# Modular Verification of Preemptive OS Kernels

Alexey Gotsman
IMDEA Software Institute
Alexey.Gotsman@imdea.org

Hongseok Yang
University of Oxford
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 Linux, FreeBSD or XNU, 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. This is achieved by establishing a novel form of refinement between an operational semantics of the real machine and an axiomatic semantics of OS processes, where the latter assumes an abstract machine with each process executing on a separate virtual CPU. The refinement is local in the sense that the logic focuses only on the relevant state of the kernel while verifying the scheduler. We illustrate the power of our logic by verifying an example scheduler, modelled on the one from Linux 2.6.11.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.4.1 [*Operating Systems*]: Process Management

***General Terms*** Languages, Theory, Verification

***Keywords*** Verification, Concurrency, Scheduling, Modularity

## 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 formal verification can tackle realistic OS kernels, such as a variant of the L4 microkernel [17] and Microsoft's Hyper-V hypervisor [3]. Having dealt with relatively small microkernels, these projects nevertheless give us hope that in the future we will be able to verify the likes of kernels from today's mainstream operating systems, such as Windows and Linux.

In this paper, we tackle one of the main challenges in realising this hope—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 [3–5, 13, 20, 23] that can reason about concurrent code. 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.

The problem is that all these logics can verify code only under so-called interleaving semantics, expressed by the well-known operation 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 reality, the processes are multiplexed onto available CPUs by a scheduler. Furthermore, in a preemptive kernel, the scheduler is part of the kernel being verified and its correctness is *interdependent* with the correctness of the rest of the kernel (which, in the following, we refer to as just the kernel). Thus, what you see in a C implementation of OS system calls and what most logics reason about is not what you execute in such a kernel. When reasoning about a system call implementation in reality, 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 scheduling 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.

In this paper we propose a logic that is able to decompose the verification of safety properties of preemptive OS code into verifying the scheduler and preemptable code separately. This is the first logic that can handle interdependencies between the scheduler and the kernel present in mainstream OS kernels, such as Linux, FreeBSD and XNU. Our logic consists of two proof systems, which we call high-level and low-level. The high-level system verifies preemptable code assuming that the scheduler is implemented cor-



*ICFP'11,* September 19–21, 2011, Tokyo, Japan.

rectly (Section 4.3). It hides the complex manipulation of control by the scheduler, which stores program counters of processes (describing their continuations) and jumps to one of them during a context switch. In this way, the high-level proof system provides the illusion that 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 4.4).

A common way to simplify reasoning about program components sharing an address space, such as the scheduler and the kernel, is to introduce the notion of *ownership* of memory areas: only the component owning an area of memory has the right to access it. The main 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. This leads to several technical challenges our logic has to deal with.

First, this setting introduces an obligation to prove that the scheduler and the kernel do not corrupt each other's data structures. To this end, we base our proof systems on concurrent separation logic [20], which 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.

Second, in reasoning about mainstream operating systems, the ownership transfer between the scheduler and the kernel can involve not only fixed memory cells, but arbitrary logical facts describing them (Section 4.3). Such 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. In this paper we resolve this problem by introducing a concept of *logical refinement* between an operational semantics of the real machine and an axiomatic (as opposed to operational) semantics of the abstract machine, defined in our logic by the high-level proof system. Namely, 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, as would be the case in the usual semantic refinement. Instead, our soundness statement (Section 6) 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.

Finally, while we would like to hide the scheduler state completely when reasoning about the kernel, the converse is not true: the scheduler has to be able to access at least some of the local state of every process, such as its register values. For this reason, the low-level proof system (Section 4.4) includes special assertions to describe the state of the OS processes the scheduler manages. These assertions are also interpreted as exclusive permissions to schedule the corresponding processes, which allows us to reason about scheduling on multiprocessors. A novel feature of the low-level proof system that allows verifying schedulers separately from the rest of the kernel is its *locality*: proofs about the scheduler focus only on a small relevant part of the state of processes.

Even though all of the OS verification projects carried out so far had to deal with a scheduler (see Section 7 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, modelled on the one from Linux 2.6.11 (Sections 2.2 and 5), which exhibits the issues mentioned above.

## 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

To keep the presentation tractable, 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 consider a machine with 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 reality they would be implemented as spin-locks). Every CPU has a single interrupt, with its handler located at a distinguished label `schedule`, which a scheduler can use to trigger a context switch. There are four special-purpose registers, `ip`, `if`, `ss` and `sp`, and $m$ general-purpose ones, $\mathtt{gr}_1, \ldots, \mathtt{gr}_m$. The `ip` register is the instruction pointer. The `if` register controls interrupts: they are disabled on the corresponding CPU when it is 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 `ss`$<$`sp`.

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—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`$(\ell)$ and `unlock`$(\ell)$ acquire and release the lock $\ell$.
- `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.

### 2.2 Motivating example

Figure 1 presents an implementation of the scheduler we use as a running example. We would like 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. The scheduler uses data structures and an interface with the rest of the kernel similar to the ones in Linux 2.6.11 [2][1]. To concentrate on key issues of scheduler verification, we make some simplifying assumptions: we do not consider virtual memory and assume that processes are never removed and never go to sleep. We have also omitted the code for data structure initialisation.

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, thus, 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 Figure 1, is mapped to our machine.

We assume that global variables are allocated at fixed addresses 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. 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`) are passed via registers. We assume macros `lock`, `unlock`, `savecpuid` and `iret` for the corresponding machine commands. We also use some library functions: e.g., `remove_node` deletes a node from the doubly-linked list it belongs to, and `insert_node_after` inserts the node given as its second argument after the list node given as its first argument.

***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, meaning that a runqueue can only be accessed with the corresponding lock held. 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 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 de-

[1] We modelled our scheduler on an older version of the Linux kernel (from 2005) because it uses simpler data structures. Newer versions use more efficient data structures [18] that would only complicate our running example without adding anything interesting.

```
#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 schedule() {
  int cpu;
  Process *old_process;
  savecpuid(&cpu);
  load_balance(cpu);
  old_process = current[cpu];
  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[0]);
  unlock(runqueue_lock[cpu]);
  iret();
}

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 = malloc(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;
}
```

**Figure 1.** The example scheduler

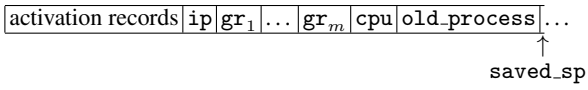


**Figure 2.** The invariant of the stack of a preempted process

scriptor. When a process is preempted, the `saved_sp` field is used to save the value of the stack pointer register `sp`. 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.

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, 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 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`, which migrates processes between CPUs to balance the load; we describe it below. The function then decrements the timeslice of the currently running process, and if it becomes zero, schedules another one. The processes are scheduled in a round-robin fashion, thus, the function inserts the current process at the end of the local runqueue and dequeues the process at the front of the runqueue, making it current. The function also refills the scheduling quantum of the process being descheduled. The runqueue manipulations are done with the corresponding lock held. 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. 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, the values of general-purpose registers saved upon the interrupt, and the activation record of `schedule`, as shown in Figure 2. This invariant holds for descriptors of all preempted processes.

The actual context switch is performed by the assignment to `sp`, which switches the current stack to another one satisfying the invariant in Figure 2. Since this changes the activation record of `schedule`, the function has to update the `cpu` variable, which lets it then retrieve and load the new value of `ss`. The `iret` command at the end of `schedule` loads the values of the registers stored on the stack and enables interrupts, thus completing the context switch.

***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. To migrate a process, the function chooses a runqueue to steal a process from and locks it together with the current runqueue in the order determined by the corresponding CPU identifiers, to avoid deadlocks. The function then removes one process from the victim runqueue, if it is non-empty, and inserts it into the current one. Note that two concurrent scheduler invocations executing `load_balance` on different CPUs may access the same runqueue. While verifying the OS, we have to ensure they synchronise their accesses correctly.

