,z,i.\, \&\mathtt{count}[cpu] \mapsto i * \&\mathtt{free\_cache}[cpu] \mapsto z * z.\mathtt{prev} \mapsto y * z.\mathtt{next} \mapsto x *$
   $z.\mathtt{data} \mapsto \_ * \mathsf{dll}^i_\Lambda(x,z,z,y) * \mathsf{CPU}(cpu) * n.\mathtt{prev} \mapsto \_ * n.\mathtt{next} \mapsto \_ * n.\mathtt{data} \mapsto \_ *$
   $\mathtt{sp}..(\mathtt{ss}+\mathtt{StackSize}-1) \mapsto \_ \wedge \mathtt{sp} = \mathtt{ss}+2\cdot\mathtt{sizeof}(\mathtt{Request}*)\}$
  insert_node_after(free_cache[cpu], n);
  count[cpu]++;
  $\{n, cpu \Vdash \exists x,y,z,i.\, \&\mathtt{count}[cpu] \mapsto i * \&\mathtt{free\_cache}[cpu] \mapsto z * z.\mathtt{prev} \mapsto y * z.\mathtt{next} \mapsto x *$
   $z.\mathtt{data} \mapsto \_ * \mathsf{dll}^i_\Lambda(x,z,z,y) * \mathsf{CPU}(cpu) *$
   $\mathtt{sp}..(\mathtt{ss}+\mathtt{StackSize}-1) \mapsto \_ \wedge \mathtt{sp} = \mathtt{ss}+2\cdot\mathtt{sizeof}(\mathtt{Request}*)\}$
  if (count[cpu] > LIMIT) {
    lock(list_lock);
    $\{n, cpu \Vdash \exists x,y,z,i.\, \&\mathtt{count}[cpu] \mapsto i * \&\mathtt{free\_cache}[cpu] \mapsto z * z.\mathtt{prev} \mapsto y * z.\mathtt{next} \mapsto x *$
     $z.\mathtt{data} \mapsto \_ * \mathsf{dll}^i_\Lambda(x,z,z,y) * \mathsf{CPU}(cpu) *$
     $\exists x',y',z'.\, \&\mathtt{free\_list} \mapsto z' * z'.\mathtt{prev} \mapsto y' * z'.\mathtt{next} \mapsto x' * z'.\mathtt{data} \mapsto \_ * \mathsf{dll}_\Lambda(x',z',z',y') *$
     $\mathtt{sp}..(\mathtt{ss}+\mathtt{StackSize}-1) \mapsto \_ \wedge \mathtt{sp} = \mathtt{ss}+2\cdot\mathtt{sizeof}(\mathtt{Request}*)\}$
    move_contents(free_cache[cpu], free_list);
    $\{n, cpu \Vdash \exists z.\, \&\mathtt{count}[cpu] \mapsto \_ * \&\mathtt{free\_cache}[cpu] \mapsto z * z.\mathtt{prev} \mapsto z * z.\mathtt{next} \mapsto z *$
     $z.\mathtt{data} \mapsto \_ * \mathsf{CPU}(cpu) *$
     $\exists x',y',z'.\, \&\mathtt{free\_list} \mapsto z' * z'.\mathtt{prev} \mapsto y' * z'.\mathtt{next} \mapsto x' * z'.\mathtt{data} \mapsto \_ * \mathsf{dll}_\Lambda(x',z',z',y') *$
     $\mathtt{sp}..(\mathtt{ss}+\mathtt{StackSize}-1) \mapsto \_ \wedge \mathtt{sp} = \mathtt{ss}+2\cdot\mathtt{sizeof}(\mathtt{Request}*)\}$
    unlock(list_lock);
    count[cpu] = 0;
  }
  $\{n, cpu \Vdash \exists x,y,z,i.\, \&\mathtt{count}[cpu] \mapsto i * \&\mathtt{free\_cache}[cpu] \mapsto z * z.\mathtt{prev} \mapsto y * z.\mathtt{next} \mapsto x *$
   $z.\mathtt{data} \mapsto \_ * \mathsf{dll}^i_\Lambda(x,z,z,y) * \mathsf{CPU}(cpu) *$
   $\mathtt{sp}..(\mathtt{ss}+\mathtt{StackSize}-1) \mapsto \_ \wedge \mathtt{sp} = \mathtt{ss}+2\cdot\mathtt{sizeof}(\mathtt{Request}*)\}$
  sti();
  $\{n, cpu \Vdash \mathtt{sp}..(\mathtt{ss}+\mathtt{StackSize}-1) \mapsto \_ \wedge \mathtt{sp} = \mathtt{ss}+2\cdot\mathtt{sizeof}(\mathtt{Request}*)\}$
}
```

Fig. 18. A memory deallocation routine using per-CPU caches of free nodes.

$$\text{For } v_1, v_2 \in \mathsf{CPUid} \cup \{\bot\} \text{ let } v_1 \circ v_2 = v \iff (v_1 = v \wedge v_2 = \bot) \vee (v_1 = \bot \wedge v_2 = v)$$

$$\begin{array}{lll}
(r,h,L,v) \models_\eta \mathsf{CPU}(e) & \text{iff} & h = [\,],\ L = \emptyset,\ [\![e]\!]_r = v \text{ and } v \in \mathsf{CPUid} \\
(r,h,L,v) \models_\eta P_1 * P_2 & \text{iff} & \exists h_1, h_2, L_1, L_2, v_1, v_2.\ h = h_1 \uplus h_2,\ L = L_1 \uplus L_2,\ v = v_1 \circ v_2, \\
 & & (r, h_1, L_1, v_1) \models_\eta P_1 \text{ and } (r, h_2, L_2, v_2) \models_\eta P_2 \\
(r,h,L,v) \models_\eta \mathsf{emp} & \text{iff} & h = [\,],\ L = \emptyset \text{ and } v = \bot \\
(r,h,L,v) \models_\eta P \wedge Q & \text{iff} & (r,h,L,v) \models_\eta P \text{ and } (r,h,L,v) \models_\eta Q
\end{array}$$

Fig. 19. Semantics of high-level assertions adjusted for handling per-CPU data structures. The semantics of assertions not shown is adjusted similarly.

We furthermore lift the restriction we made in Section 5 that prohibits the kernel code from using the `savecpuid` command. The semantics of our programming language is adjusted by adding the following clauses for the new commands to the transition relation from Figure 11:

$$\begin{array}{ll}
(k,(r[\mathtt{if}:1],h,L),l,l') \rightsquigarrow_{\mathtt{cli}} ((r[\mathtt{if}:0],h,L),l'); & (k,(r[\mathtt{if}:0],h,L),l,l') \not\rightsquigarrow_{\mathtt{cli}}\ ; \\
(k,(r[\mathtt{if}:0],h,L),l,l') \rightsquigarrow_{\mathtt{sti}} ((r[\mathtt{if}:1],h,L),l'); & (k,(r[\mathtt{if}:1],h,L),l,l') \rightsquigarrow_{\mathtt{sti}} \top.
\end{array}$$

Note that according to this semantics, calling `cli` twice on a CPU freezes it and calling `sti` twice crashes the system.

To handle the new commands in the high-level proof system, we extend its assertion language from Figure 13 with the predicate $\mathsf{CPU}(e)$, which certifies that the process owning it is running on the CPU with the identifier e:

$$P ::= \ \dots \mid \mathsf{CPU}(e)$$

This requires us to adjust the domain over which assertions of the high-level proof system are interpreted, replacing State defined in (3) by

$$\mathsf{State_I} = \mathsf{Context} \times \mathsf{Heap} \times \mathsf{Lockset} \times (\mathsf{CPUid} \cup \{\bot\}).$$

The last component records the CPU that the current process is executing on, or $\bot$ if the assertion does not carry such information. The assertion semantics from Figure 13 is adjusted as shown in Figure 19. Note that, according to this semantics, the assertion $\mathsf{CPU}(k_1) * \mathsf{CPU}(k_2)$ is inconsistent for all $k_1, k_2 \in \mathsf{CPUid}$. This ensures that an assertion can denote at most one $\mathsf{CPU}(e)$ predicate: a process cannot be at two CPUs at the same time. We denote the extended set of assertions by $\mathsf{Assert_I}$.

Judgements of the high-level proof system now have the forms $I, H, \Delta \vdash C$ or $I, H, \Delta \rhd_l \{P\}\ c\ \{Q\}$, where the additional component H is a vector of invariants describing the kernel data structures local to every CPU in the system. We do not allow invariants in H to contain registers or free occurrences of logical variables and require them to have an empty lockset (Section 4.2). We also require that invariants in I and H do not own CPU predicates: $\forall \eta, (r,h,L,v) \in [\![I(\ell)]\!]_\eta.\, v = \bot$ and the same for H. To preserve the soundness of the CREATE proof rule from Figure 16, we have to impose the same requirement on the $\mathsf{desc}(d, \gamma)$ predicate and the assertion P used in the rule. All these restrictions ensure that a CPU predicate never gets transferred between processes. This is necessary for soundness,

$$\frac{}{I,H,\Delta \rhd_l \{\mathsf{emp}\}\ \texttt{cli}\ \{\exists k.\mathsf{CPU}(k) * H_k\}}\ \textsc{Sti}$$

$$\frac{}{I,H,\Delta \rhd_l \{\exists k.\mathsf{CPU}(k) * H_k\}\ \texttt{sti}\ \{\mathsf{emp}\}}\ \textsc{Cli}$$

$$\frac{}{I,H,\Delta \rhd_l \{e \mapsto \_ * \mathsf{CPU}(\_)\}\ \texttt{savecpuid}(e)\ \{\exists k.\, e \mapsto k * \mathsf{CPU}(k)\}}\ \text{CPU}\textsc{id-fixed}$$

$$\frac{}{I,H,\Delta \rhd_l \{e \mapsto \_\}\ \texttt{savecpuid}(e)\ \{e \mapsto \_\}}\ \text{CPU}\textsc{id-any}$$

Fig. 20. Proof rules for per-CPU data structures.

since such a predicate makes a statement about the physical CPU that only a particular process is executing on.

The proof rules for the new commands accessible to kernel code are given in Figure 20; these extend the rules in Figures 15 and 16. The rules express a simple reasoning method similar to that used for lock invariants (Section 4): a process executing `cli` gets the ownership of a CPU predicate for some CPU identifier and the corresponding per-CPU data structure (Cli); it gives both up when it executes `sti` (Sti). In between calling `cli` and `sti`, the process may modify the CPU-local data structure in any way. The last two axioms are similar to CPUid from Section 5.3. CPUid-fixed ensures that the `savecpuid` command returns the value consistent with the CPU predicate owned by the process. According to CPUid-any, we cannot make any constraints on the value returned by `savecpuid` without such a predicate. We note that our proof rules for interrupts are essentially identical to those in Feng *et al.* (2008a). Our goal here is not to propose a new logic for interrupts, but to demonstrate how natural reasoning methods for such low-level features can be integrated into our logic for preemptive kernels.

Figure 18 gives a proof of the example deallocation routine using our proof rules. We assume the following lock and per-CPU data structure invariants:

$$\begin{aligned}
I(\texttt{list\_lock}) &= \exists x,y,z.\, \&\texttt{free\_list} \mapsto z * \\
&\quad z.\texttt{prev} \mapsto y * z.\texttt{next} \mapsto x * z.\texttt{data} \mapsto \_ * \mathsf{dll}_\Lambda(x,z,z,y);\\
H_k &= \exists x,y,z,i.\, \&\texttt{count}[k] \mapsto i * \&\texttt{free\_cache}[k] \mapsto z * \\
&\quad z.\texttt{prev} \mapsto y * z.\texttt{next} \mapsto x * z.\texttt{data} \mapsto \_ * \mathsf{dll}^i_\Lambda(x,z,z,y),
\end{aligned}$$

where $\Lambda(x) = x.\texttt{data} \mapsto \_$ and dll^i_Λ is the straightforward generalisation of the dll_Λ predicate from Figure 13 that specifies the number i of nodes in the list.

The soundness statement for our logic, which we discuss next, includes the extension presented in this section.

8 Soundness

A typical approach to proving the soundness of a logic such as ours would be to define an operational semantics of the abstract machine with one CPU per process the scheduler is supposed to implement. Then, the soundness statement of the high-level proof system could restrict the behaviour of this machine, and that of the low-level proof system would establish that any behaviour of the concrete machine with the scheduler is reproducible in the abstract one. As a corollary, the statements about the behaviour of the kernel proved in the high-level proof system would be carried over to the concrete machine.

Following this approach is difficult in our case for the following reason. In our logic, the pieces of state whose ownership is transferred between the scheduler and the kernel can be described by arbitrary logical assertions, e.g., $\mathsf{desc}(d, \gamma)$ from Section 5.1. In some cases, e.g., when these assertions are imprecise (O'Hearn, 2007), their transfer from the kernel to the scheduler is hard to express operationally when defining a semantics of the abstract machine; see Gotsman *et al.* (2011) for a discussion. The situation would be worse had we based our logic on a more advanced modular concurrency logic, such as deny-guarantee (Dinsdale-Young *et al.*, 2010), which would be needed to handle real OS code. This is because proofs of soundness for such logics do not give an operational semantics to separate components of a program.

For this reason, we do not define an operational semantics of the abstract machine, and neither of the two proof systems of our logic is proved sound with respect to any semantics alone. Instead, our soundness statement interprets a proof of the kernel in the high-level system and a proof of the scheduler in the low-level one together with respect to the semantics of the concrete machine. This makes it convenient for us to prove the soundness of the fragment of our logic inherited from concurrent separation logic in a way (Gotsman *et al.*, 2011) other than its original proof (Brookes, 2007), as the latter relied on giving a semantics to separate processes of the program in isolation. We therefore start by formulating the soundness statement of the baseline concurrency logic from Section 4, which allows us to introduce the basic techniques we use in stating soundness.

8.1 Soundness of the baseline logic

Assume a proof $I, \Delta \vdash C$ in the logic of Section 4 and an environment η giving the values of the logical variables used in this proof. Consider a point in an execution of the machine of Section 3 when the CPUs are at program positions $l_1, \dots, l_{\mathtt{NCPUS}} \in \mathsf{labels}(C)$. We formulate the soundness of the logic by extracting the set of configurations that the machine can be in at this point from the proof of C. We achieve this by combining the process-local states, defined by Δ, and the states protected by the free locks, defined by I, as per Figure 7. This formalises the intuitive explanations of the reasoning approach of concurrent separation logic given in Section 2.3.

The mapping Δ describes the states that can be owned by processes at the program positions $l_1, \dots, l_{\mathtt{NCPUS}}$: $[\![\Delta(l_1)]\!]_\eta, \dots, [\![\Delta(l_{\mathtt{NCPUS}})]\!]_\eta \in \mathscr{P}(\mathsf{State})$. We now combine these states to get a configuration from Config (Figure 8) that describes the contexts of all CPUs and the part of the heap and the lockset of the machine belonging to the local state of any process. To this end, we lift states to configurations using the operation $\lfloor\cdot\rfloor_k^{\mathsf{BA}}$ in Figure 22 below, which tags their contexts with a CPU identifier $k \in \mathsf{CPUid}$. We then combine the resulting configurations using the operation $\star_{\mathsf{B}}$ in Figure 23 below, which merges the contexts for different CPUs and takes the union of the heaps and locksets. In Figures 22 and 23 we also summarise all other operations for lifting states to configurations and combining the latter that we use in formulating soundness.

Using the above operations, the set of configurations describing the process-local part of the machine state is thus given by the following predicate over Config:

$$\mathsf{IProc}_\eta(l_1,\ldots,l_{\mathtt{NCPUS}}) = \underset{k\in\mathsf{CPUid}}{\circledast_{\mathsf{B}}} \lfloor [\![\Delta(l_k)]\!]_\eta \rfloor_k^{\mathsf{BA}},$$

where $\circledast_{\mathsf{B}}$ is the iterated version of $\star_{\mathsf{B}}$. This predicate, however, does not describe the whole heap and lockset of the machine, as we have not taken into account their parts belonging to the invariants of free locks, which are not included into the local state of any process (Figure 7). In any configuration $(R,h,L) \in \mathsf{IProc}_\eta(l_1,\ldots,l_{\mathtt{NCPUS}})$, L gives the set of locks held by any process, so that the set of free locks is $\mathsf{Lock} - L$. To combine their invariants, we use the operation $\lfloor\cdot\rfloor^{\mathsf{BL}}$ in Figure 22, which lifts states, meant to come from lock invariants, to configurations by discarding the context and assuming an empty lockset (recall that lock invariants cannot have free register occurrences and are required to have an empty lockset; see Section 4). The following predicate on configurations describes the part of the machine state belonging to all locks from a set L':

$$\mathsf{ILock}_{L'} = \underset{\ell\in L'}{\circledast_{\mathsf{B}}} \lfloor [\![I(\ell)]\!] \rfloor^{\mathsf{BL}}.$$

Here, we omit an environment defining the values of logical variables from $[\![I(\ell)]\!]$, since lock invariants are insensitive to these variables. The set of all configurations the machine can be in when CPUs are at program positions $l_1,\ldots,l_{\mathtt{NCPUS}}$ is obtained by combining the above predicate with $\mathsf{IProc}_\eta(l_1,\ldots,l_{\mathtt{NCPUS}})$:

$$\begin{aligned}\mathsf{IProg}_\eta(l_1,\ldots,l_{\mathtt{NCPUS}}) = \{(R,h_1\uplus h_2,L) \mid \\ (R,h_1,L) \in \mathsf{IProc}_\eta(l_1,\ldots,l_{\mathtt{NCPUS}}) \wedge ([\,],h_2,\emptyset) \in \mathsf{ILock}_{\mathsf{Lock}-L}\}.\end{aligned}$$

Theorem 1
Assume $I,\Delta \vdash C$ in the logic of Section 4, $(R,h,L) \rightarrow_C (R',h',L')$ and $R(k,\mathtt{if}) = 0$ for $k = 1..\mathtt{NCPUS}$. Then for all environments η,

$$(R,h,L) \in \mathsf{IProg}_\eta(R(1,\mathtt{ip}),\ldots,R(\mathtt{NCPUS},\mathtt{ip})),$$

entails

$$(R',h',L') \in \mathsf{IProg}_\eta(R'(1,\mathtt{ip}),\ldots,R'(\mathtt{NCPUS},\mathtt{ip})).$$

Theorem 1 shows that IProg_η defines an inductive invariant of the system. Since it excludes the error configuration $\top$, the provability of a program in our logic implies its safety.

To summarise, we obtain an overapproximation of the set of machine configurations in two stages: first, we look up the local states at the program positions given in Δ; second, we look up the lock-protected states using the lockset information extracted from the local states. Although this formulation using lookups is easy to understand, it gets unwieldy for more complicated logics, such as our logic for preemptive kernels. We therefore show how to reformulate Theorem 1 in a somewhat less intuitive, but more compact way. First, we replace the map IProc_η from program positions to process-local parts of configurations by a relation. Let

$$\mathsf{IProc}'_\eta = \underset{k\in\mathsf{CPUid}}{\circledast_{\mathsf{B}}} \bigcup_{l\in\mathsf{labels}(C)} \lfloor [\![\Delta(l)]\!]_\eta \cap \mathsf{at}_{\mathsf{B}}(l) \rfloor_k^{\mathsf{BA}},$$

where $\mathsf{at}_\mathsf{B}(l) = \{(r,h,L) \in \mathsf{State} \mid r(\mathtt{ip}) = l\}$. The predicate $\mathsf{IProc}'_\eta \subseteq \mathsf{Config}$ describes the part of the machine state belonging to processes for all possible program positions; it can thus be viewed as an invariant of the process-local state. Since assertions in Δ do not restrict the value of the ip register (Section 4.1), we have to do this explicitly using at_B. Then the set of configurations that the machine can be in at any time is now given by the following predicate:

$$\mathsf{IProg}'_\eta = \bigcup_{L \uplus L' = \mathsf{Lock}} ((\mathsf{IProc}_\eta \star_\mathsf{B} \mathsf{ILock}_{L'}) \cap \mathsf{held}_\mathsf{B}(L)),$$

where $\mathsf{held}_\mathsf{B}(L) = \{(R,h,L) \in \mathsf{Config}\}$. Here we branch over all sets of locks L that could be held by processes and compute the lock-protected state for its complement L'. We then ensure that L is indeed the set of all held locks by intersecting the result with $\mathsf{held}_\mathsf{B}(L)$. The following theorem is equivalent to Theorem 1.

Theorem 2
If $I,\Delta \vdash C$ in the logic of Section 4, then for all environments η, the set of configurations $\mathsf{IProg}'_\eta \cap \{(R,h,L) \mid \forall k = 1..\mathsf{CPUid}. R(k,\mathtt{if}) = 0\}$ is preserved by $\rightarrow_C$.

Theorems 1 and 2 follow from the proof of the soundness statement of our logic for preemptive kernels, which we now formulate using the approach just presented.

8.2 Soundness of the logic for preemptive kernels

Consider a program OS of the form introduced in Section 5 and assume its proof

$$I_\mathsf{K}, H, \Delta_\mathsf{K} \mid I_\mathsf{S}, \{\Delta^k_\mathsf{S}\}_{k \in \mathsf{CPUid}} \mid J \vdash (\mathsf{S},\mathsf{C},\mathsf{K})$$

in our logic for preemptive kernels, including the extension to per-CPU data structures from Section 7. We also assume an environment η giving the values of the logical variables used in the proof. To explain the soundness statement informally, let us fix a point in a machine execution and assume for simplicity that every CPU is executing the scheduler code. We construct an inductive system invariant by conjoining the descriptions of pieces of the machine state owned by different OS components, as per Figure 6. We extract these descriptions from the proof as follows:

- If a CPU k is at a program position l in the scheduler code, then $[\![\Delta^k_\mathsf{S}(l)]\!]_\eta \in \mathscr{P}(\mathsf{SchedState})$ describes the state local to the scheduler invocation running on the CPU, including its context, heap and lockset, and the contexts of the processes it has a permission to schedule.
- The combined lockset of all these states tells us which of the locks accessible to the scheduler are free. As in Section 8.1, this allows us to obtain a description of the whole scheduler state by combining the local states with the invariants of all free scheduler locks given by I_S.
- The combined scheduler state contains not only the part of the heap belonging to it, but also the contexts of all the processes that exist in the machine, including their program positions. By looking up the assertions at these positions in Δ_K, we obtain a description of the local states of the processes, including their heaps and locksets.

$$\begin{aligned}
\mathsf{State} &= \mathsf{Context} \times \mathsf{Heap} \times \mathsf{Lockset} \\
\mathsf{SchedState} &= \mathsf{State} \times \mathscr{M}(\mathsf{Context}) \\
\mathsf{Config} &= (\mathsf{CPUid} \to \mathsf{Context}) \times \mathsf{Heap} \times \mathsf{Lockset} \\
\mathsf{SchedConfig} &= \mathsf{Config} \times \mathscr{M}(\mathsf{Context}) \times \mathscr{P}(\mathsf{CPUid}) \\
\mathsf{KernelConfig} &= \mathscr{M}(\mathsf{Context}) \times \mathsf{Heap} \times \mathsf{Lockset} \times \mathscr{P}(\mathsf{CPUid})
\end{aligned}$$

Fig. 21. A summary of the semantic domains used in formulating soundness.

$$\begin{aligned}
\lfloor p_{\mathsf{B}} \rfloor^{\mathsf{BA}}_{k} &= \{([k:r],h,L) \in \mathsf{Config} \mid (r,h,L) \in p_{\mathsf{B}}\} \\
\lfloor p_{\mathsf{B}} \rfloor^{\mathsf{BL}} &= \{([],h,\emptyset) \in \mathsf{Config} \mid (r,h,\emptyset) \in p_{\mathsf{B}}\} \\
\lfloor p_{\mathsf{S}} \rfloor^{\mathsf{SA}}_{k,V} &= \{(([k:r],h,L),M,V) \in \mathsf{SchedConfig} \mid ((r,h,L),M) \in p_{\mathsf{S}}\} \\
\lfloor p_{\mathsf{S}} \rfloor^{\mathsf{SL}} &= \{((([],h,\emptyset),M,\emptyset) \in \mathsf{SchedConfig} \mid ((r,h,\emptyset),M) \in p_{\mathsf{S}}\} \\
\lfloor p_{\mathsf{K}} \rfloor^{\mathsf{KA}}_{r} &= \{(\{r\},h,L,\{v\}-\{\bot\}) \in \mathsf{KernelConfig} \mid \\
&\qquad (r, h \uplus [r(\mathtt{sp})..(r(\mathtt{ss})+\mathtt{StackSize}-1) : \_], L, v) \in p_{\mathsf{K}} \wedge (r(\mathtt{if}) = 0 \iff v \neq \bot)\} \\
\lfloor p_{\mathsf{K}} \rfloor^{\mathsf{KL}} &= \{(\emptyset,h,\emptyset,\emptyset) \in \mathsf{KernelConfig} \mid (r,h,\emptyset,\bot) \in p_{\mathsf{K}}\}
\end{aligned}$$

Fig. 22. Operations lifting states to configurations. Here, $p_{\mathsf{B}} \in \mathscr{P}(\mathsf{State})$, $p_{\mathsf{S}} \in \mathscr{P}(\mathsf{SchedState})$, $p_{\mathsf{K}} \in \mathscr{P}(\mathsf{State}_{\mathsf{I}})$, $k \in \mathsf{CPUid}$, $V \in \mathscr{P}(\mathsf{CPUid})$ and $r \in \mathsf{Context}$. We have a pair of operations for every domain of states, one for states coming from assertions in the code (marked by A) and another for states coming from lock or per-CPU invariants (marked by L).

- Again, their combined lockset tells us which of the locks accessible to the kernel are free, allowing us to obtain a description of all lock-protected kernel state from the invariants I_{K}.

We now define this construction formally and for the general case. As we have noted in Section 8.1, we do this by packaging the results of all the lookups mentioned in the above explanation into relations and then performing a relational composition on them.

We start by defining an invariant of the part of the machine state owned by the scheduler. Let us again consider a point in a machine execution when every CPU is executing the scheduler code. To combine the scheduler-local states given by Δ^{k}_{S} for all CPUs $k \in \mathsf{CPUid}$, we lift them to configurations in the set SchedConfig defined in Figure 21 (in the figure we also summarise all the other domains used in formulating soundness). A configuration $((R,h,L),M,V)$ describes the combined state of multiple scheduler invocations: R defines the contexts on the corresponding CPUs, h and L the combined heap and lockset, and M the contexts of the processes that the invocations have a permission to schedule. To handle per-CPU data structures, we also add a component V describing the set of CPUs on which processes have disabled interrupts using `cli`. When every CPU is executing the scheduler code, this set is empty; we use the general case below. We lift states in SchedState to configurations in SchedConfig using the operation $\lfloor \cdot \rfloor^{\mathsf{SA}}_{k,V}$, defined in Figure 22 for $k \in \mathsf{CPUid}$ and $V \in \mathscr{P}(\mathsf{CPUid})$. We combine the resulting configurations using the operation $\star_{\mathsf{S}}$ in Figure 23. This is similar to $\star_{\mathsf{B}}$, but additionally combines the information about the processes the scheduler invocations know about.

Thus, at those points in the machine execution when all CPUs are executing scheduler invocations, the part of the machine state local to these invocations is described by the

$$\begin{array}{ll}
\bullet_{\mathsf{B}} & : \mathsf{Config} \times \mathsf{Config} \rightharpoonup \mathsf{Config} \\
\bullet_{\mathsf{S}} & : \mathsf{SchedConfig} \times \mathsf{SchedConfig} \rightharpoonup \mathsf{SchedConfig} \\
\bullet_{\mathsf{K}} & : \mathsf{KernelConfig} \times \mathsf{KernelConfig} \rightharpoonup \mathsf{KernelConfig} \\
\bullet_{\mathsf{SK}} & : \mathsf{SchedConfig} \times \mathsf{KernelConfig} \rightharpoonup \mathsf{Config}
\end{array}$$

$$\begin{array}{rcll}
(R_1,h_1,L_1) \bullet_{\mathsf{B}} (R_2,h_2,L_2) & = & (R_1 \uplus R_2, h_1 \uplus h_2, L_1 \uplus L_2) & \\
((R_1,h_1,L_1),M_1,V_1) \bullet_{\mathsf{S}} ((R_2,h_2,L_2),M_2,V_2) & = & ((R_1 \uplus R_2, h_1 \uplus h_2, L_1 \uplus L_2), M_1 \uplus M_2, V_1 \uplus V_2) & \\
(M_1,h_1,L_1,V_1) \bullet_{\mathsf{K}} (M_2,h_2,L_2,V_2) & = & (M_1 \uplus M_2, h_1 \uplus h_2, L_1 \uplus L_2, V_1 \uplus V_2) & \\
((R,h_1,L_1),M_1,V_1) \bullet_{\mathsf{SK}} (M_2,h_2,L_2,V_2) & = & (R, h_1 \uplus h_2, L_1 \uplus L_2), & \text{if } M_1 = M_2 \text{ and } V_1 = V_2 \\
((R,h_1,L_1),M_1,V_1) \bullet_{\mathsf{SK}} (M_2,h_2,L_2,V_2) & & \text{undefined}, & \text{otherwise}
\end{array}$$

Let $\star_{\mathsf{B}}$, $\star_{\mathsf{S}}$, $\star_{\mathsf{K}}$, $\star_{\mathsf{SK}}$ be the pointwise liftings of $\bullet_{\mathsf{B}}$, $\bullet_{\mathsf{S}}$, $\bullet_{\mathsf{K}}$, $\bullet_{\mathsf{SK}}$ to sets of configurations. For example, for $p_1, p_2 \in \mathscr{P}(\mathsf{Config})$, we define $\star_{\mathsf{B}} : \mathscr{P}(\mathsf{Config}) \times \mathscr{P}(\mathsf{Config}) \to \mathscr{P}(\mathsf{Config})$ as follows: $p_1 \star_{\mathsf{B}} p_2 = \{(R_1,h_1,L_1) \bullet_{\mathsf{B}} (R_2,h_2,L_2) \mid (R_1,h_1,L_1) \in p_1 \wedge (R_2,h_2,L_2) \in p_2\}$.