***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, hence, its stack must satisfy the invariant in Figure 2. To prevent deadlocks, `create` must be called using `icall`, which disables interrupts. Upon a call to `create`, the ownership of the descriptor is transferred from the kernel to the scheduler.

***The*** `fork` ***function*** is 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, copies the stack of the current process to it and initialises the stack as expected by `create` (Figure 2). This amounts to discarding the topmost activation record of `fork` and pushing a fake activation record of `schedule` (note that the values of registers the new process should start from 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 as a whole, modelling every CPU as a process in one of the existing program logics for concurrency [3–5, 13, 20, 23]. 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 a system call implementation we would end up having to reason explicitly about invariants and actions of both `schedule` and all other processes, making the reasoning unintuitive 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 in detail, we give an informal overview of the reasoning principles behind it.

***Modular reasoning via memory partitioning.*** The first issue we have to deal with while designing the logic is how to verify the scheduler and the kernel separately, despite 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. 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`.

As we have noted before, our logic consists of two proof systems: the high-level system (Section 4.3) for verifying the ker-

nel and the low-level one for the scheduler (Section 4.4). These proof systems implement a form of assume-guarantee reasoning between the two components, 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.*** We use concurrent separation logic [20] as a basis for modular reasoning within a given component, i.e., either among concurrent OS processes or concurrent scheduler invocations on different CPUs. This choice 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 in the assertions in a proof of a command is guaranteed not to be changed by it. While reasoning about a component, we consider only the memory partition belonging to it. Hence, we automatically know that the component cannot modify the others.

Concurrent separation logic achieves modular reasoning by further partitioning the memory owned by the component under consideration 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 3. 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. Resource invariants restrict how processes can change the protected state, and hence, allow reasoning about them in isolation.

***Scheduler-agnostic verification of kernel code.*** The high-level proof system (Section 4.3) reasons about preemptable code assuming an abstract machine where every process has its own virtual CPU. 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 4. We have primed process identifiers in the figure to emphasise that the virtual 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 it: 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 scheduler getting executed upon an interrupt. Explicit calls to the scheduler are treated as if they were executed atomically.

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.

***Proving schedulers correct via logical refinement.*** The use of the high-level proof system is justified by verifying the scheduler implementation using a low-level proof system (Section 4.4). What does it mean for a scheduler to be functionally correct? Intuitively, a scheduler must provide an illusion of a system where every process has its own virtual CPU with a dedicated set of registers. To formalise this, we could define a semantics of such an abstract system and prove that any behaviour of the concrete system is reproducible in the abstract one, thus establishing a refinement between the two systems. The main technical challenge we have to

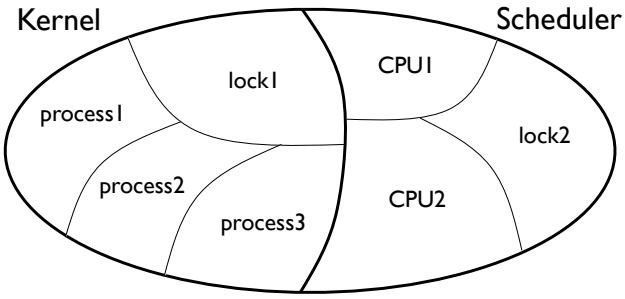


**Figure 3.** The partitioning of the system state enforced by the logic. The memory is partitioned into two parts, owned by the scheduler and the kernel, respectively. The memory of each component is further partitioned into parts local to processes or scheduler invocations on a given CPU, and parts protected by locks.

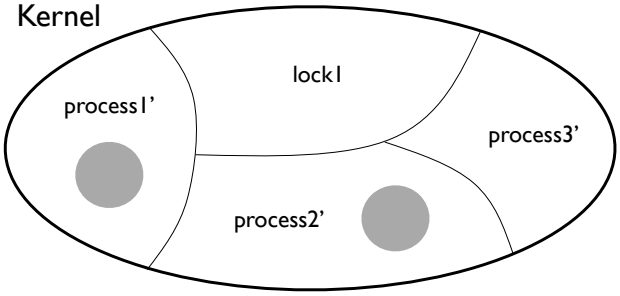


**Figure 4.** The state of the abstract system with one virtual CPU per process. Process identifiers are primed to emphasise that the virtual state of the process can be represented differently in the abstract and physical machines (cf. Figure 3). Dark regions illustrate the parts of process state that are tracked by a scheduler invocation running on a particular physical CPU.

deal with in this paper is that for realistic OS schedulers, defining a semantics for the abstract system a scheduler implements is difficult. This is because, in reasoning about mainstream operating systems, the ownership transfer between the scheduler and the kernel can involve not only fixed memory cells, but arbitrary *logical* facts describing them, which is difficult to describe operationally (see the treatment of the desc predicate in Section 4.3).

In this paper we resolve this problem in a novel way. Instead of defining the semantics of the abstract machine operationally, we define it only axiomatically as the high-level proof system described above. As expected, the low-level proof system is used to reason about the correspondence between the concrete and the abstract system, 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 6) 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 systems, the soundness statement relates logical statements about the abstract system (given by high-level proofs) to logical statements about the concrete one (given by a constraint on concrete states). We call this form of establishing a correspondence between the two systems a *logical refinement*. Note that in this case the soundness statement for the logic does not yield a semantic statement of correctness for the scheduler being considered. Rather, its correctness is established indirectly by the fact that reasoning in the high-level proof system, which assumes

$$\mathsf{Reg} = \{\mathtt{ip}, \mathtt{if}, \mathtt{ss}, \mathtt{sp}, \mathtt{gr}_1, \ldots, \mathtt{gr}_m\} \quad \mathsf{Loc} \subseteq \mathsf{Val}$$
$$\mathsf{Context} = \mathsf{Reg} \to \mathsf{Val} \quad \mathsf{CPUid} = \{1, \ldots, \mathtt{NCPUS}\}$$
$$\mathsf{GContext} = \mathsf{CPUid} \rightharpoonup \mathsf{Context} \quad \mathsf{Heap} = \mathsf{Loc} \rightharpoonup \mathsf{Val}$$
$$\mathsf{Lock} = \{\ell_1, \ell_2, \ldots, \ell_n\} \quad \mathsf{Lockset} = \mathcal{P}(\mathsf{Lock})$$
$$\mathsf{Config} = \mathsf{GContext} \times \mathsf{Heap} \times \mathsf{Lockset}$$

**Figure 5.** The set of machine configurations Config. We assume sets Loc of valid memory addresses and Val of values, respectively.

the abstract one-CPU-per-process machine, is sound with respect to the concrete machine.

To verify the scheduler separately from the processes it manages, low-level assertions focus only on a small relevant part of the state of the kernel, which we call *scheduler-visible*. Namely, the assertions relate the state local to a scheduler invocation on a particular CPU in the concrete system (e.g., the region marked CPU1 in Figure 3) to parts of abstract states of some of the OS processes (e.g., the dark regions in Figure 4). The latter parts can include, e.g., the values of registers of the virtual CPU of the process, but not the process-local memory. They are used in the low-level proof system to verify that the operations performed by the scheduler in the concrete machine correctly implement the required actions in the abstract machine. These parts also function as permissions to schedule the corresponding processes, i.e., a given part can be owned by at most one scheduler invocation at a time. For example, a scheduler invocation owning the parts of process states marked in Figure 4 has a permission to schedule processes 1 and 2, but not 3. Such a permission reading is crucial for handling scheduling on multiprocessors, as it ensures that a process may not be scheduled at two CPUs at the same time.

***Summary.*** In the following we formalise the above approach for a particular class of schedulers. Despite the formalisaton being performed for this class, the technical methods we develop here can be reused in other settings (see Section 8 for a discussion). In particular, we propose the following novel ideas:

- exploiting a logic validating the frame property to hide the state of the scheduler while verifying the kernel and vice versa;
- using a logical refinement in a context where defining an abstract semantics refined by the concrete one is difficult; and
- focusing on relevant parts of the two systems related in the refinement and giving a permission interpretation to them.