Fig. 23. Operations for combining configurations. Recall that the $\uplus$ operation on multisets adds up the number of occurrences of each element in its operands.

following predicate over $\mathsf{SchedConfig}$:

$$\bigcircledast_{\mathsf{S}\;k \in \mathsf{CPUid}} \quad \bigcup_{l \in (\mathsf{labels}(\mathsf{S} \uplus \mathsf{C}) \uplus \{l_s, l_c\})} \lfloor [\![\Delta^k_{\mathsf{S}}(l)]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l) \cap \mathsf{if}_{\mathsf{S}}(0) \rfloor^{\mathsf{SA}}_{k,\emptyset},$$

where $\mathsf{at}_{\mathsf{S}}(l) = \{((r,h,L),M) \in \mathsf{SchedState} \mid r(\mathtt{ip}) = l\}$ and $\mathsf{if}_{\mathsf{S}}(v) = \{((r,h,L),M) \in \mathsf{SchedState} \mid r(\mathtt{if}) = v\}$. Like in the definition of IProc'_η in Section 8.1, we branch over all program positions in the scheduler code and combine the local states at these positions given by Δ^k_{S}; we also restrict the value of the ip and if registers explicitly.

We now need to consider the case when a process is running on some CPU k. Let l be its program position. In this case, the scheduler still owns some state associated with the CPU, e.g., the per-CPU scheduler invariant J_k. We describe this state by the following predicate over $\mathsf{SchedState}$:

$$\mathsf{SchedSleep}_k(l) = J_k * \mathtt{sp}..(\mathtt{ss} + \mathtt{StackSize} - 1) \mapsto \_ * \\ \mathsf{Process}([\mathbf{ip} : l, \mathbf{if} : \mathtt{if}, \mathbf{ss} : \mathtt{ss}, \mathbf{sp} : \mathtt{sp}, \vec{\mathbf{gr}} : \vec{\mathtt{gr}}]).$$

Note that, when a scheduler invocation starts executing on a CPU, the invariant J_k is added to its local state, which is why previously we did not have to take it into account when defining the state local to active scheduler invocations. Although assertions in the high-level proof system mention the empty slots of the process stack, the slots in fact belong to the scheduler when the process is preempted. For the sake of uniformity, we choose to count them in the scheduler state even when the process is running and, hence, add them to $\mathsf{SchedSleep}_k(l)$. The $\mathsf{Process}$ predicate in $\mathsf{SchedSleep}_k(l)$ describes the currently running process; it corresponds to the $\mathsf{Process}$ predicate that the scheduler lost when it transferred the control to the process (see the postcondition of schedule in the OS proof rule from Section 5.4). We took the liberty of using if in $\mathsf{SchedSleep}_k(l)$, even though this is prohibited in our logic, since this assertion is used only for formulating soundness.

The following predicate over $\mathsf{SchedConfig}$ describes the scheduler state excluding that protected by free locks:

$$\mathsf{ISched}_\eta = \bigcup_{\substack{V_\mathsf{I} \subseteq V_\mathsf{K},\\ V_\mathsf{S} \uplus V_\mathsf{K} = \mathsf{CPUid}}} \Bigg(\Bigg(\mathop{\circledast_\mathsf{S}}_{k \in V_\mathsf{S}} \bigcup_{l \in (\mathsf{labels}(\mathsf{S} \uplus \mathsf{C}) \uplus \{l_s, l_c\})} \lfloor [\![\Delta^k_\mathsf{S}(l)]\!]_\eta \cap \mathsf{at}_\mathsf{S}(l) \cap \mathsf{if}_\mathsf{S}(0) \rfloor^{\mathsf{SA}}_{k,\emptyset} \Bigg) \star_\mathsf{S}$$
$$\Bigg(\mathop{\circledast_\mathsf{S}}_{k \in V_\mathsf{I}} \bigcup_{l \in \mathsf{labels}(\mathsf{K})} \lfloor [\![\mathsf{SchedSleep}_k(l)]\!]_\eta \cap \mathsf{at}_\mathsf{S}(l) \cap \mathsf{if}_\mathsf{S}(0) \rfloor^{\mathsf{SA}}_{k,\{k\}} \Bigg) \star_\mathsf{S}$$
$$\Bigg(\mathop{\circledast_\mathsf{S}}_{k \in V_\mathsf{K} - V_\mathsf{I}} \bigcup_{l \in \mathsf{labels}(\mathsf{K})} \lfloor [\![\mathsf{SchedSleep}_k(l)]\!]_\eta \cap \mathsf{at}_\mathsf{S}(l) \cap \mathsf{if}_\mathsf{S}(1) \rfloor^{\mathsf{SA}}_{k,\emptyset} \Bigg) \Bigg).$$

Here, we branch over all splittings of CPUs into those executing the scheduler and the kernel code, given by V_S and V_K. We also branch over all sets $V_\mathsf{I} \subseteq V_\mathsf{K}$ of CPUs where processes have disabled interrupts. For every CPU k, we then branch over all possible program positions l. Depending on whether l is in the scheduler or the kernel code, we use either the assertion in the scheduler proof or the invariant $\mathsf{SchedSleep}_k$. In the latter case, V_I determines the last component of the resulting configurations.

To obtain the whole state owned by the scheduler, we take into account the invariants of free locks accessible to it, similarly to how it was done in the definition of IProg'_η in Section 8.1. To this end, we use the operation $\lfloor\cdot\rfloor^{\mathsf{SL}}$ in Figure 22 that converts states in $\mathsf{SchedState}$, meant to come from lock invariants, to configurations in $\mathsf{SchedConfig}$. Then the part of the machine state belonging to the scheduler locks from a set L' is defined by the following predicate:

$$\mathsf{ISchedLock}_{L'} = \mathop{\circledast_\mathsf{S}}_{\ell \in L'} \lfloor [\![I_\mathsf{S}(\ell)]\!] \rfloor^{\mathsf{SL}}.$$

Hence, the invariant of the whole scheduler state is

$$\bigcup_{L \uplus L' = \mathsf{dom}(I_\mathsf{S})} ((\mathsf{ISched}_\eta \star_\mathsf{S} \mathsf{ISchedLock}_{L'}) \cap \mathsf{held}_\mathsf{S}(L)), \tag{5}$$

where $\mathsf{held}_\mathsf{S}(L) = \{((R,h,L),M,V) \in \mathsf{SchedConfig}\}$. In a configuration $((R,h,L), M,V)$ from the set (5), the components M and V give the information about all processes that exist in the system, whether running or preempted, with the former taken into account due to the inclusion of the corresponding $\mathsf{Process}$ predicate into $\mathsf{SchedSleep}_k$. In the following, we use this fact to connect the invariant of the scheduler with that of the kernel. We now proceed to define the latter.

Consider a process with a context r, which could come from a configuration in the scheduler invariant (5). Then its local state is given by $[\![\Delta_\mathsf{K}(r(\mathtt{ip}))]\!]_\eta \in \mathscr{P}(\mathsf{State}_\mathsf{I})$. We now combine such states for different processes in a form appropriate for composing with the scheduler invariant (5). We start by lifting them to configurations in the set $\mathsf{KernelConfig}$ defined in Figure 21. A configuration $(M,h,L,V) \in \mathsf{KernelConfig}$ describes the combined state of multiple processes, with the contexts M, the combined heap h and lockset L and the set V of CPUs on which they have disabled interrupts. We perform the lifting using the operation $\lfloor\cdot\rfloor^{\mathsf{KA}}_r$ in Figure 22 that selects the states with the context $r \in \mathsf{Context}$ and removes the empty slots of the process stack, accounted for in the scheduler state (in $\mathsf{SchedSleep}_k$, if the process is running, and in Δ^k_{S} and I_{S}, if it is preempted). We then combine the resulting configurations using the operation $\star_{\mathsf{K}}$ in Figure 23. The invariant of the process-local part of the machine state is thus given by the following predicate over $\mathsf{KernelConfig}$:

$$\mathsf{IKernel}_\eta = \bigcup_{M \in \mathscr{M}(\mathsf{Context})} \mathop{\circledast_{\mathsf{K}}}_{r \in M} \lfloor [\![\Delta_{\mathsf{K}}(r(\mathtt{ip}))]\!]_\eta \rfloor^{\mathsf{KA}}_r .$$

Here we branch over all possible finite multisets M of contexts of processes that may run on the machine. For every context r in M, the local state of the corresponding process is then determined by the assertion in the proof of the kernel at the program point $r(\mathtt{ip})$, restricted to the states with the context r. Note that the comprehension $r \in M$ over a multiset M considers every instance of an element in the multiset separately.

As before, to obtain the invariant of the whole kernel state, we need to take into account the invariants of free locks accessible to the kernel. Additionally, we need to include kernel per-CPU invariants H_k for all CPUs k where processes have not disabled interrupts (for those where they have, the per-CPU data structures have been merged into their local states). We use the operation $\lfloor \cdot \rfloor^{\mathsf{KL}}$ in Figure 22 to convert states in $\mathsf{State}_{\mathsf{l}}$, meant to come from kernel lock invariants or kernel per-CPU data structure invariants, to configurations in $\mathsf{KernelConfig}$. Since the invariants cannot contain CPU predicates (Section 7), the operation assumes that the last component of the states it is given is always $\bot$. Then the part of the machine state belonging to kernel locks from a set L' or per-CPU invariants for the set of CPUs V are respectively given by

$$\mathsf{IKernelLock}_{L'} = \mathop{\circledast_{\mathsf{K}}}_{\ell \in L'} \lfloor [\![I_{\mathsf{K}}(\ell)]\!] \rfloor^{\mathsf{KL}}; \qquad \mathsf{IPercpu}_V = \mathop{\circledast_{\mathsf{K}}}_{k \in V} \lfloor [\![H_k]\!] \rfloor^{\mathsf{KL}}.$$

Hence, the invariant of the whole kernel state is

$$\bigcup_{\substack{L \uplus L' = \mathsf{dom}(I_{\mathsf{K}}) \\ V \uplus V' = \mathsf{CPUid}}} ((\mathsf{IKernel}_\eta \star_{\mathsf{K}} \mathsf{IKernelLock}_{L'} \star_{\mathsf{K}} \mathsf{IPercpu}_{V'}) \cap \mathsf{held}_{\mathsf{K}}(L) \cap \mathsf{disabled}(V)),$$

where

$$\mathsf{held}_{\mathsf{K}}(L) = \{(M, h, L, V) \in \mathsf{KernelConfig}\};$$
$$\mathsf{disabled}(V) = \{(M, h, L, V) \in \mathsf{KernelConfig}\}.$$

Finally, we compose the above invariant of the kernel with that of the scheduler (5) to obtain an invariant of the whole system. To this end, we use the operation $\star_{\mathsf{SK}}$ in Figure 23, which combines heaps and locksets, provided that the contexts of processes and sets of CPUs on which they disabled interrupts match in both arguments. Then the system invariant is given by the following predicate over Config:

$$\mathsf{IOs}_\eta \;=\; \left(\bigcup_{L \uplus L' = \mathsf{dom}(I_{\mathsf{S}})} ((\mathsf{ISched}_\eta \star_{\mathsf{S}} \mathsf{ISchedLock}_{L'}) \cap \mathsf{held}_{\mathsf{S}}(L)) \right) \star_{\mathsf{SK}}$$
$$\left(\bigcup_{\substack{L \uplus L' = \mathsf{dom}(I_{\mathsf{K}}) \\ V \uplus V' = \mathsf{CPUid}}} ((\mathsf{IKernel}_\eta \star_{\mathsf{K}} \mathsf{IKernelLock}_{L'} \star_{\mathsf{K}} \mathsf{IPercpu}_{V'}) \cap \mathsf{held}_{\mathsf{K}}(L) \cap \mathsf{disabled}(V)) \right).$$

The following theorem, proved in Appendix A, states the soundness of our logic for preemptive kernels.

Theorem 3
If $I_\mathsf{K}, H, \Delta_\mathsf{K} \mid I_\mathsf{S}, \{\Delta_\mathsf{S}^k\}_{k \in \mathsf{CPUid}} \mid J \vdash (\mathsf{S}, \mathsf{C}, \mathsf{K})$, then for all environments η, the set of configurations IOs_η is preserved by $\rightarrow_\mathsf{OS}$.

Consequences. Theorem 3 allows carrying over statements proved in the high-level proof system about the abstract machine with one virtual CPU per process to the concrete machine. For example, it implies that the properties of the code in Figure 4 described in Section 4 hold when it is managed by the example scheduler. To demonstrate this formally, assume that the initial machine configuration satisfies IOs_η. Then the soundness statement ensures that the machine cannot reach an error label l_e on any CPU, provided that the assertion at this program point in all high-level proofs is false. Indeed, in this case the invariant IOs_η does not contain any states where one of the CPUs is at l_e. Note that the functional correctness of an OS kernel is usually formulated as a refinement between the kernel and its specification. As an OS kernel does not usually make any assumptions about user processes, complications with formulating this refinement that necessitate the unusual soundness statement of our logic do not arise, and thus proving that it can be reduced to proving an invariance property relating the kernel and its specification (Gargano *et al.*, 2005; Klein *et al.*, 2009). Thus, Theorem 3 can also be used to justify such proofs.

Ownership transfer. It is instructive to analyse how the ownership transfer between the scheduler and the kernel is handled by our soundness statement. For example, consider a transfer of a new process descriptor $\mathsf{desc}(d, \gamma)$ from the kernel to the scheduler at a call to `create`. Since the CREATE axiom requires the descriptor in its precondition, before the kernel calls `create`, the state partitioning defined by IOs_η counts the descriptor as part of $\mathsf{IKernel}_\eta$. Since, by the OS proof rule, the implementation of `create` receives the descriptor in its precondition, in the configuration immediately after the call to `create`, IOs_η defines it to be part of ISched_η. Thus, ownership transfer repartitions program state among the parts defined above.

Proof idea and other concurrency logics. The proof of Theorem 3 is relatively straightforward, if technical. When the transition in $\rightarrow_\mathsf{OS}$ considered in the theorem corresponds to a command associated with an ownership transfer in our logic, we prove that the target configuration belongs to IOs_η by redistributing the state among components used to construct this invariant, following the above explanation of ownership transfers. When the transition in $\rightarrow_\mathsf{OS}$ corresponds to a command that is not associated with an ownership transfer between components, such as an assignment, we first ‘unpack’ the IOs_η invariant to get the local state of the process or the scheduler invocation executing the command. We then replace this local state with the new one specified by the proof and show that we can ‘pack’ the invariant back to obtain the target state of the machine.

We based our logic for preemptable code on concurrent separation logic, which would not be able to handle complicated concurrency mechanisms employed in modern OS kernels (Bovet & Cesati, 2005). The approach we take in stating and proving the soundness of our logic has been applied extensively to various concurrent derivatives of separation logic (Gotsman, 2009; Gotsman *et al.*, 2011). This leads us to believe that we can integrate more advanced logics from this class (Feng *et al.*, 2007a; Vafeiadis & Parkinson, 2007; Dinsdale-Young *et al.*, 2010; Dinsdale-Young *et al.*, 2013) without problems.

9 Related work

There have been a number of OS verification projects; see Klein (2009) for a survey. To our knowledge, none of these has included the verification of a scheduler in a preemptive kernel with the realistic features we consider. A representative example is the seL4 project (Klein *et al.*, 2009), which verified a variant of the L4 microkernel as a whole, together with the scheduler. There, proofs about kernel components other than the scheduler had to ensure the preservation of its invariants, e.g., the well-formedness of its runqueue. The proof was still tractable because the kernel was running on a uniprocessor and used an event-based execution model, so that preemption was disabled most of the time. However, such architecture is not used by mainstream operating systems. If fact, as noted by Klein *et al.* (2009), the absence of verification technology dealing with preemption was one of the reasons for the choice of this architecture in seL4.

The closest work to ours is the one by Feng *et al.* (2007b; 2008a; 2008b), who proposed a logic for verifying OS kernels, also based on separation logic. Like us, they structure the logic into separate proof systems for the scheduler and preemptable code. We thus share their vision (Feng *et al.*, 2008b; Shao, 2010) of verifying different components of systems software using specialised logics that allow reasoning on an appropriate level of abstraction. However, there are differences between our work and theirs in the OS features handled and in the general approach to formulating the logic and proving it sound.

As far as OS features are concerned, Feng *et al.* consider a uniprocessor and verify an idealised scheduler without dynamic process creation or ownership transfer between the scheduler and processes. As a consequence, they do not have an analogue of our affine Process predicate needed to handle multiprocessing. On the other hand, Feng *et al.* support modular reasoning about procedures, which we do not. As for the general approach, Feng *et al.* formulate the logics for the scheduler and preemptive code and justify their soundness by embedding them into OCAP (Feng *et al.*, 2007b), a logic supporting first-class code pointers. This support is then used to handle transfers of control between the scheduler and the kernel and to reason modularly about procedures. In contrast, we establish the soundness of our proof systems by a direct correspondence to an operational semantics, without going through an intermediate logic.

The verification of realistic kernels requires supporting modular reasoning about procedures as well as multiprocessing and ownership transfer. Thus, both the logic of Feng *et al.* and ours would need to be extended before they are up to the task. In the case of Feng *et al.*'s logic, this would require extending OCAP to accommodate multiprocessing. This is not completely trivial, since on a multiprocessor, the scheduler and the kernel can run at the same time on different CPUs, and OCAP currently requires the control to be within a single component at any point of time. One would also need to extend OCAP to treat assertions about code pointers affinely, as our Process predicates. Conversely, to provide support for modular reasoning about procedures in our logic, we would have to borrow the corresponding proof rules from one of the available logics for storable code, possibly a relative of OCAP (Feng *et al.*, 2006; Feng *et al.*, 2007b; Ni & Shao, 2006; Schwinghammer *et al.*, 2009; Charlton, 2011). The soundness could still be proved by a correspondence to an operational semantics, but our current proof would have to be adjusted. Thus, both our approach and the one of Feng *et al.* can potentially be extended to cover a wider range of OS features, possibly by exploiting techniques from the other. It remains to be seen in which settings a given approach works better.

Even though in this paper we focus on a low-level programming language, the reasoning principles we propose are high level and analogous to those developed for control flow in functional programs. For example, the Process predicate in our low-level proof system can be viewed as an assertion about an affine continuation, providing a clean model for capturing and resuming process state. Our use of separating conjunction over such predicates is analogous to the use of linear typing in the study of continuations in functional programs (Berdine *et al.*, 2002; Hasegawa, 2002; Thielecke, 2003; Hasegawa, 2004; Laird, 2005). One can thus think of our work as layering clean functional reasoning on top of low-level OS code.

Maeda and Yonezawa have proved a simple context-switch routine using an extension of alias types (Maeda & Yonezawa, 2009). Their proof expresses the disjointness of data structures belonging to the scheduler and the rest of the kernel using the tensor operator of alias types, which corresponds to our separating conjunction. However, their type system does not hide the internal data structures of the scheduler while proving the rest of the kernel, and is thus non-modular.

Yang and Hawblitzel have recently developed a kernel where most of the codebase is typechecked and therefore cannot directly access data structures belonging to the core part of the kernel, including the scheduler (Yang & Hawblitzel, 2010). However, the guarantees established by the type system do not take into account the contents of data structures, so the kernel can still subvert the scheduler by leaving them in an inconsistent state. The OS resorts to runtime checks in such cases, introducing a performance penalty. The relationship to this work is that of a trade-off: type safety guarantees are easier to get, but are not as strong as those provided by a program logic.

Refinement is a well-known approach in the verification of both operating systems and general concurrent programs (Back, 1981; Gargano *et al.*, 2005; Jones, 2007; Klein *et al.*, 2009; Turon & Wand, 2011). Our logic can be viewed as implementing a form of refinement where the semantics of the abstract system is defined axiomatically by the high-level proof system and refinement relations, defined by the low-level proof system, focus only on the relevant state of the systems related. We thus advance the refinement theory to systems with complex ownership transfers.

10 Conclusion

In this paper, we have neither verified a complete operating system nor built an automatic tool. Instead, we have proposed a proof rule that allows decomposing the verification of a preemptive OS kernel into two simpler tasks—verifying the scheduler and preemptable code separately. Furthermore, we have, for the first time, achieved this for the patterns of interaction between the scheduler and the kernel present in mainstream operating systems.

Such a result is relevant no matter what type of formal analysis of OS code one is performing: manual or automatic verification, or even bug-finding. Moreover, as we argued in Section 2.2, the straightforward approach of verifying the scheduler together with the rest of the kernel makes reasoning intractable; thus, a result such as ours is in fact indispensable for verifying realistic OS kernels.

Despite our development being carried out for a particular baseline concurrency logic and a class of scheduling interfaces, the key technical methods we proposed in this paper are transferable and can be reused in OS verification projects. These include:

- exploiting a logic validating the frame property to hide the state of the scheduler while verifying the kernel and vice versa;
- using the $\mathsf{Process}$ assertions to reason about the correct treatment of process states by the scheduler and the affine semantics of $*$ on them to reason about scheduling on multiprocessors;
- dealing with features breaking through the scheduler abstraction, such as interrupt disabling, by axiomatising their intended uses when reasoning about the kernel; and
- formulating soundness by constructing a global property from local assertions on different levels of abstraction using a combination of the separating conjunction and relational composition.

Acknowledgments

We thank Anindya Banerjee, Xinyu Feng, Boris Köpf, Mark Marron, Peter O'Hearn, Matthew Parkinson, Noam Rinetzky, Zhong Shao, Viktor Vafeiadis and Jules Villard for comments and discussions that helped improve the paper. Gotsman was supported by the EU FET ADVENT project. Yang was supported by EPSRC.

Supplementary materials

For supplementary material for this article, please visit dx.doi.org/10.1017/S0956796813000075.

A Appendix. Proof of soundness

Auxiliary definitions. In the following, we write $\{E(\_)\}$, where E is an expression with occurrences of $\_$, to mean the set of values arising from evaluating E with $\_$ substituted for any values from the corresponding domains.

For a set Σ let $\mathscr{P}(\Sigma)^\top$ be the domain of subsets of Σ with a special element $\top$. The order $\sqsubseteq$ in the domain $\mathscr{P}(\Sigma)^\top$ is subset inclusion with $\top$ being the greatest element. We define two partial operations interpreting the $*$ connectives in the high- and low-level proof systems, respectively:

$$\begin{aligned} *_{\mathsf{K}} &: \mathscr{P}(\mathsf{State})^\top \times \mathscr{P}(\mathsf{State})^\top \to \mathscr{P}(\mathsf{State})^\top; \\ *_{\mathsf{S}} &: \mathscr{P}(\mathsf{SchedState})^\top \times \mathscr{P}(\mathsf{SchedState})^\top \to \mathscr{P}(\mathsf{SchedState})^\top. \end{aligned}$$

For $p, q \in \mathscr{P}(\mathsf{State})$ we let

$$p *_{\mathsf{K}} q = \{(r, h_1 \uplus h_2, L_1 \uplus L_2) \mid (r, h_1, L_1) \in p \wedge (r, h_2, L_2) \in q\}; \quad \top * p = p * \top = \top.$$

For $p, q \in \mathscr{P}(\mathsf{SchedState})$ we let

$$p *_{\mathsf{S}} q = \{((r, h_1 \uplus h_2, L_1 \uplus L_2), M_1 \uplus M_2) \mid ((r, h_1, L_1), M_1) \in p \wedge ((r, h_2, L_2), M_2) \in q\};$$
$$\top * p = p * \top = \top.$$

We use the following definitions: for $p \subseteq \mathsf{State}_{\mathsf{I}}$, $q \subseteq \mathsf{SchedState}$, $l \in \mathsf{Label}$, $k \in \mathsf{CPUid}$, $\ell \in \mathsf{Lock}$ let

$$\begin{aligned}
\mathsf{at}_{\mathsf{K}}(l) &= \{(r, h, L, v) \in \mathsf{State}_{\mathsf{I}} \mid r(\mathtt{ip}) = l\};\\
\mathsf{lk}_{\mathsf{K}}(\ell) &= \{(r, [\,], \{\ell\}, \bot) \in \mathsf{State}_{\mathsf{I}}\};\\
\mathsf{lk}_{\mathsf{S}}(\ell) &= \{((r, [\,], \{\ell\}), \emptyset) \in \mathsf{SchedState}\};\\
\mathsf{int}(k) &= \{(r, [\,], \emptyset, k) \in \mathsf{State}_{\mathsf{I}}\};\\
\mathsf{to}_{\mathsf{K}}(l, p) &= \{(r[\mathtt{ip} : l], h, L, v) \in \mathsf{State}_{\mathsf{I}} \mid (r, h, L, v) \in p\};\\
\mathsf{to}_{\mathsf{S}}(l, q) &= \{((r[\mathtt{ip} : l], h, L), M) \in \mathsf{SchedState} \mid ((r, h, L), M) \in q\}.
\end{aligned}$$

Finally, consider a process descriptor predicate $\mathsf{desc}(d, \gamma)$ with free logical variables d and γ and an environment η. We define $\boldsymbol{desc}_\eta : \mathsf{Val} \times \mathsf{Context} \to \mathsf{State}$ as follows: for $u \in \mathsf{Val}$ and $r \in \mathsf{Context}$ we let $\boldsymbol{desc}_\eta(u, r) = [\![\mathsf{desc}(d, \gamma)]\!]_{\eta[d:u, \gamma:r]}$.

Transformers for primitive commands. It is convenient for us to reformulate the semantics of primitive commands c in Figure 11 and Section 7 in terms of transformers

$$f_c^k : \mathsf{Label} \times \mathsf{Label} \times \mathsf{State} \to \mathscr{P}(\mathsf{State})^\top, \quad k \in \mathsf{CPUid}$$

for $c \in \mathsf{PComm}$, defined as follows: $f_c^k(l, l', (r, h, L)) = \top$, if $k, (r, h, L), l, l' \rightsquigarrow_c \top$; otherwise,

$$f_c^k(l, l', (r, h, L)) = \bigcup \big\{ (r'[\mathtt{ip} : l''], h', L') \mid (k, (r, h, L), l, l') \rightsquigarrow_c ((r', h', L'), l'') \big\}.$$

We extend the transformers to operate on states in $\mathsf{SchedState}$ and $\mathsf{State}_{\mathsf{I}}$:

$$\begin{aligned}
&f_c^k : \mathsf{Label} \times \mathsf{Label} \times \mathsf{SchedState} \to \mathscr{P}(\mathsf{SchedState})^\top, && k \in \mathsf{CPUid};\\
&f_c^k : \mathsf{Label} \times \mathsf{Label} \times \mathsf{State}_{\mathsf{I}} \to \mathscr{P}(\mathsf{State}_{\mathsf{I}})^\top, && k \in \mathsf{CPUid}.
\end{aligned}$$

We let

$$f_c^k(l, l', ((r, h, L), M)) = \{((r', h', L'), M) \mid (r', h', L') \in f_c^k(l, l', (r, h, L))\}, \tag{A 1}$$

if $f_c^k(l, l', (r, h, L)) \neq \top$, and $f_c^k(l, l', ((r, h, L), M)) = \top$, otherwise. We let

$$f_c^k(l, l', (r, h, L, v)) = \{(r', h', L', v) \mid (r', h', L') \in f_c^k(l, l', (r, h, L))\}, \tag{A 2}$$

if $f_c^k(l, l', (r, h, L)) \neq \top$, and $f_c^k(l, l', (r, h, L, v)) = \top$, otherwise. We then lift these transformers to the corresponding domains pointwise. For example, for $p \in \mathscr{P}(\mathsf{State}_{\mathsf{I}})^\top$ we let

$$f_c^k(l, l', p) = \begin{cases} \bigsqcup\{f_c^k(l, l', (r, h, L, v)) \mid (r, h, L, v) \in p\}, & \text{if } p \neq \top;\\ \top, & \text{if } p = \top. \end{cases}$$

The transformers thus defined satisfy the property of *locality* (Calcagno *et al.*, 2007) with respect to the operations $*_{\mathsf{S}}$ and $*_{\mathsf{K}}$:

$$\forall p,q \in \mathscr{P}(\mathsf{SchedState}).\, f_c^k(p *_{\mathsf{S}} q) \sqsubseteq f_c^k(p) *_{\mathsf{S}} q; \tag{A 3}$$

$$\forall p,q \in \mathscr{P}(\mathsf{State_l}).\, f_c^k(p *_{\mathsf{K}} q) \sqsubseteq f_c^k(p) *_{\mathsf{K}} q. \tag{A 4}$$

Semantic proofs. To prove Theorem 3, we translate a syntactic proof in our logic into a semantic form, which annotates every program point in the OS code with a description of the state local to the process or the scheduler invocation executing the code. Namely, given an environment η, a *semantic proof* (Gotsman *et al.*, 2011) of the OS program is defined as a tuple $(G_{\mathsf{S}}, G_{\mathsf{K}}, \mathscr{I}_{\mathsf{S}}, \mathscr{I}_{\mathsf{K}}, \mathscr{H}, \mathscr{J})$, where

- $G_{\mathsf{S}}^k : \mathsf{Label} \to \mathscr{P}(\mathsf{SchedState})$, $k \in \mathsf{CPUid}$;
- $G_{\mathsf{K}} : \mathsf{Label} \to \mathscr{P}(\mathsf{State_l})$;
- $\mathscr{I}_{\mathsf{S}} \in \mathsf{Lock} \rightharpoonup \mathscr{P}(\mathsf{SchedState})$;
- $\mathscr{I}_{\mathsf{K}} \in \mathsf{Lock} \rightharpoonup \mathscr{P}(\mathsf{State_l})$;
- $\mathscr{H}_k \in \mathscr{P}(\mathsf{State_l})$, $k \in \mathsf{CPUid}$,
- $\mathscr{J}_k \in \mathscr{P}(\mathsf{SchedState})$, $k \in \mathsf{CPUid}$,

such that $\mathscr{I}_{\mathsf{K}}$, $\mathscr{I}_{\mathsf{S}}$, $\mathscr{H}$, $\mathscr{J}$ satisfy the analogues of the well-formedness restrictions previously imposed on I_{K}, I_{S} , H, J, and the conditions in Figure A 1 hold. The latter conditions are semantic counterparts of the axioms in the high- and low-level proof systems. The following lemma shows that a syntactic proof can be converted into a semantic one.