## 3. Preliminaries

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

### 3.1 Storage model

Figure 5 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, the state of the heap and the set of locks taken by some CPU. The configurations in which the heap or the global context is a partial function are not encountered in the semantics we define in this section. They come in handy in Sections 4 and 6 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 Commands

Programs for our machine consist of primitive commands $c$:

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

In addition to the primitive commands listed in Section 2, 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 $\mathtt{r}$; $[e] := e'$ updates the contents of cell $e$ by $e'$; $\mathtt{assume}(b)$ acts as a filter on the state space of programs, choosing those satisfying $b$. We write PComm for the set of primitive commands. Note that the commands cannot access the `ip` register directly.

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

The language constructs used in the example scheduler of Section 2, such as loops and conditionals, can be expressed as commands in a standard way, with conditions translated using `assume`.

### 3.3 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:

$$\mathsf{State} = \mathsf{Context} \times \mathsf{Heap} \times \mathsf{Lockset}$$
$$\rightsquigarrow_c \subseteq \big(\mathsf{CPUid} \times \mathsf{State} \times \mathsf{Label}^2\big) \times \big(\mathsf{State} \times \mathsf{Label} \cup \{\top\}\big)$$

The input to $\rightsquigarrow_c$ consists of four components. The first is the identifier of the CPU executing the command, and the next the configuration of the system projected to this CPU, which we call a state. The latter includes the context of the CPU and the information about the shared resources—the heap and locks. The last two components of the input are the labels of the command $c$ and a primitive command following it in the program. Given this input, the transition relation $\rightsquigarrow_c$ for $c$ nondeterministically 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 the label given as the last component of the input, when $c$ is a call or a return.

The relation $\rightsquigarrow_c$ appears in Figure 6. In the figure and in the rest of the paper, we write $\_$ for an expression whose value is irrelevant and implicitly existentially quantified. The relation follows the informal meaning of primitive commands given in Sections 2.1 and 3.2. Note that $\rightsquigarrow_c$ may yield no post-state for a given pre-state. Unlike a transition to the $\top$ state, this represents the divergence of the command. For example, according to Figure 6, acquiring the same lock twice leads to a deadlock, and releasing a lock that is not held crashes the system. Note that we do not prevent a held lock from being released by a CPU that did not acquire it, so locks behave like binary semaphores.

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 7. The first rule in the figure describes a normal execution, where the `ip` register of CPU $k$ is used to choose the primitive command $c$ to run. After choosing $c$, the machine picks the label $l'$ of a following command, runs $c$ according to the semantics $\rightsquigarrow_c$,

$$
\begin{array}{lll}
(k,(r,h[[\![e]\!]r:u],L),l,l') & \rightsquigarrow_{\mathtt{r}:=[e]} & ((r[\mathtt{r}:u],h[[\![e]\!]r:u],L),l') \\
(k,(r,h,L),l,l') & \rightsquigarrow_{\mathtt{assume}(b)} & ((r,h,L),l'), \text{ if } [\![b]\!]r = \mathsf{true} \\
(k,(r,h,L),l,l') & \not\rightsquigarrow_{\mathtt{assume}(b)} & \text{if } [\![b]\!]r = \mathsf{false} \\
(k,(r,h,L),l,l') & \rightsquigarrow_{\mathtt{lock}(\ell)} & ((r,h,L\cup\{\ell\}),l'), \text{ if } \ell \notin L \\
(k,(r,h,L),l,l') & \not\rightsquigarrow_{\mathtt{lock}(\ell)} & \text{if } \ell \in L \\
(k,(r,h,L),l,l') & \rightsquigarrow_{\mathtt{unlock}(\ell)} & ((r,h,L-\{\ell\}),l'), \text{ if } \ell \in L \\
(k,(r,h[[\![e]\!]r:\_],L),l,l') & \rightsquigarrow_{\mathtt{savecpuid}(e)} & ((r,h[[\![e]\!]r:k],L),l') \\
(k,(r,h[r(\mathtt{sp}):\_,\ldots,r(\mathtt{sp})+m:\_],L),l,l') & & \\
\quad \rightsquigarrow_{\mathtt{call}(l'')} & ((r[\mathtt{sp}:r(\mathtt{sp})+m+1], & \\
\quad h[r(\mathtt{sp}):l',r(\mathtt{sp})+1:r(\mathtt{gr}_1),\ldots,r(\mathtt{sp})+m:r(\mathtt{gr}_m)],L),l'') & & \\
(k,(r,h[r(\mathtt{sp})-m-1:l'',r(\mathtt{sp})-m:g_1,\ldots,r(\mathtt{sp})-1:g_m],L),l,l') & & \\
\quad \rightsquigarrow_{\mathtt{ret}} & ((r[\mathtt{sp}:r(\mathtt{sp})-m-1,\mathtt{gr}_1:g_1,\ldots,\mathtt{gr}_m:g_m], & \\
\quad h[r(\mathtt{sp})-m-1:l'',r(\mathtt{sp})-m:g_1,\ldots,r(\mathtt{sp})-1:g_m],L),l'') & & \\
(k,(r,h,L),l,l') \rightsquigarrow_c \top, \text{ otherwise} & &
\end{array}
$$

**Figure 6.** Semantics of primitive commands. We have omitted standard definitions for `skip` and most of assignments (see [21]). We have also omitted them for `icall` and `iret`: the definitions are the same as for `call` and `ret`, but additionally modify `if`. In the figure $\rightsquigarrow_c \top$ indicates that the command $c$ crashes, and $\not\rightsquigarrow_c$ means that it does not crash, but diverges. The function $[\![\cdot]\!]r$ evaluates expressions with respect to the context $r$.

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

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

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

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

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

**Figure 7.** Operational semantics of the machine

and uses the result of this run to update the registers of CPU $k$ and the heap and the lockset of the machine. The next rule 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. The logic

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. (Our results can be extended when new scheduler routines are introduced; see Section 8 for a discussion.) Our logic thus reasons about programs of the form:

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

where $\mathsf{C}$ and $\mathsf{S}$ are pieces of code implementing the `create` and `schedule` routines of the scheduler and $\mathsf{K}$ is the rest of the kernel code. Our high-level proof system is designed for proving $\mathsf{K}$, and the low-level system for proving $\mathsf{C}$ and $\mathsf{S}$.

We place several restrictions on programs. First, we require that $\mathsf{C}$ and $\mathsf{S}$ 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 simplicity, we assume that the scheduler data structures are properly initialised when the program starts executing.

To ensure that the scheduler routines execute with interrupts disabled, we require that $\mathsf{C}$ and $\mathsf{S}$ may not contain `icall`, `iret` and assignments accessing the `if` register. We also need to 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, $\mathsf{K}$ may not contain `savecpuid`, `icall` and `iret` (except calls to the scheduler routines `schedule` and `create`), assignments accessing `if` or writing to `ss`. In reality, a kernel might need to disable interrupts. We discuss how our results can be extended to handle this in Section 8. Finally, we require that the kernel $\mathsf{K}$ and the scheduler $\mathsf{C}$ and $\mathsf{S}$ access disjoint sets of locks. This condition simplifies the soundness statement in Section 6 and can be lifted.

The core part of our logic is the low-level proof system for verifying scheduler code, which we present in Section 4.4. It extends the high-level proof system used for verifying kernel code, which, in turn, adapts concurrent separation logic to our setting. For this reason, we present the high-level system first.

### 4.1 Assertion language

We now present the assertion language of the high-level proof system. Assertions describe properties of a single process, as if it were running on a separate virtual CPU. The state of the process thus consists of the values of the CPU registers (its context), the heap local to the process and the locks the process has a permission to release (its lockset). Mathematically, states of a process are just elements of $\mathsf{State}$ defined in Section 3.3: $\mathsf{State} = \mathsf{Context} \times \mathsf{Heap} \times \mathsf{Lockset}$. However, unlike in the semantics of Section 3.3, a heap here can be a partial function, with its domain defining the part of the heap owned by the process. A lockset is now meant to contain only the set of locks that the process has a permission to release (in our logic such permissions can be transferred between processes).