Lemma 1
Given a proof $I_{\mathsf{K}}, H, \Delta_{\mathsf{K}} \mid I_{\mathsf{S}}, \{\Delta_{\mathsf{S}}^k\}_{k \in \mathsf{CPUid}} \mid J \vdash (\mathsf{S}, \mathsf{C}, \mathsf{K})$ and an environment η, there exists a semantic proof $(G_{\mathsf{S}}, G_{\mathsf{K}}, [\![I_{\mathsf{S}}]\!]_\eta, [\![I_{\mathsf{K}}]\!]_\eta, [\![H]\!]_\eta, [\![J]\!]_\eta)$ such that for all $l \in \mathsf{Label}$ and $k \in \mathsf{CPUid}$ we have $G_{\mathsf{K}}(l) = [\![\Delta_{\mathsf{K}}(l)]\!]_\eta \cap \mathsf{at}_{\mathsf{K}}(l)$ and $G_{\mathsf{S}}^k(l) = [\![\Delta_{\mathsf{S}}^k(l)]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l)$.

We omit the straightforward proof of the lemma and proceed to prove the main soundness theorem.

Proof of Theorem 3. Let us fix an environment η. We first apply Lemma 1 to construct a semantic proof $(G_{\mathsf{S}}, G_{\mathsf{K}}, \mathscr{I}_{\mathsf{S}}, \mathscr{I}_{\mathsf{K}}, \mathscr{H}, \mathscr{J})$ from the given syntactic one. Assume now that $\sigma \in \mathsf{IOs}_\eta$ and $\sigma \to_{\mathsf{OS}} \sigma'$ for some $\sigma' \in \mathsf{Config} \cup \{\top\}$. We need to show that $\sigma' \in \mathsf{IOs}_\eta$. Let the command in $\sigma \to_{\mathsf{OS}} \sigma'$ be executed by CPU k. We can thus assume

$$\sigma = (R[k:r], h, L),\;\; R(k) \text{ is undefined},\;\; r(\mathtt{ip}) = l,\;\; c = \mathsf{comm}(\mathsf{OS}, l),\;\; l' \in \mathsf{next}(\mathsf{OS}, l).$$

By the definition of IOs_η, there exist

$$h_1, h_2 \in \mathsf{Heap},\;\; L_1 \subseteq \mathsf{dom}(I_{\mathsf{S}}),\;\; L_2 \subseteq \mathsf{dom}(I_{\mathsf{K}}),\;\; M \in \mathscr{M}(\mathsf{Context}),\;\; V \in \mathscr{P}(\mathsf{CPUid})$$

such that

$$((R[k:r], h_1, L_1), M, V) \in \mathsf{ISched}_\eta \star_{\mathsf{S}} \mathsf{ISchedLock}_{\mathsf{dom}(I_{\mathsf{S}})-L_1}; \tag{A 24}$$

$$(M, h_2, L_2, V) \in \mathsf{IKernel}_\eta \star_{\mathsf{K}} \mathsf{IKernelLock}_{\mathsf{dom}(I_{\mathsf{K}})-L_2} \star_{\mathsf{K}} \mathsf{IPercpu}_{\mathsf{CPUid}-V} \tag{A 25}$$

$$h = h_1 \uplus h_2, \quad L = L_1 \uplus L_2. \tag{A 26}$$

We now consider several cases of how σ' may be obtained.

$$\forall l \in \mathsf{Label}.\, l \notin \mathsf{dom}(\mathsf{K}) \implies G_{\mathsf{K}}(l) = \emptyset; \tag{A 5}$$

$$\forall l \in \mathsf{Label}, k \in \mathsf{CPUid}.\, l \notin \mathsf{dom}(\mathsf{S}) \uplus \mathsf{dom}(\mathsf{C}) \uplus \{l_s, l_c\} \implies G^k_{\mathsf{S}}(l) = \emptyset; \tag{A 6}$$

$$\forall k \in \mathsf{CPUid}.\, G^k_{\mathsf{S}}(\mathtt{schedule}) = [\![\mathsf{SchedState}_k]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(\mathtt{schedule}); \tag{A 7}$$

$$\forall k \in \mathsf{CPUid}.\, G^k_{\mathsf{S}}(l_s) = [\![\mathsf{SchedState}_k]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l_s); \tag{A 8}$$

$$\begin{aligned}&\forall k \in \mathsf{CPUid}.\, G^k_{\mathsf{S}}(\mathtt{create}) = \\ &\qquad [\![\exists \gamma.\, \gamma(\mathbf{if}) = 1 \wedge \mathsf{SchedState}_k * \mathsf{desc}(\mathtt{gr}_1, \gamma) * \mathsf{Process}(\gamma)]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(\mathtt{create});\end{aligned} \tag{A 9}$$

$$\forall k \in \mathsf{CPUid}.\, G^k_{\mathsf{S}}(l_c) = [\![\mathsf{SchedState}_k]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l_c); \tag{A 10}$$

$$\begin{aligned}&\forall l \in \mathsf{Label}, (r,h,L,v) \in G_{\mathsf{K}}(l).\, 0 \leqslant r(\mathtt{sp}) - r(\mathtt{ss}) \leqslant \mathtt{StackBound} \wedge \\ &\qquad \mathsf{dom}(h) \supseteq \{r(\mathtt{sp}), \dots, r(\mathtt{ss}) + \mathtt{StackSize} - 1\} \wedge \\ &\forall h'.\, (\forall u \notin \{r(\mathtt{sp}), \dots, r(\mathtt{ss}) + \mathtt{StackSize} - 1\}.\, h(u) = h'(u)) \implies (r, h', L, v) \in G_{\mathsf{K}}(l),\end{aligned} \tag{A 11}$$

and for all $l \in \mathsf{labels}(\mathsf{OS})$, $l' \in \mathsf{next}(\mathsf{OS}, l)$, $c = \mathsf{comm}(\mathsf{OS}, l)$ and $k \in \mathsf{CPUid}$, we have:

- if c is not $\mathtt{lock}$ or $\mathtt{unlock}$, and $l \in \mathsf{labels}(\mathsf{S}) \uplus \mathsf{labels}(\mathsf{C})$, then

$$f^k_c(l, l', G^k_{\mathsf{S}}(l)) \neq \top \wedge \forall ((r,h,L),M) \in f^k_c(l, l', G^k_{\mathsf{S}}(l)).\, ((r,h,L),M) \in G^k_{\mathsf{S}}(r(\mathtt{ip})); \tag{A 12}$$

- if c is not $\mathtt{lock}$, $\mathtt{unlock}$, $\mathtt{icall}$, $\mathtt{cli}$ or $\mathtt{sti}$ and $l \in \mathsf{labels}(\mathsf{K})$, then

$$f^k_c(l, l', G_{\mathsf{K}}(l)) \neq \top \wedge \forall (r,h,L,v) \in f^k_c(l, l', G_{\mathsf{K}}(l)).\, (r,h,L,v) \in G_{\mathsf{K}}(r(\mathtt{ip})); \tag{A 13}$$

- if c is $\mathtt{lock}(\ell)$ and $l \in \mathsf{labels}(\mathsf{S}) \uplus \mathsf{labels}(\mathsf{C})$, then

$$\mathsf{to}_{\mathsf{S}}(l', (G^k_{\mathsf{S}}(l) *_{\mathsf{S}} \mathscr{I}_{\mathsf{S}}(\ell) *_{\mathsf{S}} \mathsf{lk}_{\mathsf{S}}(\ell))) \subseteq G^k_{\mathsf{S}}(l'); \tag{A 14}$$

- if c is $\mathtt{lock}(\ell)$ and $l \in \mathsf{labels}(\mathsf{K})$, then

$$\mathsf{to}_{\mathsf{K}}(l', (G_{\mathsf{K}}(l) *_{\mathsf{K}} \mathscr{I}_{\mathsf{K}}(\ell) *_{\mathsf{K}} \mathsf{lk}_{\mathsf{K}}(\ell))) \subseteq G_{\mathsf{K}}(l'); \tag{A 15}$$

- if c is $\mathtt{unlock}(\ell)$ and $l \in \mathsf{labels}(\mathsf{S}) \uplus \mathsf{labels}(\mathsf{C})$, then

$$\mathsf{to}_{\mathsf{S}}(l', G^k_{\mathsf{S}}(l)) \subseteq G^k_{\mathsf{S}}(l') *_{\mathsf{S}} \mathscr{I}_{\mathsf{S}}(\ell) *_{\mathsf{S}} \mathsf{lk}_{\mathsf{S}}(\ell); \tag{A 16}$$

- if c is $\mathtt{unlock}(\ell)$ and $l \in \mathsf{labels}(\mathsf{K})$, then

$$\mathsf{to}_{\mathsf{K}}(l', G_{\mathsf{K}}(l)) \subseteq G_{\mathsf{K}}(l') *_{\mathsf{K}} \mathscr{I}_{\mathsf{K}}(\ell) *_{\mathsf{K}} \mathsf{lk}_{\mathsf{K}}(\ell); \tag{A 17}$$

- if c is $\mathtt{cli}$ and $l \in \mathsf{labels}(\mathsf{K})$, then

$$\mathsf{to}_{\mathsf{K}}(l', (G_{\mathsf{K}}(l) *_{\mathsf{K}} \mathscr{H}_k *_{\mathsf{K}} \mathsf{int}(k))) \subseteq G_{\mathsf{K}}(l'); \tag{A 18}$$

- if c is $\mathtt{sti}$ and $l \in \mathsf{labels}(\mathsf{K})$, then

$$\mathsf{to}_{\mathsf{K}}(l', G_{\mathsf{K}}(l)) \subseteq G_{\mathsf{K}}(l') *_{\mathsf{K}} \mathscr{H}_k *_{\mathsf{K}} \mathsf{int}(k); \tag{A 19}$$

- if c is $\mathtt{icall}(\mathtt{schedule})$, then

$$\mathsf{to}_{\mathsf{K}}(l', G_{\mathsf{K}}(l)) \subseteq G_{\mathsf{K}}(l'); \tag{A 20}$$

- if c is $\mathtt{icall}(\mathtt{create})$, then for some $P, Q \in \mathsf{Assert}_{\mathsf{I}}$ such that $\mathsf{free}(P) \cap \mathsf{Reg} = \emptyset$, P does not own CPU predicates and has an empty lockset, we have

$$G_{\mathsf{K}}(l) \subseteq [\![\exists \gamma.\, \gamma(\mathbf{if}) = 1 \wedge \mathsf{desc}(\mathtt{gr}_1, \gamma) * P * Q]\!]_\eta \cap \mathsf{at}_{\mathsf{K}}(l); \tag{A 21}$$

$$G_{\mathsf{K}}(l') \supseteq [\![\exists \gamma.\, Q]\!]_\eta \cap \mathsf{at}_{\mathsf{K}}(l'); \tag{A 22}$$

$$\forall r \in \mathsf{Context}.\, \{(r, [r(\mathtt{sp})..(r(\mathtt{ss}) + \mathtt{StackSize} - 1) : \_], \emptyset, \bot)\} *_{\mathsf{K}} [\![P]\!]_{\eta[\gamma:r]} \subseteq G_{\mathsf{K}}(r(\mathtt{ip})). \tag{A 23}$$

Fig. A 1. Conditions for a semantic proof $(G_{\mathsf{S}}, G_{\mathsf{K}}, \mathscr{I}_{\mathsf{S}}, \mathscr{I}_{\mathsf{K}}, \mathscr{H}, \mathscr{J})$.

Case 1. σ' is obtained by applying the fourth rule in Figure 12. This case is impossible, since by (A 24) and the definition of ISched_η we have $l = r(\mathtt{ip}) \in \mathsf{labels}(\mathsf{OS})$.

Case 2. σ' is obtained by applying the first or the third rule in Figure 12, with the command executed by the scheduler and different from `lock`, `unlock` or `iret`. In this case, $l \in \mathsf{labels}(\mathsf{S}) \uplus \mathsf{labels}(\mathsf{C})$ and

$$\begin{aligned}&(f_c^k(l,l',(r,h,L)) = \top \implies \sigma' = \top) \wedge \\ &(f_c^k(l,l',(r,h,L)) \neq \top \implies \sigma' \in (\{(R,[\,],\emptyset)\} \star_{\mathsf{B}} \lfloor f_c^k(l,l',(r,h,L)) \rfloor_k^{\mathsf{BA}})).\end{aligned} \tag{A 27}$$

From (A 24), for some $h_{\mathsf{S}} \in \mathsf{Heap}$, $L_{\mathsf{S}} \in \mathsf{Lockset}$, $M_{\mathsf{S}} \in \mathscr{M}(\mathsf{Context})$, $V_{\mathsf{S}}, V_{\mathsf{K}} \in \mathscr{P}(\mathsf{CPUid})$ we have $V_{\mathsf{S}} \uplus V_{\mathsf{K}} \uplus \{k\} = \mathsf{CPUid}$, $V \subseteq V_{\mathsf{K}}$, $r(\mathtt{if}) = 0$,

$$((r,h_1,L_1),M) \in G_{\mathsf{S}}^k(l) *_{\mathsf{S}} \{((\_,h_{\mathsf{S}},L_{\mathsf{S}}),M_{\mathsf{S}})\} \tag{A 28}$$

and for $W = V$ and $W_1 = L_1$ we have

$$\begin{aligned}((R,h_{\mathsf{S}},L_{\mathsf{S}}),M_{\mathsf{S}},W) \in\ & \left(\mathop{\circledast_{\mathsf{S}}}_{j\in V_{\mathsf{S}}} \bigcup_{l\in(\mathsf{labels}(\mathsf{S}\uplus\mathsf{C})\uplus\{l_s,l_c\})} \lfloor [\![\Delta_{\mathsf{S}}^j(l)]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l) \cap \mathsf{if}_{\mathsf{S}}(0) \rfloor_{j,\emptyset}^{\mathsf{SA}}\right) \star_{\mathsf{S}} \\ & \left(\mathop{\circledast_{\mathsf{S}}}_{j\in V_{\mathsf{I}}} \bigcup_{l\in\mathsf{labels}(\mathsf{K})} \lfloor [\![\mathsf{SchedSleep}_j(l)]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l) \cap \mathsf{if}_{\mathsf{S}}(0) \rfloor_{j,\{j\}}^{\mathsf{SA}}\right) \star_{\mathsf{S}} \\ & \left(\mathop{\circledast_{\mathsf{S}}}_{j\in V_{\mathsf{K}}-V_{\mathsf{I}}} \bigcup_{l\in\mathsf{labels}(\mathsf{K})} \lfloor [\![\mathsf{SchedSleep}_j(l)]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l) \cap \mathsf{if}_{\mathsf{S}}(1) \rfloor_{j,\emptyset}^{\mathsf{SA}}\right) \star_{\mathsf{S}} \\ & \mathsf{ISchedLock}_{\mathsf{dom}(I_{\mathsf{S}})-W_1}.\end{aligned} \tag{A 29}$$

We have:

$$\begin{aligned} & f_c^k(l,l',((r,h,L),M)) \\ = \ & f_c^k(l,l',\{((r,h_1,L_1),M)\} *_{\mathsf{S}} \{((\_,h_2,L_2),\emptyset)\}) && \textit{by (A 26)} \\ \sqsubseteq \ & f_c^k(l,l',G_{\mathsf{S}}^k(l) *_{\mathsf{S}} \{((\_,h_2 \uplus h_{\mathsf{S}},L_2 \uplus L_{\mathsf{S}}),M_{\mathsf{S}})\}) && \textit{by (A 28)} \\ \sqsubseteq \ & f_c^k(l,l',G_{\mathsf{S}}^k(l)) *_{\mathsf{S}} \{((\_,h_2 \uplus h_{\mathsf{S}},L_2 \uplus L_{\mathsf{S}}),M_{\mathsf{S}})\} && \textit{by (A 3)} \end{aligned}$$

From this and (A 12), $f_c^k(l,l',G_{\mathsf{S}}^k(l)) \neq \top$, hence, by (A 1) and (A 27), $\sigma' \neq \top$. Let $\sigma' = (R[k:r'],h',L)$. Then, $r'(\mathtt{if}) = r(\mathtt{if}) = 0$ and from the above, (A 27) and (A 12), we have

$$\begin{aligned}((R[k:r'],h',L),M,V) \in \{((R,[\,],\emptyset),\emptyset,\emptyset)\} \star_{\mathsf{S}} & \\ \lfloor (G_{\mathsf{S}}^k(r'(\mathtt{ip})) \cap \mathsf{if}_{\mathsf{S}}(0)) *_{\mathsf{S}} \{((\_,h_2 \uplus h_{\mathsf{S}},L_2 \uplus L_{\mathsf{S}}),M_{\mathsf{S}})\} \rfloor_{k,V}^{\mathsf{SA}}&.\end{aligned}$$

From this and (A 6) we get $r'(\mathtt{ip}) \in \mathsf{dom}(\mathsf{S}) \uplus \mathsf{dom}(\mathsf{C}) \uplus \{l_s,l_c\}$. Hence, by (A 29) we have

$$((R[k:r'],h',L),M,V) \in \mathsf{ISched}_\eta \star_{\mathsf{S}} \mathsf{ISchedLock}_{\mathsf{dom}(I_{\mathsf{S}})-L_1} \star_{\mathsf{S}} \{(([\,],h_2,L_2),\emptyset,\emptyset)\}.$$

Then, from (A 25) and the definition of $\star_{\mathsf{SK}}$ we get $\sigma' \in \mathsf{IOs}_\eta$.

Case 3. σ' is obtained by applying the first rule in Figure 12, with the scheduler executing `lock`. In this case

$$l \in \mathsf{labels}(\mathsf{S}) \uplus \mathsf{labels}(\mathsf{C}), \quad c = \mathtt{lock}(\ell), \quad \ell \notin L, \quad \sigma' = (R[k:r[\mathtt{ip}:l']],h,L \cup \{\ell\}).$$

From (A 24), for some $h_{\mathsf{S}}, L_{\mathsf{S}}, M_{\mathsf{S}}, V_{\mathsf{S}}, V_{\mathsf{K}}$ we have $V_{\mathsf{S}} \uplus V_{\mathsf{K}} \uplus \{k\} = \mathsf{CPUid}$, $V \subseteq V_{\mathsf{K}}$, $r(\mathtt{if}) = 0$,

$$((r, h_1, L_1), M) \in G^k_{\mathsf{S}}(l) *_{\mathsf{S}} \mathscr{I}_{\mathsf{S}}(\ell) *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), M_{\mathsf{S}})\}$$

and (A 29) holds for $W = V$ and $W_1 = L_1 \cup \{\ell\}$. Then,

$$((r[\mathtt{ip} : l'], h_1, L_1 \cup \{\ell\}), M) \in \mathsf{to}_{\mathsf{S}}(l', (G^k_{\mathsf{S}}(l) *_{\mathsf{S}} \mathscr{I}_{\mathsf{S}}(\ell) *_{\mathsf{S}} \mathsf{lk}_{\mathsf{S}}(\ell))) *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), M_{\mathsf{S}})\}.$$

By (A 14), this implies

$$((r[\mathtt{ip} : l'], h_1, L_1 \cup \{\ell\}), M) \in G^k_{\mathsf{S}}(l') *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), M_{\mathsf{S}})\}.$$

From this and (A 6) we get $l' \in \mathsf{dom}(\mathsf{S}) \uplus \mathsf{dom}(\mathsf{C}) \uplus \{l_s, l_c\}$. Hence, by (A 29) for $W = V$ and $W_1 = L_1 \cup \{\ell\}$,

$$((R[k : r[\mathtt{ip} : l']], h_1, L_1 \cup \{\ell\}), M, V) \in \mathsf{ISched}_\eta \star_{\mathsf{S}} \mathsf{ISchedLock}_{\mathsf{dom}(I_{\mathsf{S}}) - (L_1 \cup \{\ell\})}.$$

Then, from (A 25) and the definition of $\star_{\mathsf{SK}}$ we get $\sigma' \in \mathsf{IOs}_\eta$.

Case 4. σ' is obtained by applying the first or the third rule in Figure 12, with the scheduler executing `unlock`. In this case, $l \in \mathsf{labels}(\mathsf{S}) \uplus \mathsf{labels}(\mathsf{C})$, $c = \mathtt{unlock}(\ell)$ and (A 27) holds.

From (A 24), there exist $h_{\mathsf{S}}, L_{\mathsf{S}}, M_{\mathsf{S}}, V_{\mathsf{S}}, V_{\mathsf{K}}$ such that $V_{\mathsf{S}} \uplus V_{\mathsf{K}} \uplus \{k\} = \mathsf{CPUid}$, $V \subseteq V_{\mathsf{K}}$, $r(\mathtt{if}) = 0$ and (A 28) and (A 29) hold for $W = V$ and $W_1 = L_1$. From (A 28) we then get

$$((r[\mathtt{ip} : l'], h_1, L_1), M) \in \mathsf{to}_{\mathsf{S}}(l', G^k_{\mathsf{S}}(l)) *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), M_{\mathsf{S}})\}.$$

Then by (A 16)

$$((r[\mathtt{ip} : l'], h_1, L_1), M) \in G^k_{\mathsf{S}}(l') *_{\mathsf{S}} \mathscr{I}_{\mathsf{S}}(\ell) *_{\mathsf{S}} \mathsf{lk}_{\mathsf{S}}(\ell) *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), M_{\mathsf{S}})\}.$$

Hence, $\ell \in L_1$, which by (A 27) implies $\sigma' \neq \top$. Then, $\sigma' = (R[k : r[\mathtt{ip} : l']], h, L - \{\ell\})$. The above also implies

$$((r[\mathtt{ip} : l'], h_1, L_1 - \{\ell\}), M) \in G^k_{\mathsf{S}}(l') *_{\mathsf{S}} \mathscr{I}_{\mathsf{S}}(\ell) *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), M_{\mathsf{S}})\}.$$

From this and (A 6) we get $l' \in \mathsf{dom}(\mathsf{S}) \uplus \mathsf{dom}(\mathsf{C}) \uplus \{l_s, l_c\}$. Hence, by (A 29)

$$(R[k : r[\mathtt{ip} : l']], h_1, L_1 - \{\ell\}) \in \mathsf{ISched}_\eta \star_{\mathsf{S}} \mathsf{ISchedLock}_{\mathsf{dom}(I_{\mathsf{S}}) - (L_1 - \{\ell\})}.$$

Then, from (A 25) and the definition of $\star_{\mathsf{SK}}$ we get $\sigma' \in \mathsf{IOs}_\eta$.

Case 5. σ' is obtained by applying the first or the third rule in Figure 12, with the command executed by the kernel and different from `lock`, `unlock`, `icall`, `sti` or `cli`. In this case, $l \in \mathsf{labels}(\mathsf{K})$ and (A 27) holds.

From (A 24), there exist $h_{\mathsf{S}}, L_{\mathsf{S}}, M_{\mathsf{S}}, V_{\mathsf{S}}, V_{\mathsf{K}}$ such that $V_{\mathsf{S}} \uplus V_{\mathsf{K}} \uplus \{k\} = \mathsf{CPUid}$, $V \subseteq V_{\mathsf{K}} \cup \{k\}$, $r(\mathtt{if}) = 0 \iff k \in V$, (A 29) holds for $W = V - \{k\}$ and $W_1 = L_1$ and

$$((r, h_1, L_1), M) \in ([\![\mathsf{SchedSleep}_k(l)]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l)) *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), M_{\mathsf{S}})\}. \tag{A 30}$$

The latter implies

$$((r, h_1, L_1), M) \in \mathscr{J}_k *_{\mathsf{S}} \{((\_, [r(\mathtt{sp})..(r(\mathtt{ss}) + \mathtt{StackSize} - 1 : \_)] \uplus h_{\mathsf{S}}, L_{\mathsf{S}}), \{r\} \uplus M_{\mathsf{S}})\}.$$

Then, for some $h'_1 \in \mathsf{Heap}$ and

$$h_0 \in \{[r(\mathtt{sp})..(r(\mathtt{ss}) + \mathtt{StackSize} - 1) : \_]\}$$

we have $h_1 = h'_1 \uplus h_0$ and

$$((r, h'_1, L_1), M) \in \mathscr{I}_k *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), \{r\} \uplus M_{\mathsf{S}})\}. \tag{A 31}$$

Let $h'_2 = h_2 \uplus h_0$, then

$$h = h'_1 \uplus h'_2, \quad L = L_1 \uplus L_2. \tag{A 32}$$

Also, let $v = k$, if $k \in V$, and $v = \bot$, otherwise.

Note that from (A 30) it follows that $r \in M$. Then by (A 25) and (A 11) for some $h_{\mathsf{K}} \in \mathsf{Heap}$ and $L_{\mathsf{K}} \in \mathsf{Lockset}$ we have

$$(r, h'_2, L_2, v) \in G_{\mathsf{K}}(l) *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \emptyset)\} \tag{A 33}$$

and[4]

$$(M - \{r\}, h_{\mathsf{K}}, L_{\mathsf{K}}, V - \{k\}) \in \underset{r'' \in M - \{r\}}{\circledast_{\mathsf{K}}} \lfloor \llbracket \Delta_{\mathsf{K}}(r''(\mathtt{ip})) \rrbracket_\eta \rfloor^{\mathsf{KA}}_{r''} \star_{\mathsf{K}}$$
$$\mathsf{IKernelLock}_{\mathsf{dom}(I_{\mathsf{K}}) - L_2} \star_{\mathsf{K}} \mathsf{IPercpu}_{\mathsf{CPUid} - V}. \tag{A 34}$$

We have:

$$\begin{array}{lll}
& f^k_c(l, l', (r, h, L, v)) & \\
= & f^k_c(l, l', \{(r, h'_2, L_2, v)\} *_{\mathsf{K}} \{(\_, h'_1, L_1, \bot)\}) & \textit{by (A 32)} \\
\sqsubseteq & f^k_c(l, l', G_{\mathsf{K}}(l) *_{\mathsf{K}} \{(\_, h'_1 \uplus h_{\mathsf{K}}, L_1 \uplus L_{\mathsf{K}}, \bot)\}) & \textit{by (A 33)} \\
\sqsubseteq & f^k_c(l, l', G_{\mathsf{K}}(l)) *_{\mathsf{K}} \{(\_, h'_1 \uplus h_{\mathsf{K}}, L_1 \uplus L_{\mathsf{K}}, \bot)\} & \textit{by (A 4)}
\end{array}$$

By (A 13), $f^k_c(l, l', G_{\mathsf{K}}(l)) \neq \top$, hence, by (A 2) and (A 27), $\sigma' \neq \top$. Let $\sigma' = (R[k : r'], h', L)$. Then, by (A 27) and (A 13), for some $h_3 \in \mathsf{Heap}$ and $L_3 \in \mathsf{Lockset}$, we have $h' = h_3 \uplus h'_1 \uplus h_{\mathsf{K}}, L = L_3 \uplus L_1 \uplus L_{\mathsf{K}}$ and $(r', h_3, L_3) \in G_{\mathsf{K}}(r'(\mathtt{ip}))$. Using (A 11), we conclude that for some $h''_2 \in \mathsf{Heap}$ and

$$h'_0 \in [r'(\mathtt{sp})..(r'(\mathtt{ss}) + \mathtt{StackSize} - 1) : \_]$$

we have $h' = h''_2 \uplus h'_0 \uplus h'_1$ and

$$(\{r'\}, h''_2, L_2, \{k\} - (\mathsf{CPUid} - V)) \in \lfloor \llbracket \Delta_{\mathsf{K}}(r'(\mathtt{ip})) \rrbracket_\eta \rfloor^{\mathsf{KA}}_{r'} \star_{\mathsf{K}} \{(\emptyset, h_{\mathsf{K}}, L_{\mathsf{K}}, \emptyset)\}.$$

From this and (A 5), we get $r'(\mathtt{ip}) \in \mathsf{dom}(\mathsf{K})$. Let $M' = (M - \{r\}) \uplus \{r'\}$. Then by (A 34) this implies

$$(M', h''_2, L_2, V) \in \mathsf{IKernel}_\eta \star_{\mathsf{K}} \mathsf{IKernelLock}_{\mathsf{dom}(I_{\mathsf{K}}) - L_2} \star_{\mathsf{K}} \mathsf{IPercpu}_{\mathsf{CPUid} - V}. \tag{A 35}$$

Let $h''_1 = h'_1 \uplus h'_0$. Then from (A 31) we get

$$((r', h''_1, L_1), M') \in (\llbracket \mathsf{SchedSleep}_k(r'(\mathtt{ip})) \rrbracket_\eta \cap \mathsf{at}_{\mathsf{S}}(r'(\mathtt{ip}))) *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), M_{\mathsf{S}})\}.$$

Since $r'(\mathtt{ip}) \in \mathsf{dom}(\mathsf{K})$, together with (A 29) for $W = V$ and $W_1 = L_1$, this implies

$$((R[k : r'], h''_1, L_1), M', V) \in \mathsf{ISched}_\eta \star_{\mathsf{S}} \mathsf{ISchedLock}_{\mathsf{dom}(I_{\mathsf{S}}) - L_1}.$$

[4] Note that if there are several occurrences of r in M, $M - \{r\}$ removes only one of them.