To denote sets of process states in our logic, we use a minor extension of the assertion language of separation logic [21]. Let $\mathsf{NVar}$ and $\mathsf{CVar}$ be disjoint sets containing logical variables for values and contexts, respectively. Assertions are defined as follows:

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

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 $G$. A context $G$ is either a logical variable or a finite map from register

$$
\begin{array}{ll}
(r,h,L) \models_\eta B & \text{iff } [\![B]\!]_\eta r = \mathit{true} \\
(r,h,L) \models_\eta P_1 \wedge P_2 & \text{iff } (r,h,L) \models_\eta P_1 \text{ and } (r,h,L) \models_\eta P_2 \\
(r,h,L) \models_\eta \mathsf{emp} & \text{iff } h = [\,] \text{ and } L = \emptyset \\
(r,h,L) \models_\eta E_0 \mapsto E_1 & \text{iff } h = [[\![E_0]\!]_\eta r : [\![E_1]\!]_\eta r] \text{ and } L = \emptyset \\
(r,h,L) \models_\eta E_0..E_1 \mapsto \Sigma & \text{iff } \exists j \geq 0.\, \exists v_1,\ldots,v_j \in \mathsf{Val}. \\
\quad L = \emptyset,\ \ j = [\![E_1]\!]_\eta r - [\![E_0]\!]_\eta r + 1,\ \ v_1 v_2 \ldots v_j = [\![\Sigma]\!]_\eta r & \\
\quad \text{and } h = [[\![E_0]\!]_\eta r : v_1, \ldots, [\![E_1]\!]_\eta r : v_j] & \\
(r,h,L) \models_\eta \mathsf{locked}(\ell) & \text{iff } h = [\,] \text{ and } L = \{\ell\} \\
(r,h,L) \models_\eta P_1 * P_2 & \text{iff } \exists h_1, h_2, L_1, L_2.\ h = h_1 \uplus h_2, \\
\quad L = L_1 \uplus L_2,\ \ (r,h_1,L_1) \models_\eta P_1 \text{ and } (r,h_2,L_2) \models_\eta P_2 &
\end{array}
$$

Predicate $\mathsf{dll}_\Lambda$ is the least one satisfying the equivalence below:

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

**Figure 8.** Semantics of high-level assertions. 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 $\eta$.

labels $\mathbf{r}$ to expressions. We denote the set of assertions defined here with $\mathsf{Assert}_\mathsf{K}$. Let a logical variable environment $\eta$ 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 $\mathsf{State}$ as defined by the satisfaction relation $\models_\eta$ in Figure 8. For an environment $\eta$ and an assertion $P$, we denote with $[\![P]\!]_\eta$ the set of states satisfying $P$.

The assertions in the first line of the definition of $P$ except $\mathsf{emp}$ are connectives from the first-order logic with the standard semantics. We can define the missing connectives from the given ones. The following assertions from $\mathsf{emp}$ up to the $\mathsf{dll}_\Lambda$ predicate are standard assertions of separation logic [21]. Informally, $\mathsf{emp}$ 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 $\Sigma$. For a value $u$ of a C type `t` taking several cells, we shorten $E..(E+\mathtt{sizeof(t)}-1) \mapsto u$ to just $E \mapsto u$. For a field `f` of a C structure, we use $E.\mathtt{f} \mapsto E'$ as a shortcut for $E + \mathit{off} \mapsto E'$, where *off* is the offset of `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. It assumes a C structure definition with fields `prev` and `next`. Here $E_h$ is the address of the head of the list, $E_t$ the address of its tail, $E_p$ the pointer in the `prev` field of the head node, and $E_n$ the pointer in the `next` field of the tail node. The $\Lambda$ parameter is a formula with one free logical variable describing the shape of each node in the list, excluding the `prev` and `next` fields; the logical variable defines the address of the node. For instance, a simple doubly-linked list can be expressed using $\Lambda(x) = \mathsf{emp}$. We included $\mathsf{dll}_\Lambda$ to describe the runqueues of the scheduler in our example. Predicates for other data structures can be added straightforwardly [21].

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 $\ell$, i.e., it denotes a permission to release the lock $\ell$. Note that $\mathsf{locked}(\ell) * \mathsf{locked}(\ell)$ is inconsistent: acquiring the same lock twice leads to a deadlock.

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

### 4.2 Interface parameters

As we noted in Section 2.3, our logic can be viewed as implementing a form of assume-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. We note that the interface parameters we present here are tied to a particular class of schedulers for which we present our logic. As we argue in Section 8, our results can be carried over to schedulers with more elaborate interfaces.

Ownership transfer happens at calls to and returns from the scheduler routines `create` and `schedule`. When the kernel calls the `create` routine of the scheduler, the latter should get the ownership of the process descriptor supplied as the parameter. In the two proof systems, we specify this descriptor using an assertion $\mathsf{desc}(d,\gamma) \in \mathsf{Assert}_\mathsf{K}$ with two free logical variables and no register occurrences. Our intention is that it describes the descriptor of a process with the context $\gamma$, 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. As the scheduler and the kernel access disjoint sets of locks, we require that all states in $[\![\mathsf{desc}(d,\gamma)]\!]_\eta$ have an empty lockset.

We fix the piece of state transferred from the kernel to the `schedule` routine upon an interrupt to be the free part of the stack of the process being preempted. The parameters determining its size are the size of the stack $\mathtt{StackSize} \in \mathbb{N}$ and the upper bound $\mathtt{StackBound} \in \mathbb{N}$ on the stack usage by the kernel (excluding the scheduler). To ensure that the stack does not overflow while calling an interrupt hander, we require that $\mathtt{StackSize} - \mathtt{StackBound} \geq m+1$, where $m$ is the number of general-purpose registers.

### 4.3 High-level proof system

The high-level proof system reasons about the kernel code $\mathsf{K}$. It is obtained by adapting concurrent separation logic to our setting and adding proof rules axiomatising the effect of scheduler routines.

The judgements of the high-level proof system are of the form $I, \Delta \vdash C$, where $I : \mathsf{Lock} \rightharpoonup \mathsf{Assert}_\mathsf{K}$ is a partial mapping from locks accessible in the kernel code to their invariants (see Section 2.3) and $\Delta : \mathsf{Label} \to \mathsf{Assert}_\mathsf{K}$ is a total mapping from code labels to preconditions. The parameter $\Delta$ in our judgement specifies local states of the process at various program points, which induce pre- and post-conditions for all primitive commands in $C$. When considering a complete system in Section 4.5, we restrict $\Delta$ so that it is $\mathsf{false}$ everywhere except at labels in the kernel code. An example of a lock invariant is

$$
\exists x, y.\, 10.\mathtt{prev} \mapsto y * 10.\mathtt{next} \mapsto x * \mathsf{dll}_\Lambda(x, 10, 10, y),
$$

where $\Lambda(x) = \mathsf{emp}$. It states that the lock protects a non-empty cyclic doubly-linked list with the head node at address 10. We forbid lock invariants to contain registers or free occurrences of logical variables. We consider a version of concurrent separation logic where resource invariants are allowed to be imprecise [20] at the expense of excluding the conjunction rule from the proof system [12].

The rule PROG-H for deriving the judgements is given in Figure 9. The first premise of the rule says that all assertions in $\Delta$ have to satisfy some restrictions regarding stack usage, formulated using parameters `StackSize` and `StackBound` introduced in Section 4.2. These ensure that the interrupt handler can safely execute on the stack of the process it preempts:

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

$$\frac{P \Rightarrow P' \quad I, \Delta \rhd_l \{P'\}\, c\, \{Q'\} \quad Q' \Rightarrow Q}{I, \Delta \rhd_l \{P\}\, c\, \{Q\}}\ \textsc{Conseq} \qquad \frac{I, \Delta \rhd_l \{P\}\, c\, \{Q\} \quad \mathsf{notCallRet}(c)}{I, \Delta \rhd_l \{\exists x.\, P\}\, c\, \{\exists x.\, Q\}}\ \textsc{Exists}$$

$$\frac{I, \Delta \rhd_l \{P_1\}\, c\, \{Q_1\} \quad I, \Delta \rhd_l \{P_2\}\, c\, \{Q_2\}}{I, \Delta \rhd_l \{P_1 \vee P_2\}\, c\, \{Q_1 \vee Q_2\}}\ \textsc{Disj} \qquad \frac{I, \Delta \rhd_l \{P\}\, c\, \{Q\} \quad \mathsf{mod}(c) \cap \mathsf{free}(F) = \emptyset \quad \mathsf{notCallRet}(c)}{I, \Delta \rhd_l \{P * F\}\, c\, \{Q * F\}}\ \textsc{Frame}$$

$$\frac{}{I, \Delta \rhd_l \{P\}\, \mathtt{assume}(b)\, \{P \wedge b\}}\ \textsc{Assume} \qquad \frac{}{I, \Delta \rhd_l \{e \mapsto \_\}\, [e]{:=}e'\, \{e \mapsto e'\}}\ \textsc{Store}$$

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

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

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

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

**Figure 9.** High-level proof system. Here $\mathsf{mod}(c)$ is the set of registers modified by $c$, $\mathsf{free}(F)$ is the set of registers appearing in $F$, and $\mathsf{notCallRet}(c)$ means that $c$ is not one of `call`, `icall`, `ret` and `iret`. Finally, $\mathsf{id} = [\mathbf{ip} : \_, \mathbf{if} : \mathtt{if}, \mathbf{ss} : \mathtt{ss}, \mathbf{sp} : \mathtt{sp}, \vec{\mathbf{gr}} : \vec{\mathtt{gr}}]$.

- the free part of the stack of the process must always be in its local state so that it can be transferred to the 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 other condition in the PROG-H rule is that for every primitive command $c$ in $C$ and the label $l'$ of a command following $c$, we have to prove $I, \Delta \rhd_{l'} \{\Delta(l)\}\ c\ \{\Delta(l')\}$. 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. Note that the italicised clause enforces the frame property (Section 2.3). The proof rules for such judgements are also given in Figure 9. The rules CONSEQ, DISJ and EXISTS are standard rules of Hoare logic. The FRAME rule is inherited from separation logic; it 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. To keep the logic sound we have to forbid EXISTS and FRAME to be applied to calls or returns. The logic also provides standard separation logic axioms for primitive commands. In Figure 9 we show two of them, ASSUME and STORE, and omit the others to save space; see [21].

The LOCK and UNLOCK axioms are inherited from concurrent separation logic and provide tools for modular reasoning about concurrent processes. The LOCK axiom says that, upon acquiring a lock, the process gets the ownership of its invariant and a permission to release it. 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.

The CALL and RET axioms mirror the operational semantics of `call` and `ret` (see Section 2.1 and Figure 6). 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 $\Delta$ at the target label. RET similarly requires the precondition to establish the assertion at the target label after the values of general-purpose registers and `ip` (denoted with $\vec{E}$ 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 FRAME rule 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. This issue is orthogonal to the problem of scheduler verification we are concerned with, thus, in this paper we chose the simplest high-level logic possible. See Section 8 for pointers to more expressive logics for procedures.

What we have presented so far is just an adaptation of concurrent separation logic to our setting. We now provide axioms for calling the scheduler routines `schedule` and `create`, which are specific to our logic. As the high-level proof system hides the implementation of scheduler routines, the corresponding axioms are significantly different from CALL. In particular, the axioms are formulated as if after these `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-flow 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 heap. The axiom does not place any requirements on the process, as the preconditions necessary for the execution of `schedule`, which anyway can be invoked at any time as the interrupt handler, are established by the first condition in 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 $\gamma$. 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 4.5). 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 [13]. The premise 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 $\gamma$ 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 2: $\mathsf{desc}(d, \gamma) = d.\mathtt{prev} \mapsto \_ * d.\mathtt{next} \mapsto \_ * \mathsf{desc}_0(d, \gamma)$, where

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

and `SCHED_FRAME` is the size of the activation record of `schedule` (Figure 1). The descriptor does not include filled stack slots; they can be passed to the process directly in the precondition $P$.

As we have noted before, $\mathsf{desc}(d, \gamma)$ can be an arbitrary logical predicate. In some cases, e.g., when it is imprecise [20], its transfer from the kernel to the scheduler is hard to express operationally when defining a semantics of the kernel separately from the implementation of the scheduler; see [12] for a discussion. The situation would be worse had we based our logic on one of advanced modular concurrency logics, such as deny-guarantee [4], which are 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. The above difficulties with an operational definition of ownership transfer are a prime reason for using logical refinement in this paper.

The high-level proof system provides modern tools for modular reasoning about concurrent processes using proof rules of concurrent separation logic. The PROG-H rule of the system subsumes the usual sequential composition rule of Hoare logic, which assumes that the control-flow follows the structure of the process code and ignores the possibility of scheduler code getting executed at an interrupt. The axioms SCHED and CREATE abstract the implementation of scheduler routines by treating them like atomic commands. Thus, the state and the control-flow of the scheduler is completely hidden by the proof system. The soundness of such an illusion is established by verifying the scheduler code using a low-level proof system, which we describe next.

### 4.4 Low-level proof system

We now present the core of our logic—the low-level proof system, which is used to prove that the commands C and S of the OS program implement scheduling correctly. As we explained in Section 2.3, assertions of the proof system relate the states of the concrete machine and an abstract one, where every process has its own virtual CPU. The state of the concrete machine can be described using separation logic assertions introduced in Section 4.1. To describe states of the abstract machine, we extend the assertion language of Section 4.3 with an additional predicate: $P ::= \ldots \mid \mathsf{Process}(G)$, where $G$ ranges over context expressions. We denote the set of such assertions with $\mathsf{Assert_S}$. The $\mathsf{Process}(G)$ predicate describes a process with the values of registers of its virtual CPU given by the context $G$.

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

**Figure 10.** Semantics of low-level assertions. The $\uplus$ operation on multisets adds up the number of occurrences of each element in its operands.

The addition of the Process predicate changes objects described by assertions: they now denote relations defined by subsets of $\mathsf{RelState} = \mathsf{State} \times \mathcal{M}(\mathsf{Context})$, where $\mathcal{M}(A)$ is the set of all finite multisets with elements from $A$. Relations in RelState connect the states of the concrete machine and the abstract machine with one CPU per process. As we have noted before, 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 3). The multiset in the second part records the scheduler-visible states of processes described by Process predicates in the assertion, i.e., parts of their local states that may be referred to by proofs about the scheduler (cf. the dark regions in Figure 4). These include the context of a process, but exclude its local heap and lockset: the latter are irrelevant for the schedulers we consider here and are therefore invisible to them. The low-level logic we present in this section is based on separation logic, hence, the invisibility of parts of process state to the scheduler automatically guarantees that it cannot access them.

Apart from keeping track of the state of a process, a Process predicate serves in the logic as an exclusive permission for the scheduler invocation owning it to schedule the corresponding process. To enforce this, the semantics of assertions defined below forbids the duplication of Process predicates: $\mathsf{Process}(G) \not\Rightarrow \mathsf{Process}(G) * \mathsf{Process}(G)$. Furthermore, the proof obligations for the scheduler we define in Section 4.5 state that it needs a Process predicate to schedule a process. Such a permission interpretation of Process is a key feature of our logic that allows us to reason about schedulers for multiprocessors: it ensures that, at a given time, only one scheduler invocation can own a Process predicate for a process, and hence, it can be scheduled only on one CPU at a time.

We give the formal semantics of assertions using the satisfaction relation $\models_\eta$ in Figure 10, parameterised by environments $\eta$. The first two cases in the figure are the most interesting ones. $\mathsf{Process}(G)$ relates a scheduler invocation having the empty heap and the empty lockset to a single process with the register values $G$. To be related by the separating conjunction $P * Q$, all parts of the state-multiset pair except the context should be split such that the first part is related by $P$ and the second by $Q$. The semantic definitions of the remaining assertions are obtained from the corresponding cases in our high-level proof system (Figure 8) 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 $P \wedge Q$. We denote with $[\![P]\!]_\eta$ the set of states satisfying $P$.