By the definition of $\star_{\mathsf{SK}}$, from this and (A 35) we get $\sigma' \in \mathsf{IOs}_\eta$.

Case 6. σ' is obtained by applying the first rule in Figure 12, with the kernel executing `lock`. In this case

$$l \in \mathsf{labels}(\mathsf{K}), \quad c = \mathtt{lock}(\ell), \quad \ell \notin L, \quad \sigma' = (R[k : r[\mathtt{ip} : l']], h, L \cup \{\ell\}).$$

As in Case 5, there exist $h_{\mathsf{S}}, L_{\mathsf{S}}, M_{\mathsf{S}}, V_{\mathsf{S}}, V_{\mathsf{K}}, h'_1, h_0, h'_2, v$ satisfying the conditions stated there. Additionally, from (A 25) for some $h_{\mathsf{K}}, L_{\mathsf{K}}$ we get

$$(r, h'_2, L_2, v) \in G_{\mathsf{K}}(l) *_{\mathsf{K}} \mathscr{I}_{\mathsf{K}}(\ell) *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \bot)\}$$

and

$$(M - \{r\}, h_{\mathsf{K}}, L_{\mathsf{K}}, V - \{k\}) \in \underset{r'' \in M - \{r\}}{\circledast_{\mathsf{K}}} \lfloor [\![\Delta_{\mathsf{K}}(r''(\mathtt{ip}))]\!]_\eta \rfloor^{\mathsf{KA}}_{r''} \star_{\mathsf{K}}$$
$$\mathsf{IKernelLock}_{\mathsf{dom}(I_{\mathsf{K}}) - (L_2 \cup \{\ell\})} \star_{\mathsf{K}} \mathsf{IPercpu}_{\mathsf{CPUid} - V}. \quad \text{(A 36)}$$

This implies

$$(r[\mathtt{ip} : l'], h'_2, L_2 \cup \{\ell\}, v) \in \mathsf{to}_{\mathsf{K}}(l', (G_{\mathsf{K}}(l) *_{\mathsf{K}} \mathscr{I}_{\mathsf{K}}(\ell) *_{\mathsf{K}} \mathsf{lk}_{\mathsf{K}}(\ell))) *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \bot)\}.$$

Hence, by (A 15)

$$(r[\mathtt{ip} : l'], h'_2, L_2 \cup \{\ell\}, v) \in G_{\mathsf{K}}(l') *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \bot)\}.$$

Then from (A 11) it follows that

$$(\{r[\mathtt{ip} : l']\}, h_2, L_2 \cup \{\ell\}, \{k\} - (\mathsf{CPUid} - V)) \in \lfloor [\![\Delta_{\mathsf{K}}(l')]\!]_\eta \rfloor^{\mathsf{KA}}_{r[\mathtt{ip}:l']} \star_{\mathsf{K}} \{(\emptyset, h_{\mathsf{K}}, L_{\mathsf{K}}, \emptyset)\}.$$

From this and (A 5), we get $l' \in \mathsf{dom}(\mathsf{K})$. Let $M' = (M - \{r\}) \uplus \{r[\mathtt{ip} : l']\}$. Then by (A 36) we get

$$(M', h_2, L_2 \cup \{\ell\}, V) \in \mathsf{IKernel}_\eta \star_{\mathsf{K}} \mathsf{IKernelLock}_{\mathsf{dom}(I_{\mathsf{K}}) - (L_2 \cup \{\ell\})} \star_{\mathsf{K}} \mathsf{IPercpu}_{\mathsf{CPUid} - V}. \quad \text{(A 37)}$$

From (A 30) we get

$$((r[\mathtt{ip} : l'], h_1, L_1), M') \in ([\![\mathsf{SchedSleep}_k(l')]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l')) *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), M_{\mathsf{S}})\}.$$

Since $l' \in \mathsf{dom}(\mathsf{K})$, together with (A 29) for $W = V - \{k\}$ and $W_1 = L_1$, this implies

$$((R[k : r[\mathtt{ip} : l']], h_1, L_1), M', V) \in \mathsf{ISched}_\eta \star_{\mathsf{S}} \mathsf{ISchedLock}_{\mathsf{dom}(I_{\mathsf{S}}) - L_1}. \quad \text{(A 38)}$$

By the definition of $\star_{\mathsf{SK}}$, from this and (A 37) we get $\sigma' \in \mathsf{IOs}_\eta$.

Case 7. σ' is obtained by applying the first or the third rule in Figure 12, with the kernel executing `unlock`. In this case, $l \in \mathsf{labels}(\mathsf{K})$, $c = \mathtt{unlock}(\ell)$ and (A 27) holds.

As in Case 5, there exist $h_{\mathsf{S}}, L_{\mathsf{S}}, M_{\mathsf{S}}, V_{\mathsf{S}}, V_{\mathsf{K}}, h'_1, h_0, h'_2, v, h_{\mathsf{K}}, L_{\mathsf{K}}$ satisfying the conditions stated there. Then using (A 33), we get

$$(r[\mathtt{ip} : l'], h'_2, L_2, v) \in \mathsf{to}_{\mathsf{K}}(l', G_{\mathsf{K}}(l)) *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \bot)\}.$$

Hence, by (A 17)

$$(r[\mathtt{ip} : l'], h'_2, L_2, v) \in G_{\mathsf{K}}(l') *_{\mathsf{K}} \mathscr{I}_{\mathsf{K}}(\ell) *_{\mathsf{K}} \mathsf{lk}_{\mathsf{K}}(\ell) *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \bot)\}.$$

Hence, $\ell \in L_2$, which means that $\sigma' \neq \top$. Then $\sigma' = (R[k : r[\mathtt{ip} : l']], h, L - \{\ell\})$. The above also implies

$$(r[\mathtt{ip} : l'], h_2', L_2 - \{\ell\}, v) \in G_{\mathsf{K}}(l') *_{\mathsf{K}} \mathscr{I}_{\mathsf{K}}(\ell) *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \bot)\}.$$

Then from (A 11) it follows that

$$(\{r[\mathtt{ip} : l']\}, h_2, L_2 - \{\ell\}, \{k\} - (\mathsf{CPUid} - V)) \in \\ \lfloor [\![\Delta_{\mathsf{K}}(l')]\!]_\eta \rfloor^{\mathsf{KA}}_{r[\mathtt{ip}:l']} \star_{\mathsf{K}} \lfloor \mathscr{I}_{\mathsf{K}}(\ell) *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \bot)\} \rfloor^{\mathsf{KL}}.$$

From this and (A 5) we get $l' \in \mathsf{dom}(\mathsf{K})$. Let $M' = (M - \{r\}) \uplus \{r[\mathtt{ip} : l']\}$. Then by (A 34) we get

$$(M', h_2, L_2 - \{\ell\}, V) \in \mathsf{IKernel}_\eta \star_{\mathsf{K}} \mathsf{IKernelLock}_{\mathsf{dom}(I_{\mathsf{K}}) - (L_2 - \{\ell\})} \star_{\mathsf{K}} \mathsf{IPercpu}_{\mathsf{CPUid} - V}.$$

As in the previous case, from (A 30) and (A 29) for $W = V - \{k\}$ and $W_1 = L_1$, we can establish (A 38). Together with the last inclusion, this implies $\sigma' \in \mathsf{IOs}_\eta$.

Case 8. σ' is obtained by applying the first or the third rule in Figure 12, with the kernel executing `cli` or `sti`. These cases are similar to the previous two and are omitted. They rely on (A 18) and (A 19) instead of (A 15) and (A 17).

Case 9. σ' is obtained by applying the first or the third rule in Figure 12, with the kernel executing `icall(schedule)`. In this case, $l \in \mathsf{labels}(\mathsf{K})$, $c = \mathtt{icall(schedule)}$ and (A 27) holds.

As in Case 5, there exist $h_{\mathsf{S}}, L_{\mathsf{S}}, M_{\mathsf{S}}, V_{\mathsf{S}}, V_{\mathsf{K}}, h_1', h_0, h_2', v, h_{\mathsf{K}}, L_{\mathsf{K}}$ satisfying the conditions stated there. Additionally, $r(\mathtt{if}) = 1$ and hence, $k \notin V$ and $v = \bot$. From (A 33) we then get

$$(r[\mathtt{ip} : l'], h_2', L_2, \bot) \in \mathsf{to}_{\mathsf{K}}(l', G_{\mathsf{K}}(l)) *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \bot)\}.$$

By (A 20), this implies

$$(r[\mathtt{ip} : l'], h_2', L_2, \bot) \in G_{\mathsf{K}}(l') *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \bot)\}.$$

Then using (A 11) we get

$$(\{r[\mathtt{ip} : l']\}, h_2, L_2, \emptyset) \in \lfloor [\![\Delta_{\mathsf{K}}(l')]\!]_\eta \rfloor^{\mathsf{KA}}_{r[\mathtt{ip}:l']} \star_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \emptyset)\}.$$

Let $M' = (M - \{r\}) \uplus \{r[\mathtt{ip} : l']\}$. Then by (A 34) we have

$$(M', h_2, L_2, V) \in \mathsf{IKernel}_\eta \star_{\mathsf{S}} \mathsf{IKernelLock}_{\mathsf{dom}(I_{\mathsf{K}}) - L_2} \star_{\mathsf{S}} \mathsf{IPercpu}_{\mathsf{CPUid} - V}. \tag{A 39}$$

From (A 30) we get $\mathsf{dom}(h) \supseteq \{r(\mathtt{sp}), \ldots, r(\mathtt{sp}) + m + 1\}$, which implies that $\sigma' \neq \top$. Then $\sigma' = (R[k : r''], h_1'' \uplus h_2, L)$, where

$$r'' = r[\mathtt{ip} : \mathtt{schedule}, \mathtt{sp} : (r(\mathtt{sp}) + m + 1), \mathtt{if} : 0]$$

and

$$h_1'' = h_1[r(\mathtt{sp}) : l', (r(\mathtt{sp}) + 1) : r(\mathtt{gr}_1), \ldots, (r(\mathtt{sp}) + m) : r(\mathtt{gr}_m)]. \tag{A 40}$$

From (A 30) we also get

$$((r, h_1, L_1), M') \in \mathscr{J}_k *_{\mathsf{S}} \\ \{((\_, [r(\mathtt{sp})..(r(\mathtt{ss}) + \mathtt{StackSize} - 1) : \_] \uplus h_{\mathsf{S}}, L_{\mathsf{S}}), \{r[\mathtt{ip} : l']\} \uplus M_{\mathsf{S}})\}.$$

Hence,

$$((r'',h''_1,L_1),M') \in \mathscr{J}_k *_{\mathsf{S}} \{((\_,\ [(r''(\mathtt{sp})-m-1)..(r''(\mathtt{sp})-1) : l'r(\mathtt{gr}_1)\dots r(\mathtt{gr}_m),\\ r''(\mathtt{sp})..(r''(\mathtt{ss})+\mathtt{StackSize}-1) : \_] \uplus h_{\mathsf{S}}, L_{\mathsf{S}}), \{r[\mathtt{ip} : l']\} \uplus M_{\mathsf{S}})\}.$$

From (A 33) and (A 11) we get $0 \leqslant r(\mathtt{sp}) - r(\mathtt{ss}) \leqslant \mathtt{StackBound}$, so that $0 \leqslant r''(\mathtt{sp}) - r''(\mathtt{ss}) - m - 1 \leqslant \mathtt{StackBound}$. Thus,

$$((r'',h''_1,L_1),M') \in ([\![\mathsf{SchedState}_k]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(\mathtt{schedule})) *_{\mathsf{S}} \{((\_,h_{\mathsf{S}},L_{\mathsf{S}}),M_{\mathsf{S}})\}.$$

Together with (A 29) for $W = V - \{k\}$ and $W_1 = L_1$ and (A 7), this implies

$$((R[k:r''],h''_1,L_1),M') \in \mathsf{ISched}_\eta \star_{\mathsf{S}} \mathsf{ISchedLock}_{\mathsf{dom}(I_{\mathsf{S}})-L_1}.$$

By the definition of $\star_{\mathsf{SK}}$, from this and (A 39) we get $\sigma' \in \mathsf{IOs}_\eta$.

Case 10. σ' is obtained by applying the second or the last rule in Figure 12, i.e., by executing an interrupt. This case is virtually identical to the previous one and is omitted.

Case 11. σ' is obtained by applying the first or the third rule in Figure 12, with the scheduler executing `iret` at l_s or l_c. In this case

$$l \in \{l_s,l_c\}, \quad l' \in \{l_s+1,l_c+1\}, \quad c = \mathtt{iret}$$

and (A 27) holds.

From (A 24), there exist $h_{\mathsf{S}},L_{\mathsf{S}},M_{\mathsf{S}},V_{\mathsf{S}},V_{\mathsf{K}}$ satisfying the conditions stated there, in particular, (A 28) and (A 29) for $W = V$ and $W_1 = L_1$. Then from (A 28), (A 8) and (A 10) we get

$$((r,h_1,L_1),M) \in ([\![\mathsf{SchedState}_k]\!]_\eta \cap (\mathsf{at}_{\mathsf{S}}(l_s) \cup \mathsf{at}_{\mathsf{S}}(l_c))) *_{\mathsf{S}} \{((\_,h_{\mathsf{S}},L_{\mathsf{S}}),M_{\mathsf{S}})\}. \qquad \text{(A 41)}$$

Hence, $\mathsf{dom}(h_1) \supseteq \{r(\mathtt{sp})-m-1,\dots,r(\mathtt{sp})-1\}$ and $\sigma' \neq \top$. Let

$$l'' = h_1(r(\mathtt{sp})-m-1), \quad g_1 = h_1(r(\mathtt{sp})-m), \quad \dots, \quad g_m = h_1(r(\mathtt{sp})-1).$$

Then, $\sigma' = (R[k:r'],h,L)$, where

$$r' = r[\mathtt{ip} : l'', \mathtt{sp} : (r(\mathtt{sp})-m-1), \mathtt{gr}_1 : g_1, \dots, \mathtt{gr}_m : g_m, \mathtt{if} : 1].$$

From (A 41) we now obtain

$$((r[\mathtt{ip}:l''],h_1,L_1),M) \in ([\![\mathsf{SchedState}_k]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l'')) *_{\mathsf{S}} \{((\_,h_{\mathsf{S}},L_{\mathsf{S}}),M_{\mathsf{S}})\}.$$

Hence,

$$((r',h_1,L_1),M) \in \\ \mathsf{at}_{\mathsf{S}}(l'') \cap (\{((\_,[\mathtt{sp}..(\mathtt{ss}+\mathtt{StackSize}-1):\_] \uplus h_{\mathsf{S}},L_{\mathsf{S}}),\{r'\} \uplus M_{\mathsf{S}})\} *_{\mathsf{S}} \mathscr{J}_k),$$

which is equivalent to

$$((r',h_1,L_1),M) \in ([\![\mathsf{SchedSleep}_k(l'')]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l'')) *_{\mathsf{S}} \{((\_,h_{\mathsf{S}},L_{\mathsf{S}}),M_{\mathsf{S}})\}.$$

Note that $r' \in M$. Hence, from (A 25) and (A 5) we get $l'' \in \mathsf{labels}(\mathsf{K})$. By (A 29) for $W = V$ and $W_1 = L_1$ we then have

$$((R[k:r'],h_1,L_1),M) \in \mathsf{ISched}_\eta \star_{\mathsf{S}} \mathsf{ISchedLock}_{\mathsf{dom}(I_{\mathsf{S}})-L_1}.$$

From (A 25) and the definition of $\star_{\mathsf{SK}}$, we get $\sigma' \in \mathsf{IOs}_\eta$.