The judgements of the low-level proof system have the form $I, \Delta \vdash_k C$, where $k \in \mathsf{CPUid}$, $I : \mathsf{Lock} \rightharpoonup \mathsf{Assert_S}$ is a vector of resource invariants for locks accessible to the scheduler, and $\Delta : \mathsf{Label} \to \mathsf{Assert_S}$ is a mapping from program positions to low-level assertions. When considering a complete system in Section 4.5,

we restrict $\Delta$ so that it is false everywhere except at labels in the scheduler code. The intuitive meaning of the judgements is the same as in the high-level system (Section 4.3), with the component describing scheduler-visible 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:

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

Note that the syntactic structure of the OS program (see the beginning of Section 4) 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 9, 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 $k$:

$$\frac{}{I, \Delta \rhd^k_l \{e \mapsto \_\}\ \texttt{savecpuid}(e)\ \{e \mapsto k\}}\ \textsc{CPUid}$$

### 4.5 Putting the two proof systems together

The proof systems presented in Sections 4.3 and 4.4 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 4.

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 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 be maintained when the scheduler is not running on it. Similarly to lock invariants, we forbid $J_k$ 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 with $J$ the vector of invariants $J_k$.

Consider assertions $I_\mathsf{K}, \Delta_\mathsf{K}$ and $I_\mathsf{S}, \Delta^k_\mathsf{S}$ for all $k \in \mathsf{CPUid}$, such that:

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

The proof rule for the program OS is as follows:

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

where

$$\begin{array}{l} \mathsf{SchedState}_k = \exists l, \vec{g}.\, \mathtt{if}{=}0 \wedge 0 \le \mathtt{sp}{-}\mathtt{ss}{-}m{-}1 \le \mathtt{StackBound} \\ \wedge\ (\mathtt{sp}{-}m{-}1)..(\mathtt{sp}{-}1) \mapsto l\vec{g} * \mathtt{sp}..(\mathtt{ss}{+}\mathtt{StackSize}{-}1) \mapsto \_ * \\ \qquad J_k * \mathsf{Process}([\mathbf{ip} : l, \mathbf{if} : 1, \mathbf{ss} : \mathtt{ss}, \mathbf{sp} : \mathtt{sp}{-}m{-}1, \vec{\mathbf{gr}} : \vec{g}]) \end{array}$$

The first three premises 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 $k$ right after it is called using `icall` or before it returns by executing `iret`.

When `schedule` is called, the stack satisfies the bound on stack usage and interrupts are disabled. 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 precondition of `schedule` mirrors the first premise 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.

The precondition of `create` is similar to that of `schedule`, but additionally assumes a process descriptor for a new process with the address in $\mathtt{gr}_1$, 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. 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 proof rule in Section 4.5. By the soundness theorem for our logic (presented in Section 6), 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. The detailed proof is given in [14, Appendix A]. Here we present only lock and per-CPU scheduler invariants together with some informal explanations.

The invariants of runqueue locks are as follows:

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

where $\Lambda(d) = \exists \gamma.\, \mathsf{desc}_0(d, \gamma) * \mathsf{Process}(\gamma)$ and $\mathsf{desc}_0$ is defined in Section 4.3. The per-CPU scheduler invariants are:

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

According to these definitions, a runqueue for a CPU $k$ contains a list of descriptors of preempted processes together with Process predicates matching the state stored in them. When an invocation of `schedule` acquires the runqueue lock and removes a node from the list, it gets the ownership of the corresponding Process predicate, which lets it schedule the process by establishing the postcondition $\mathsf{SchedState}_k$ of `schedule` (see Section 4.5). The descriptor of the process just scheduled, pointed to by an entry in the `current` array, forms the scheduler's per-CPU state and is described by $J_k$. 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.

## 6. Soundness

In this section, we explain the guarantees about the entire kernel that follow from proofs in our logic. Consider a program $\mathsf{OS}$ of the form introduced in Section 4. We formulate a theorem, proved in [14, Appendix B], which describes how proofs of a scheduler and the kernel in our logic can be combined to construct an inductive invariant of the entire system. To aid understanding, we first state the theorem and explain the components used to formulate it informally. Only after this do we provide formal definitions.

THEOREM 1. *If* $I_\mathsf{K}, \Delta_\mathsf{K} \mid I_\mathsf{S}, \{\Delta_\mathsf{S}^k\}_{k\in\mathsf{CPUid}} \mid J \vdash (\mathsf{S}, \mathsf{C}, \mathsf{K})$*, then for all environments* $\eta$*, the following set of configurations* $\mathcal{R}_k$ *is preserved by* $\rightarrow_\mathsf{OS}$*:*

$$\begin{array}{l}\mathsf{compose}(\bigcup_{L\uplus L'=\mathsf{dom}(I_\mathsf{S})} \mathsf{held}_\mathsf{S}(L) \cap (\mathsf{lowinv}_\eta \star_\mathsf{S} \mathsf{lowlock}_{L'}),\\ \qquad\bigcup_{L\uplus L'=\mathsf{dom}(I_\mathsf{K})} \mathsf{held}_\mathsf{K}(L) \cap (\mathsf{highinv}_\eta \star_\mathsf{K} \mathsf{highlock}_{L'}))\end{array}$$

***Informal explanation.*** The invariant $\mathcal{R}$ is constructed in several steps by conjoining the descriptions of pieces of program state owned by different OS components. First, from assertions $\Delta_\mathsf{S}^k$ and $J$ in the proof of the scheduler, we construct a predicate

$$\mathsf{lowinv}_\eta \subseteq \mathsf{RelConfig} \stackrel{\text{def}}{=} \mathsf{Config} \times \mathcal{M}(\mathsf{Context})$$

Consider $((R, h, L), M) \in \mathsf{lowinv}_\eta$. For register values of the CPUs in the machine given by $R$, the components $h$ and $L$ describe the part of the machine state belonging to the scheduler, and $M$ the contexts of the processes it has a permission to schedule. Similarly, from assertions $\Delta_\mathsf{K}$ in the proof of the kernel, we construct a predicate

$$\mathsf{highinv}_\eta \subseteq \mathsf{HighConfig} \stackrel{\text{def}}{=} \mathcal{M}(\mathsf{Context}) \times \mathsf{Heap} \times \mathsf{Lockset}$$

Consider $(M, h, L) \in \mathsf{highinv}_\eta$. For any set of processes with the contexts given by $M$, the components $h$ and $L$ describe the part of the machine state belonging to these processes.

To construct the complete machine state, we also have to take into account the parts of the heap protected by free locks. Thus, for any set of free locks $L'$ accessible to the scheduler, from resource invariants $I_\mathsf{S}$ we construct a predicate $\mathsf{lowlock}_{L'} \subseteq \mathsf{RelConfig}$ describing the state protected by the locks. A similar predicate $\mathsf{highlock}_{L'} \subseteq \mathsf{HighConfig}$, constructed from $I_\mathsf{K}$, describes the state protected by a set of free locks $L'$ accessible to the kernel. The predicates $\mathsf{lowlock}_{L'}$ and $\mathsf{highlock}_{L'}$ are then combined with $\mathsf{lowinv}_\eta$ and $\mathsf{highinv}_\eta$ using operations

$$\begin{array}{l}\star_\mathsf{S} : \mathcal{P}(\mathsf{RelConfig}) \times \mathcal{P}(\mathsf{RelConfig}) \to \mathcal{P}(\mathsf{RelConfig})\\ \star_\mathsf{K} : \mathcal{P}(\mathsf{HighConfig}) \times \mathcal{P}(\mathsf{HighConfig}) \to \mathcal{P}(\mathsf{HighConfig})\end{array}$$

To ensure that $L'$ is indeed the set of all free locks, we require that the rest of the locks $L$ are held by intersecting the result with $\mathsf{held}_\mathsf{S}(L) \subseteq \mathcal{P}(\mathsf{RelConfig})$ or $\mathsf{held}_\mathsf{K}(L) \subseteq \mathcal{P}(\mathsf{HighConfig})$.

Finally, we connect the resulting predicates describing the states of the scheduler and the kernel using a form of relational composition, implemented by

$$\mathsf{compose} : \mathcal{P}(\mathsf{RelConfig}) \times \mathcal{P}(\mathsf{HighConfig}) \to \mathcal{P}(\mathsf{Config})$$

The operation conjoins the heaps and locksets described by the predicates and makes sure that the scheduler-visible states of processes they describe match. The result is an invariant of the entire machine maintained by each step of the kernel or the scheduler.

We now formally define the above operations and predicates.

***Composition operations.*** Each of the operations $\star_\mathsf{S}$, $\star_\mathsf{K}$ and $\mathsf{compose}$ is obtained by lifting a partial function in $A \times B \rightharpoonup C$ to a function in $\mathcal{P}(A) \times \mathcal{P}(B) \to \mathcal{P}(C)$ pointwise. To define $\star_\mathsf{K}$ we lift the operation $\bullet_\mathsf{K}$ on $\mathsf{HighConfig}$ that combines the information about processes, heaps and locksets:

$$(M_1, h_1, L_1) \bullet_\mathsf{K} (M_2, h_2, L_2) = (M_1 \uplus M_2, h_1 \uplus h_2, L_1 \uplus L_2)$$

(Recall that the $\uplus$ operation on multisets adds up the number of occurrences of each element in its operands.)

To define $\star_\mathsf{S}$ we similarly lift $\bullet_\mathsf{S}$ on $\mathsf{RelConfig}$ that combines the information about contexts, heaps, locksets and processes:

$$\begin{array}{l}((R_1, h_1, L_1), M_1) \bullet_\mathsf{S} ((R_2, h_2, L_2), M_2)\\ \qquad = ((R_1 \uplus R_2, h_1 \uplus h_2, L_1 \uplus L_2), M_1 \uplus M_2)\end{array}$$

Finally, we lift $\bullet_{\mathsf{compose}} : \mathsf{RelConfig} \times \mathsf{HighConfig} \rightharpoonup \mathsf{Config}$ that combines heaps and locksets provided the scheduler-visible states of processes in both arguments match:

$$((R, h_1, L_1), M_1) \bullet_{\mathsf{compose}} (M_2, h_2, L_2) = (R, h_1 \uplus h_2, L_1 \uplus L_2)$$

if both unions are defined and $M_1 = M_2$; undefined otherwise. It is this operation that carries over statements proved in the high-level proof system about the abstract machine with one virtual CPU per process to the concrete machine: the second operand $(M_2, h_2, L_2)$ represents the state owned by the processes running on the abstract machine, and the first $((R, h_1, L_1), M_1)$ relates the scheduler state in the concrete machine to the processes it has permissions to schedule. The components $M_1$ and $M_2$ are used to ensure that the two operands describe the same set of processes.

***Predicate definitions.*** Consider $p \subseteq \mathsf{RelState}$ and $q \subseteq \mathsf{State}$. Given $k \in \mathsf{CPUid}$ and $r \in \mathsf{Context}$, we define the following embedding operations converting states to configurations:

$$\begin{array}{ll}\lfloor p \rfloor_k &= \{(([k : r], h, L), M) \in \mathsf{RelConfig} \mid ((r, h, L), M) \in p\}\\ \lfloor q \rfloor_r &= \{(\{r\}, h, L) \in \mathsf{HighConfig} \mid\\ &\qquad (r, h \uplus [r(\mathtt{sp})..(r(\mathtt{ss})+\mathtt{StackSize}-1) : \_], L) \in q\}\\ \lfloor p \rfloor &= \{(([], h, L), M) \in \mathsf{RelConfig} \mid ((r, h, L), M) \in p\}\\ \lfloor q \rfloor &= \{(\emptyset, h, L) \in \mathsf{HighConfig} \mid (r, h, L) \in q\}\end{array}$$

The first one tags states with CPU identifiers and is used to construct $\mathsf{lowinv}_\eta$. The second selects the states with a given context $r$ and is used for $\mathsf{highinv}_\eta$. For technical reasons it removes the empty slots of the process stack, which are accounted for in the scheduler state (see the definition of $\mathsf{SchedSleep}_k$ below). The remaining two operations are used for $\mathsf{lowlock}_{L'}$ and $\mathsf{highlock}_{L'}$. As resource invariants do not restrict registers, they ignore contexts. We also need predicates defining states where the CPU is at a particular label $l$, or configurations with a particular lockset $L$:

$$\begin{array}{rl}\mathsf{at}_\mathsf{S}(l) &= \{((r, h, L), M) \in \mathsf{RelState} \mid r(\mathtt{ip}) = l\}\\ \mathsf{held}_\mathsf{S}(L) &= \{((R, h, L), M) \in \mathsf{RelConfig}\}\\ \mathsf{held}_\mathsf{K}(L) &= \{(M, h, L) \in \mathsf{HighConfig}\}\end{array}$$

The following predicate describes the state of the scheduler on CPU $k$, when a process is running on this CPU and is at label $l$:

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

Finally, let $\circledast_{k\in\mathsf{CPUid}}^{\star_\mathsf{S}}$ and $\circledast_{k\in\mathsf{CPUid}}^{\star_\mathsf{K}}$ be the iterated versions of $\star_\mathsf{S}$ and $\star_\mathsf{K}$.

Using the above notation, we can define the predicates from the theorem. For $L_\mathsf{S} \subseteq \mathsf{dom}(I_\mathsf{S})$ and $L_\mathsf{K} \subseteq \mathsf{dom}(I_\mathsf{K})$, we have:

$$\begin{array}{l}\mathsf{lowinv}_\eta = \circledast_{k\in\mathsf{CPUid}}^{\star_\mathsf{S}} (\bigcup_{l\in\mathsf{labels}(\mathsf{K})} \lfloor [\![\mathsf{SchedSleep}_k(l)]\!]_\eta \cap \mathsf{at}_\mathsf{S}(l) \rfloor_k \cup\\ \qquad\bigcup_{l\in(\mathsf{labels}(\mathsf{S}\uplus\mathsf{C})\uplus\{l_s, l_c\})} \lfloor [\![\Delta_\mathsf{S}^k(l)]\!]_\eta \cap \mathsf{at}_\mathsf{S}(l) \rfloor_k)\end{array}$$

$$\mathsf{highinv}_\eta = \bigcup_{M\in\mathcal{M}(\mathsf{Context})} \circledast_{r\in M}^{\star_\mathsf{K}} \lfloor [\![\Delta_\mathsf{K}(r(\mathtt{ip}))]\!]_\eta \rfloor_r$$

$$\mathsf{lowlock}_{L_\mathsf{S}} = \circledast_{\ell\in L_\mathsf{S}}^{\star_\mathsf{S}} \lfloor [\![I_\mathsf{S}(\ell)]\!] \rfloor$$

$$\mathsf{highlock}_{L_\mathsf{K}} = \circledast_{\ell\in L_\mathsf{K}}^{\star_\mathsf{K}} \lfloor [\![I_\mathsf{K}(\ell)]\!] \rfloor$$

The definitions follow the informal explanation given at the beginning of this section. To determine the state of the scheduler on a given CPU when defining $\mathsf{lowinv}_\eta$, we branch over all possible program positions $l$ of that CPU. Depending on whether $l$ is in the scheduler or the kernel code, we use either the assertion in the scheduler proof or the invariant $\mathsf{SchedSleep}_k$, describing the state of the scheduler when it is not running. Since assertions do not restrict the value of the `ip` register, we have to do this explicitly using $\mathsf{at}_\mathsf{S}$. Note that, 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 simplicity we choose always to count them in the scheduler state (the assertion in $\Delta^k_\mathsf{S}$ or the scheduler invariant $\mathsf{SchedSleep}_k$).