Case 12. σ' is obtained by applying the first or the third rule in Figure 12, with the kernel executing $\mathtt{icall(create)}$. In this case, $l \in \mathsf{labels}(\mathsf{K})$, $c = \mathtt{icall(create)}$ and (A 27) holds.

As in Case 5, there exist $h_{\mathsf{S}}, L_{\mathsf{S}}, M_{\mathsf{S}}, V_{\mathsf{S}}, V_{\mathsf{K}}, h'_1, h_0, h'_2, h_{\mathsf{K}}, L_{\mathsf{K}}$ satisfying the conditions stated there. Additionally, $r(\mathtt{if}) = 1$ and hence, $k \notin V$ and $v = \bot$. From (A 33) and (A 21) we get

$$(r, h'_2, L_2, \bot) \in \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}})\} *_{\mathsf{K}} \llbracket \exists \gamma.\, \gamma(\mathbf{if}) = 1 \wedge \mathsf{desc}(\mathtt{gr}_1, \gamma) * P * Q \rrbracket_\eta.$$

Hence, there exists r' such that $r'(\mathtt{if}) = 1$ and

$$(r, h'_2, L_2, \bot) \in \boldsymbol{desc}_\eta(u, r') *_{\mathsf{K}} \llbracket P \rrbracket_{\eta'} *_{\mathsf{K}} \llbracket Q \rrbracket_{\eta'} *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \bot)\},$$

where $u = r(\mathtt{gr}_1)$ and $\eta' = \eta[\gamma : r']$. Since $\mathsf{free}(P) \cap \mathsf{Reg} = \emptyset$ and $\mathsf{free}(\mathsf{desc}(d, \gamma)) \cap \mathsf{Reg} = \emptyset$, we have

$$(r[\mathtt{ip} : l'], h'_2, L_2, \bot) \in \boldsymbol{desc}_\eta(u, r') *_{\mathsf{K}} \llbracket P \rrbracket_{\eta'} *_{\mathsf{K}} \mathsf{to}_{\mathsf{K}}(l', \llbracket Q \rrbracket_{\eta'}) *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \bot)\}.$$

Using (A 22), we then get

$$(r[\mathtt{ip} : l'], h'_2, L_2, \bot) \in \boldsymbol{desc}_\eta(u, r') *_{\mathsf{K}} \llbracket P \rrbracket_{\eta'} *_{\mathsf{K}} G_{\mathsf{K}}(l') *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \bot)\}.$$

According to (A 11), this implies

$$(\{r[\mathtt{ip} : l']\}, h_2, L_2, \emptyset) \in \lfloor \boldsymbol{desc}_\eta(u, r') \rfloor^{\mathsf{KL}} \star_{\mathsf{K}} \lfloor \llbracket P \rrbracket_{\eta'} \rfloor^{\mathsf{KL}} \star_{\mathsf{K}}$$
$$\lfloor \llbracket \Delta_{\mathsf{K}}(l') \rrbracket_\eta \rfloor^{\mathsf{KA}}_{r[\mathtt{ip}:l']} \star_{\mathsf{K}} \{(\emptyset, h_{\mathsf{K}}, L_{\mathsf{K}}, \emptyset)\}.$$

Then for some $h''_2, h_d \in \mathsf{Heap}$ such that $h_2 = h''_2 \uplus h_d$ we have $\{(\_, h_d, \emptyset, \bot)\} \subseteq \boldsymbol{desc}_\eta(u, r')$ (recall that all states from $\boldsymbol{desc}_\eta(u, r')$ have an empty lockset) and

$$(\{r[\mathtt{ip} : l']\}, h''_2, L_2, \emptyset) \in \lfloor \llbracket P \rrbracket_{\eta'} \rfloor^{\mathsf{KL}} \star_{\mathsf{K}} \lfloor \llbracket \Delta_{\mathsf{K}}(l') \rrbracket_\eta \rfloor^{\mathsf{KA}}_{r[\mathtt{ip}:l']} \star_{\mathsf{K}} \{(\emptyset, h_{\mathsf{K}}, L_{\mathsf{K}}, \emptyset)\}.$$

Then from (A 23) and (A 11) we get

$$(\{r[\mathtt{ip} : l'], r'\}, h''_2, L_2, \emptyset) \in \lfloor \llbracket \Delta_{\mathsf{K}}(r'(\mathtt{ip})) \rrbracket_\eta \rfloor^{\mathsf{KA}}_{r'} \star_{\mathsf{K}} \lfloor \llbracket \Delta_{\mathsf{K}}(l') \rrbracket_\eta \rfloor^{\mathsf{KA}}_{r[\mathtt{ip}:l']} \star_{\mathsf{K}} \{(\emptyset, h_{\mathsf{K}}, L_{\mathsf{K}}, \emptyset)\}.$$

Let $M' = (M - \{r\}) \uplus \{r[\mathtt{ip} : l'], r'\}$. Then by (A 34) we have

$$(M', h''_2, L_2, V) \in \mathsf{IKernel}_\eta \star_{\mathsf{S}} \mathsf{IKernelLock}_{\mathsf{dom}(I_{\mathsf{K}}) - L_2} \star_{\mathsf{S}} \mathsf{IPercpu}_{\mathsf{CPUid} - V}. \tag{A 42}$$

As in Case 9, we can assume that

$$\sigma' = (R[k : r''], h''_1 \uplus h_2, L) = (R[k : r''], h''_1 \uplus h_d \uplus h''_2, L),$$

where

$$r'' = r[\mathtt{ip} : \mathtt{create}, \mathtt{sp} : (r(\mathtt{sp}) + m + 1), \mathtt{if} : 0]$$

and h''_1 is defined by (A 40). Let $h'''_1 = h''_1 \uplus h_d$. Then from (A 30) we get

$$((r, h'''_1, L_1), M') \in \mathscr{J}_k *_{\mathsf{S}} (\boldsymbol{desc}_\eta(u, r') \times \{\emptyset\}) *_{\mathsf{S}}$$
$$\{((\_, [r(\mathtt{sp})..(r(\mathtt{sp}) + m) : l' r(\mathtt{gr}_1) \ldots r(\mathtt{gr}_m),$$
$$(r(\mathtt{sp}) + m + 1)..(r(\mathtt{ss}) + \mathtt{StackSize} - 1) : \_] \uplus h_{\mathsf{S}}, L_{\mathsf{S}}), \{r[\mathtt{ip} : l'], r'\} \uplus M_{\mathsf{S}})\}.$$

Similar to how it was done in Case 9, using (A 9) we now establish

$$((r'', h_1''', L_1), M') \in G_{\mathsf{S}}^k(\texttt{create}) *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), M_{\mathsf{S}})\}.$$

Together with (A 29) for $W = V$ and $W_1 = L_1$, this implies

$$((R[k : r''], h_1''', L_1), M', V) \in \mathsf{ISched}_\eta \star_{\mathsf{S}} \mathsf{ISchedLock}_{\mathsf{dom}(I_{\mathsf{S}}) - L_1}.$$

By the definition of $\star_{\mathsf{SK}}$, from this and (A 42) we get $\sigma' \in \mathsf{IOs}_\eta$. □

References

Back, R.-J. (1981) On correct refinement of programs. *J. Comput. Syst. Sci.* **23**, 49–68.

Berdine, J., O'Hearn, P. W., Reddy, U. S. & Thielecke, H. (2002) Linear continuation-passing. *Higher-order Symb. Comput.* **15**(2–3), 181–208.

Bovet, D. & Cesati, M. (2005) *Understanding the Linux Kernel*, 3rd ed. O'Reilly.

Brookes, S. D. (2007) A semantics of concurrent separation logic. *Theor. Comput. Sci.* **375**(1–3), 227–270.

Calcagno, C., O'Hearn, P. W. & Yang, H. (2007) Local action and abstract separation logic. In *Symposium on Logic in Computer Science (LICS'07)*. IEEE, pp. 366–378.

Charlton, N. (2011) Hoare logic for higher order store using simple semantics. In *Conference on Logic, Language, Information and Computation (WoLLIC'11)*. LNCS, vol. 6642. Springer, pp. 52–66.

Clarke, D. G., Noble, J. & Potter, J. (2001) Simple ownership types for object containment. In *European Conference on Object-Oriented Programming (ECOOP'01)*. LNCS, vol. 2072. Springer, pp. 53–76.

Cohen, E., Schulte, W. & Tobies, S. (2010) Local verification of global invariants in concurrent programs. In *Conference on Computer-Aided Verification (CAV'10)*. LNCS, vol. 6174. Springer, pp. 480–494.

Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M. & Yang, H. (2013) Views: Compositional reasoning for concurrent programs. In *Symposium on Principles of Programming Languages (POPL'13)*. ACM, pp. 287–300.

Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M. & Vafeiadis, V. (2010) Concurrent abstract predicates. In *European Conference on Object-Oriented Programming (ECOOP'10)*. LNCS, vol. 6183. Springer, pp. 504–528.

Feng, X., Ferreira, R. & Shao, Z. (2007a) On the relationship between concurrent separation logic and assume-guarantee reasoning. In *European Conference on Programming (ESOP'07)*. LNCS, vol. 4421. Springer, pp. 173–188.

Feng, X., Ni, Z., Shao, Z. & Guo, Y. (2007b) An open framework for foundational proof-carrying code. In *Workshop on Types in Language Design and Implementation (TLDI'07)*. ACM, pp. 67–78.

Feng, X., Shao, Z., Dong, Y. & Guo, Y. (2008a) Certifying low-level programs with hardware interrupts and preemptive threads. In *Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, pp. 170–182.

Feng, X., Shao, Z., Guo, Y. & Dong, Y. (2008b) Combining domain-specific and foundational logics to verify complete software systems. In *Conference on Verified Software: Theories, Tools, Experiments (VSTTE'08)*. LNCS, vol. 5295. Springer, pp. 54–69.

Feng, X., Shao, Z., Vaynberg, A., Xiang, S. & Ni, Z. (2006) Modular verification of assembly code with stack-based control abstractions. In *Conference on Programming Language Design and Implementation (PLDI'06)*. ACM, pp. 401–414.

Gargano, M., Hillebrand, M., Leinenbach, D. & Paul, W. (2005) On the correctness of operating system kernels. In *Conference on Theorem Proving in Higher-Order Logics (TPHOLs'05)*. LNCS, vol. 3603. Springer, pp. 1–16.

Gotsman, A. (2009) *Logics and Analyses for Concurrent Heap-Manipulating Programs*. PhD Thesis, University of Cambridge.

Gotsman, A., Berdine, J. & Cook, B. (2011) Precision and the conjunction rule in concurrent separation logic. *ENTCS* **276**(1), 171–190. MFPS'11: Mathematical Foundations of Programming Semantics.

Gotsman, A., Berdine, J., Cook, B., Rinetzky, N. & Sagiv, M. (2007) Local reasoning for storable locks and threads. In *Asian Symposium on Programming Languages and Systems (APLAS'07)*. LNCS, vol. 4807. Springer, pp. 19–37.

Gotsman, A. & Yang, H. (2013) *Electronic Appendix for This Paper*. Available from http://dx.doi.org/10.1017/S0956796813000075.

Hasegawa, M. (2002) Linearly used effects: Monadic and CPS transformations into the linear lambda calculus. In *International Symposium on Functional and Logic Programming (FLOPS'02)*. LNCS, vol. 2441. Springer, pp. 167–182.

Hasegawa, M. (2004) Semantics of linear continuation-passing in call-by-name. In *International Symposium on Functional and Logic Programming (FLOPS'04)*. LNCS, vol. 2998. Springer, pp. 229–243.

Jones, C. (2007) Splitting atoms safely. *Theor. Comput. Sci.* **375**, 109–119.

Jones, C. B. (1983) Specification and design of (parallel) programs. In *IFIP Congress*, pp. 321–332.

Klein, G. (2009) Operating system verification–an overview. *Sādhanā* **34**, 26–69.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. & Winwood, S. (2009) seL4: Formal verification of an OS kernel. In *Symposium on Operating Systems Principles (SOSP'09)*. ACM, pp. 207–220.

Laird, J. (2005) Game semantics and linear CPS interpretation. *Theor. Comput. Sci.* **333**(1–2), 199–224.

Love, R. (2010) *Linux Kernel Development*, 3rd ed. Addison Wesley.

Maeda, T. & Yonezawa, A. (2009) Writing an OS kernel in a strictly and statically typed language. In *Formal to Practical Security*. LNCS, vol. 5458. Springer, pp. 181–197.

Ni, Z. & Shao, Z. (2006) Certified assembly programming with embedded code pointers. In *Symposium on Principles of Programming Languages (POPL'06)*. ACM, pp. 320–333.

O'Hearn, P. W. (2007) Resources, concurrency and local reasoning. *Theor. Comput. Sci.* **375**, 271–307.

Parkinson, M. & Bierman, G. (2005) Separation logic and abstraction. In *Symposium on Principles of Programming Languages (POPL'05)*. ACM, pp. 247–258.

Pnueli, A. (1985) In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*. Springer, pp. 123–144.

Reynolds, J. C. (2002) Separation logic: A logic for shared mutable data structures. In *Symposium on Logic in Computer Science (LICS'02)*. IEEE, pp. 55–74.

Schwinghammer, J., Birkedal, L., Reus, B. & Yang, H. (2009) Nested Hoare triples and frame rules for higher-order store. In *Conference on Computer Science Logic (CSL'09)*. LNCS, vol. 5771. Springer, pp. 440–454.

Shao, Z. (2010) Certified software. *Commun. ACM* **53**(12), 56–66.

Thielecke, H. (2003) From control effects to typed continuation passing. In *Symposium on Principles of Programming Languages (POPL'03)*. ACM, pp. 139–149.

Turon, A. & Wand, M. (2011) A separation logic for refining concurrent objects. In *Symposium on Principles of Programming Languages (POPL'11)*. ACM, pp. 247–258.

Vafeiadis, V. & Parkinson, M. J. (2007) A marriage of rely/guarantee and separation logic. In *Conference on Concurrency Theory (CONCUR'07)*. LNCS, vol. 4703. Springer, pp. 256–271.

Yang, J. & Hawblitzel, C. (2010) Safe to the last instruction: Automated verification of a type-safe operating system. In *Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, pp. 99–110.