To define $\mathsf{highinv}_\eta$ we branch over all possible finite multisets of contexts $M$, representing 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 duplicate of an element in the multiset separately.

Finally, $\mathsf{lowlock}_{L_\mathsf{S}}$ and $\mathsf{highlock}_{L_\mathsf{K}}$ are straightforward combinations of resource invariants for the given sets of locks.

***Ownership transfer.*** It is instructive to analyse how 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 $\mathcal{R}$ counts the descriptor as part of $\mathsf{highinv}_\eta$. Since the implementation of `create` receives the descriptor in its precondition, in the configuration immediately after the call to `create`, $\mathcal{R}$ defines it to be part of $\mathsf{lowinv}_\eta$. Thus, ownership transfer repartitions program state among the parts defined in Theorem 1.

***Consequences.*** Theorem 1 allows us to check invariance properties of preemptable code. For example, assume that the initial configuration satisfies $\mathcal{R}$. Then the soundness statement ensures that the machine cannot reach an error label $l_e$ on any CPU, provided the assertion at this program point in all high-level proofs is $\mathsf{false}$. Indeed, in this case the invariant $\mathcal{R}$ 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 simulation between the kernel and its specification. As an OS kernel does not usually make any assumptions about user processes, proving the simulation can be reduced to proving an invariance property relating the two (e.g., [10, 17]). Thus, Theorem 1 can be also used to justify such proofs.

## 7. Related work

There have been a number of OS verification projects; see [16] 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 L4.verified project [17], which verified 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 preservation of its runqueue. The proof was still tractable because the kernel was running on a uniprocessor and preemption was disabled most of the time. However, such architecture is not used by mainstream operating systems.

The closest work to ours is the one by Feng et al. [6–8], who verified an idealised scheduler without dynamic process creation. Their logic considers a uniprocessor and does not handle ownership transfer between the scheduler and processes. Like us, they have separate proof systems for the scheduler and preemptable code. However, their high-level system is non-modular in that it does not have a notion of a process-local state. Their approach to low-level reasoning and proving the soundness of the logic is also different from ours. Because Feng et al. consider a restricted scheduler and high-level proof system, they are able to avoid designing a special relational low-level logic. Instead, they view calls to and returns from the scheduler as jumps and compile proofs of the scheduler and the rest of the system into OCAP [6], a logic supporting first-class code pointers. According to our understanding, extending this approach to handle multiprocessing, ownership transfer and a modular high-level proof system would be non-trivial.

Maeda and Yonezawa have proved a simple context-switch routine using an extension of alias types [19]. 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 [24] have recently proposed 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. 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 verification of both operating systems and general concurrent programs [1, 10, 15, 17, 22]. We advance it further by proposing its novel form where the target of the refinement is defined axiomatically and refinement relations focus only on the relevant state of the systems related. This allows us to handle systems with complex ownership transfers.

## 8. Discussion

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. 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.

The only way we could communicate the proposed reasoning principles understandably is by presenting our results in a simplified setting. Besides, we could not cover all the interesting features of mainstream OS kernels, even in regards to scheduling, in one paper. Below we list some of the limitations of our results and possible ways to lift them, which also provide avenues for future work:

- 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. The proof of soundness of our logic follows an approach that has been applied extensively to various concurrent derivatives of separation logic [11, 12]. This leads us to believe that we can integrate more advanced logics from this class [4, 5, 23] without problems.
- Our treatment of procedure calls is naive in that it does not allow us to reason about procedures modularly. We consider this problem orthogonal to our goal and believe that our logic can be combined with more powerful logics for procedures in low-level code, such as [9].

- We have considered schedulers with only two procedures in their interface, and fixed the piece of state transferred between the scheduler and the kernel at `schedule` to be the empty slots of the process stack. It is straightforward to add new procedures and define their pre- and postconditions abstractly, like `desc` in the precondition of `create`. The real issue is how to restrict the ways the scheduler is allowed to change the state it receives before giving the state back to the kernel. For example, in some operating systems (e.g., XNU), `schedule` can receive the ownership of the whole stack of the process being preempted and may reallocate the stack when it schedules the process again, while preserving its contents. Such an interference is routinely described in combinations of separation logic and rely-guarantee [4, 5, 23] and can be integrated into our logic.
- Modern OS kernels have a number of features that break through the abstraction of a virtual CPU implemented by the scheduler. For example, they allow preemptable code to disable interrupts, e.g., to access data structures local to a particular CPU. The effects of such features can be axiomatised in the high-level logic in much the same way as we axiomatise the effect of the `create` routine of the scheduler. We plan to report on extensions of our logic to such features in future papers.
- Our logic is designed for proving safety properties only. Proof methods for liveness properties or the absence of deadlocks usually rely on modular methods for safety properties. Thus, our logic is a prerequisite for attacking liveness in the future.

Despite the above limitations, our logic is the first to handle patterns of interaction between the scheduler and the kernel that are present in mainstream operating systems. Even though the logic has been formalised in a particular setting, its key technical ideas—the use of proof systems validating the frame property, logical refinement and a local way of establishing it—are transferable and can be reused in OS verification projects.

## Acknowledgements

We would like to thank Anindya Banerjee, Xinyu Feng, Boris Koepf, 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. Yang was supported by EPSRC.

## References

[1] R.-J. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23:49–68, 1981.

[2] D. Bovet and M. Cesati. *Understanding the Linux Kernel, 3rd ed.* O'Reilly, 2005.

[3] E. Cohen, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In *CAV'10: Conference on Computer-Aided Verification*, volume 6174 of *LNCS*, pages 480–494. Springer, 2010.

[4] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP'10: European Conference on Object-Oriented Programming*, pages 504–528. Springer, 2010.

[5] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP'07: European Conference on Programming*, volume 4421 of *LNCS*, pages 173–188. Springer, 2007.

[6] X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *TLDI'07: Workshop on Types in Language Design and Implementation*, pages 67–78. ACM, 2007.

[7] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI'08: Conference on Programming Language Design and Implementation*, pages 170–182. ACM, 2008.

[8] X. Feng, Z. Shao, Y. Guo, and Y. Dong. Combining domain-specific and foundational logics to verify complete software systems. In *VSTTE'08: Conference on Verified Software: Theories, Tools, Experiments*, volume 5295 of *LNCS*, pages 54–69. Springer, 2008.

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

[10] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In *TPHOLs: Conference on Theorem Proving in Higher Order Logics*, volume 3603 of *LNCS*, pages 1–16. Springer, 2005.

[11] A. Gotsman. Logics and analyses for concurrent heap-manipulating programs. PhD Thesis, University of Cambridge, 2009.

[12] A. Gotsman, J. Berdine, and B. Cook. Precision and the conjunction rule in concurrent separation logic. In *MFPS'11: Conference on Mathematical Foundations of Programming Semantics*, 2011. To appear.

[13] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS'07: Asian Symposium on Programming Languages and Systems*, volume 4807 of *LNCS*, pages 19–37. Springer, 2007.

[14] A. Gotsman and H. Yang. Modular verification of preemptive OS kernels (extended version). Available from `www.software.imdea.org/~gotsman`.

[15] C. Jones. Splitting atoms safely. *Theoretical Computer Science*, 375:109–119, 2007.

[16] G. Klein. Operating system verification–an overview. *Sādhanā*, 34:26–69, 2009.

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

[18] R. Love. *Linux Kernel Development, 3rd ed.* Addison Wesley, 2010.

[19] T. Maeda and A. Yonezawa. Writing an OS kernel in a strictly and statically typed language. In *Formal to Practical Security*, volume 5458 of *LNCS*, pages 181–197. Springer, 2009.

[20] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375:271–307, 2007.

[21] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02: Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[22] A. Turon and M. Wand. A separation logic for refining concurrent objects. In *POPL'11: Symposium on Principles of Programming Languages*, pages 247–258. ACM, 2011.

[23] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR'07: Conference on Concurrency Theory*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.

[24] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *PLDI'10: Conference on Programming Language Design and Implementation*, pages 99–110. ACM, 2010.